ARM Prep Questions

What is wrap boundary 4kb in AXI Protocol?

In AXI (Advanced eXtensible Interface), a 4KB (4096 bytes) boundary refers to the address alignment requirement for memory access. When a transaction crosses a 4KB boundary, it means the transfer's address range extends beyond a single 4KB-aligned block of memory. This can lead to issues like data corruption or undefined behavior, especially with devices that are read-sensitive or where memory mapping is crucial, such as FIFOs or memory controllers.

The constraint below will ensure that a transaction does not cross the 4Kb boundary:

```
constraint axi_4kb {
   awaddr % 4096 + (2**awsize * (awlen + 1)) <= 4096;
}
```

Check out the Below Executable Code and Output Below:

```
class axi_tx;
  rand bit [31:0] awaddr;
  rand bit [ 3:0] awsize;
  rand bit [ 7:0] awlen;

  constraint axi_4kb {
    awaddr % 4096 + (2**awsize * (awlen + 1)) <= 4096;
  }
endclass

module top;
  axi_tx tx=new();
  initial begin
        repeat (5) begin
        assert(tx.randomize());
```

```
        $display("awaddr = %0d         awsize = %0d   awlen =
%0d",tx.awaddr,tx.awsize,tx.awlen);

        $display("addr_offset = %0d Total_txn =
%0d",tx.awaddr%4096,(2^tx.awsize*(tx.awlen+1)));

        $display("offset + total_txn <= 4096");

        $display("    %0d   <= 4096 \n",(tx.awaddr%4096+(2**tx.awsize*(tx.awlen+1))));

   end

 end

endmodule
```

Data in axi3/axi4 packed array or unpacked array with unique or shuffle?

In AXI3/AXI4 protocols, data is typically transmitted as packed arrays, not unpacked arrays. AXI focuses on transferring data as a series of words (typically 32 or 64 bits) rather than individual bits or bytes, which is characteristic of packed arrays. The concepts of "unique" or "shuffle" primarily relate to the ordering of data within a transaction, not the structure of the data itself, and are handled by the AXI protocol's addressing and ordering mechanisms, not the array type.

- Why AXI Prefers Packed Arrays:

AXI's burst transfers, which are a common way to transfer large amounts of data, are designed to work with whole words. Transferring unpacked arrays would require multiple small transactions or complex data formatting, which would reduce efficiency.

- "Unique" and "Shuffle" in AXI:
    - Unique: The AXI protocol ensures that data is transferred in a specific order, and each data value is unique within a given transaction. The address and data are used to ensure that the data is transferred to the correct location in memory, and that each data value is processed in the correct order.
    - Shuffle: AXI generally does not support "shuffling" of data within a single transaction. The ordering of data within a burst transfer is determined by the order in which the data was written to the AXI master, and AXI does not support rearranging the data on the bus.

How axi4 works with no WID?

In AXI4, the WID (Write ID) signal is not required because AXI4 removes support for write data interleaving. This means that all write data for a transaction must be transferred in a consecutive order, and there's no need to distinguish between different IDs for write operations. Without write interleaving, the WID signal becomes redundant, and it's not specified in the AXI4 protocol.

Elaboration:

- Write Interleaving:

In AXI3, write data could be interleaved (sent out of order) using the WID signal to track which write data belonged to which transaction. This allowed multiple write transactions to be in progress simultaneously.

- AXI4 Simplification:

AXI4 simplifies the protocol by removing write interleaving. This means that all write data for a particular transaction must be transferred in the same order as the write addresses, and there's no need to track multiple IDs for write operations.

- WID Redundancy:

With the removal of write interleaving, the WID signal becomes unnecessary and is not included in the AXI4 protocol.

- Simplified Implementation:

The removal of WID simplifies the implementation of AXI4 masters and slaves, as there's no need for logic to handle write interleaving.

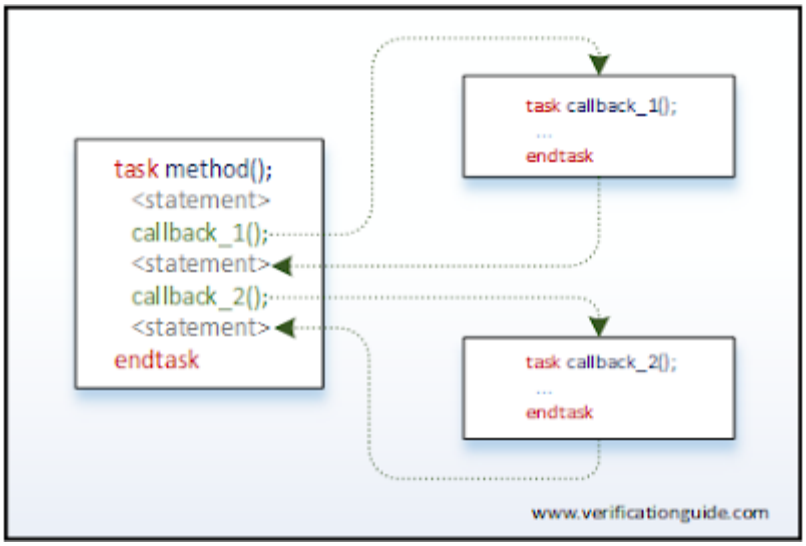
- Connection in AXI4:

If an AXI4 interface is connected to an AXI3 interface, the WID signal from the AXI3 master can be connected to the AWID signal of the AXI4 interface.

Callback function in uvm?

What is Callback?

Callbacks are empty methods with a call to them.



UVM Callback

Before Starting with UVM Callbacks, It is good to go through SystemVerilog Callback.

SYSTEM VERILOG CALLBACK:

What is Callback?

As per the English dictionary, *Callback* definition is 'a return call'

What is SystemVerilog Callback?

SystemVerilog callback specifies the *rules* to define the methods and placing method calls to achieve 'a return call to methods'.

In simple words,
Callbacks are empty methods with a call to them.

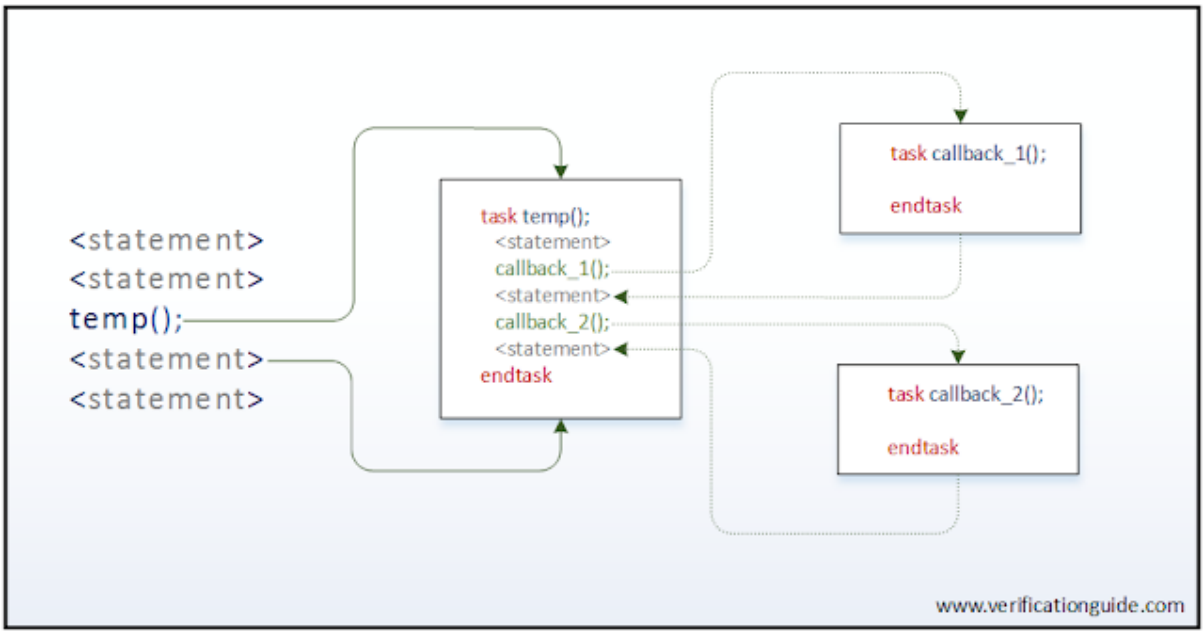
or

A method of the class is implemented with calls to dummy methods. On a need basis, the user can extend the class and implement the dummy methods.

Where,
Dummy methods are callback methods
Calls to dummy methods are callback hooks

Note:
There won't be any impact for not implementing the Callback methods.

How callback works?



SystemVerilog Callback

In the above diagram,

- temp() is the method call to invoke a method temp
- method temp has few statements (logic) also calls to callback_1 and callback_2
    - callback_1 and callback_2 are dummy methods without any logic in it.
- On calling callback_1() and callback_2() there won't be any impact to regular logic
- On a need basis, the user can extend the class and add required logic to callback_1 and callback_2 without modifying the method temp

"randomize" is one of the SystemVerilog builtin methods which has the callback features implemented in it.

randomize method is implemented with the callback features to enable the execution of pre_randomize() and post_randomize() methods before and after the randomize() method execution respectively.

This is done by placing the callback hooks to pre_randomize() and post_randomize() methods in it.

Methods will get executed in the order mentioned below,

- pre_randomize();
- randomize();
- post_randomize();

If required user can implement the pre_randomize() and post_randomize() methods.

How to implement callback?

Below is one of the ways to implement the SystemVerilog callback.

- Write a method which has calls (callback hooks) to dummy methods
- Write dummy callback methods

How to use callback?

- Extend the class and implement the callback method
- Override the existing class by extended class

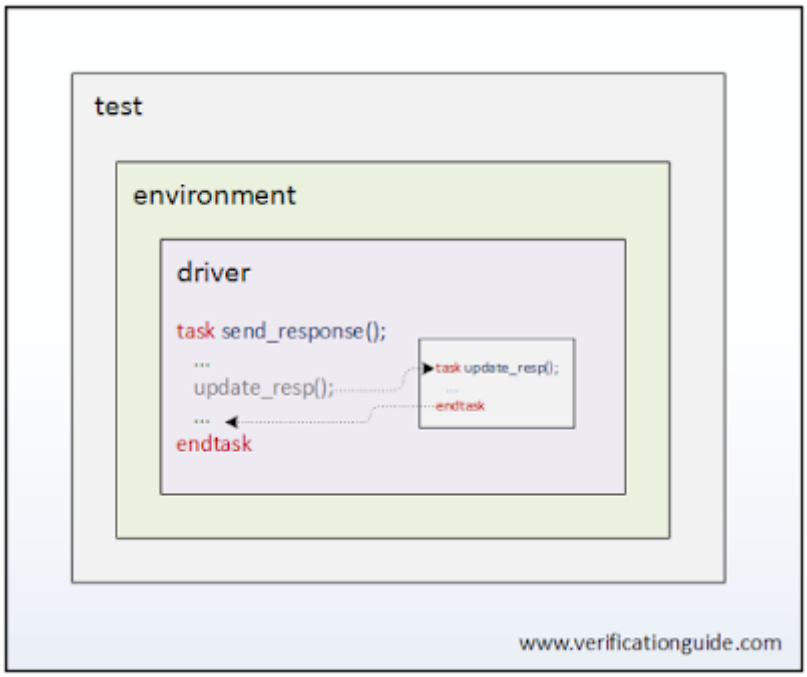
Callback example

Let's consider an example of a Slave Driver (*Slave Driver:* drives response to the master).

For simplicity, one callback hook is implemented and only response generation logic is captured in the code.

Components of slave_driver example are,

- slave_driver – Normal driver to drive response
    - response types are OKAY, EXOKAY, SLVERR, DECERR
    - slave_driver is constrained to send OKAY response always to see to the callback usage difference
- slave_env – Environment which has a slave driver created in it
- basic_test – Which sends the normal response
- error_test – Testcase with callback method used to generate an error response
- err_inject – extended driver class to implement the callback method



SystemVerilog Callback TestBench Example

By using a callback, will implement the testcase to drive SLVERR response.

Will see though the below steps,

- Adding Callback Support
- Implementing Callback Method
- Using Callback

Adding Callback Support

In this step, will write the driver with a dummy method and callback hooks to it.

- Writing slave_driver class with enum variable resp

typedef enum {OKAY, EXOKAY, SLVERR, DECERR} resp_type;

class slave_driver;

resp_type resp;

endclass

- Adding response driving task send_response with response randomize logic

typedef enum {OKAY, EXOKAY, SLVERR, DECERR} resp_type;

```
class slave_driver;
  resp_type resp;

  //send response task
  task send_response;
    std::randomize(resp) with { resp == OKAY;};
  endtask
endclass
```

- Defining the callback method by declaring the virtual task update_resp

```
typedef enum {OKAY, EXOKAY, SLVERR, DECERR} resp_type;

class slave_driver;
  resp_type resp;

  //callback hook
  virtual task update_resp; endtask

  //send response task
  task send_response;
    std::randomize(resp) with { resp == OKAY;};
  endtask
endclass
```

- Placing hook for callback. In this example, as it is required to change the response once it is generated, so it is good to place callback hook after calling the randomize method

```
typedef enum {OKAY, EXOKAY, SLVERR, DECERR} resp_type;

class slave_driver;
  resp_type resp;
```

```
  //callback hook
  virtual task update_resp; endtask


  //send response task
  task send_response;
    std::randomize(resp) with { resp == OKAY;};
    update_resp();
  endtask
endclass
```

Implementing Callback Method

For error response generation scenario, it is required to corrupt the response generated. This can be achieved by adding the logic in update_resp() callback to overwrite the response generated.

Which involves below steps.

- extend the slave_driver and write new driver err_inject

```
class err_inject extends slave_driver;


endclass
```

- Implement the callback method update_resp

```
class err_inject extends slave_driver;
  virtual task update_resp;
    $display("Injecting SLVERR");
    resp = SLVERR;
  endtask
endclass
```

Using Callback

A callback can be used by,

- Writing new error response testcase

```
program error_test;
  slave_env  env;

  initial begin
    //Create env
    env = new();
    //Calling run of env
    env.run();
  end
endprogram
```

- Declare and create an err_inject driver

```
program error_test;
  slave_env  env;
  err_inject err_driver;

  initial begin
    //Create env
    env = new();
    err_driver = new();

    //Calling run of env
    env.run();
  end
endprogram
```

- Override the slave_driver by the new err_inject driver

```
program error_test;
  slave_env  env;
  err_inject err_driver;
```

```
 initial begin
   //Create env
   env = new();
   err_driver = new();
   //Overriding slave_driver by error_driver
   env.slv_driver = err_driver;

   //Calling run of env
   env.run();
 end
endprogram
```

Simulator Output of basic_test

Slave generated response is OKAY

Slave generated response is OKAY

Simulator of error_test

Injecting SLVERR

Slave generated response is SLVERR

Injecting SLVERR

Slave generated response is SLVERR

Note:

- To execute basic_test comment error_test.sv
- To execute error_test comment basic_test.sv

Benefits of callback?

By looking into the above slave_driver examples we could say that without modifying the existing environment it is possible to achieve error injection.

- Easy to add additional functionality to existing logic
- Makes component reusable

Where callbacks can be implemented?

Callbacks can be implemented in an object or component.

What are the benefits of callback?

Different flavors of the component can be obtained by customizing the empty callback methods.

About UVM Callbacks

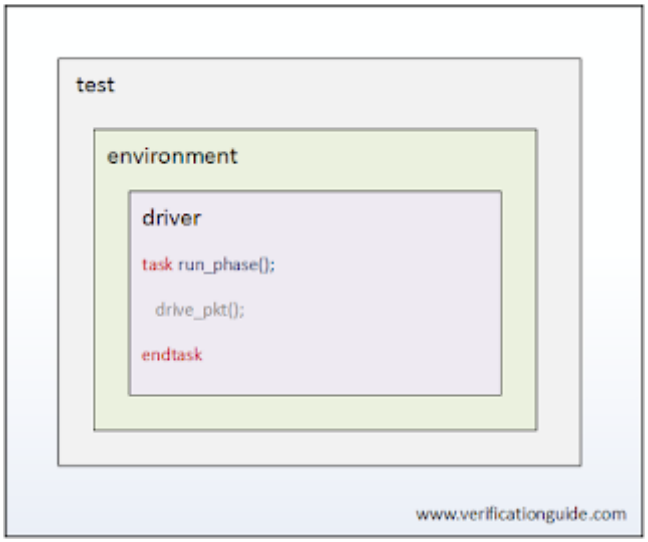
UVM provides a set of classes, methods, and macros to implement the callbacks.

Only the required callback methods are explained here.

Callback Example

With a simple driver example will see to UVM Callback.
Below are components of example,

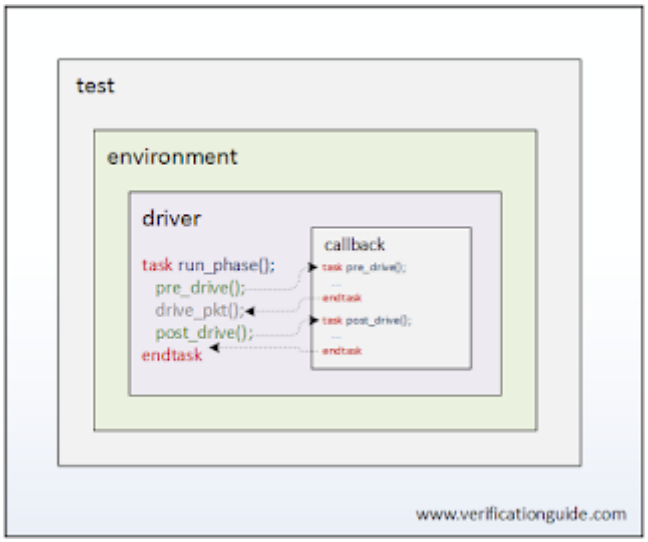
- Driver
- Environment
- Test



UVM testbench without callback

The driver has drive() task, which revives the seq_item and drives to DUT (Current example code doesn't have any logic to receive and drive seq_item).

In this example,
With help of callback, will add pre_drive() and post_drive() callbacks.



UVM testbench with callback

TestBench without callbacks

Driver

class driver extends uvm_component;

```
 `uvm_component_utils(driver)

 function new(string name, uvm_component parent);
  super.new(name,parent);
 endfunction

 task run_phase(uvm_phase phase);
    drive_pkt();
 endtask

 task drive_pkt();
  `uvm_info("DRIVER","Inside drive_pkt method",UVM_LOW);
 endtask
```

```
endclass
```

Environment

```
class environment extends uvm_env;
  driver driv;

  `uvm_component_utils(environment)

  function new(string name, uvm_component parent);
    super.new(name,parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    driv = driver::type_id::create("driv", this);
  endfunction

endclass
```

TestCase

```
class basic_test extends uvm_test;
  environment env;

  `uvm_component_utils(basic_test)

  function new(string name = "basic_test", uvm_component parent=null);
    super.new(name,parent);
  endfunction

  function void build_phase(uvm_phase phase);
```

```
    super.build_phase(phase);
    env = environment::type_id::create("env", this);
  endfunction
endclass
```

Simulator Output:

```
UVM_INFO @ 0: reporter [RNTST] Running test basic_test...
UVM_INFO driver.sv(18) @ 0: uvm_test_top.env.driv [DRIVER] Inside drive_pkt method
```

TestBench with callbacks

For easy understanding will see through below steps,

- Adding callback support
- Implementing the callback methods
- Using callback

Adding Callback Support

In order to have user-defined callbacks,

- User has to write user-defined callback class by extending the uvm_callback class

```
class driver_callback extends uvm_callback;
  ...
endclass
```

- Define callback methods in user-defined callback class
    - In this example pre_drive() and post_drive() are the callback methods.

```
  virtual task pre_drive; endtask
  virtual task post_drive; endtask
```

Complete user-defined class code

```
class driver_callback extends uvm_callback;

  `uvm_object_utils(driver_callback)

  function new(string name = "driver_callback");
```

```
  super.new(name);
 endfunction

 virtual task pre_drive; endtask
 virtual task post_drive; endtask
endclass
```

In order to use callbacks defined,

- Callback class has to be registered with the object/component where callbacks are going to be used.
- Callback class can be registered by using macro `uvm_register_cb

`uvm_register_cb(driver,driver_callback)

In order to call the callback methods,

- Need to place method call's (callback hook) to methods defined in user-defined callback class (pre_drive() and post_drive())
- Callback methods can be called by using macro `uvm_do_callbacks

`uvm_do_callbacks(driver,driver_callback,pre_drive());

`uvm_do_callbacks(driver,driver_callback,post_drive());

Complete driver code:

```
class driver extends uvm_component;

 `uvm_component_utils(driver)
 `uvm_register_cb(driver,driver_callback)

 function new(string name, uvm_component parent);
  super.new(name,parent);
 endfunction

 task run_phase(uvm_phase phase);
    `uvm_do_callbacks(driver,driver_callback,pre_drive());
```

```
    drive_pkt();


    `uvm_do_callbacks(driver,driver_callback,post_drive());
  endtask


  task drive_pkt();
    `uvm_info("DRIVER","Inside drive_pkt method",UVM_LOW);
  endtask


endclass
```

Implementing Callback Method

We have looked into, how to define the callback methods (In user-defined callback class) and placing hook to callback methods (In driver class using `uvm_do_callbacks macro)

Here will see how to add logic to callback methods.

- Need to extend the user-defined callback class and write one more callback class callback_1

```
class user_callback extends driver_callback;
  ...
endclass
```

- Implement the methods with logic in it (this example we are adding only display's)

```
  task pre_drive;
    `uvm_info("USER_CALLBACK","Inside pre_drive method",UVM_LOW);
  endtask


  task post_drive;
    `uvm_info("USER_CALLBACK","Inside post_drive method",UVM_LOW);
  endtask
```

Complete callback class code

```
class user_callback extends driver_callback;


  `uvm_object_utils(user_callback)


  function new(string name = "user_callback");
    super.new(name);
  endfunction


  task pre_drive;
    `uvm_info("USER_CALLBACK","Inside pre_drive method",UVM_LOW);
  endtask


  task post_drive;
    `uvm_info("USER_CALLBACK","Inside post_drive method",UVM_LOW);
  endtask
endclass
```

Using Callback

In the testcase where callbacks need to be applied,

- Declare and create an object of callback class in which methods are implemented (callback_1).

```
  user_callback callback_1;
  callback_1 = user_callback::type_id::create("callback_1", this);
```

- In order to execute the callback method, register the callback object to the driver using the ‘add’ method

```
  uvm_callbacks#(driver,driver_callback)::add(env.driv,callback_1);
```

Complete testcase code

```
class user_callback_test extends basic_test;
  user_callback callback_1;
```

```
 `uvm_component_utils(user_callback_test)

 function new(string name = "user_callback_test", uvm_component parent=null);
  super.new(name,parent);
 endfunction

 function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  callback_1 = user_callback::type_id::create("callback_1", this);

  uvm_callbacks#(driver,driver_callback)::add(env.driv,callback_1);
 endfunction
endclass
```

callback_test is the test with callbacks added, and basic_test is without callbacks.
Simulator Output:

```
UVM_INFO @ 0: reporter [RNTST] Running test basic_test...
UVM_INFO driver.sv(23) @ 0: uvm_test_top.env.driv [DRIVER] Inside drive_pkt method
```

callback_test simulator output (With Callback)

```
UVM_INFO @ 0: reporter [RNTST] Running test user_callback_test...
UVM_INFO user_callback.sv(14) @ 0: reporter [USER_CALLBACK] Inside pre_drive method
UVM_INFO driver.sv(23) @ 0: uvm_test_top.env.driv [DRIVER] Inside drive_pkt method
UVM_INFO user_callback.sv(18) @ 0: reporter [USER_CALLBACK] Inside post_drive method
```

With the above result, we could see that displays added in callback methods are getting printed in callback_test.

The addition of logic to the callback methods will give more control on stimulus driving.

In the next example will see an example of how driving stimulus can be controlled using callbacks.

Callback Class and Macros

uvm_callback class

user-defined callback classes can be written by extending the uvm_callback class.
In it, the user defines one or more virtual methods.

These virtual methods are referred to as callback methods, these are empty methods with users have the option of overriding them.

```
class CB extends uvm_callback;

  ...

endclass
```

uvm_register_cb

`uvm_register_cb(T,CB)

This macro is used to register the Callback (CB) with the Object(T).

Where,
CB => user defined callback class.
T => Object in which CB is used.

w.r.t above example:
CB => driver_callback class
T => driver class

uvm_do_callbacks

`uvm_do_callbacks(T,CB,METHOD)

This macro calls the callback methods defined in the user-defined callback class.

CB => user defined callback class.
T => Object in which CB is used.
METHOD => callback method to be called

w.r.t above example:

CB => driver_callback class
T => driver class
METHOD => pre_drive and post_drive

add method

uvm_callbacks#(T, CB)::add(t,cb);

Registers the callback object with the object in which callback is used.

CB => user defined callback class type.
T => Object type in which CB is used.
cb => callback class object
t => object in which callback is used

w.r.t above example:

CB => driver_callback class
T => driver class
cb => callback_1
t => driver class (differs if multiple objects of same type present)

Another Explanation:

1. What is a callback ?
2. Callback Macros
3. UVM Callback Example

What is a callback ?

The uvm_callback class serves as the base class for user-defined callback classes. Typically, a component developer creates an application-specific callback class by extending this base class. In the derived class, the developer defines one or more virtual methods, collectively known as the callback interface, which provide the hooks that users can override.

A callback is useful because it allows a flexible and modular way to modify or extend the behavior of a system without altering the original code. Callbacks decouple the code that triggers an action from the code that defines the action itself. It is required in scenarios where customization, or dynamic behavior is necessary.

Callback Macros

`uvm_register_cb macro is used to register the Callback(CB) with the Object(T) where CB is the user-defined callback class and T is the object in which CB is used.

// Macro definition

```
`define uvm_register_cb(T,CB) \
  static local bit m_register_cb_``CB = uvm_callbacks#(T,CB)::m_register_pair(`"T`",`"CB`");
```

And a specific callback method is executed by invoking `uvm_do_callbacks macro.

```
// Macro definition
`define uvm_do_callbacks(T,CB,METHOD) \
  `uvm_do_obj_callbacks(T,CB,this,METHOD)

// Basically iterates through and executes the requested method
`define uvm_do_obj_callbacks(T,CB,OBJ,METHOD) \
  begin \
    uvm_callback_iter#(T,CB) iter = new(OBJ); \
    CB cb = iter.first(); \
    while(cb != null) begin \
      `uvm_cb_trace_noobj(cb,$sformatf(`"Executing callback method 'METHOD' for callback %s (CB) from %s (T)`",cb.get_name(), OBJ.get_full_name())) \
      cb.METHOD; \
      cb = iter.next(); \
    end \
  end
```

UVM Callback Example

Let's create a callback mechanism that allows us to extend the behavior of a monitor by adding custom code to a call_pre_check() and call_post_check() methods.

1. Define the Callback Class

Users can create custom callback classes by extending the uvm_callback class. In these classes, users define one or more virtual methods. These virtual methods, known as callback methods, are initially empty and can be overridden by the user to implement specific behavior.

```
class my_monitor_cb extends uvm_callback;
  `uvm_object_utils(my_monitor_cb)

  function new(string name="my_monitor_cb");
    super.new(name);
```

```
 endfunction

 virtual function void call_pre_check();
  // Placeholder
 endfunction

 virtual function void call_post_check();
  // Placeholder
 endfunction

endclass
```

2. Define a Custom Callback

custom_monitor_cb gives a way to override the check_transaction() method with user-defined logic, such as additional checks.

```
class custom_monitor_cb extends my_monitor_cb;
 `uvm_object_utils(custom_monitor_cb)
 function new(string name="custom_monitor_cb");
  super.new(name);
 endfunction

 // Override the callback method with custom behavior
 virtual function void call_pre_check();
  `uvm_info(get_type_name(), $sformatf("[call_pre_check] start pre_check"), UVM_LOW)
 endfunction

 virtual function void call_post_check();
  `uvm_info(get_type_name(), $sformatf("[call_post_check] start post_check"), UVM_LOW)
 endfunction
```

```
endclass
```

3. Add Callback Hooks and Register the Callback

```
class my_monitor extends uvm_monitor;

 `uvm_component_utils(my_monitor)
 function new(string name="my_monitor", uvm_component parent=null);
  super.new(name, parent);
 endfunction

 // Register the callback class for this component
 `uvm_register_cb(my_monitor, my_monitor_cb)

 // Method that processes a transaction and uses the callback
 function void check_transaction();
  // Call the registered callback(s)
  `uvm_do_callbacks(my_monitor, my_monitor_cb, call_pre_check());

  // Normal checking logic
  `uvm_info("MONITOR", "Checking transaction", UVM_MEDIUM)

  // Or use a callback macro
  `uvm_do_callbacks(my_monitor, my_monitor_cb, call_post_check());
 endfunction

 // Run phase where the transaction check happens
 virtual task run_phase(uvm_phase phase);
  super.run_phase(phase);
```

```
    // Call the checking function, which triggers callbacks
    check_transaction();
  endtask
endclass
```

4. Register the Callback in the Test

Test creates an instance of my_monitor and registers the custom_monitor_cb callback.

```
class my_test extends uvm_test;
  `uvm_component_utils(my_test)
  function new(string name="my_test", uvm_component parent=null);
    super.new(name, parent);
  endfunction

  my_monitor         mon;
  custom_monitor_cb   my_cb;

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    // Create the monitor
    mon = my_monitor::type_id::create("mon", null);

    // Create and register the custom callback
    my_cb = custom_monitor_cb::type_id::create("my_cb");
    uvm_callbacks#(my_monitor)::add(mon, my_cb);

  endfunction
endclass
```

Finally create a module to run the test.

```
module tb;
  initial
    run_test("my_test");
endmodule
```

Simulation Log

```
UVM_INFO @ 0: reporter [RNTST] Running test my_test...
UVM_INFO testbench.sv(60) @ 0: reporter [custom_monitor_cb] [call_pre_check] start pre_check
UVM_INFO testbench.sv(37) @ 0: mon [MONITOR] Checking transaction
UVM_INFO testbench.sv(64) @ 0: reporter [custom_monitor_cb] [call_post_check] start post_check
UVM_INFO /xcelium23.09/tools/methodology/UVM/CDNS-1.2/sv/src/base/uvm_report_server.svh(847) @ 0: reporter [UVM/REPORT/SERVER]
--- UVM Report Summary ---
```

Note in the log above that user-defined methods for call_pre_check and call_post_check are executed instead of the placeholder.

Virtual Sequence and Virtual Sequence & Virtual Sequencer?

Virtual Sequence and Virtual Sequencers

A virtual sequence is nothing but a container that starts multiple sequences on different sequencers.

Virtual sequencer controls other sequencers and it is not attached to any driver.

Virtual Sequence and Virtual Sequencer Usage

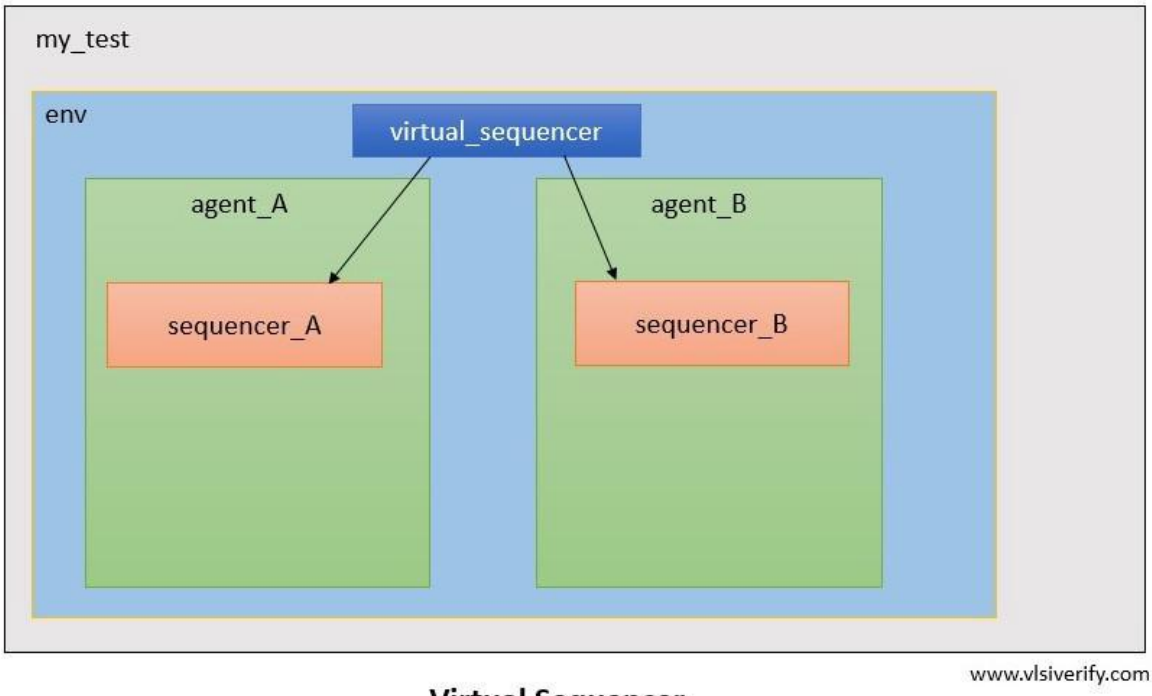
In SOC, there could be different modules that interact with different protocols So, we need different drivers to drive corresponding interfaces. So we usually keep separate agents to handle the different protocols. So, we need to execute sequences on corresponding sequencers.

Another example can be thought of as multiple cores in SOC. There can be multiple cores present in SOC that can handle different operations on input provided and respond to the

device differently. In this case, as well, different sequence execution becomes important on different sequencers.

A virtual sequence is usually executed on the virtual sequencer. A virtual sequence gives control to start different sequences.

It is recommended to use a virtual sequencer if you have multiple agents and stimulus coordination is required.



Virtual Sequencer

Why are the virtual_sequence and virtual_sequencer named virtual?

System Verilog has virtual methods, virtual interfaces, and virtual classes. “virtual” keyword is common in all of them. But, virtual_sequence and virtual_sequencer do not require any virtual keyword. UVM does not have uvm_virtual_sequence and uvm_virtual_sequencer as base classes. A virtual sequence is derived from uvm_sequence. A virtual_sequencer is derived from uvm_sequencer as a base class.

Virtual sequencer controls other sequencers. It is not attached to any driver and can not process any sequence_items too. Hence, it is named virtual.

Examples

The complete code is available on the EDA playground executable link for the below examples.

Without virtual sequence and virtual sequencer

// No Virtual Sequencer

```
class core_A_sequencer extends uvm_sequencer #(seq_item);
  `uvm_component_utils(core_A_sequencer)

  function new(string name = "core_A_sequencer", uvm_component parent = null);
    super.new(name, parent);
  endfunction

endclass

class core_B_sequencer extends uvm_sequencer #(seq_item);
  `uvm_component_utils(core_B_sequencer)

  function new(string name = "core_B_sequencer", uvm_component parent = null);
    super.new(name, parent);
  endfunction
endclass

// base_test
class base_test extends uvm_test;
  env env_o;

  core_A_seq Aseq;
  core_B_seq Bseq;

  `uvm_component_utils(base_test)

  function new(string name = "base_test", uvm_component parent = null);
    super.new(name, parent);
```

```
  endfunction


  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env_o = env::type_id::create("env_o", this);
  endfunction


  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    Aseq = core_A_seq::type_id::create("Aseq");
    Bseq = core_B_seq::type_id::create("Bseq");


    Aseq.start(env_o.agt_A.seqr_A);
    Bseq.start(env_o.agt_B.seqr_B);


    phase.drop_objection(this);
  endtask
endclass
```

Output:

```
UVM_INFO sequence.sv(10) @ 0: uvm_test_top.env_o.agt_A.seqr_A@@Aseq [core_A_seq] core_A_seq: Inside Body

UVM_INFO driver.sv(38) @ 0: uvm_test_top.env_o.agt_A.drv_A [core_A_driver] Driving from core A

UVM_INFO sequence.sv(30) @ 50: uvm_test_top.env_o.agt_B.seqr_B@@Bseq [core_B_seq] core_B_seq: Inside Body

UVM_INFO driver.sv(55) @ 50: uvm_test_top.env_o.agt_B.drv_B [core_B_driver] Driving from core B
```

With virtual sequence and without a virtual sequencer

```
// virtual sequence
class virtual_seq extends uvm_sequence #(seq_item);
  core_A_seq Aseq;
  core_B_seq Bseq;

  core_A_sequencer seqr_A;
  core_B_sequencer seqr_B;

  `uvm_object_utils(virtual_seq)

  function new (string name = "virtual_seq");
    super.new(name);
  endfunction

  task body();
    `uvm_info(get_type_name(), "virtual_seq: Inside Body", UVM_LOW);
    Aseq = core_A_seq::type_id::create("Aseq");
    Bseq = core_B_seq::type_id::create("Bseq");

    Aseq.start(seqr_A);
    Bseq.start(seqr_B);
  endtask
endclass

// No Virtual sequencer
class core_A_sequencer extends uvm_sequencer #(seq_item);
  `uvm_component_utils(core_A_sequencer)
```

```
  function new(string name = "core_A_sequencer", uvm_component parent = null);
    super.new(name, parent);
  endfunction

endclass

class core_B_sequencer extends uvm_sequencer #(seq_item);
  `uvm_component_utils(core_B_sequencer)

  function new(string name = "core_B_sequencer", uvm_component parent = null);
    super.new(name, parent);
  endfunction
endclass
```

Output:

```
UVM_INFO sequence.sv(56) @ 0: reporter@@v_seq [virtual_seq] virtual_seq: Inside Body
UVM_INFO sequence.sv(10) @ 0: uvm_test_top.env_o.agt_A.seqr_A@@Aseq [core_A_seq] core_A_seq: Inside Body
UVM_INFO driver.sv(38) @ 0: uvm_test_top.env_o.agt_A.drv_A [core_A_driver] Driving from core A
UVM_INFO sequence.sv(30) @ 50: uvm_test_top.env_o.agt_B.seqr_B@@Bseq [core_B_seq] core_B_seq: Inside Body
UVM_INFO driver.sv(55) @ 50: uvm_test_top.env_o.agt_B.drv_B [core_B_driver] Driving from core B
```

With virtual sequence and virtual sequencer using p_senquencer handle

```
// Virtual sequence
class virtual_seq extends uvm_sequence #(seq_item);
  core_A_seq Aseq;
  core_B_seq Bseq;

  core_A_sequencer seqr_A;
  core_B_sequencer seqr_B;
  `uvm_object_utils(virtual_seq)
  `uvm_declare_p_sequencer(virtual_sequencer)

  function new (string name = "virtual_seq");
    super.new(name);
  endfunction

  task body();
    `uvm_info(get_type_name(), "virtual_seq: Inside Body", UVM_LOW);
    Aseq = core_A_seq::type_id::create("Aseq");
    Bseq = core_B_seq::type_id::create("Bseq");

    Aseq.start(p_sequencer.seqr_A);
    Bseq.start(p_sequencer.seqr_B);
  endtask
endclass

// Virtual p_sequencer
class virtual_sequencer extends uvm_sequencer;
  `uvm_component_utils(virtual_sequencer)
```

```
  core_A_sequencer seqr_A;
  core_B_sequencer seqr_B;


  function new(string name = "virtual_sequencer", uvm_component parent = null);
    super.new(name, parent);
  endfunction
endclass
```

Output:

```
UVM_INFO sequence.sv(56) @ 0: uvm_test_top.env_o.v_seqr@@v_seq [virtual_seq] virtual_seq: Inside Body

UVM_INFO sequence.sv(10) @ 0: uvm_test_top.env_o.agt_A.seqr_A@@Aseq [core_A_seq] core_A_seq: Inside Body

UVM_INFO driver.sv(38) @ 0: uvm_test_top.env_o.agt_A.drv_A [core_A_driver] Driving from core A

UVM_INFO sequence.sv(30) @ 50: uvm_test_top.env_o.agt_B.seqr_B@@Bseq [core_B_seq] core_B_seq: Inside Body

UVM_INFO driver.sv(55) @ 50: uvm_test_top.env_o.agt_B.drv_B [core_B_driver] Driving from core B
```

With virtual sequence and virtual sequencer but without using p_senquencer handle

```
// virtual sequence
class virtual_seq extends uvm_sequence #(seq_item);
  core_A_seq Aseq;
  core_B_seq Bseq;

  core_A_sequencer seqr_A;
  core_B_sequencer seqr_B;
  `uvm_object_utils(virtual_seq)

  function new (string name = "virtual_seq");
    super.new(name);
  endfunction

  task body();
    env env_s;
    `uvm_info(get_type_name(), "virtual_seq: Inside Body", UVM_LOW);
    Aseq = core_A_seq::type_id::create("Aseq");
    Bseq = core_B_seq::type_id::create("Bseq");

    // virtual_sequencer is created in env, so we need env handle to find v_seqr.
    if(!$cast(env_s, uvm_top.find("uvm_test_top.env_o"))) `uvm_error(get_name(), "env_o is not found");

    Aseq.start(env_s.v_seqr.seqr_A);
    Bseq.start(env_s.v_seqr.seqr_B);
  endtask
endclass
```

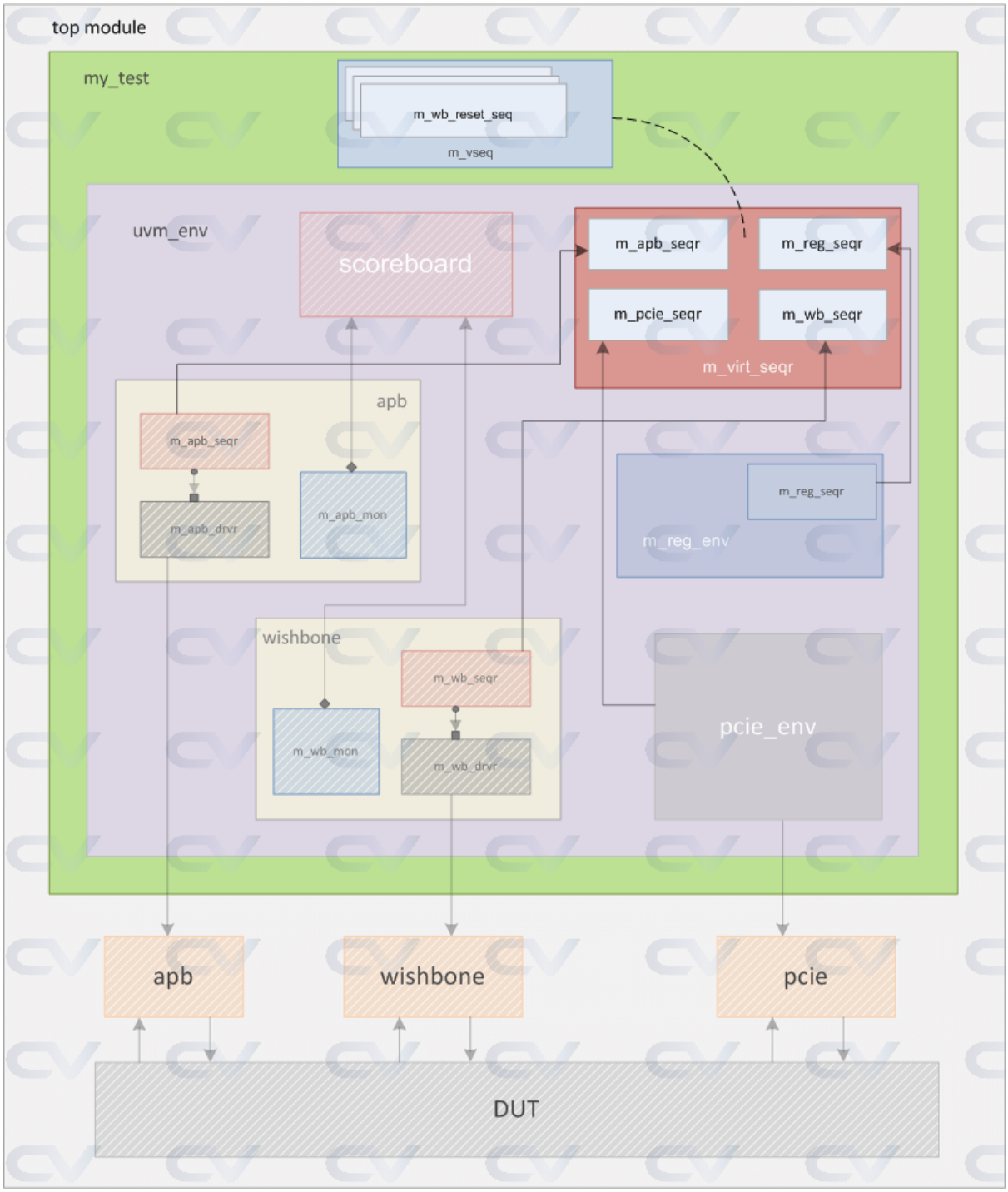
```
// virtual_sequencer
class virtual_sequencer extends uvm_sequencer;
  `uvm_component_utils(virtual_sequencer)
  core_A_sequencer seqr_A;
  core_B_sequencer seqr_B;

  function new(string name = "virtual_sequencer", uvm_component parent = null);
    super.new(name, parent);
  endfunction
endclass
```

Output:

```
UVM_INFO sequence.sv(55) @ 0: uvm_test_top.env_o.v_seqr@@v_seq [virtual_seq] virtual_seq: Inside Body

UVM_INFO sequence.sv(10) @ 0: uvm_test_top.env_o.agt_A.seqr_A@@Aseq [core_A_seq] core_A_seq: Inside Body

UVM_INFO driver.sv(38) @ 0: uvm_test_top.env_o.agt_A.drv_A [core_A_driver] Driving from core A

UVM_INFO sequence.sv(30) @ 50: uvm_test_top.env_o.agt_B.seqr_B@@Bseq [core_B_seq] core_B_seq: Inside Body

UVM_INFO driver.sv(55) @ 50: uvm_test_top.env_o.agt_B.drv_B [core_B_driver] Driving from core B
```

UVM Virtual Sequence:

A *virtual* sequence is a container to start multiple sequences on different sequencers in the environment. This virtual sequence is usually executed by a virtual *sequencer* which has handles to real sequencers. The need for a virtual sequence arises when you require different sequences to be run on different environments. For example, an SoC design might have multiple different interfaces that might need to be driven by a different set of sequences on individual sequencers. Hence the best way to start and control these different sequences would be from a **virtual** sequence. **It becomes virtual because it is not associated with any particular data type.**



Use of a virtual sequencer

class my_virtual_seq extends uvm_sequence;

`uvm_object_utils (my_virtual_seq)

`uvm_declare_p_sequencer (my_virtual_sequencer)

function new (string name = "my_virtual_seq");

```
        super.new (name);
    endfunction

    apb_rd_wr_seq       m_apb_rd_wr_seq;
    wb_reset_seq m_wb_reset_seq;
    pcie_gen_seq  m_pcie_gen_seq;

    task pre_body();
        m_apb_rd_wr_seq = apb_rd_wr_seq::type_id::create ("m_apb_rd_wr_seq");
        m_wb_reset_seq  = wb_reset_seq::type_id::create ("m_wb_reset_seq");
        m_pcie_gen_seq  = pcie_gen_seq::type_id::create ("m_pcie_gen_seq");
    endtask

    task body();
        ...
        m_apb_rd_wr_seq.start (p_sequencer.m_apb_seqr);
        fork
            m_wb_reset_seq.start (p_sequencer.m_wb_seqr);
            m_pcie_gen_seq.start (p_sequencer.m_pcie_seqr);
        join
        ...
    endtask
endclass
```

Note the following from the example above.

- **my_virtual_seq** is derived from uvm_sequence just like any other sequence
- A handle called p_sequencer is created within the sequence via macro `uvm_declare_p_sequencer and assigned to be run with **my_virtual_sequencer**
- Each sequence is started on its corresponding sequencer using the start() method

- Each sequencer is referenced by **p_sequencer** handle which points to the virtual sequencer

Once a virtual sequence is defined, you can start this in your test as shown below.

```
class my_test extends uvm_test;
    `uvm_component_utils (my_test)

    my_env      m_env;

    ...

    task run_phase (uvm_phase phase);
        my_virtual_seq m_vseq = my_virtual_seq::type_id::create ("m_vseq");
        phase.raise_objection (this);
        m_vseq.start (m_env.m_virtual_seqr);
        phase.drop_objection (this);
    endtask
endclass
```

Without a virtual sequencer

You can also put handles to individual sequencers within the virtual sequence, and then would not need a virtual sequencer.

```
class my_virtual_seq extends uvm_sequence;
    `uvm_object_utils (my_virtual_seq)
    ...
    apb_sequencer      m_apb_seqr;
    reg_sequencer      m_reg_seqr;
    wb_sequencer       m_wb_seqr;

    apb_rd_wr_seq      m_apb_rd_wr_seq;
    wb_reset_seq m_wb_reset_seq;
```

```
    pcie_gen_seq  m_pcie_gen_seq;

    virtual task pre_body();
        m_apb_rd_wr_seq = apb_rd_wr_seq::type_id::create ("m_apb_rd_wr_seq");
        ...
        // Instantiate other sequences here
    endtask

    virtual task body ();
        m_apb_rd_wr_seq.start (m_apb_seqr);
        ...
    endtask
endclass
```

The handles to sequencers within the sequence needs to be assigned prior to starting this virtual sequence.

```
class my_test extends uvm_test;
    ...

    virtual task run_phase (uvm_phase phase);
        my_virtual_seq m_vseq = my_virtual_seq::type_id::create ("m_vseq");
        phase.raise_objection (this);

        // Assign all sequencer handles
        m_vseq.m_apb_seqr = m_env.m_apb_agent.m_apb_seqr;
        ...

        m_vseq.start (null);
        phase.drop_objection (this);
```

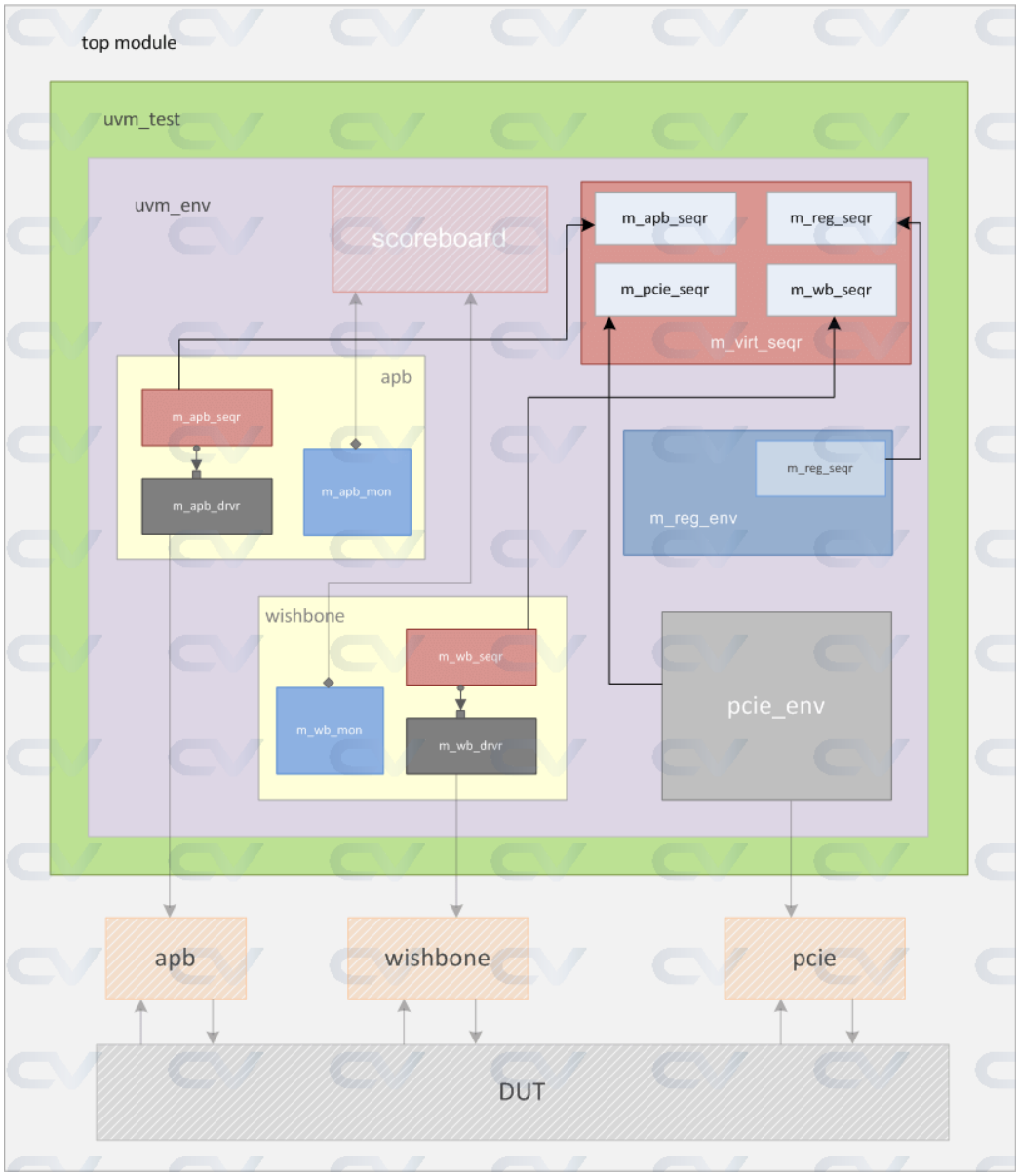
endtask

endclass

If you would like, you can also split the creation and assignment of sequence into build_phase and connect_phase within **my_test**.

UVM Virtual Sequencer:

In simple terms it's a UVM sequencer that contain handles to other sequencers. Why do we need this ? Because we plan to use virtual sequences and want to have control over all sequencers from a central place. A *request* type is not required here because this sequencer is generic and not limited to handle only one particular data type.



The environment shown above has an APB agent, Wishbone agent, PCIE environment and a register layer environment. Each of these components have their own sequences and the respective sequencers on which they are launched. A virtual sequencer called **m_virt_seqr** is instantiated to hold references to each individual sequencer. Hence a virtual sequence executing on this virtual sequencer will have access to all the sequencers in the testbench.

Example

```
class my_virtual_sequencer extends uvm_sequencer;

        `uvm_component_utils (my_virtual_sequencer)


        function new (string name = "my_virtual_sequencer", uvm_component parent);

                super.new (name, parent);

        endfunction


        // Declare handles to other sequencers here

        apb_sequencer      m_apb_seqr;

        reg_sequencer      m_reg_seqr;

        wb_sequencer       m_wb_seqr;

        pcie_sequencer     m_pcie_seqr;

endclass
```

In our top environment, we'll create an instance of the virtual sequencer and connect real sequencers with their respective handles in the virtual sequencer.

```
class top_env extends uvm_env;

        ...


        my_virtual_sequencer          m_virt_seqr;


        virtual function void build_phase (uvm_phase phase);

                ...

                m_virt_seqr = my_virtual_sequencer::type_id::create ("m_virt_seqr", this);
```

```
        ...
    endfunction

    // Connect virtual sequencer handles to actual sequencers
    virtual function void connect_phase (uvm_phase phase);
        ...
        m_virt_seqr.m_apb_seqr   = m_apb_agent.m_apb_seqr;
        m_virt_seqr.m_reg_seqr   = m_reg_env.m_reg_seqr;
        m_virt_seqr.m_pcie_seqr  = m_pcie_env.m_pcie_agent.m_pcie_seqr;
        ...
    endfunction
endclass
```

A virtual sequencer operates best when used with a *virtual sequence*. Read more on Virtual Sequence.

Simulation Log

```
----------------------------------------------------------------
CDNS-UVM-1.1d
(C) 2007-2013 Mentor Graphics Corporation
(C) 2007-2013 Cadence Design Systems, Inc.
(C) 2006-2013 Synopsys, Inc.
(C) 2011-2013 Cypress Semiconductor Corp.
----------------------------------------------------------------
UVM_INFO @ 0: reporter [RNTST] Running test base_test...
UVM_INFO @ 0: reporter [UVMTOP] UVM testbench topology:
----------------------------------------------------------------
Name                Type              Size  Value
----------------------------------------------------------------
uvm_test_top        base_test         -     @2667
```

```
m_top_env            top_env            -    @2733
  m_apb_agent          apb_agent          -    @2780
    m_apb_drv          apb_driver         -    @4089
      rsp_port         uvm_analysis_port  -    @4237
      seq_item_port    uvm_seq_item_pull_port -    @4188
    m_apb_mon          apb_monitor        -    @4217
    m_apb_seqr         uvm_sequencer      -    @3512
      rsp_export       uvm_analysis_export -    @3569
      seq_item_export  uvm_seq_item_pull_imp -    @4109
      arbitration_queue  array            0    -
      lock_queue       array              0    -
      num_last_reqs    integral           32   'd1
      num_last_rsps    integral           32   'd1
  m_spi_agent          spi_agent          -    @2840
    m_spi_drv          spi_driver         -    @4897
      rsp_port         uvm_analysis_port  -    @5043
      seq_item_port    uvm_seq_item_pull_port -    @4995
    m_spi_mon          spi_monitor        -    @5024
    m_spi_seqr         uvm_sequencer      -    @4321
      rsp_export       uvm_analysis_export -    @4377
      seq_item_export  uvm_seq_item_pull_imp -    @4917
      arbitration_queue  array            0    -
      lock_queue       array              0    -
      num_last_reqs    integral           32   'd1
      num_last_rsps    integral           32   'd1
  m_virt_seq           virtual_sequencer  -    @2870
    rsp_export         uvm_analysis_export -    @2928
    seq_item_export    uvm_seq_item_pull_imp -    @3478
```

```
    arbitration_queue    array              0    -
    lock_queue           array              0    -
    num_last_reqs        integral           32   'd1
    num_last_rsps        integral           32   'd1
  m_wb_agent             wb_agent           -    @2810
    m_wb_drv             wb_driver          -    @5712
      rsp_port           uvm_analysis_port  -    @5858
      seq_item_port      uvm_seq_item_pull_port  -    @5810
    m_wb_mon             wb_monitor         -    @5839
    m_wb_seqr            uvm_sequencer      -    @5136
      rsp_export         uvm_analysis_export  -    @5192
      seq_item_export    uvm_seq_item_pull_imp  -    @5732
      arbitration_queue  array              0    -
      lock_queue         array              0    -
      num_last_reqs      integral           32   'd1
      num_last_rsps      integral           32   'd1
----------------------------------------------------------------
```

UVM_INFO ./tb/my_pkg.sv(68) @ 0: uvm_test_top.m_top_env.m_virt_seq@@m_virt_seq [VSEQ] Start of virtual sequence

UVM_INFO ./tb/wb_agent.sv(63) @ 0: uvm_test_top.m_top_env.m_wb_agent.m_wb_seqr@@m_wb_reset_seq [RESET_SEQ] Starting wb_reset_seq

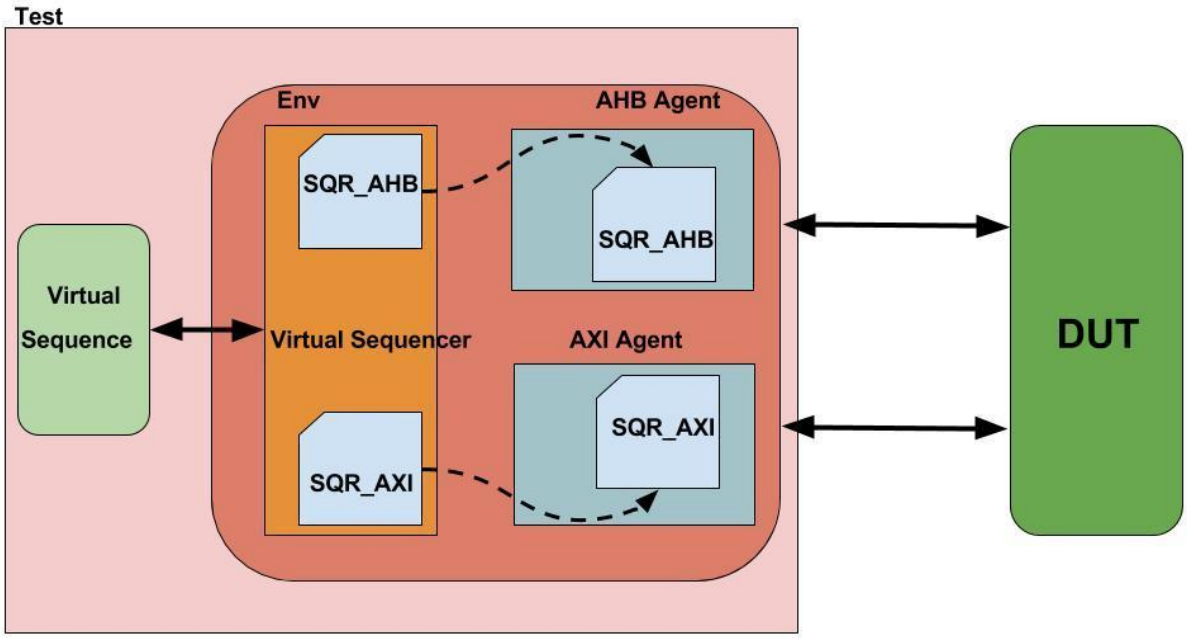
UVM_INFO ./tb/apb_agent.sv(52) @ 20000: uvm_test_top.m_top_env.m_apb_agent.m_apb_seqr@@m_apb_rw_seq [RW_SEQ] Starting apb_rw_seq

UVM_INFO ./tb/spi_agent.sv(74) @ 30000: uvm_test_top.m_top_env.m_spi_agent.m_spi_seqr@@m_spi_tx_seq [tx_SEQ] Starting spi_tx_seq

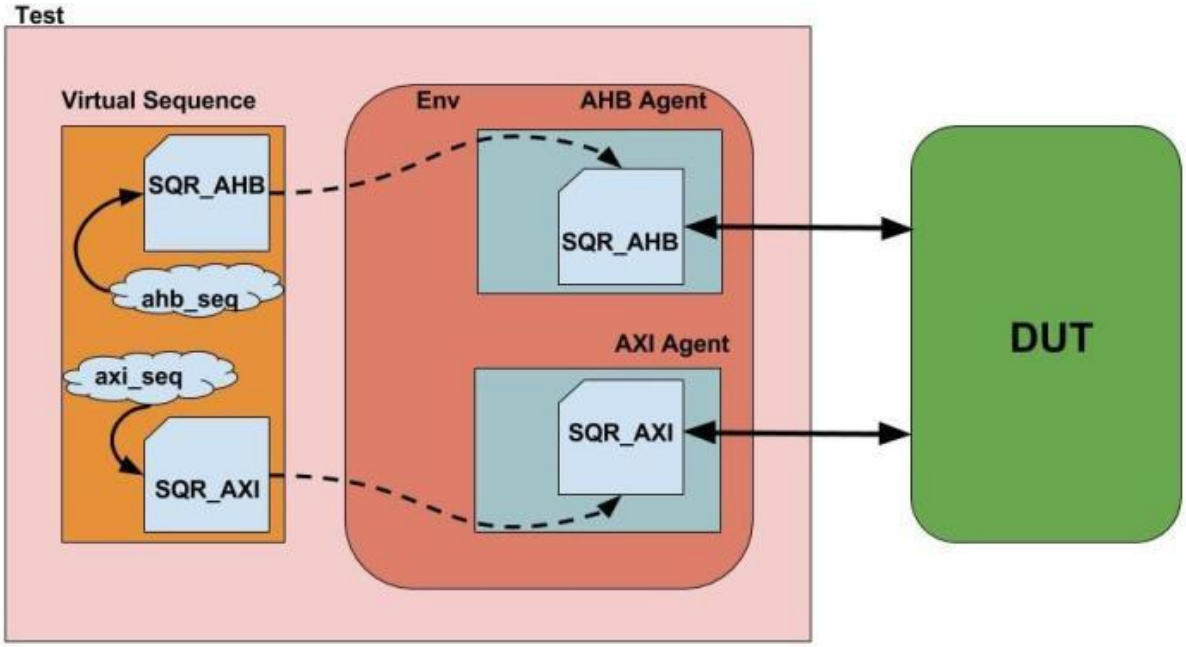
UVM_INFO ./tb/my_pkg.sv(75) @ 30000:
uvm_test_top.m_top_env.m_virt_seq@@m_virt_seq [VSEQ] End of virtual sequence

UVM_INFO ./tb/my_pkg.sv(108) @ 30000: uvm_test_top [SHUT] Shutting down test ...

--- UVM Report catcher Summary ---



Virtual Sequence and Virtual Sequencer



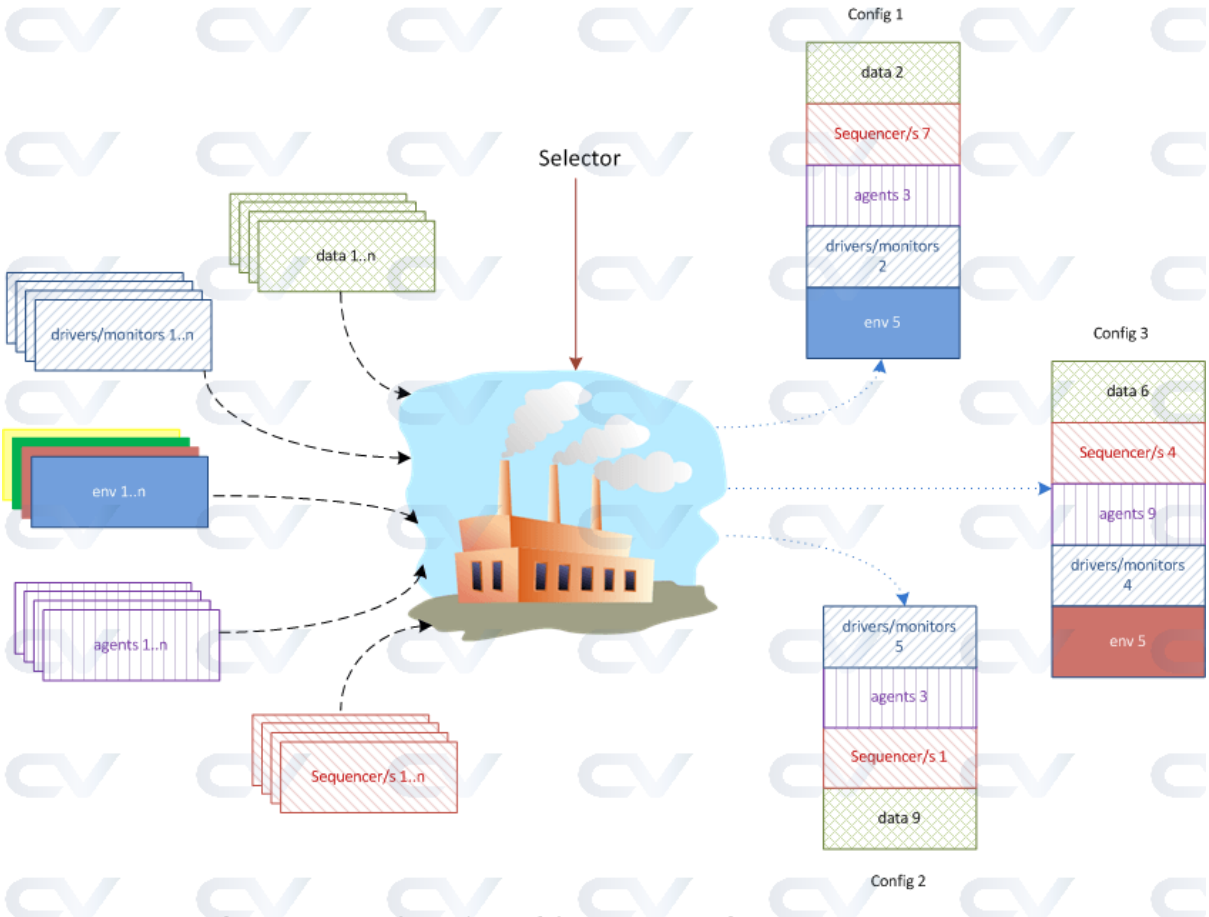
Virtual Sequence only

set_type_override and set_type_instance?

UVM FACTORY OVERRIDE:

UVM factory is a mechanism to improve flexibility and scalability of the testbench by allowing the user to substitute an existing class object by any of its inherited child class objects.

For this purpose, the *factory* needs to know all the types of classes created within the testbench by a process called as *registration*. There are UVM macros that allow classes to be registered with the factory, and methods that allow certain *types* and *instances* of class objects to be overridden by its derived *types*.



A factory can provide an object of the type we specify

Factory Override Methods

// Override all the objects of a particular type

set_type_override_by_type (uvm_object_wrapper original_type,

uvm_object_wrapper override_type,

bit replace=1);

```
set_type_override_by_name ( string original_type_name,
                            string override_type_name,
                            bit replace=1);

// Override a type within a particular instance
set_inst_override_by_type (uvm_object_wrapper original_type,
                           uvm_object_wrapper override_type,
                           string full_inst_path);

set_inst_override_by_name (string original_type_name,
                           string override_type_name,
                           string full_inst_path);
```

Method Examples

Shown below is an example testbench structure comprising of a base agent inside a base environment that is instantiated within the test class. We will try to use factory override methods to override the base class with child class.

```
// Define a base class agent
class base_agent extends uvm_agent;
  `uvm_component_utils(base_agent)
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
endclass

// Define child class that extends base agent
class child_agent extends base_agent;
  `uvm_component_utils(child_agent)
  function new(string name, uvm_component parent);
    super.new(name, parent);
```

```
  endfunction
endclass

// Environment contains the agent
class base_env extends uvm_env;
  `uvm_component_utils(base_env)
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  // 'm_agent' is a class handle to hold base_agent
  // type class objects
  base_agent m_agent;

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    // Use create method to request factory to return a base_agent
    // type of class object
    m_agent = base_agent::type_id::create("m_agent", this);

    // Now print the type of the object pointing to by the 'm_agent' class handle
    `uvm_info("AGENT", $sformatf("Factory returned agent of type=%s, path=%s",
m_agent.get_type_name(), m_agent.get_full_name()), UVM_LOW)
  endfunction
endclass
```

1. Type override by Type/Name

```
class base_test extends uvm_test;
```

```
 `uvm_component_utils(base_test)
 function new(string name, uvm_component parent);
  super.new(name, parent);
 endfunction

 base_env m_env;

 virtual function void build_phase(uvm_phase phase);
        // Get handle to the singleton factory instance
  uvm_factory factory = uvm_factory::get();

  super.build_phase(phase);

  // Set factory to override 'base_agent' by 'child_agent' by type
  set_type_override_by_type(base_agent::get_type(), child_agent::get_type());

  // Or set factory to override 'base_agent' by 'child_agent' by name
  // factory.set_type_override_by_name("base_agent", "child_agent");

  // Print factory configuration
  factory.print();

  // Now create environment
  m_env = base_env::type_id::create("m_env", this);
 endfunction
endclass
```

The following log shows that a type override is configured and replaces base_agent with child_agent . This is also evident from the info statement that prints *type* of the returned object.

Simulation Log

UVM_INFO @ 0: reporter [RNTST] Running test base_test...

UVM_INFO /playground_lib/uvm-1.2/src/base/uvm_factory.svh(1645) @ 0: reporter [UVM/FACTORY/PRINT]

Factory Configuration (*)

No instance overrides are registered with this factory

Type Overrides:

Requested Type Override Type

---------- ----------

base_agent child_agent

All types registered with the factory: 54 total

Type Name

---

base_agent

base_env

base_test

child_agent

(*) Types with no associated type name will be printed as

####

UVM_INFO testbench.sv(32) @ 0: uvm_test_top.m_env [AGENT] Factory returned agent of type=child_agent, path=uvm_test_top.m_env.m_agent

UVM_INFO /playground_lib/uvm-1.2/src/base/uvm_report_server.svh(847) @ 0: reporter [UVM/REPORT/SERVER]

--- UVM Report Summary ---

2. Instance override by Type/Name

This is most useful when only a few instances of the given type has to be overriden. A type override by type will replace all instances of the given type by the new type.

```
class base_test extends uvm_test;

  `uvm_component_utils(base_test)

  function new(string name, uvm_component parent);

    super.new(name, parent);

  endfunction


  base_env m_env;


  virtual function void build_phase(uvm_phase phase);

          // Get handle to the singleton factory instance

    uvm_factory factory = uvm_factory::get();


    super.build_phase(phase);


    // Set factory to override all instances under m_env of type 'base_agent' by 'child_agent'

    set_inst_override_by_type("m_env.*", base_agent::get_type(), child_agent::get_type());


    // Or set factory to override all instances under 'm_env' called 'base_agent' by 'child_agent' by name

    // factory.set_inst_override_by_name("base_agent", "child_agent", {get_full_name(), ".m_env.*"});


    // Print factory configuration
```

```
    factory.print();


    // Now create environment
    m_env = base_env::type_id::create("m_env", this);
  endfunction
endclass
```

Simulation Log

```
UVM_INFO @ 0: reporter [RNTST] Running test base_test...
UVM_INFO /playground_lib/uvm-1.2/src/base/uvm_factory.svh(1645) @ 0: reporter
[UVM/FACTORY/PRINT]
#### Factory Configuration (*)

Instance Overrides:

  Requested Type  Override Path        Override Type
  --------------  -------------------  -------------
  base_agent      uvm_test_top.m_env.*  child_agent

No type overrides are registered with this factory

All types registered with the factory: 54 total
  Type Name
  ---------
  base_agent
  base_env
  base_test
  child_agent
(*) Types with no associated type name will be printed as
```

####

UVM_INFO testbench.sv(32) @ 0: uvm_test_top.m_env [AGENT] Factory returned agent of type=child_agent, path=uvm_test_top.m_env.m_agent

UVM_INFO /playground_lib/uvm-1.2/src/base/uvm_report_server.svh(847) @ 0: reporter [UVM/REPORT/SERVER]

--- UVM Report Summary ---

Why is UVM factory required ?

The new function is used in SystemVerilog to create a class object and is perfectly valid to be used in UVM as well. Assume that an existing testbench uses Wishbone v1.0 protocol data packet class and is used throughout the testbench in components like driver, monitor, scoreboard, and many other sequences. **If Wishbone v2.0 is released and the testbench is required to update and start using packet definition for the new protocol, there would be many places in the testbench that would require an update in code which can prove to be cumbersome.**

```
class wb_seq extends uvm_sequence_item;
    ...

    virtual task body();
        // new() function allocates space for the new class object
        // and assigns the handle 'm_wb_pkt' to new object
        wb_pkt m_wb_pkt = new();

        // This new object may be used everywhere in the sequence
        start_item(m_wb_pkt);
        m_wb_pkt.randomize();
        ...
    endtask
```

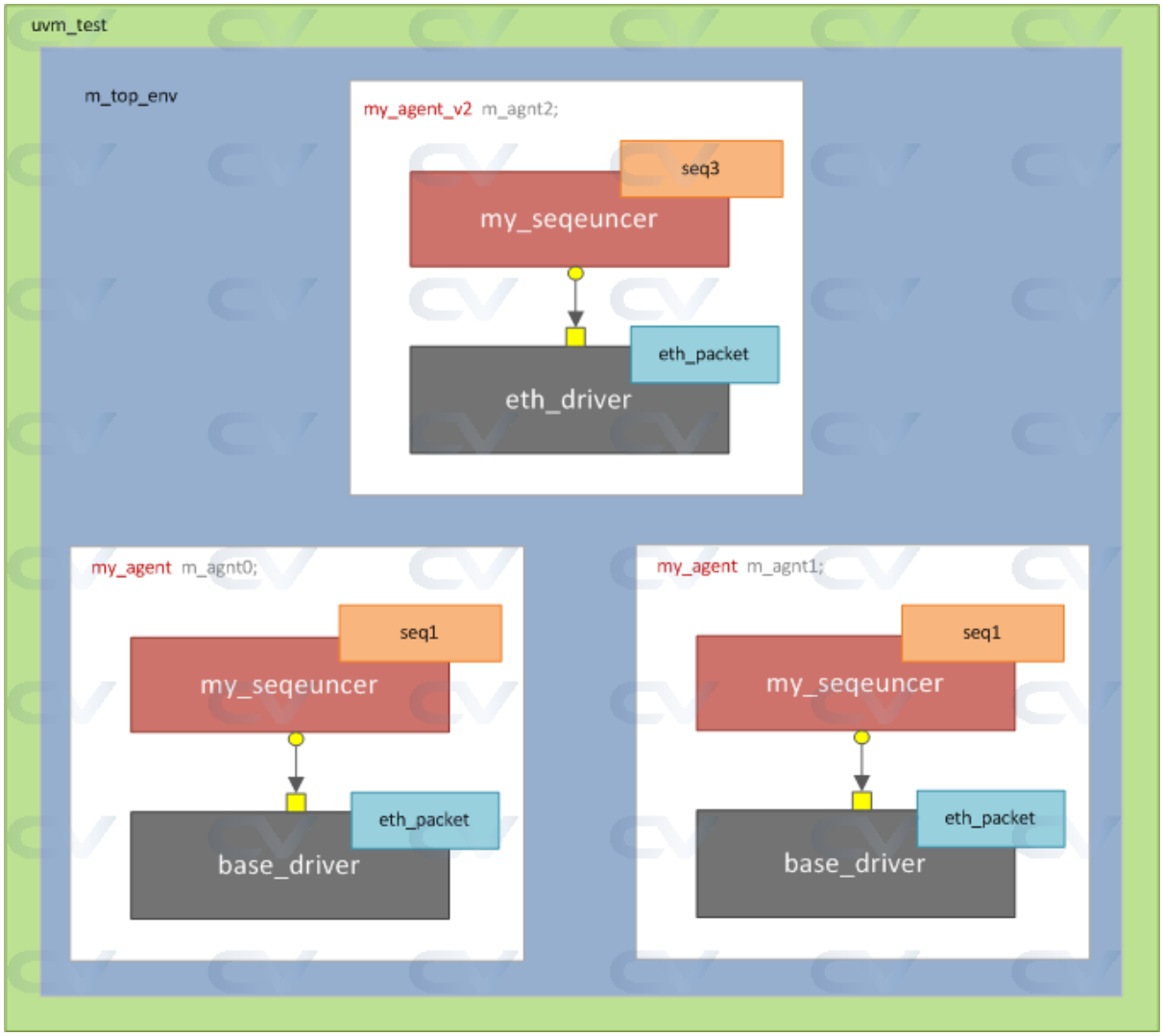
```
endclass
```

UVM has a feature called *factory* which allows users to modify or substitute *type* of the item created by the factory without having to modify existing class instantiations. Instead of text substitution of class name of existing data packet, a child class object can be created that makes necessary modifications for 2.0 and the *factory* can be used to return the newly defined class object in all places within the testbench instead of the first one. So, the preferred method of object creation in UVM testbench is by using create method.

```
class wb_seq extends uvm_sequence_item;
        ...

        virtual task body();
                // By calling create() method, the factory will create an instance of the
requested
                // type and assign the handle 'm_wb_pkt' to the new object. This override of
type
                // can be done at a higher level like the test class
                wb_pkt m_wb_pkt = wb_pkt::type_id::create("wb_pkt", this);

                // This new object may be used everywhere in the sequence
                start_item(m_wb_pkt);
                m_wb_pkt.randomize();
                ...
        endtask

endclass
```

Practical Example

To illustrate the example of a factory override, we'll create an environment as shown above with a couple of different drivers, sequences, and data object types. The image shown above is the default configuration.



Components:

- Agents
    - my_agent (Base)
        1. my_agent_v2 (child)
- Drivers

    base_driver (Base)
        1. eth_driver (child)
        2. spi_driver (child)
- Sequencer - my_seqeuncer
- Sequences
    - base_sequence (Base)

1. seq1 (child)

2. seq2 (child)

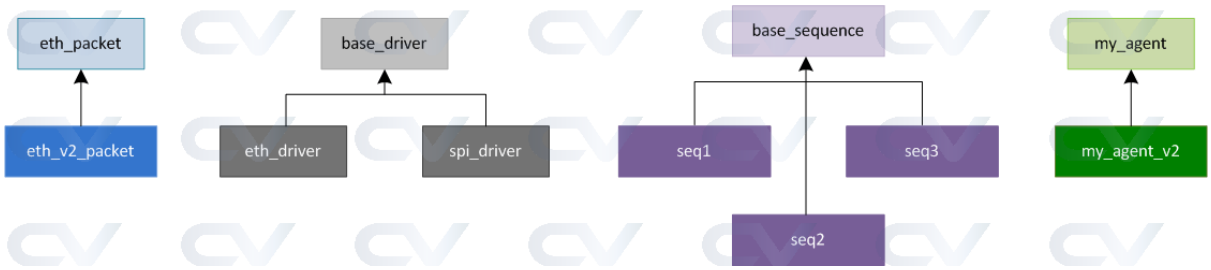
3. seq3 (child)

- Data

    - eth_packet (Basse)

        - eth_v2_packet (child)

Sequencer within m_agnt2 operate on seq3, while the other two operate on seq1 by default.



Let us first define a base data packet and derive a child class for it.

```
//------------------ eth_packet----------------------------------
class eth_packet extends uvm_sequence_item;
  `uvm_object_utils (eth_packet)

        ...

  function new (string name = "eth_packet");
    super.new (name);
    `uvm_info (get_type_name(), "Packet created", UVM_MEDIUM)
  endfunction
endclass

//------------------ eth_v2_packet----------------------------------
class eth_v2_packet extends eth_packet;
```

```
  `uvm_object_utils (eth_v2_packet)

        ...

  function new (string name = "eth_v2_packet");
    super.new (name);
  endfunction
endclass
```

Now let us create a base driver class and extend two more driver classes **eth_driver** and **spi_driver**. To show the effect of a factory override, we'll keep these classes to have minimum data and methods.

```
//----------------- base_driver---------------------------------

class base_driver #(type T=eth_packet) extends uvm_driver;
  `uvm_component_utils (base_driver #(T))

  T pkt;

  function new (string name, uvm_component parent);
    super.new (name, parent);
  endfunction

  virtual function void build_phase (uvm_phase phase);
    super.build_phase (phase);
    pkt = T::type_id::create ("pkt0");
  endfunction

  virtual task run_phase (uvm_phase phase);
    super.run_phase (phase);
```

```
    `uvm_info (get_type_name(), $sformatf("Driver running ...with packet of type : %s",
pkt.get_type_name()), UVM_MEDIUM)
  endtask

endclass

//----------------- eth_driver----------------------------------

class eth_driver #(type T=eth_packet) extends base_driver #(T);
  `uvm_component_utils (eth_driver #(T))

  function new (string name, uvm_component parent);
    super.new (name, parent);
  endfunction
endclass

//----------------- spi_driver----------------------------------

class spi_driver #(type T=eth_packet) extends base_driver #(T);
  `uvm_component_utils (spi_driver #(T))

  function new (string name, uvm_component parent);
    super.new (name, parent);
  endfunction
endclass
```

Coming to the sequences which will be executed by our sequencer

```
//----------------- base_sequence----------------------------------
class base_sequence extends uvm_sequence;
```

```
        `uvm_object_utils (base_sequence)
endclass

//----------------- seq1 ----------------------------------------
class seq1 extends base_eth_sequence;
  `uvm_object_utils (seq1)
        ...
endclass

//----------------- seq2 ----------------------------------------
class seq2 extends base_eth_sequence;
  `uvm_object_utils (seq2)
  ...
endclass

//----------------- seq3 ----------------------------------------
class seq3 extends base_eth_sequence;
  `uvm_object_utils (seq3)
        ...
endclass
```

We'll create two agents as described below.

```
//----------------- my_agent ----------------------------------------
class my_agent extends uvm_agent;
  `uvm_component_utils (my_agent)

  base_driver    m_drv0;
  my_sequencer   m_seqr0;
```

```
  function new (string name, uvm_component parent);
    super.new (name, parent);
  endfunction

  virtual function void build_phase (uvm_phase phase);
    super.build_phase (phase);
    m_drv0 = base_driver::type_id::create ("m_drv0", this);
    m_seqr0 = my_sequencer::type_id::create ("m_seqr0", this);
  endfunction

  virtual function void connect_phase (uvm_phase phase);
    super.connect_phase (phase);
    m_drv0.seq_item_port.connect (m_seqr0.seq_item_export);
  endfunction
endclass

//----------------- my_agent_v2 ------------------------------------------
class my_agent_v2 extends uvm_agent;
  `uvm_component_utils (my_agent_v2)

  eth_driver     m_drv0;
  my_sequencer   m_seqr0;

  function new (string name, uvm_component parent);
    super.new (name, parent);
  endfunction

  virtual function void build_phase (uvm_phase phase);
```

```
    super.build_phase (phase);
    m_drv0 = eth_driver::type_id::create ("m_drv0", this);
    m_seqr0 = my_sequencer::type_id::create ("m_seqr0", this);
  endfunction

  virtual function void connect_phase (uvm_phase phase);
    super.connect_phase (phase);
    m_drv0.seq_item_port.connect (m_seqr0.seq_item_export);
  endfunction
endclass
```

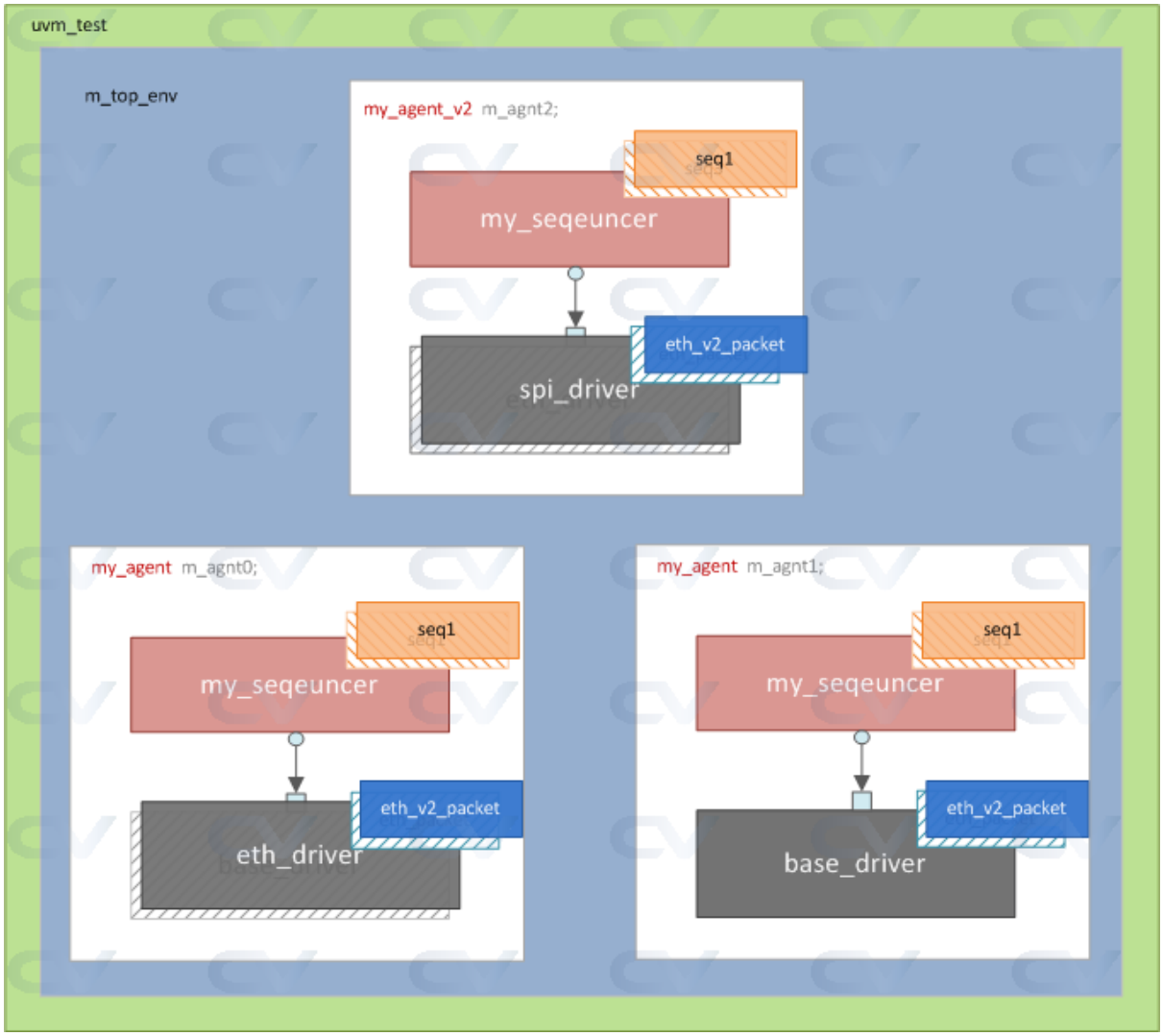
Note that **my_agent** has **base_driver**, while **my_agent_v2** contains **eth_driver**. All drivers and sequencers are instantiated within the build_phase() and connected in connect_phase().

The top container "my_env" is still left.

```
class my_env extends uvm_env ;
  `uvm_component_utils (my_env)

  my_agent      m_agnt0;
  my_agent      m_agnt1;
  my_agent_v2   m_agnt2;

  function new (string name, uvm_component parent);
    super.new (name, parent);
  endfunction : new

  virtual function void build_phase (uvm_phase phase);
    super.build_phase (phase);
    m_agnt0 = my_agent::type_id::create ("m_agnt0", this);
    m_agnt1 = my_agent::type_id::create ("m_agnt1", this);
```

```
    m_agnt2 = my_agent_v2::type_id::create ("m_agnt2", this);
  endfunction : build_phase
endclass : my_env
```

We have put our components into boxes, and arranged everything. It's time to write the test, and this is where factory override helps. We'll be able to substitute drivers, data packets and sequences by calling a few factory methods.




```
class feature_test extends base_test;
    `uvm_component_utils (feature_test)

    function new (string name, uvm_component parent = null);
      super.new (name, parent);
    endfunction
```

```
    virtual function void build_phase (uvm_phase phase);
      super.build_phase (phase);

`ifdef PKT_OVERRIDE
      // Substitute all eth_packets with eth_v2_packet
      set_type_override_by_type (eth_packet::get_type(), eth_v2_packet::get_type());
`endif

// These are the three different styles to override something

`ifdef DRV_STYLE1
      // Substitute all instances of base_driver with driver2
      set_type_override_by_type (base_driver::get_type(), spi_driver::get_type());
`elsif DRV_STYLE2
      // Substitute only eth_driver in agnt2 with spi_driver - by calling the component to be replaced method
      eth_driver::type_id::set_inst_override (spi_driver::get_type(), "m_top_env.m_agnt2.m_drv0", this);
`elsif DRV_STYLE3
      // Substitute base_driver only in agnt0 - by calling the factory method
      factory.set_inst_override_by_type (base_driver::get_type(), eth_driver::get_type(), {get_full_name(), ".m_top_env.m_agnt0.*"});
`endif

// Trying to override a sequence

`ifdef SEQ_TYPE
      // Substitute seq1 with seq2
```

set_type_override_by_type (seq1::get_type(), seq3::get_type());

`elsif SEQ_INST

// Substitute seq1 with seq2 only for agnt1

set_inst_override_by_type ("m_top_env.m_agnt1.m_seqr0.*", seq1::get_type(), seq2::get_type());

`else

`endif

factory.print();

endfunction

// Enter test code for feature here

endclass

These results are for running the test with
defines PKT_OVERRIDE, DRV_STYLE3 and SEQ_INST. Also, note that the factory configuration is shown in the log.

Simulation Log

CDNS-UVM-1.1d (14.10-s013)

(C) 2007-2013 Mentor Graphics Corporation

(C) 2007-2013 Cadence Design Systems, Inc.

(C) 2006-2013 Synopsys, Inc.

(C) 2011-2013 Cypress Semiconductor Corp.

UVM_INFO @ 0: reporter [RNTST] Running test feature_test...

Factory Configuration (*)

Instance Overrides:

Requested Type	Override Path	Override Type

```
---------------  ---------------------------------------  ---------------
base_driver #(T)  uvm_test_top.m_top_env.m_agnt0.*         eth_driver #(T)
seq1              uvm_test_top.m_top_env.m_agnt1.m_seqr0.*  seq2
```

Type Overrides:

```
Requested Type    Override Type
---------------  ---------------------------------------
eth_packet       eth_v2_packet
```

All types registered with the factory: 51 total

(types without type names will not be printed)

```
Type Name
---------
base_driver #(T)
base_eth_sequence
base_test
eth_driver #(T)
eth_packet
eth_v2_packet
feature_test
my_agent
my_agent_v2
my_env
my_sequencer
reg_test
seq1
```

seq2

seq3

(*) Types with no associated type name will be printed as

####

UVM_INFO ./tb/my_pkg.sv(45) @ 0: reporter@@eth_v2_packet [eth_v2_packet] Packet created

UVM_INFO ./tb/my_pkg.sv(45) @ 0: reporter@@eth_v2_packet [eth_v2_packet] Packet created

UVM_INFO ./tb/my_pkg.sv(45) @ 0: reporter@@eth_v2_packet [eth_v2_packet] Packet created

UVM_INFO @ 0: reporter [UVMTOP] UVM testbench topology:

Name Type Size Value

---

uvm_test_top feature_test - @2657

m_top_env my_env - @218

m_agnt0 my_agent - @2772

m_drv0 eth_driver #(T) - @2770

rsp_port uvm_analysis_port - @2963

seq_item_port uvm_seq_item_pull_port - @2912

m_seqr0 my_sequencer - @2996

rsp_export uvm_analysis_export - @3054

seq_item_export uvm_seq_item_pull_imp - @3602

arbitration_queue array 0 -

lock_queue array 0 -

num_last_reqs integral 32 'd1

num_last_rsps integral 32 'd1

```
  m_agnt1            my_agent            -    @2751
    m_drv0           base_driver #(T)    -    @3676
      rsp_port       uvm_analysis_port   -    @3773
      seq_item_port  uvm_seq_item_pull_port  -    @3725
    m_seqr0          my_sequencer        -    @3754
      rsp_export     uvm_analysis_export -    @3859
      seq_item_export  uvm_seq_item_pull_imp  -    @4399
      arbitration_queue  array            0    -
      lock_queue     array               0    -
      num_last_reqs  integral            32   'd1
      num_last_rsps  integral            32   'd1
  m_agnt2            my_agent_v2         -    @2802
    m_drv0           eth_driver #(T)     -    @4455
      rsp_port       uvm_analysis_port   -    @4553
      seq_item_port  uvm_seq_item_pull_port  -    @4505
    m_seqr0          my_sequencer        -    @3803
      rsp_export     uvm_analysis_export -    @4639
      seq_item_export  uvm_seq_item_pull_imp  -    @5179
      arbitration_queue  array            0    -
      lock_queue     array               0    -
      num_last_reqs  integral            32   'd1
      num_last_rsps  integral            32   'd1
----------------------------------------------------------------
```

UVM_INFO ./tb/my_pkg.sv(94) @ 0: uvm_test_top.m_top_env.m_agnt2.m_drv0 [eth_driver #(T)] Driver running ...with packet of type : eth_v2_packet

UVM_INFO ./tb/my_pkg.sv(94) @ 0: uvm_test_top.m_top_env.m_agnt1.m_drv0 [base_driver #(T)] Driver running ...with packet of type : eth_v2_packet

UVM_INFO ./tb/my_pkg.sv(94) @ 0: uvm_test_top.m_top_env.m_agnt0.m_drv0 [eth_driver #(T)] Driver running ...with packet of type : eth_v2_packet

UVM_INFO ./tb/my_pkg.sv(278) @ 0: uvm_test_top.m_top_env.m_agnt2.m_seqr0@@seq3 [seq3] Executing pre_body

UVM_INFO ./tb/my_pkg.sv(250) @ 0: uvm_test_top.m_top_env.m_agnt1.m_seqr0@@seq1 [seq2] Executing pre_body

UVM_INFO ./tb/my_pkg.sv(282) @ 0: uvm_test_top.m_top_env.m_agnt2.m_seqr0@@seq3 [SEQ3] Starting seq3

UVM_INFO ./tb/my_pkg.sv(255) @ 0: uvm_test_top.m_top_env.m_agnt1.m_seqr0@@seq1 [SEQ2] Starting seq2

UVM_INFO ./tb/my_pkg.sv(232) @ 0: uvm_test_top.m_top_env.m_agnt0.m_seqr0@@seq1 [SEQ1] Starting seq1

UVM_INFO ./tb/my_pkg.sv(284) @ 10000: uvm_test_top.m_top_env.m_agnt2.m_seqr0@@seq3 [SEQ3] Ending seq3

UVM_INFO ./tb/my_pkg.sv(257) @ 10000: uvm_test_top.m_top_env.m_agnt1.m_seqr0@@seq1 [SEQ2] Ending seq2

UVM_INFO ./tb/my_pkg.sv(234) @ 10000: uvm_test_top.m_top_env.m_agnt0.m_seqr0@@seq1 [SEQ1] Ending seq1

UVM_INFO ./tb/my_pkg.sv(288) @ 10000: uvm_test_top.m_top_env.m_agnt2.m_seqr0@@seq3 [seq3] Executing post_body

UVM_INFO ./tb/my_pkg.sv(262) @ 10000: uvm_test_top.m_top_env.m_agnt1.m_seqr0@@seq1 [seq2] Executing post_body

UVM_INFO ./tb/my_pkg.sv(292) @ 10000: uvm_test_top.m_top_env.m_agnt2.m_seqr0@@seq3 [seq3] Executing post_start

--- UVM Report catcher Summary ---

Instance Override in UVM factory

Unlike type override does override all instances of the type, instance override does override only specified positions in the uvm component hierarchy.

Methods of instance override in UVM factory

1. set_inst_override_by_type

2. set_inst_override_by_name

Syntax for set_inst_override_by_type

```
function void set_inst_override_by_type (uvm_object_wrapper original_type,
                                         uvm_object_wrapper override_type,
                                         string full_inst_path)
```

Syntax for set_inst_override_by_name

```
function void set_inst_override_by_name (string original_type_name,
                                         string override_type_name,
                                         string full_inst_path)
```

Example with set_inst_override_by_type

```
class component_A extends uvm_component;
  `uvm_component_utils(component_A)

  function new(string name = "component_A", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  virtual function display();
    `uvm_info(get_type_name(), $sformatf("inside component_A"), UVM_LOW);
  endfunction
endclass

class component_B extends component_A;
  `uvm_component_utils(component_B)

  function new(string name = "component_B", uvm_component parent = null);
    super.new(name, parent);
  endfunction
```

```
  function display();
    `uvm_info(get_type_name(), "inside component_B", UVM_LOW);
  endfunction
endclass

class my_test extends uvm_test;
  `uvm_component_utils(my_test)
  component_A comp_A;

  function new(string name = "my_test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    uvm_factory factory = uvm_factory::get();
    super.build_phase(phase);

    set_inst_override_by_type("*", component_A::get_type(), component_B::get_type());
    comp_A = component_A::type_id::create("comp_A", this);

    factory.print();
  endfunction

  function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    uvm_top.print_topology();
  endfunction
```

```
  task run_phase(uvm_phase phase);
    super.run_phase(phase);
    comp_A.display();
  endtask
endclass

module tb_top;
  initial begin
    run_test("my_test");
  end
endmodule
```

Output:

```
#### Factory Configuration (*)

Instance Overrides:

  Requested Type  Override Path   Override Type
  --------------  --------------  -------------
  component_A     uvm_test_top.*  component_B

No type overrides are registered with this factory

All types registered with the factory: 55 total
  Type Name
  ---------
  component_A
  component_B
```

```
  my_test
  snps_uvm_reg_bank_group
  snps_uvm_reg_map
(*) Types with no associated type name will be printed as <unknown>

####

UVM_INFO /apps/vcsmx/vcs/Q-2020.03-SP1-1//etc/uvm-1.2/src/base/uvm_root.svh(675) @
0: reporter [UVMTOP] UVM testbench topology:
----------------------------------------
Name         Type        Size  Value
----------------------------------------
uvm_test_top  my_test     -    @336
  comp_A      component_B  -    @349
----------------------------------------

UVM_INFO testbench.sv(26) @ 0: uvm_test_top.comp_A [component_B] inside
component_B
```

Example with set_inst_override_by_name

```
class component_A extends uvm_component;
  `uvm_component_utils(component_A)

  function new(string name = "component_A", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  virtual function display();
    `uvm_info(get_type_name(), $sformatf("inside component_A"), UVM_LOW);
  endfunction
```

```
endclass

class component_B extends component_A;
  `uvm_component_utils(component_B)

  function new(string name = "component_B", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function display();
    `uvm_info(get_type_name(), "inside component_B", UVM_LOW);
  endfunction
endclass

class my_test extends uvm_test;
  `uvm_component_utils(my_test)
  component_A comp_A;

  function new(string name = "my_test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    uvm_factory factory = uvm_factory::get();
    super.build_phase(phase);

    factory.set_inst_override_by_name("component_A", "component_B", "*");
    comp_A = component_A::type_id::create("comp_A", this);
```

```
  factory.print();
 endfunction

 function void end_of_elaboration_phase(uvm_phase phase);
  super.end_of_elaboration_phase(phase);
  uvm_top.print_topology();
 endfunction

 task run_phase(uvm_phase phase);
  super.run_phase(phase);
  comp_A.display();
 endtask
endclass

module tb_top;
 initial begin
  run_test("my_test");
 end
endmodule
```

Output:

```
#### Factory Configuration (*)

Instance Overrides:

 Requested Type  Override Path  Override Type
 --------------  -------------  -------------
 component_A     *              component_B
```

No type overrides are registered with this factory

All types registered with the factory: 53 total

Type Name

---

component_A

component_B

my_test

(*) Types with no associated type name will be printed as <unknown>

####

UVM_INFO /xcelium20.09/tools//methodology/UVM/CDNS-1.2/sv/src/base/uvm_root.svh(605) @ 0: reporter [UVMTOP] UVM testbench topology:

Name Type Size Value

---

uvm_test_top my_test - @1807

comp_A component_B - @1883

UVM_INFO testbench.sv(26) @ 0: uvm_test_top.comp_A [component_B] inside component_B

Copy

There are also similar two methods used that behave similarly as mentioned above.

1. set_type_override
2. set_inst_override

set_type_override and set_inst_override Methods

Syntax for set_type_override

```
<original_type>::type_id::set_type_override(<substitute_type>::get_type(),
                                replace);
```

Syntax for set_inst_override

```
<original_type>::type_id::set_inst_override(<substitute_type>::get_type(),
                                <path_string>);
```

Example with set_type_override

```
class component_A extends uvm_component;
  `uvm_component_utils(component_A)

  function new(string name = "component_A", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  virtual function display();
    `uvm_info(get_type_name(), $sformatf("inside component_A"), UVM_LOW);
  endfunction
endclass

class component_B extends component_A;
  `uvm_component_utils(component_B)

  function new(string name = "component_B", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function display();
    `uvm_info(get_type_name(), "inside component_B", UVM_LOW);
```

```
  endfunction
endclass

class my_test extends uvm_test;
  `uvm_component_utils(my_test)
  component_A comp_A;

  function new(string name = "my_test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    uvm_factory factory = uvm_factory::get();
    super.build_phase(phase);

    component_A::type_id::set_type_override(component_B::get_type());
    comp_A = component_A::type_id::create("comp_A", this);
    factory.print();
  endfunction

  function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    uvm_top.print_topology();
  endfunction

  task run_phase(uvm_phase phase);
    super.run_phase(phase);
    comp_A.display();
```

```
  endtask
endclass

module tb_top;
  initial begin
    run_test("my_test");
  end
endmodule
```

Output:

```
#### Factory Configuration (*)

No instance overrides are registered with this factory

Type Overrides:

  Requested Type  Override Type
  --------------  -------------
  component_A     component_B

All types registered with the factory: 53 total
  Type Name
  ---------
  component_A
  component_B
  my_test
(*) Types with no associated type name will be printed as <unknown>

####
```

```
UVM_INFO /xcelium20.09/tools//methodology/UVM/CDNS-
1.2/sv/src/base/uvm_root.svh(605) @ 0: reporter [UVMTOP] UVM testbench topology:
----------------------------------
Name        Type        Size  Value
----------------------------------
uvm_test_top  my_test     -    @1807
  comp_A     component_B  -    @1876
----------------------------------

UVM_INFO testbench.sv(26) @ 0: uvm_test_top.comp_A [component_B] inside
component_B
```

Example with set_inst_override

```
class component_A extends uvm_component;
  `uvm_component_utils(component_A)

  function new(string name = "component_A", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  virtual function display();
    `uvm_info(get_type_name(), $sformatf("inside component_A"), UVM_LOW);
  endfunction
endclass

class component_B extends component_A;
  `uvm_component_utils(component_B)
```

```
  function new(string name = "component_B", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function display();
    `uvm_info(get_type_name(), "inside component_B", UVM_LOW);
  endfunction
endclass

class my_test extends uvm_test;
  `uvm_component_utils(my_test)
  component_A comp_A;

  function new(string name = "my_test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    uvm_factory factory = uvm_factory::get();
    super.build_phase(phase);

    component_A::type_id::set_inst_override(component_B::get_type(), "*");
    comp_A = component_A::type_id::create("comp_A", this);
    factory.print();
  endfunction

  function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
```

```
    uvm_top.print_topology();
  endfunction

  task run_phase(uvm_phase phase);
    super.run_phase(phase);
    comp_A.display();
  endtask
endclass

module tb_top;
  initial begin
    run_test("my_test");
  end
endmodule
```

Output:

```
#### Factory Configuration (*)

Instance Overrides:

  Requested Type  Override Path  Override Type
  --------------  -------------  -------------
  component_A     *              component_B

No type overrides are registered with this factory

All types registered with the factory: 53 total
  Type Name
  ---------
```

component_A

component_B

my_test

(*) Types with no associated type name will be printed as <unknown>

####

UVM_INFO /xcelium20.09/tools//methodology/UVM/CDNS-1.2/sv/src/base/uvm_root.svh(605) @ 0: reporter [UVMTOP] UVM testbench topology:

Name Type Size Value

---

uvm_test_top my_test - @1807

comp_A component_B - @1881

UVM_INFO testbench.sv(26) @ 0: uvm_test_top.comp_A [component_B] inside component_B

Why create, why not new?

In UVM, create() is preferred over new() for object instantiation because it leverages the UVM factory for object creation and overrides, allowing for runtime flexibility. new() is a SystemVerilog constructor, while create() is a UVM factory method that can potentially override the object type based on registered overrides.

Here's a more detailed explanation:

- new():

This is the standard SystemVerilog constructor, used to create instances of classes. It always creates an object of the specified class type.

- create():

This UVM factory method constructs an object, but it can be overridden at runtime. If type or instance overrides are registered before the object is created, the factory will return an object of the overridden type.

Key Advantages of using create():

- **Runtime Overrides:**

You can change the type of object being created without modifying the source code.

- **Modular Design:**

The factory decouples object creation from the code that uses the objects, making it easier to maintain and extend your testbench.

- **Flexibility:**

Allows for dynamic substitution of objects, enabling more sophisticated testbench setups.

When to use new():

- **Direct Instantiation:** When you need to create an object of a specific type and don't expect any overrides.
- **SystemVerilog Classes:** When using SystemVerilog classes that are not derived from uvm_object.

Example:

Code

```
// Create a base class object using 'create()'
uvm_object base_obj = uvm_object::type_id::create("base_obj", this);

// Create a derived class object using 'new()'
derived_class derived_obj = new("derived_obj", this);
```

In this example, create() allows for overrides based on registered type overrides, while new() creates a specific instance of the derived_class.

How associative arrays are iterated or accessed? using first, last, prev, next linked list type of accesses?

System Verilog Associative Array:

When size of a collection is unknown or the data space is sparse, an associative array is a better option. Associative arrays do not have any storage allocated until it is used, and the index expression is not restricted to integral expressions, but can be of any type.

An associative array implements a look-up table of the elements of its declared type. The data type to be used as an index serves as the lookup key and imposes an ordering.

Syntax

```
// Value     Array_Name         [ key ];
data_type    array_identifier    [ index_type ];
```

Initialization Example

```
module tb;

        int       array1 [int];                    // An integer array with integer index
        int       array2 [string];                 // An integer array with string index
        string  array3 [string];           // A string array with string index

        initial begin
        // Initialize each dynamic array with some values
        array1 = '{ 1 : 22,
                6 : 34 };

                array2 = '{ "Ross" : 100,
                "Joey" : 60 };

                array3 = '{ "Apples" : "Oranges",
                "Pears" : "44" };

        // Print each array
    $display ("array1 = %p", array1);
    $display ("array2 = %p", array2);
    $display ("array3 = %p", array3);
   end
```

endmodule

Simulation Log

ncsim> run

array1 = '{1:22, 6:34}

array2 = '{"Joey":60, "Ross":100}

array3 = '{"Apples":"Oranges", "Pears":"44"}

ncsim: *W,RNQUIE: Simulation is complete.

Associative Array Methods

Function	**Description**
function int num ();	Returns the number of entries in the associative array
function int size ();	Also returns the number of entries, if empty 0 is returned
function void delete ([input index]);	*index* when specified deletes the entry at that index, else the whole array is deleted
function int exists (input index);	Checks whether an element exists at specified index; returns 1 if it does, else 0
function int first (ref index);	Assigns to the given index variable the value of the first index; returns 0 for empty array
function int last (ref index);	Assigns to given index variable the value of the last index; returns 0 for empty array
function int next (ref index);	Finds the smallest index whose value is greater than the given index
function int prev (ref index);	Finds the largest index whose value is smaller than the given index

Associative Array Methods Example

```
module tb;
   int     fruits_l0 [string];


   initial begin
```

```
fruits_l0 = '{ "apple"  : 4,
              "orange" : 10,
              "plum"   : 9,
              "guava"  : 1 };



// size() : Print the number of items in the given dynamic array
$display ("fruits_l0.size() = %0d", fruits_l0.size());



// num() : Another function to print number of items in given array
$display ("fruits_l0.num() = %0d", fruits_l0.num());



// exists() : Check if a particular key exists in this dynamic array
if (fruits_l0.exists ("orange"))
  $display ("Found %0d orange !", fruits_l0["orange"]);

if (!fruits_l0.exists ("apricots"))
  $display ("Sorry, season for apricots is over ...");

// Note: String indices are taken in alphabetical order
// first() : Get the first element in the array
begin
      string f;
  // This function returns true if it succeeded and first key is stored
  // in the provided string "f"
  if (fruits_l0.first (f))
```

```
      $display ("fruits_l0.first [%s] = %0d", f, fruits_l0[f]);
    end

    // last() : Get the last element in the array
    begin
      string f;
      if (fruits_l0.last (f))
        $display ("fruits_l0.last [%s] = %0d", f, fruits_l0[f]);
    end

    // prev() : Get the previous element in the array
    begin
      string f = "orange";
      if (fruits_l0.prev (f))
        $display ("fruits_l0.prev [%s] = %0d", f, fruits_l0[f]);
    end

    // next() : Get the next item in the array
    begin
      string f = "orange";
      if (fruits_l0.next (f))
        $display ("fruits_l0.next [%s] = %0d", f, fruits_l0[f]);
    end
  end
endmodule
```

Simulation Log

```
ncsim> run
fruits_l0.size() = 4
```

```
fruits_l0.num() = 4
Found 10 orange !
Sorry, season for apricots is over ...
fruits_l0.first [apple] = 4
fruits_l0.last [plum] = 9
fruits_l0.prev [guava] = 1
fruits_l0.next [plum] = 9
ncsim: *W,RNQUIE: Simulation is complete.
```

Dynamic array of Associative arrays

```
module tb;
  // Create an associative array with key of type string and value of type int
  // for each index in a dynamic array
  int fruits [] [string];

  initial begin
    // Create a dynamic array with size 2
    fruits = new [2];

    // Initialize the associative array inside each dynamic array index
    fruits [0] = '{ "apple" : 1, "grape" : 2 };
    fruits [1] = '{ "melon" : 3, "cherry" : 4 };

    // Iterate through each index of dynamic array
    foreach (fruits[i])
      // Iterate through each key of the current index in dynamic array
      foreach (fruits[i][fruit])
        $display ("fruits[%0d][%s] = %0d", i, fruit, fruits[i][fruit]);
```

```
  end
endmodule
```

Simulation Log

```
ncsim> run
fruits[0][apple] = 1
fruits[0][grape] = 2
fruits[1][cherry] = 4
fruits[1][melon] = 3
ncsim: *W,RNQUIE: Simulation is complete.
```

Dynamic array within each index of an Associative array

```
// Create a new typedef that represents a dynamic array
typedef int int_da [];

module tb;
  // Create an associative array where key is a string
  // and value is a dynamic array
  int_da fruits [string];

  initial begin
    // For key "apple", create a dynamic array that can hold 2 items
    fruits ["apple"] = new [2];

    // Initialize the dynamic array with some values
    fruits ["apple"] = '{ 4, 5};

    // Iterate through each key, where key represented by str1
    foreach (fruits[str1])
      // Iterate through each item inside the current dynamic array ie.fruits[str1]
```

```
    foreach (fruits[str1][i])
      $display ("fruits[%s][%0d] = %0d", str1, i, fruits[str1][i]);

  end
endmodule
```

Simulation Log

```
ncsim> run
fruits[apple][0] = 4
fruits[apple][1] = 5
ncsim: *W,RNQUIE: Simulation is complete.
```

Associative Array

An **associative array**, also known as a **map**, **dictionary**, or **hash table**, is a data structure that stores key-value pairs. Each key is unique and is used to retrieve the corresponding value. Associative arrays allow for efficient data retrieval based on keys, making them ideal for scenarios where quick access to data is needed.

- **Key Characteristics:**
- **Key-Value Pairs:** Data is stored as pairs, where each key maps to a value.
- **Fast Lookup:** Typically allows average-case O(1) time complexity for lookups, insertions, and deletions.
- **Dynamic Size:** Can grow or shrink as elements are added or removed.
- **Common Use Cases:**
- Storing user preferences or settings.
- Implementing caches.
- Counting occurrences of items (like word frequencies).

Linked List

A **linked list** is a linear data structure where elements, called **nodes**, are stored in a sequence. Each node contains two parts: the data (or value) and a reference (or pointer) to the next node in the sequence. This structure allows for efficient insertion and deletion of elements.

- **Key Characteristics:**
- **Dynamic Size:** Can grow and shrink as needed, unlike arrays which have a fixed size.
- **Non-contiguous Storage:** Nodes can be scattered throughout memory, as they are linked using pointers.
- **Types of Linked Lists:**
    - **Singly Linked List:** Each node points to the next node, with no backward links.
    - **Doubly Linked List:** Each node has pointers to both the next and the previous nodes, allowing traversal in both directions.
    - **Circular Linked List:** The last node points back to the first node, forming a circle.
- **Common Use Cases:**
- Implementing stacks and queues.
- Managing dynamic memory (e.g., in garbage collection).
- Representing complex data structures like graphs.

Summary

In summary, associative arrays are used for efficient key-value storage and retrieval, while linked lists are used for dynamic data storage where elements can be easily inserted and deleted. Each data structure has its strengths and is suited for different types of applications.

OCP (precise and unprecise, posted and non-posted) and MDIO protocol (phases with request and accept acknowledgements)?

OCP (Open Core Protocol) transactions can be precise or imprecise, and posted or non-posted. MDIO (Management Data Input/Output) transactions, used for managing Ethernet PHYs, follow a phased protocol with request and acceptance acknowledgements.

OCP Transactions:

- **Posted vs. Non-Posted:**

A posted transaction does not require a response from the slave to confirm the transaction's completion. A non-posted transaction, on the other hand, requires a response from the slave.

- **Precise vs. Imprecise:**

Precise bursts in OCP guarantee that each transfer in a burst is executed, while imprecise bursts might only provide an estimate of the total transfers.

- **Phases:**

OCP transactions are generally split into request, data, and response phases.

MDIO Transactions:

- **Phased Protocol:**

MDIO transactions involve a sequence of phases, including request and accept acknowledgements, for reading and writing data to Ethernet PHY registers, according to the 1999 IEEE standard.

- **Request and Accept:**

The master sends a request to the PHY, and the PHY sends an accept acknowledgment if it is ready to process the request.

- **Data Transfer:**

Once the request is accepted, data is transferred between the master and the PHY, as explained in the document.

- **Phases:**

MDIO transactions include turnaround and idle phases in addition to request, accept, and data phases

In the Open Core Protocol (OCP), precise and imprecise bursts refer to how the number of transfers in a burst is handled. In precise bursts, the number of transfers is explicitly defined and constant throughout the burst. In imprecise bursts, the number of transfers is not fixed and can change during the burst, with the protocol relying on other signals to indicate the end of the burst.

Elaboration:

- **Precise Bursts:**
    - The MBurstLength signal specifies the exact number of transfers in the burst.
    - This value remains constant throughout the burst.
    - The burst ends when the specified number of transfers is completed.
- **Imprecise Bursts:**
    - The MBurstLength signal provides a best guess of the number of transfers.

- The length can change during the burst, and the end of the burst is indicated by other signals, such as MReqLast.
- This approach offers flexibility, allowing for more dynamic burst lengths.

Key Differences:

Feature	**Precise Burst**	**Imprecise Burst**
MBurstLength	Exact value, constant	Best guess, can change
Burst End	By count	By signal (e.g., MReqLast)
Flexibility	Less	More

SDIO protocol revise?

The Secure Digital Input/Output (SDIO) protocol is an extension of the SD protocol that allows for the connection of various I/O devices to SD cards. It enables devices to connect to an SD card and exchange data. SDIO operates in both SD mode and SPI mode. The protocol has been revised and updated over time, with various versions introducing new features and functionalities.

Key aspects of the SDIO protocol and its revisions:

- **SDIO vs. SD:**

SDIO builds upon the SD memory card interface, adding support for general-purpose I/O functions.

- **Data Transfer:**

SDIO supports both 1-bit and 4-bit data transfer modes.

- **Bus Speed:**

SDIO supports different bus speeds, with earlier versions using 50MHz (SD2.0) and newer versions using 200MHz (UHS I, SD3.0).

- **Function Mapping:**

SDIO allows for up to seven I/O functions to be mapped into its interface, enabling connection of devices like Bluetooth, Wi-Fi, GPS, etc.

- **Revisions and Enhancements:**

SDIO specifications have evolved over time, with versions 1.01, 2.00, and later versions introducing features like bus speed changes, voltage range extensions, and support for different card types.

- **SPI Mode:**

SDIO also supports SPI mode, allowing for communication using the SPI protocol.

- **Host-Device Communication:**

The host (e.g., microcontroller) sends commands over the CMD line, and the device (e.g., SDIO card) responds.

In summary, the SDIO protocol is a versatile interface that allows for the expansion of SD card capabilities by enabling communication with various I/O devices.

AXI3/AXI4/APB revise?

The AXI3, AXI4, and APB are all part of the AMBA (Advanced Microcontroller Bus Architecture) family, which provides interfaces for connecting components within a System-on-Chip (SoC). AXI (Advanced eXtensible Interface) is designed for high-performance and high-bandwidth applications, while APB (Advanced Peripheral Bus) is optimized for low-cost, low-power peripherals.

AXI3 vs. AXI4:

- **AXI3:** The earlier version of the AXI protocol.
- **AXI4:** An updated version that addresses some limitations of AXI3, offering improvements in features and performance.
    - **Burst Length:** AXI3 supports burst lengths up to 16 beats, while AXI4 supports up to 256 beats.
    - **WID Signal:** AXI4 removes the WID signal from the W channel because write data reordering is no longer allowed.
    - **User-Defined Signals:** AXI4 adds user-defined signals to each channel, providing more flexibility.

APB:

- **Purpose:**

APB is a low-cost, low-power interface designed for connecting to low-bandwidth peripherals that don't require high-performance data transfer.

- **Characteristics:**
    - **APB is not pipelined, meaning each transfer takes at least two cycles.**
    - APB relates signal transitions to the rising edge of the clock, simplifying integration.
    - APBs are used to access the programmable control registers of peripheral devices.

In essence:

- AXI4 is the preferred protocol for high-performance applications in modern SOCs, while AXI3 is often used for older designs or when specific AXI3 features are required.
- APB is a good choice for low-bandwidth peripherals, where power consumption and cost are important factors.
- The AXI-to-APB bridge is a common way to connect AXI masters to APB slaves.

AXI3 and AXI4 are two versions of the Advanced eXtensible Interface (AXI) protocol, a standard for on-chip communication in System-on-a-Chip (SoC) designs. While AXI3 and AXI4 share similarities, AXI4 offers enhanced features and is generally preferred for modern designs. Key differences include burst length, write interleaving, locked transfers, and quality of service (QoS) support.

Key Differences between AXI3 and AXI4:

- **Burst Length:**

AXI3 supports burst lengths up to 16 beats, while AXI4 supports burst lengths up to 256 beats, allowing for more efficient data transfers.

- **Write Interleaving:**

AXI3 supports write interleaving, which allows multiple write transactions to be interleaved, potentially improving performance. AXI4 does not support write interleaving.

- **Locked Transfers:**

AXI3 supports locked transfers, where a master can lock a slave's memory region to prevent other masters from accessing it. AXI4 does not support locked transfers.

- **Quality of Service (QoS):**

AXI4 supports QoS, which allows for prioritization of different types of traffic, potentially improving performance and reliability. AXI3 does not support QoS.

- **Read Address Length:**

AXI4 uses a wider signal (ARLEN) for specifying read address length, enabling longer read bursts.

- **ARLOCK Signal:**

AXI4 reduces the ARLOCK signal to a single bit for exclusive transfers, as locked transfers are not supported.

- **User-Defined Signals:**

AXI4 adds user-defined signals to the read channels, offering more flexibility.

- **AXI4-Stream:**

AXI4 introduces the AXI4-Stream protocol, which is a fast, unidirectional protocol for transferring data from master to slave.

Usefulness of Changes in AXI4:

The changes in AXI4 are generally aimed at improving performance, efficiency, and flexibility in on-chip communication. The longer burst lengths in AXI4 allow for more efficient data transfers, while the QoS support in AXI4 enables better management of different traffic types. The removal of write interleaving and locked transfers in AXI4 simplifies the protocol and may offer performance advantages in certain scenarios.

In essence, AXI4 is a more advanced and versatile protocol compared to AXI3, offering better performance and flexibility for modern SoC designs.

AHB5 (retry and split transactions) revise?

In AHB, both RETRY and SPLIT responses allow a slave to release the bus when it cannot complete a transfer immediately, preventing the bus from being stalled. However, they differ in how the bus is granted after the response: RETRY prioritizes higher-priority masters, while SPLIT allows any master to access the bus, even lower-priority ones.

Here's a more detailed breakdown:

RETRY:

- Indicates the bus master should re-attempt the transfer after the bus is granted again.

- The arbiter will prioritize higher-priority masters for the next bus grant.
- The master must cancel any pending transfers and re-attempt the current failed one.

SPLIT:

- Indicates the slave will split the transfer into multiple cycles.
- The arbiter can grant the bus to other masters before the SPLIT transaction is complete.
- The master must wait for the slave to signal completion of the SPLIT transfer before attempting to complete the transaction.
- SPLIT responses are more complex to implement than RETRY but can improve bus bandwidth efficiency.

Key Differences Summarized:

Feature	**RETRY**	**SPLIT**
Arbiter Action	Grants bus to higher priority masters	Grants bus to any master, even lower-priority ones
Master Action	Re-attempts transfer after bus grant	Waits for slave to signal completion, then re-attempts transfer
Complexity	Simpler to implement	More complex, but potentially more efficient in using bus bandwidth

I2C (SDA/SCL)/I3C (secondary master to manage transactions) and I2S (right and left channel) and SPI (MISO/MOSI) revise?

I2C (SDA/SCL), I3C (MIPI Alliance Improved Inter-Integrated Circuit), I2S (Inter-IC Sound), and SPI (Serial Peripheral Interface) are different serial communication protocols used in various applications. I2C and I3C are primarily for inter-integrated circuit communication, I2S is designed for audio, and SPI is used for various peripherals.

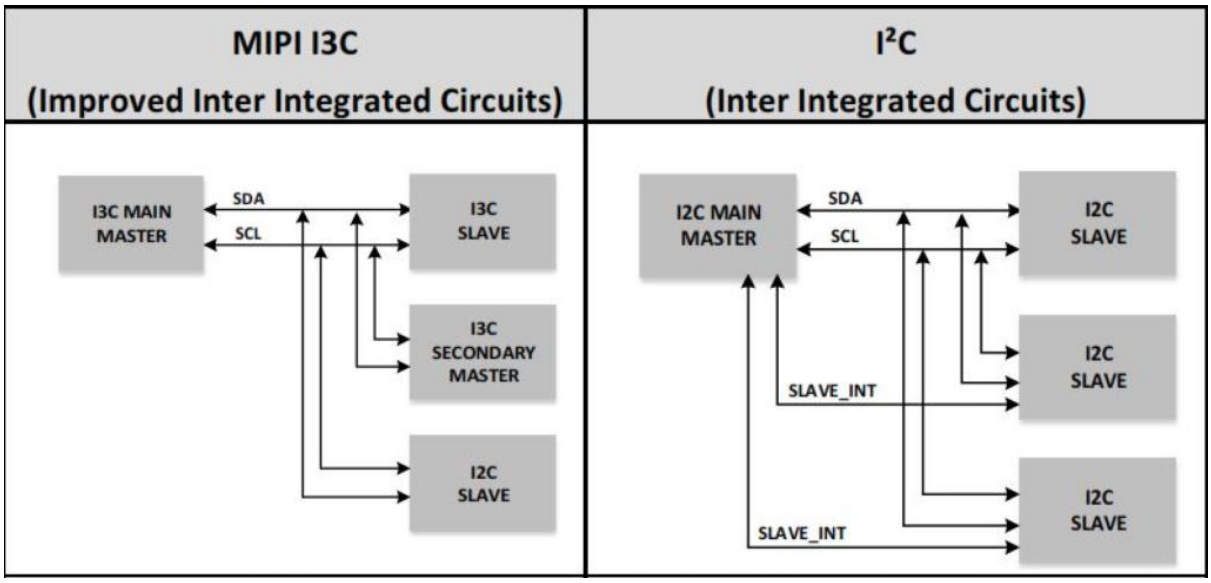
I2C (Inter-Integrated Circuit):

- Uses SDA (Serial Data) and SCL (Serial Clock) lines for communication.

- Supports both master and slave devices.
- Typically used for slower data transfers and for devices like EEPROMs and sensors.
- Can operate in different speed modes (Standard-mode, Fast-mode, Fast-mode Plus, High-speed).
- Can have address conflicts if not managed properly.
- I2C can co-exist with I3C, but I3C is preferred for faster and more efficient communication.

I3C (MIPI I3C):

- Also uses SDA and SCL lines but with enhancements for speed and power efficiency.
- Introduces a secondary master for managing transactions, allowing multiple masters to share the bus.
- Can achieve higher data rates compared to I2C, up to 12.5 Mbps.
- Designed for lower power consumption.
- Can be used for a wider range of applications, including sensors and other peripherals.
- Has been gaining popularity as a more efficient alternative to I2C, especially for applications requiring high data rates and low power consumption.



I3C offers significantly higher data rates compared to I2C, with the potential to reach 100 Mbps, while I2C typically operates at speeds of up to 1 MHz. I3C achieves this through its Single Data Rate (SDR) and High Data Rate (HDR) modes, allowing for faster data transfer and improved power efficiency.

I3C Data Rates:

- **Standard Data Rate (SDR):** 10-12.5 Mbps.
- **High Data Rate (HDR):** Up to 100 Mbps.
- **High Data Rate (HDR-DDR):** Up to 25 Mbps.

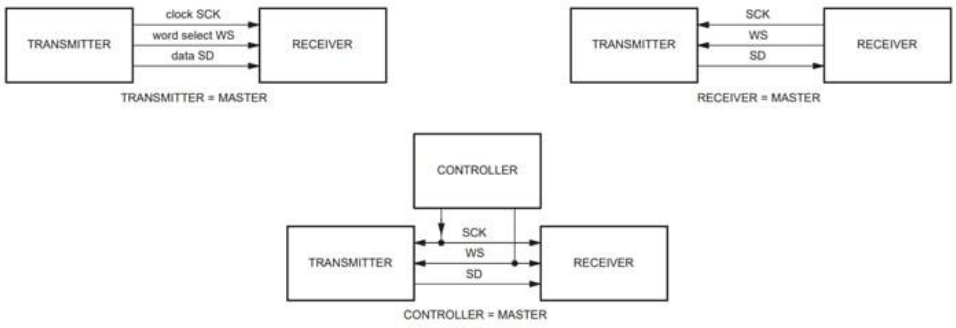
I2C Data Rates:

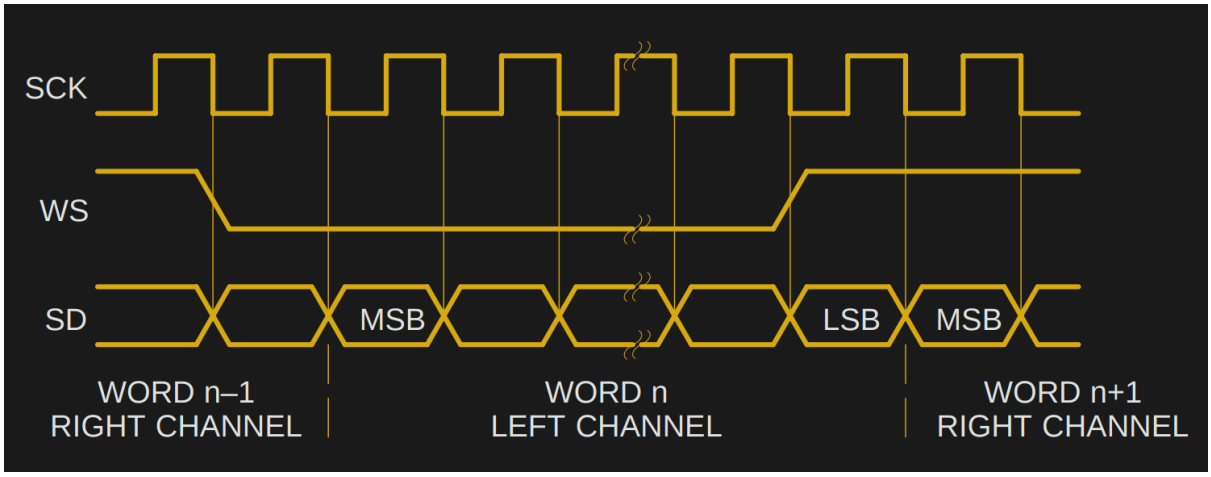
- **Standard Mode:** 100 kbit/s.
- **Fast Mode:** 400 kbit/s.
- **Fast Mode Plus:** 1 Mbit/s.
- **High-speed Mode:** 3.4 Mbit/s.

In Summary: I3C provides significantly faster data transfer rates than I2C, making it suitable for applications requiring higher bandwidth and faster communication.

I2S (Inter-IC Sound):

- Used for audio data transfer, typically between digital audio devices.
- Uses three primary lines: data, clock, and word select (WS or FS).
- WS/FS signal is used to distinguish between left and right channels.
- Can transfer data in stereo, with the left and right channels being interleaved or separate.





SPI (Serial Peripheral Interface):

- Uses MISO (Master In Slave Out) and MOSI (Master Out Slave In) lines for data transfer, as well as SCLK (Serial Clock) and optionally chip select lines.
- Can be used for a variety of applications, including memory chips and other peripherals.
- Offers high data rates and flexible configurations.
- Can be configured for full-duplex (MISO/MOSI separate) or half-duplex (MISO/MOSI combined).
- SPI is suitable for applications where speed and flexibility are needed, such as memory chips.

In summary, I2C and I3C are for general inter-IC communication, I2S is for audio, and SPI is for various peripherals, with I3C and SPI offering higher speeds and more flexibility than I2C and I2S respectively.

CAN and CAN-FD difference? and how it works?

CAN-FD (Flexible Data-rate) is an extension of the original CAN protocol that increases data rates and payload size, allowing for faster and more efficient data transmission. While CAN is limited to 1 Mbit/s and 8 bytes per frame, CAN-FD can achieve higher speeds (up to 5 Mbit/s) and larger payloads (up to 64 bytes per frame).

How CAN works:

1. **1. Arbitration:**

When multiple devices on the CAN bus want to transmit, they "arbitrate" for priority based on a unique 11-bit or 29-bit identifier in the frame. The device with the highest priority (lower identifier) wins and can send its data.

2. **2. Data Transmission:**

The winning device transmits its data (up to 8 bytes in standard CAN, or up to 64 bytes in CAN-FD) along with control bits and error detection mechanisms (CRC).

3. **3. Reception:**

All devices on the bus receive the transmitted data and check if it's intended for them based on the identifier.

4. **4. Acknowledgement:**

The receiving device acknowledges successful reception by responding with a specific bit pattern.

CAN-FD vs. CAN:

- **Data Rate:**

CAN-FD allows for a higher data rate (up to 5 Mbit/s) for the data payload, while the arbitration phase (where the message ID is transmitted) still operates at the standard CAN rate (1 Mbit/s).

- **Payload Size:**

CAN-FD can transmit up to 64 bytes of data per frame, compared to 8 bytes in standard CAN.

- **Frame Structure:**

While the basic frame structure is similar, CAN-FD introduces new fields like the Flexible Data Field (FDF) and the Data Bit Rate Selection (DBR) bit, which enable the higher data rates and larger payloads.

- **Compatibility:**

CAN-FD is designed to be compatible with standard CAN, meaning that standard CAN devices can still communicate with CAN-FD devices under certain conditions.

Benefits of CAN-FD:

- **Increased Throughput:**

The higher data rates and larger payloads lead to significantly increased throughput, allowing for more data to be transmitted in a given time.

- **Reduced Overhead:**

Larger payloads mean fewer frames are needed to transmit the same amount of data, reducing overhead on the bus.

- **Improved Efficiency:**

The ability to switch between different data rates in the data phase can lead to more efficient use of the bus bandwidth.

- **Enhanced Reliability:**

Improved CRC and other error detection mechanisms in CAN-FD reduce the risk of undetected errors.

Applications:

CAN-FD is commonly used in automotive applications, especially in vehicles with advanced features that require more data transmission capacity. It's also used in industrial automation and other applications where higher bandwidth and larger payloads are needed.

Introduction to CAN (Controller Area Network)

February 19, 2019 by **Stephen St. Michael**

-
-
-
-

This article introduces the Controller Area Network (CAN) serial communication bus, detailing message frames, bus arbitration, and signaling.

This article introduces the Controller Area Network (CAN) serial communication bus, detailing message frames, bus arbitration, and signaling.

Supporting Information

- Serial Communication in Harsh Environments: A New CAN Transceiver from Maxim
- The Why and How of Differential Signaling

What Is CAN?

The Controller Area Network (CAN) is a serial communication bus designed for robust and flexible performance in harsh environments, and particularly for industrial and automotive applications.

Originally invented by Bosch and later codified into the ISO11898-1 standard, CAN defines the data link and physical layer of the Open Systems Interconnection (OSI) model, providing a low-level networking solution for high-speed in-vehicle communications. In particular, CAN was developed to reduce cable wiring, so the separate electronic control units (ECUs) inside a vehicle could communicate with only a single pair of wires. Figure 1 shows the ECUs in a car connected to a CAN bus.

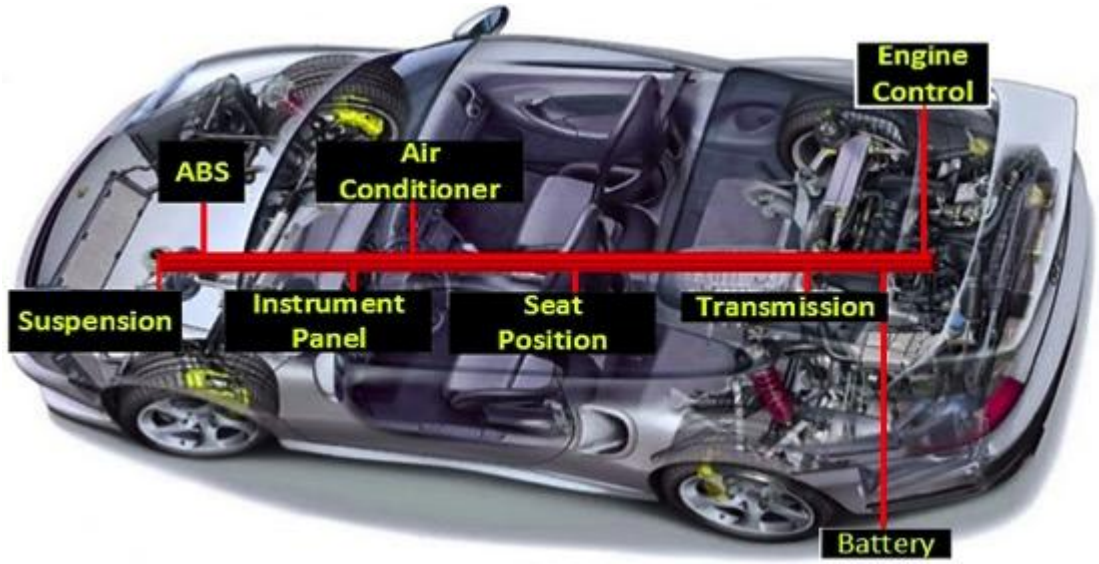


Figure 1. CAN is used for ECU communication in cars. Image used courtesy of the Infosec Institute.

Onboard diagnostics (OBD) is your vehicle's diagnostic and reporting system that allows you or a technician to troubleshoot problems via diagnostic trouble codes (DTCs). When the “check engine” light comes on, a technician will often use a handheld device to read the engine codes off of the vehicle. At the lowest level, this data is transmitted via a signaling protocol, which in most cases is CAN.

DeviceNet is a high-level networking protocol used in industrial applications. It greatly reduces the wiring needed between a control system and I/O devices. Rather than connecting each device to a separate input/output on a PLC's I/O modules, devices can be linked together via a four-wire connector and connected to a network scanner on the PLC. At the lowest level, we find CAN working its magic within the DeviceNet protocol. Figure 2 shows a PLC scanning a network of industrial devices communicating over DeviceNet.

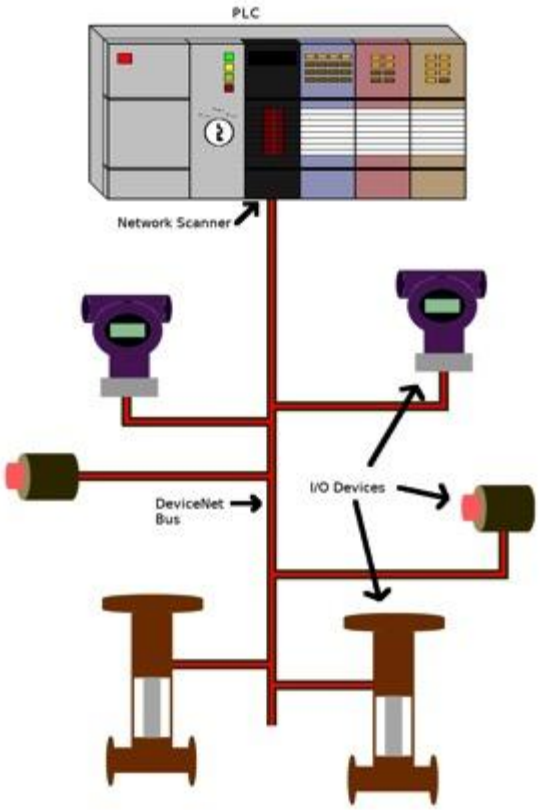


Figure 2. PLC connected to a DeviceNet network that uses CAN for the data link and physical layer.

CAN Message Frames

So what does a CAN message actually look like? The original ISO standard laid out what is called Standard CAN. Standard CAN uses an 11-bit identifier for different messages, which comes to a total of 2^{11}, i.e. 2048, different message IDs. CAN was later modified; the identifier was expanded to 29 bits, giving 2^{29} identifiers. This is called Extended CAN. CAN uses a multi-master bus, where all messages are broadcast on the entire network. The identifiers provide a message priority for arbitration.

CAN uses a differential signal with two logic states, called recessive and dominant. Recessive indicates that the differential voltage is less than a minimum threshold voltage. Dominant indicates that the differential voltage is greater than this minimum threshold. Interestingly, the dominant state is achieved by driving a logic '0' onto the bus, while the recessive state is achieved by a logic '1'. This is inverted from the traditional high and low used in most systems. These two states will be detailed later on in the article. The important thing is that a dominant state overrides a recessive during arbitration.

Standard CAN

The standard CAN message frame consists of a number of bit fields. These are shown in Figure 3.

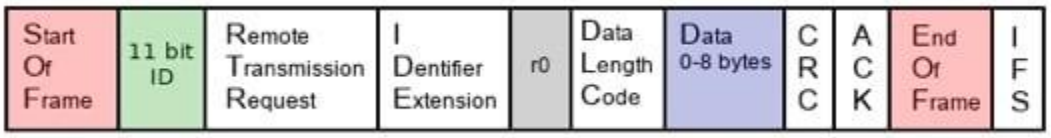


Figure 3. Standard CAN message frame

The first bit is the start of frame (SOF). This dominant bit represents the start of a CAN message. Next is the 11-bit identifier, which establishes the priority of the CAN message. The smaller the identifier, the higher the priority of the message.

The remote transmission request (RTR) bit is normally dominant, but it goes recessive when one node is requesting data from another. The identifier extension (IDE) bit is dominant when a standard CAN frame is being sent and not an extended one. The r0 bit is reserved and not currently used. The data length code (DLC) nibble signifies how many bytes of data are in this message.

Next is the data itself, being as many bytes as represented in the DLC bits. The cyclic redundancy check (CRC) is a 16-bit checksum for detecting errors in the transmitted data. If the message is properly received, the receiving node overwrites the recessive acknowledge bit (ACK) with a dominant bit. The ACK also contains a delimiter bit to keep things synchronized. The end of frame (EOF) signifies the end of the CAN message and is 7 bits wide, for detecting bit-stuffing errors. The last part of a CAN message is the interframe space (IFS), used as a time delay. This time delay is precisely the amount of time needed for a CAN controller to move the received message into a buffer for further processing.

Extended CAN

Extended CAN uses a 29-bit identifier along with a few additional bits. An extended message has a substitute remote request (SRR) bit after the 11-bit identifier, which acts as a placeholder to keep the same structure as standard CAN. This time the identifier extension (IDE) should be recessive, indicating that the extended identifier follows it. The RTR bit is after the 18-bit ID and is followed by a second reserve bit, r1. The rest of the message remains the same.

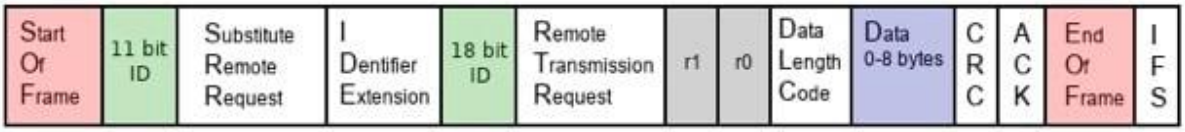


Figure 4. Extended CAN message frame

CAN Message Types

Now that you know what a CAN message looks like, you might be wondering what kinds of messages are passed along the bus. CAN allows for four different message types. They are the data frame, remote frame, overload frame, and error frame.

A standard CAN data frame makes use of the identifier, the data, and data length code, the cyclic redundancy check, and the acknowledgment bits. Both the RTR and IDE bits are dominant in data frames. If the recessive acknowledge bit at the receiving end is overwritten by a dominant bit, both the transmitter and receiver recognize this as a successful transmission.

A CAN remote frame looks similar to a data frame except for the fact that it does not contain any data. It is sent with the RTR bit in a recessive state; this indicates that it is a remote frame. Remote frames are used to request data from a node.

When a node detects an error in a message on the CAN bus, it transmits an error frame. This results in all other nodes sending an error frame. Following this, the node where the error occurred retransmits the message. The overload frame works similarly but is used when a node is receiving frames faster than it can process them. This frame provides a time buffer so the node can catch up.

Bus Arbitration & Signaling

CAN is a CSMA/CD protocol, meaning each node on the bus can detect collisions and back off for a certain amount of time before trying to retransmit. This collision detection is achieved through a priority arbitration based on the message identifiers. Before we discuss arbitration, let's take a closer look at the dominant and recessive bits used on the CAN bus.

Inverted Logic

An interesting aspect of the CAN bus is that it uses an inverted form of logic with two states, dominant and recessive. Figure 5, below, shows a simplified version of a CAN transceiver's output and input. The '101' bitstream is coming from/going to a CAN controller and/or microcontroller. Notice that when the controller sends a stream of bits, these are complemented and placed on the CANH line. The CANL line is always the complement of CANH. For arbitration to work, a CAN device must monitor both what it is sending and what is currently on the bus, i.e., what it's receiving.

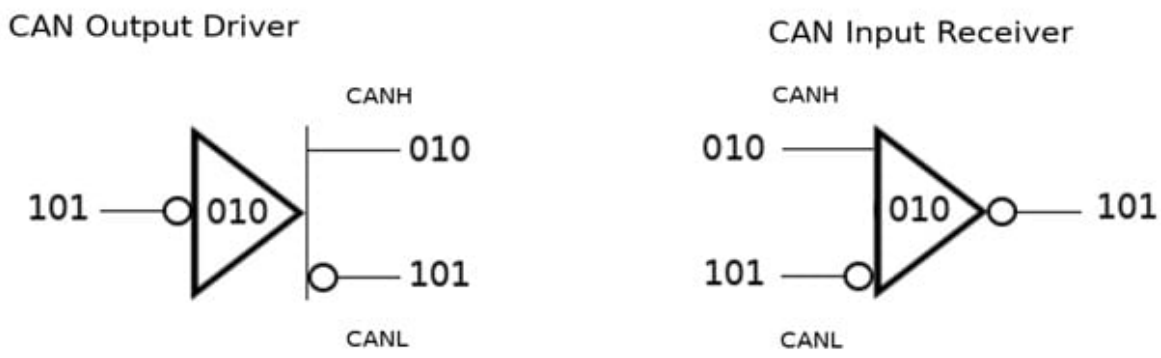


Figure 5. CAN output/input

Figure 6 shows both the CANH and CANL signals simultaneously so that you can see the CAN bus in action. Plotted below the bus signals is the differential voltage that corresponds to the dominant and recessive states of the CAN signals. The first three segments in time, t1–t3, are drawn to match up with the three bits shown above in Figure 5. We will look at this from the perspective of the output driver. The driver's input initially sees a '1' and complements this to a zero, which is placed on CANH. CANL sees the complement of CANH and goes high. This is shown as t1 in Figure 6. Notice that the CANH and CANL voltages are offset from one another. During time t1, CANH – CANL is very close to zero, since CANH and CANL are almost the same voltage. This period, where the driver is sending a logic '1' resulting in CANH and CANL being close to the same voltage, is what we call the CAN recessive state.

The next bit sent is a '0'. CANH gets its complement and CANL again gets the complement of CANH. Notice this time that the CANH and CANL voltages are not close to one another. Therefore, the differential voltage (VDIFF) is larger. This is the CAN dominant state. We say that the logic is inverted because a '1' takes the bus low and a '0' brings it high. The input receiver works in a similar fashion.

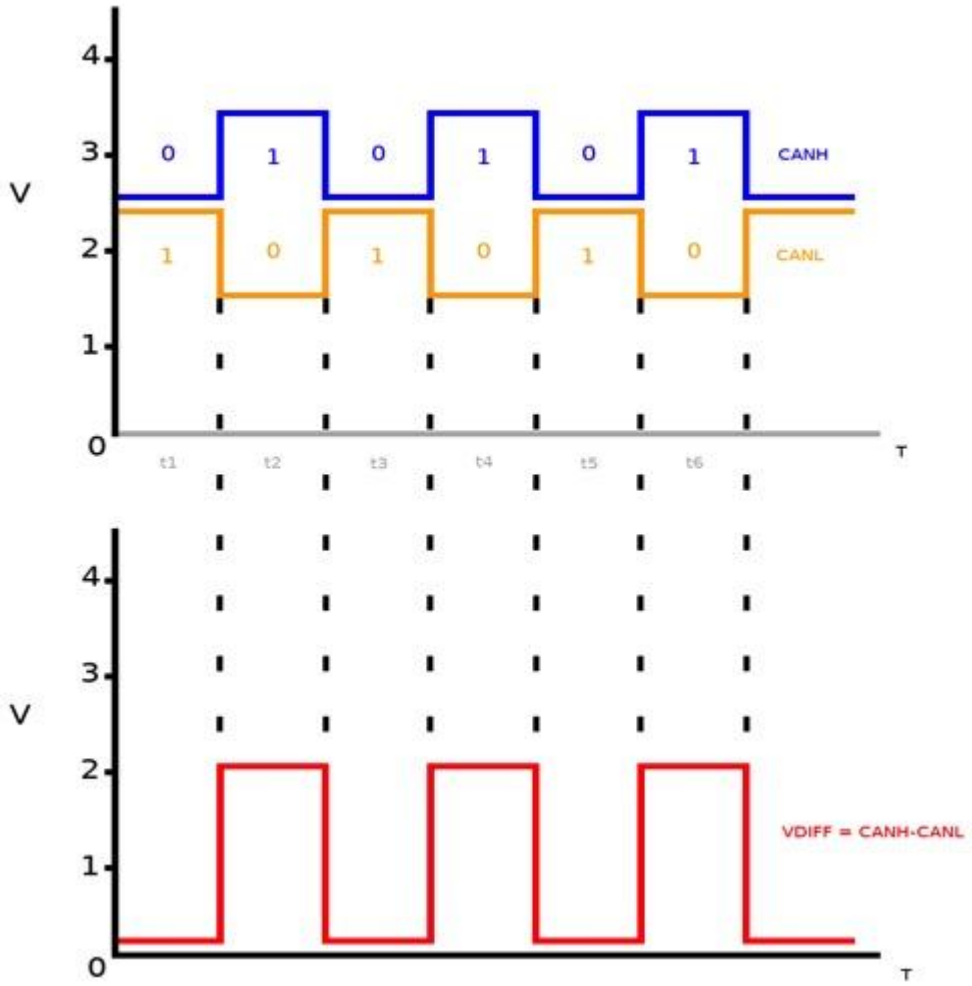


Figure 6. CAN recessive and dominant states with differential voltage shown

Priority Arbitration

As mentioned earlier, the smaller the 11-bit identifier is the greater the message's priority. Every bit that a node transmits, it monitors. This is how a node detects that a higher-priority message is being placed on the bus. The moment a node sends a recessive bit but detects a dominant bit on the bus, it backs off. This is called non-destructive arbitration because the winning message continues to be transmitted without any problem. Notice that a recessive, logic '1', loses out to a dominant, logic '0'. This makes sense since a lower identifier value represents a higher priority. To get a better idea of what this means, take a look at Figure 7, which shows three nodes on a CAN bus attempting to take control. It's important to remember that each time a recessive bit is shown, the controller is sending a '1', while dominant bits correspond to the sending of '0'.

Nodes 1–3 are all sending a stream of bits. This stream of bits represents the message identifiers and their priority. For starters, all three nodes send a '1', which is represented on the CAN bus as a recessive bit. Next, each node sends a '0', or dominant bit. The third bit placed on the bus is another '1', or recessive bit. At this point, none of the nodes have detected any conflict with another node on the bus, so they keep transmitting.

For the fourth bit, Node 1 sends a '0' or dominant bit. Node 2 transmits a recessive bit, yet detects a dominant bit on the bus. It immediately backs off, knowing that a higher-priority message is currently being sent. Node 3 continues to transmit since it read back the same dominant bit that it transmitted. When the fifth bit is placed on the bus, Node 3 then recognizes that it is at a lower priority and stops transmitting. Both Node 2 and Node 3 wait a certain amount of time before attempting again. This is shown on the right half of Figure 7, with Node 3 winning arbitration. As you can see, a logic '0' dominant bit corresponding to a lower message identifier allows the arbitration to take place.

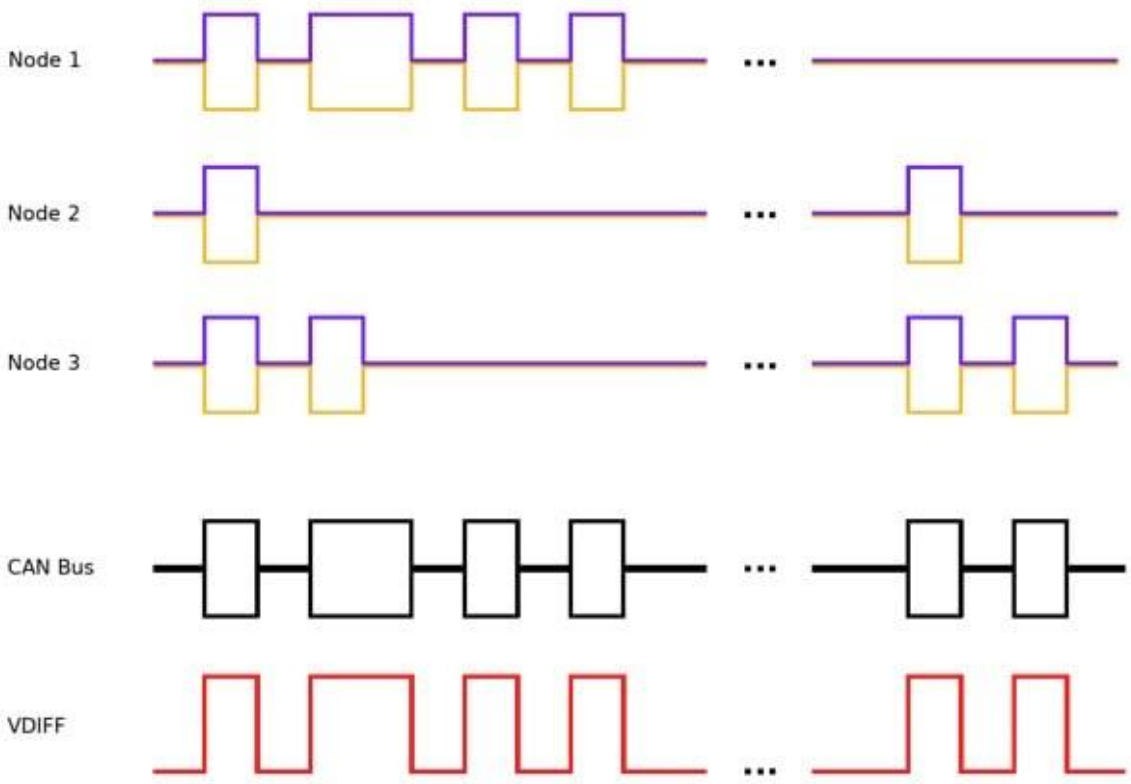


Figure 7. CAN bus arbitration with differential voltage

Conclusion

This article introduced the Controller Area Network or CAN. CAN is a robust serial communication bus found mostly in automotive and industrial environments. CAN uses a differential signal, which makes it more resistant to noise, along with a priority arbitration scheme for non-destructive message transmission. CAN is great for embedded applications that end up in hazardous environments or areas with a lot of electromagnetic interference. Whether you're building a remote-controlled submarine, setting up a microbrewery with pumps and sensors, or just hacking your car's computer, CAN is a great way to further your embedded knowledge while beefing up your next design project.

PCIe revise? flit and non-flit modes, and posted and non-posted, no response needed whereas non-posted completion status needed or response is needed?

In PCI Express (PCIe), "posted" and "non-posted" transactions determine if a completion packet is expected after a request. A posted transaction does not require a response, while a non-posted transaction does, and the response is called a completion. "Flit" and "non-flit" refer to data stream modes, with flit mode being introduced in PCIe 6.0 for improved bandwidth and error correction. Non-flit mode is mandatory, while flit mode is required only for data rates exceeding 32.0 GT/s.

Elaboration:

- **Posted vs. Non-posted:**
    - **Posted:** The requester doesn't expect a completion packet. This is often used for bulk data transfers like memory writes, where the requester doesn't need to wait for a confirmation.
    - **Non-posted:** The requester expects a completion packet from the completer. This is used for operations like memory reads, where the requester needs to receive the data.
- **Flit Mode:**
    - **Introduced in PCIe 6.0:** Flit mode uses fixed-size packets (flits) for data transfer.
    - **Benefits:** Improves bandwidth efficiency, reduces latency, and simplifies data management at the controller level.
    - **Required for certain data rates:** Flit mode is mandatory for data rates exceeding 32.0 GT/s.
    - **Low power state:** PCIe 6.0 introduces L0p, a low-power state, that works in conjunction with flit mode, allowing lanes to be shut down while still maintaining active lanes for data flow.
- **Non-posted Completion Status:**
    - **Required for non-posted transactions:** Non-posted transactions require a completion status to indicate the success or failure of the operation.
    - **Response:** The completion packet provides the requester with information about the request's outcome.
    - **No response needed for posted transactions:** Posted transactions do not require a completion status or response from the completer.

In summary, while both flit and non-flit modes are related to data transfer within PCIe, posted and non-posted are transaction types that define whether a response (completion) is expected. Non-posted transactions require a completion, while posted transactions do not.

Parameterization in uvm of address or data width, and sequence_item parameterization in sequence?

In UVM, parameterization of address and data widths can be handled at the interface level, while sequence item parameterization is done within the sequence class itself. This approach allows for flexibility and reuse of UVM components across different bus or protocol configurations.

Here's a more detailed explanation:

1. Parameterization at the Interface Level:

- **Abstraction:**

Define the bus or protocol interface (e.g., APB, AHB) with parameters for address and data width. This allows the interface to be instantiated with different configurations without changing the interface's core logic.

- **Example:**

An APB interface could have parameters
like APB_ADDR_WIDTH and APB_DATA_WIDTH. When creating an APB agent, you would pass in the desired width values as parameters, as explained in the DVCon Proceedings.

- **Benefits:**
    - **Reusability:** The same interface logic can be used for different bus protocols by simply changing the parameter values.
    - **Abstraction:** The higher-level sequence and test logic don't need to be aware of the exact implementation details of the bus or protocol.

2. Sequence Item Parameterization:

- **Flexibility:**

Sequence items (the basic units of UVM sequences) can have fields to hold data values like address or data content. These fields can be parameterized within the sequence class, allowing for different data values to be used in different sequences.

- **Example:**

A sequence item for an APB transaction could have a field address and a field data. The values of address and data can be parameterized within the sequence class, allowing for different addresses and data to be used in different sequences.

- **Benefits:**
    - **Data Specificity:** Each sequence can have unique data values associated with it, making it possible to control the stimulus more precisely.
    - **Reusability of Sequence Items:** The same sequence item class can be used to represent different types of transactions by changing the values of its parameterized fields.

In summary:

- **Interface level:** Parameterize the bus/protocol interface to handle address and data widths.
- **Sequence item level:** Parameterize data values within sequence items to create flexible and reusable sequences.

Align and unaligned transfer in axi?

In the AXI protocol, aligned transfers refer to data transfers where the starting address is a multiple of the data width (e.g., 32-bit data starting at an address that's a multiple of 32), while unaligned transfers occur when the starting address is not a multiple of the data width. AXI supports unaligned transfers, but the first transfer in a burst may need to be adjusted to accommodate the unaligned start, and subsequent transfers within the burst will be aligned.

Aligned Transfers:

- The starting address of the transfer is a multiple of the data bus width.
- For example, a 32-bit wide AXI bus, a transfer starting at address 0x0000 is aligned.
- This simplifies data processing and minimizes the need for realignment on the slave side.

Unaligned Transfers:

- The starting address of the transfer is not a multiple of the data bus width.
- For example, a 32-bit wide AXI bus, a transfer starting at address 0x0002 is unaligned.
- AXI supports unaligned transfers, but the first transfer in a burst might be adjusted to fit the unaligned start.

- Subsequent transfers within the burst will be aligned to the natural boundary of the data width.
- A master might need to send two transfers for a 32-bit data write on a 32-bit bus if the start address is 0x48E, as explained on Stack Overflow.

Key Considerations:

- **Data Realignment Engines (DREs):**

Some AXI cores, like AMD's AXI-CDM, offer DREs that can handle unaligned data and realign it to the memory map datapath, according to AMD's documentation.

- **Strobing:**

The AXI protocol uses strobing signals to indicate which data bits are valid, which helps in handling unaligned transfers.

- **First Transfer Adjustment:**

In unaligned transfers, the first transfer in a burst may have a different width than subsequent transfers to account for the misalignment.

- **Slave Responsibilities:**

The slave device (e.g., memory) might be responsible for handling the alignment of the first transfer or for performing any necessary realignment, depending on the core architecture.

Locked and unlocked transfer in axi3?

In AXI (Advanced eXtensible Interface), locked transfers ensure that only one master can access a specific slave region until an unlocked transfer from the same master completes. This is enforced by the interconnect's arbiter. Unlocked transfers, in contrast, do not require bus locking and allow other masters to access the same slave region simultaneously.

Locked Transfers:

- **Purpose:**

Ensure exclusive access to a slave region, preventing interference from other masters during a sequence of operations.

- **Mechanism:**

The ARLOCK or AWLOCK signals are used to indicate a locked transaction. The interconnect must ensure that only the master initiating the locked transaction can access the slave until an unlocked transfer from that same master completes.

- **Impact:**

Can impact interconnect performance as it requires preventing other transactions during the locked sequence.

- **AXI Versions:**

AXI3 supports locked accesses, but AXI4 does not.

Unlocked Transfers:

- **Purpose:** Allow multiple masters to access the same slave region concurrently.
- **Mechanism:** Do not use ARLOCK or AWLOCK signals.
- **Impact:** More efficient than locked transfers as they do not block other masters during the transaction.
- **AXI Versions:** Preferred in AXI4 and newer protocols.

Exclusive access in axi3/axi4?

In the Advanced eXtensible Interface (AXI) protocol, exclusive access is a mechanism that allows a master to perform atomic operations, such as read-modify-write sequences, on a shared memory region without risking data corruption due to concurrent access from other masters. It's a more efficient alternative to locked accesses, allowing other masters to access the same memory region while the exclusive access is in progress.

How Exclusive Access Works:

1. **1. Exclusive Read:**

A master initiates an exclusive read transaction to a specific address range. This read is used to obtain the current value of the memory location.

2. **2. Exclusive Write:**

After the read, the master attempts to perform an exclusive write to the same address range. This write includes data to be written and, crucially, a check to ensure that no other master has modified the memory location since the exclusive read.

3. **3. Exclusive Access Monitor:**

A hardware exclusive access monitor on the slave side (the memory controller) tracks the exclusive read and subsequent writes. It checks if any other master has accessed the monitored address range between the exclusive read and write.

4. **4. Response:**

- If the exclusive write is successful (no other master accessed the memory), the slave responds with EXOKAY.
- If another master accessed the memory between the read and write, the exclusive write is considered failed, and the slave responds with OKAY.

Advantages of Exclusive Access:

- **Atomicity:**

Ensures that operations are completed without data corruption due to interleaved access from other masters.

- **Efficiency:**

More efficient than locked accesses as it doesn't require the bus to be locked for the duration of the operation.

- **Semaphore Operations:**

Enables the implementation of semaphore-type operations without requiring the bus to remain dedicated to a particular manager.

Key Considerations:

- **Exclusive Access Monitor:**

The slave must implement a hardware exclusive access monitor to track exclusive accesses and ensure atomicity.

- **Address Range:**

Exclusive accesses operate on a specific address range, not just a single address.

- **Error Handling:**

If an exclusive write fails (another master accessed the memory), the master needs to handle the failure appropriately (e.g., retry the operation).

What I suggest is creating a 2-dimensional dynamic array. Since you seem to be working with bytes widths, you could do:

bit[7:0] data[][];

OOPs Concept?

System Verilog introduced OOPs concept which was missing in verilog. As we have seen that System Verilog was introduced keeping in mind the needs of complex design verification. In

complex test benches we make lots of components and using modules was not an efficient way to create components as the components defined as module would not have features like inheritance, polymorphism reducing the reusability. In this article we will focus more on the generic concepts of OOP which will help understand future articles.

Introduction

Most of you might be familiar with OOPs already and they can skip this article. So, let's get started with the introduction of OOPs.

What is OOP?

OOP stands for **object-oriented programming**. This type of programming gets its inspiration from real word objects where every object has its own property, characteristics defined by a blueprint. As this programming model is inspired by reads-life objects writing code becomes more easier as we can relate it to real objects.

Some key features of OOP are:

- Inheritance
- Polymorphism
- Encapsulation
- Data Abstraction

Some popular OOPs language is C++, C#, Java, etc.

OOPs feature in System Verilog

System Verilog is an HDL, i.e., hardware define language and thus all features of OOP are not needed in SV. Some of the complex OOP feature are omitted in the latest version of OOP. Some of the supported features are:

- Single & multi level inheritance
- Function overriding
- Virtual classes and pure virtual functions
- Virtual functions

We will slowly understand all the features in upcoming articles.

OOPs in detail!

OOPs as we have seen is based on real-life objects which are created from a blueprint. This blueprint is Kwan as class in OOP which defines the properties and behaviour of the object. Using this class multiple objects can be created from a single class each having unique name and properties.

Let us understand this with help of some real-life example. A car is an object which is created from the blueprint of a car. Blueprint will define the properties of the car, like it will have 4 wheels, steering wheel, breaks, design, etc. Now from this blueprint multiple cars (objects) can be created with some unique properties like chassis number. Also, different cars can have different colour, different configuration which may/may not be unique.

We will use similar real-life examples to understand distinct features of OOPs.

Class and Objects

Class is a blueprint which defines the properties and behaviour of an object. In OOP classes are the entity which encapsulates all the data and methods.

Objects are the unique entity created from class. Objects are dynamic in nature, i.e, it is created dynamically during runtime.

Illustraion of class and objects

Inheritance

This is the property of OOP by virtue of which a class can inherit properties and behaviour of another class called parent or base class. The child or derived class can add more property or behaviour on the base class.

Continuing the example of car, we can say that car is a child class of vehicle. Where vehicle would define properties like it should have brake, accelerator, tyres, etc. Car as a derived class of vehicle would have all the properties and add few other properties like it should have four tyres, steering wheel, parking brakes, etc. Similarly different car models can be child of car class.

Illustraion of parent and child class and object from these classes

Parent class can be known as base class or super class. Similarly, child class is known as derived class and sub class.

Encapsulation

This is a property by which we can bundle all the data and methods into one unit. This also helps in black boxing a unit where users can focus more on using the unit without knowing the underlying process or complexity.

With our car analogy, we can see car as a black box where we as a user of car are interested in driving the car without knowing how the car works. What happens when we apply brake??

Illustraion of encapsulation within a class

Polymorphism

Poly mean many and morph means form. Thus, **polymorphism** is a concept in which same method can act differently in child class or when inputs are different. OOPs provide us two ways to enable polymorphism – function overloading and function overriding.

For example – Let's suppose there is a method which calculates area. When there is 1 input, it would calculate area of square, when there are 2 inputs it will calculate area for rectangle and when 3 inputs are given it will calculate area for triangle. We see that the name of the method remains the same but the method perform different functionalities when number of inputs are different.

Data Abstraction

Data Abstraction means hiding unnecessary and representing only what is necessary for the user basically that particular use case.

We will use another example to understand this. Let us suppose there is a library in a school. It will keep the data of student who have borrowed books from the library. Now a student has a lot of properties such as roll number, name, address, grade, subject marks, extra-curricular activities, etc.
But for library not all the data are important. They need data like roll number, name, class, no of books borrowed, duration of borrowed book, etc. This is known as data abstraction where we focus on the data which is necessary for the use case and hide other data.

Data abstraction in OOP is implemented by *virtual class*, *pure virtual functions*.

Virtual class is kind of a framework which defines what type of data and methods are needed for some use case and depending on that different child classes can be created.

Please note we cannot create an object from a virtual class directly. We need to create a class from a virtual class and then only we can create object from it.

These were the four main concepts which build up the OOPs language.

Static and dynamic objects

We have seen that whenever we need to use class, we first need to create an object. So why creation of object is needed? We will explore the concept of static and dynamic nature of objects in this section.

Dynamic Objects

When we define a *class*, its just a blueprint which defines the properties and behaviour of the object. But this does not require any memory. When we create an object, memory is allocated for all the variables defined inside the class.
Just like, blueprint of a car will not require space, but when car (*object*) is created from blueprint it will occupy space and will be a unique entity by itself. Thus, variables in different

object will occupy different memory space and the values can be different in different instance of objects.

These are known dynamic objects as the objects are created dynamically during runtime as and when needed. Objects can be deleted once it is no longer needed, freeing up memory.

Static

Static objects are the one which cannot have different instances. There will be only one instance globally. For these types of objects, the object creation is done during compilation only and thus it will live in the memory throughout the runtime. This is why it is of static nature.

The exact mechanism for dynamic and static objects are out of the scope of this article. The basic concept related to static and dynamic objects should be kept in mind as it is very crucial in writing test benches.

SV components like module, interface is all static in nature, but classes are dynamic in nature and thus, we must be careful about how we use these components in our code.

Conclusion

In this article we mainly focused on understanding the theoretical concept of OOPs. In next few articles we will see how to apply these concepts in System Verilog. We will also see why these concepts are really needed to make a scalable and reusable test bench.
OOP is one of the most common types of programming language used in modern days mainly because the concepts can be easily related to real life objects.

A static variable exists for the whole simulation; an automatic variable exists only for the lifetime of the task, function or block - they are created when the task, function or block is entered and destroyed when it is left.

Inheritance in SystemVerilog

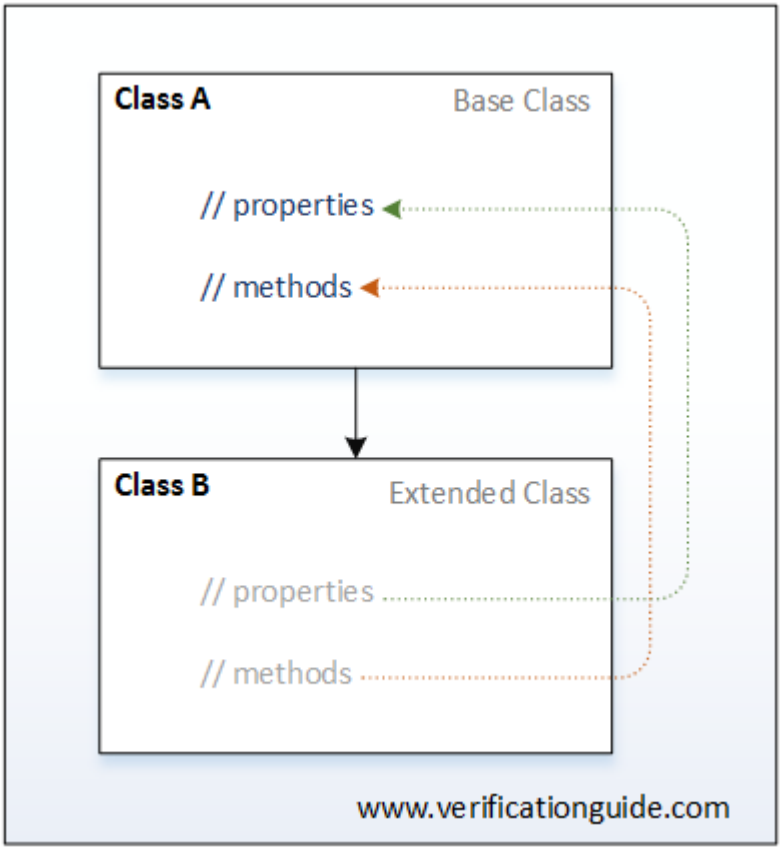
SystemVerilog Inheritance

Table of Contents

- SystemVerilog Inheritance
    - Inheritance Terminology
        - Parent Class
        - Child Class
    - Inheritance Example

SystemVerilog Inheritance. Inheritance is an OOP concept that allows the user to create classes that are built upon existing classes. The new class will be with new properties and methods along with having access to all the properties and methods of the original class. Inheritance is about inheriting base class members to the extended class.

- New classes can be created based on existing classes, this is referred to as class inheritance
- A derived class by default inherits the properties and methods of its parent class
- An inherited class is called a subclass of its parent class
- A derived class may add new properties and methods, or modify the inherited properties and methods
- Inheritance allows re-usability. i.e. derived class by default includes the properties and methods, which is ready to use
- If the class is derived from a derived class, then it is referred to as Multilevel inheritance



SystemVerilog Inheritance

Inheritance Terminology

Parent Class

- It's an existing class;
- The class whose features are inherited
- The parent class is also known as a base class, superclass

Child Class

- It's an extended class;
- The class that inherits the other class is known as subclass
- The child class is also known as an extended class, derived class, subclass

Inheritance Example

Parent class properties are accessed using child class handle, i.e child class will have (*inherit*) parent class properties and methods.

In the below example,
parent_class is base class and child_class is written by extending the parent_class.
so child_class is derived from a base class, and it inherits the properties of the parent class.

Though the addr is not declared in child_class, it is accessible. because it is inherited from the parent class.

```
class parent_class;
  bit [31:0] addr;
endclass


class child_class extends parent_class;
  bit [31:0] data;
endclass


module inheritence;
  initial begin
    child_class c = new();
    c.addr = 10;
```

```
  c.data = 20;
  $display("Value of addr = %0d data = %0d",c.addr,c.data);
 end
endmodule
```

Simulator Output

```
Value of addr = 10 data = 20
```

SystemVerilog Polymorphism

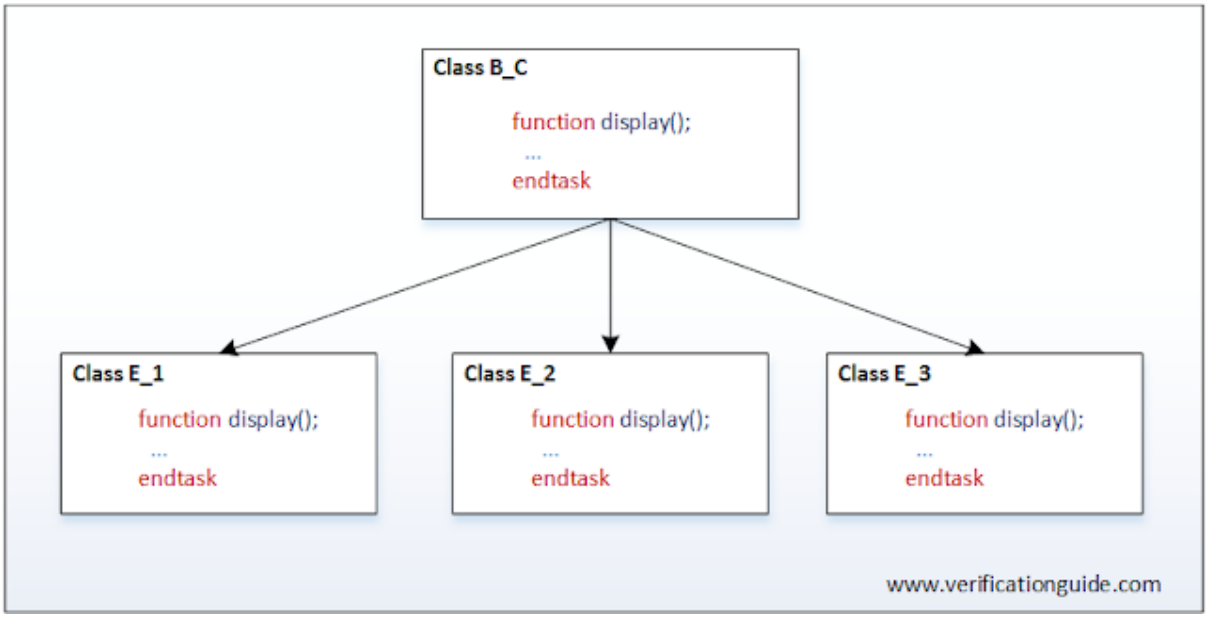
Polymorphism in SystemVerilog

Polymorphism means many forms. Polymorphism in SystemVerilog provides an ability to an object to take on many forms.

Method handle of super-class can be made to refer to the subclass method, this allows polymorphism or different forms of the same method.

How, many forms of a method can be made by referring to the subclass method?
will see with an example,

Polymorphism example



SystemVerilog Polymorphism

Let's write the base_class with a method display();

```
class base_class;
 virtual function void display();
```

```
    $display("Inside base class");
  endfunction
endclass
```

Writing three extended classes of base_class, with display method overridden in it.

```
class ext_class_1 extends base_class;
  function void display();
    $display("Inside extended class 1");
  endfunction
endclass

class ext_class_2 extends base_class;
  function void display();
    $display("Inside extended class 2");
  endfunction
endclass

class ext_class_3 extends base_class;
  function void display();
    $display("Inside extended class 3");
  endfunction
endclass
```

Create an object of each extended class,

```
ext_class_1 ec_1 = new();
ext_class_2 ec_2 = new();
ext_class_3 ec_3 = new();
```

Declare an array of a base class,

```
base_class b_c[3];
```

Assign extend class handles to base class handles.

```
b_c[0] = ec_1;

b_c[1] = ec_2;

b_c[2] = ec_3;
```

Call the display method using base_class handle,

```
b_c[0].display();

b_c[1].display();

b_c[2].display();
```

In the above method calls,
Though all the methods are called using base_class handle, different methods are getting called. this shows the many forms of the same method, this is called *polymorphism*.

Complete code:

```
// base class

class base_class;

  virtual function void display();

    $display("Inside base class");

  endfunction

endclass


// extended class 1

class ext_class_1 extends base_class;

  function void display();

    $display("Inside extended class 1");

  endfunction

endclass


// extended class 2

class ext_class_2 extends base_class;

  function void display();

    $display("Inside extended class 2");
```

```
  endfunction
endclass

// extended class 3
class ext_class_3 extends base_class;
  function void display();
    $display("Inside extended class 3");
  endfunction
endclass

// module
module class_polymorphism;

  initial begin

    //declare and create extended class
    ext_class_1 ec_1 = new();
    ext_class_2 ec_2 = new();
    ext_class_3 ec_3 = new();

    //base class handle
    base_class b_c[3];

    //assigning extended class to base class
    b_c[0] = ec_1;
    b_c[1] = ec_2;
    b_c[2] = ec_3;
```

```
  //accessing extended class methods using base class handle
  b_c[0].display();
  b_c[1].display();
  b_c[2].display();
 end


endmodule
```

Simulator Output

Inside extended class 1

Inside extended class 2

Inside extended class 3

Static and dynamic casting?
SystemVerilog Casting

Casting

Table of Contents

- Casting
  - Static casting
    - Static casting example
  - Dynamic casting
  - Dynamic Casting examples
    - assigning child class handle to parent class handle
    - assigning parent class handle to child class handle
    - assigning parent class handle to child class handle
    - Use of $cast or casting

In Manufacturing, Casting is a process in which liquid metal is converted into the desired object. Similarly, SystemVerilog casting means the conversion of one data type to another datatype. During value or variable assignment to a variable, it is required to assign value or variable of the same data type. Some situations need assignment of different data type, in

such situations, it is necessary to convert data type and assign. Otherwise, the assignment of different data type results in a compilation error. The method of data type conversion is called *casting*.
In systemVerilog, there are two types of casting,

- Static casting
- Dynamic casting

Static casting

- SystemVerilog static casting is not applicable to OOP
- Static casting converts one data type to another compatible data types (example string to int)
- As the name says 'Static', the conversion data type is fixed
- Static casting will be checked during compilation, so there won't be run-time checking and error reporting
- Casting is applicable to value, variable or to an expression
- A data type can be changed by using a cast (') operation
- The vale/variable/expression to be cast must be enclosed in parentheses or within concatenation or replication braces

Static casting example

In the below example,
the real type is converted into int type. i.e multiplication of two real numbers results in real value, the result is converted into int and then assigned to a variable of int type.
Note: the casting is applied to expression here.

```
module casting;


 real r_a;

 int  i_a;


 initial begin


  r_a = (2.1 * 3.2);
```

```
  //real to integer conversion
  i_a = int'(2.1 * 3.2); //or i_a = int'(r_a);

  $display("real value is %f",r_a);
  $display("int  value is %d",i_a);
 end
endmodule
```

Simulator Output

```
real value is 6.720000
int  value is          7
```

Dynamic casting

- Dynamic casting is used to, safely cast a super-class pointer (reference) into a subclass pointer (reference) in a class hierarchy
- Dynamic casting will be checked during run time, an attempt to cast an object to an incompatible object will result in a run-time error
- Dynamic casting is done using the $cast(destination, source) method
- With $cast compatibility of the assignment will not be checked during compile time, it will be checked during run-time

Let's see how we can use the casting,

It is always legal to assign a child class variable to a variable of a class higher in the inheritance tree (parent class).

```
parent_class = child_class; //allowed
```

It is never legal to directly assign a super-class (parent class) variable to a variable of one of its subclasses (child class).

```
child_class  = parent_class; //not-allowed
```

However, it is legal to assign a super-class (parent class) handle to a subclass (child class) variable **if** the super-class (parent class) handle refers to an object of the given subclass(child class).

```
parent_class = child_class ;
child_class  = parent_class; //allowed because parent_class is pointing to child_class.
```

Though parent_class is pointing to the child_class, we will get a *compilation error* saying its not compatible type for the assignment.

This we can over come by make use of $cast method, i.e,

$cast(child_class,parent_class);

Why is it called as dynamic casting?

In the above parent class assignment with child class example. type of parent class is changing dynamically i.e on declaration it is of parent class type, on child class assignment it is of child class type.
Parent class handle during $cast execution is considered for the assignment, so it referred to as dynamic casting.

Dynamic Casting examples

assigning child class handle to parent class handle

```
class parent_class;
  bit [31:0] addr;
  function display();
    $display("Addr = %0d",addr);
  endfunction
endclass


class child_class extends parent_class;
  bit [31:0] data;
  function display();
    super.display();
    $display("Data = %0d",data);
  endfunction
endclass


module inheritence;
```

```
 initial begin
   parent_class p=new();
   child_class  c=new();
   c.addr = 10;
   c.data = 20;
   p = c;       //assigning child class handle to parent class handle
   c.display();
 end
endmodule
```

Simulator Output

```
Addr = 10
Data = 20
```

assigning parent class handle to child class handle

This assignment is invalid, it leads to a compilation error.

```
class parent_class;
 bit [31:0] addr;
 function display();
   $display("Addr = %0d",addr);
 endfunction
endclass

class child_class extends parent_class;
 bit [31:0] data;

 function display();
   super.display();
   $display("Data = %0d",data);
 endfunction
```

```
endclass


module inheritence;
  initial begin
    parent_class p=new();
    child_class  c=new();
    c.addr = 10;
    c.data = 20;
    c = p;       //assigning child class handle to parent class handle
    c.display();
  end
endmodule
```

Simulator Output

"c = p;"

Expression 'p' on rhs is not a class or a compatible class and hence cannot

be assigned to a class handle on lhs.

Please make sure that the lhs and rhs expressions are compatible.

assigning parent class handle to child class handle

assigning parent class handle (which is pointing to child class handle) to child class handle, leads to compile error.

```
class parent_class;
  bit [31:0] addr;


  function display();
    $display("Addr = %0d",addr);
  endfunction
endclass
```

```
class child_class extends parent_class;
  bit [31:0] data;
  function display();
    super.display();
    $display("Data = %0d",data);
  endfunction
endclass

module inheritence;
  initial begin
    parent_class p;
    child_class  c=new();
    child_class  c1;
    c.addr = 10;
    c.data = 20;
    p  = c;       //p is pointing to child class handle c.
    c1 = p;       //type check fails during compile time.
    c1.display();
  end
endmodule
```

Simulator Output

```
"c1 = p;"
Expression 'p' on rhs is not a class or a compatible class and hence cannot
be assigned to a class handle on lhs.
Please make sure that the lhs and rhs expressions are compatible.
```

Use of $cast or casting

In the above example, assigning parent class handle (which is pointing to child class handle) to child class handle is valid but compilation error is observed.

During the compile time, as the handle of p is of parent class type which leads to compile error.

With the use of $cast(), type check during compile time can be skipped.

```
class parent_class;
  bit [31:0] addr;

  function display();
    $display("Addr = %0d",addr);
  endfunction
endclass

class child_class extends parent_class;
  bit [31:0] data;

  function display();
    super.display();
    $display("Data = %0d",data);
  endfunction
endclass

module inheritence;
  initial begin
    parent_class p;
    child_class  c=new();
    child_class  c1;
    c.addr = 10;
    c.data = 20;
```

```
  p = c;        //p is pointing to child class handle c.
  $cast(c1,p);  //with the use of $cast, type chek will occur during runtime


  c1.display();
 end
endmodule
```

Simulator Output

```
Addr = 10
Data = 20
```

Systemverilog Data Hiding and Encapsulation

Data hiding and Encapsulation

Table of Contents

- Data hiding and Encapsulation
    - Access Control
        - local class members
        - Local Class members examples
        - Protected class members
        - Protected Class members examples

The technique of hiding the data within the class and making it available only through the methods, is known as encapsulation.
Because it seals the data (and internal methods) safely inside the “capsule” of the class, where it can be accessed only by trusted users (i.e., by the methods of the class).

Access Control

By default all the members and methods of a class are accessible from anywhere using the object handle, sometimes this could corrupt the class members values, which should not be touched at all.
Access control rules that restrict the members of a class from being used outside the class, this is achieved by prefixing the class members with the keywords,

- local

- protected

local class members

External access to the class members can be avoided by declaring members as local.
Any violation could result in a compilation error.

Syntax:

local integer x;

Local Class members examples

Accessing local variable outside the class (Not allowed)

In below example,
The local variable declared inside the class is trying to access from outside the class by using object handle.
As the variable is declared as local, which leads to a compilation error.

```
class parent_class;
  local bit [31:0] tmp_addr;

  function new(bit [31:0] r_addr);
    tmp_addr = r_addr + 10;
  endfunction

  function display();
    $display("tmp_addr = %0d",tmp_addr);
  endfunction
endclass


//   module
module encapsulation;
  initial begin
    parent_class p_c = new(5);
```

```
    p_c.tmp_addr = 20;  //Accessing local variable outside the class
    p_c.display();
  end
endmodule
```

Simulator Output

```
Error- Illegal class variable access
testbench.sv,
Local member 'tmp_addr' of class 'parent_class' is not visible to scope
'encapsulation'.
```

Accessing local variable within the class (Allowed)

In the below example, The local variable declared inside the class is being accessed inside the class. as it is allowed, no compilation error is observed.

```
class parent_class;
  local bit [31:0] tmp_addr;

  function new(bit [31:0] r_addr);
    tmp_addr = r_addr + 10;
  endfunction

  function display();
    $display("tmp_addr = %0d",tmp_addr);
  endfunction
endclass

//   module
module encapsulation;
  initial begin
```

```
  parent_class p_c = new(5);
  p_c.display();
 end
endmodule
```

Simulator Output

```
Addr = 15
```

Protected class members

In some use cases, it is required to access the class members only by the derived class's, this can be done by prefixing the class members with the protected keyword.
Any violation could result in a compilation error.

Syntax:

```
protected integer x;
```

Protected Class members examples

Accessing a protected variable outside the class (Not allowed)

In the below example, The protected variable declared inside the class is trying to access from outside the class by using object handle.
As the variable is declared as protected, which leads to a compilation error.

```
class parent_class;
 protected bit [31:0] tmp_addr;

 function new(bit [31:0] r_addr);
  tmp_addr = r_addr + 10;
 endfunction

 function display();
  $display("tmp_addr = %0d",tmp_addr);
 endfunction
endclass
```

```
class child_class extends parent_class;
  function new(bit [31:0] r_addr);
    super.new(r_addr);
  endfunction

  function void incr_addr();
    tmp_addr++;
  endfunction
endclass

//   module
module encapsulation;
  initial begin
    parent_class p_c = new(5);
    child_class  c_c = new(10);

    // variable declared as protected cannot be accessed outside the class
    p_c.tmp_addr = 10;
    p_c.display();

    c_c.incr_addr();  //Accessing protected variable in extended class
    c_c.display();
  end
endmodule
```

Simulator Output

```
Error- Illegal class variable access
testbench.sv,
Protected member 'tmp_addr' of class 'parent_class' is not visible to scope
```

'encapsulation'.

Accessing a protected variable in the extended class (allowed)

In the below example, The protected variable declared inside the class is being accessed inside the extended class. as it is allowed, no compilation error is observed.

```
class parent_class;
  protected bit [31:0] tmp_addr;

  function new(bit [31:0] r_addr);
    tmp_addr = r_addr + 10;
  endfunction

  function display();
    $display("tmp_addr = %0d",tmp_addr);
  endfunction
endclass

class child_class extends parent_class;
  function new(bit [31:0] r_addr);
    super.new(r_addr);
  endfunction

  function void incr_addr();
    tmp_addr++;
  endfunction
endclass

//   module
module encapsulation;
```

```
 initial begin
   child_class  c_c = new(10);


   c_c.incr_addr();  //Accessing protected variable in extended class
   c_c.display();
 end
endmodule
```

Simulator Output

Addr = 21

glitch and clock generation vip assertion?

A few comments:

1. I fail to undestand your "[*1:$]" in the following assertion:

```
property glitch_p;
realtime first_change;
realtime duration = 10;
@(a)
// detecting every 2 changes duration
(1, first_change = $realtime) |=> (($realtime - first_change) >= duration))[*1:$]; // ??
endproperty
```

It states that at every edge of "a" you start a new thread, and in that thread, from then on, and and forever, every edge of "a" must be >= duration. That creates multiple, unneeded threads. I strongly suggest that you read my paper
https://verificationacademy.com/forums/systemverilog/paper-understanding-sva-engine-simple-alternate-solutions
Abstract: Understanding the engine behind SVA provides not only a better appreciation and limitations of SVA, but in some situations provide features that cannot be simply implemented with the current definition of SVA. This paper first explains, by example, how a relatively simple assertion example can be written without SVA with the use of SystemVerilog tasks; this provides the basis for understanding the concepts of multithreading and exit of threads upon a condition, such as an error in the assertion. The paper then provides examples that uses computational variables within threads; those

variables can cause, in some cases, errors in SVA. The strictly emulation model with tasks solves this issue.
2. On realtime, SystemVerilog is laxed and allows to specify a time in integer. From my roots in VHDL, and I prefer to specify a realtime variable with a real number
3. Add timeunits

Below is my corrected code. I believe that this is more of what you are looking for. DO NOT use the [*1:$], it serves no purpose, except overload your simulator.

```
import uvm_pkg::*; `include "uvm_macros.svh"

module top;

        timeunit 1ns; timeprecision 100ps;

        bit clk, a, b, signal;

        default clocking @(posedge clk); endclocking

        initial forever #10 clk=!clk;

        realtime duration=45.0ns;


 property glitch_p;

   realtime first_change;

  // realtime duration = 10;

  @(signal)  // pos and neg edge

    // detecting every 2 changes duration

   (1, first_change = $realtime) |=> (($realtime - first_change) >= duration); // [*1:$];

 endproperty

 ap_glitch_p: assert property(glitch_p);


 always_ff  @(posedge clk)  begin

end


 initial begin
```

```
    repeat(200) begin
      @(posedge clk);
      if (!randomize(signal)  with
         { signal dist {1'b1:=1, 1'b0:=3};
           b dist {1'b1:=1, 1'b0:=2};

         }) `uvm_error("MYERR", "This is a randomize error")
      end
      $stop;
   end
endmodule
```

Lets say there is a signal a . When the signal goes high, it has to stay high at least for three positive clock edges.

We can write the property as

property p;

@(posedge clk) $rose(a) -> a[*3];

endproperty

The property fails for the case below.

clk _ _ _ | = = = | _ _ _ | = = = | _ _ _ | = = = | _ _ _ | = = = |

a _ _ | = = = | _ _ | = = = = = = = = = = = = = = = = = =

This is not in accordance with the specification where a is going low in the middle but will be pulled high by the next posedge and hence the above assertion wont catch this.

Can anyone tell if there is any way to write assertion to catch this bug?

Tudor right, that in most of cases it does not matter what happening between clock edges. But at CDC, or asynchronous design we must verify that the design is glitch free. There is a inside-out way to do this. (I found this solution at http://www.verificationguild.com/modules.php?name=Forums&file=viewtopic&p=20045)

```
property detect_glitch;
   time leading;                // declare the last edge occurence variable
   @(glitch)                    // At every change of glitch signal
      //The following line saves the current time and check
      // the delay from the previous edge.
      (1, leading = $time) |=> (($time - leading) >= duration);
endproperty : detect_glitch


DETECT_GLITCH : assert property (detect_glitch)
else
   $display ("ERROR");
```

ARM, what it does and what type of chips they have?

ARM (Advanced RISC Machine) designs and licenses processor cores, not chip manufacturing. It's a leading provider of processor IP, with its technology used in billions of devices. ARM focuses on power efficiency and performance, enabling devices from sensors to supercomputers. They offer various cores, including Cortex-A, Cortex-M, Cortex-R, Neoverse, and Ethos.

Here's a more detailed breakdown:

What ARM Does:

- **Designs Processor Cores:**

ARM develops and licenses the instruction sets (the fundamental language processors understand) for chips.

- **Doesn't Manufacture Chips:**

ARM is not a chip manufacturer like Intel or AMD. They license their IP to companies that then produce chips.

- **Focus on Power Efficiency and Performance:**

ARM designs cores optimized for both power consumption and computational speed.

- **Wide Range of Devices:**

ARM's technology powers everything from small sensors and wearables to smartphones, laptops, servers, and even supercomputers.

- **Large Ecosystem:**

ARM has a large ecosystem of partners who develop and manufacture chips based on its technology.

Types of Chips with ARM Cores:

- **Cortex-A:** Designed for **general-purpose processing** in smartphones, tablets, and some laptops.
- **Cortex-M:** Optimized for **low-power applications** like microcontrollers in embedded systems, IoT devices, and smartwatches.
- **Cortex-R:** Designed for **real-time applications** in industrial automation, automotive, and other systems where speed and reliability are critical.
- **Neoverse:** High-performance cores for **servers and cloud computing**.
- **Ethos:** Accelerators for **AI workloads**.
- **Examples:** Apple's M1 chip, Qualcomm's Snapdragon chips, and many chips used in smartphones, tablets, and other devices.

Key Benefits of ARM Technology:

- **Power Efficiency:** ARM-based chips are known for their low power consumption, making them ideal for battery-powered devices.
- **Performance:** ARM cores offer a good balance of performance and power efficiency.
- **Scalability:** ARM's architecture can be used in a wide variety of applications, from low-power sensors to high-performance servers.

The Apple M1 chip is a System-on-a-Chip (SoC) that Apple designed specifically for its Mac computers. It's based on the ARM architecture, which is different from the Intel x86 architecture previously used in Macs. The M1 chip integrates various components like the CPU, GPU, and Neural Engine into a single chip, resulting in improved performance and efficiency.

Here's a more detailed breakdown:

- **ARM Architecture:**

The M1 chip uses the ARM (Advanced RISC Machine) instruction set architecture, a RISC (Reduced Instruction Set Computing) design. This architecture is known for its efficiency and

power-saving capabilities, making it well-suited for mobile devices and, as seen with the M1, high-performance laptops.

- **SoC Design:**

The M1 chip is a System-on-a-Chip (SoC), meaning it integrates multiple components, such as the CPU, GPU, Neural Engine, and memory controllers, onto a single chip. This integration leads to improved performance and efficiency compared to traditional systems where these components are separate.

- **Performance and Efficiency:**

The M1 chip provides significant performance gains over previous Intel-based Macs while also offering improved battery life and efficiency. This is due to the combination of its ARM architecture, efficient design, and unified memory architecture.

- **Software Compatibility:**

The M1 chip uses Rosetta 2, a dynamic binary translation layer, to allow Macs with the M1 chip to run x86-64 applications. This ensures compatibility with the vast majority of existing Mac software.

The M1 has four high-performance "Firestorm" and four energy-efficient "Icestorm" cores, first seen on the A14 Bionic. It has a hybrid configuration similar to ARM big.LITTLE and Intel's Lakefield processors. This combination allows power-use optimizations not possible with previous Apple–Intel architecture devices.

sample interview questions for Verilog, system Verilog, and uvm previously asked in ARM interviews?

SoC Verification?

SoC verification is the process of ensuring a System-on-Chip (SoC) design functions correctly and meets its specifications. It involves testing the overall system functionality, including how individual components interact and interface with each other. This critical phase in SoC development often accounts for a significant portion of the overall design cycle.

Here's a more detailed explanation:

What is a System-on-Chip (SoC)?

- SoCs are complex integrated circuits that integrate multiple functions (like processors, memory, peripherals, etc.) onto a single chip.
- They are common in mobile devices, embedded systems, and other applications where integration and performance are crucial.

Why is SoC Verification Important?

- Ensures the SoC meets its functional requirements and performs as intended.
- Identifies and resolves potential design flaws early in the development process.
- Helps to ensure the reliability and performance of the final chip.
- Verification often consumes a large part of the development cycle, highlighting its importance.

Key Aspects of SoC Verification:

- **Functional Verification:**

Testing the behavior of the SoC under various conditions and scenarios, including its interactions with external devices.

- **Interface Verification:**

Ensuring that different components and modules within the SoC communicate correctly.

- **Hardware-Software Co-verification:**

Verifying the interactions between the hardware and software components of the SoC.

- **Emulation and Simulation:**

Using emulators and simulators to model the SoC and its environment for testing purposes.

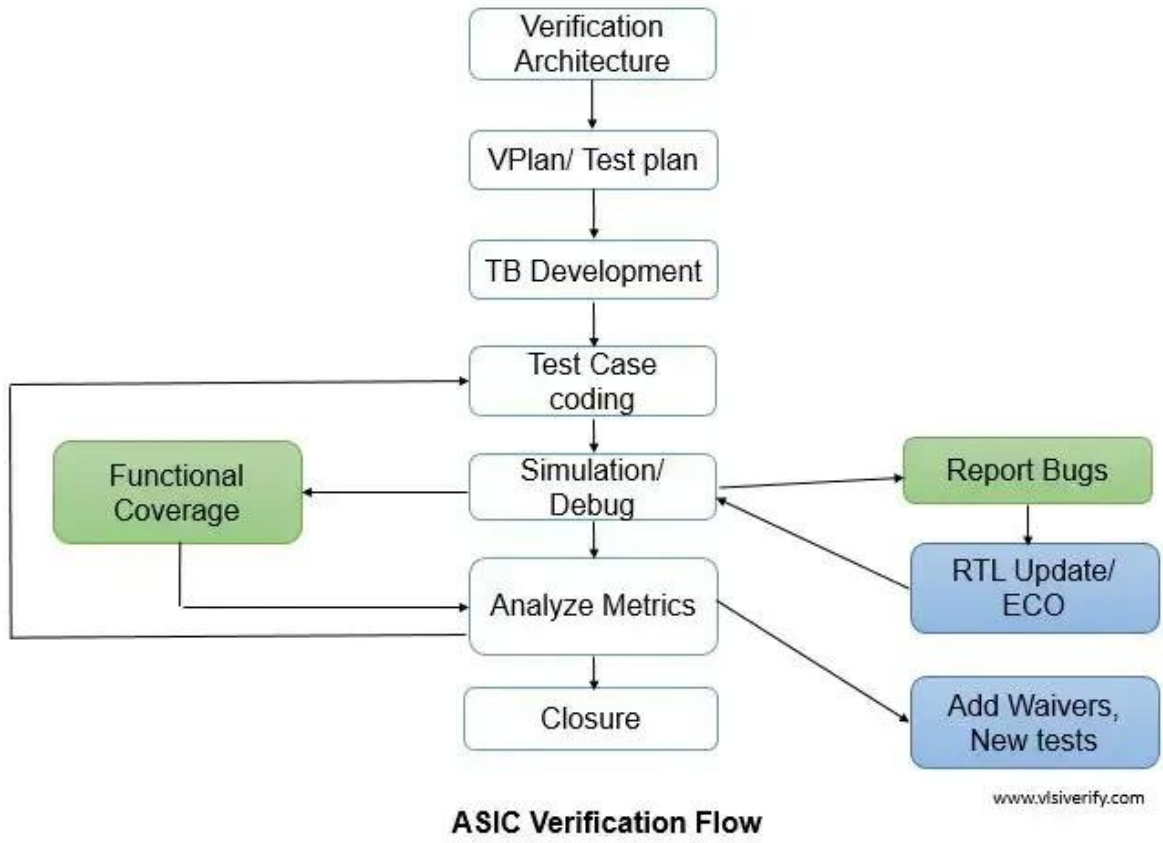
- **Formal Verification:**

Using mathematical methods to formally prove the correctness of the design.

In essence, SoC verification aims to:

- Verify that the SoC functions correctly as a whole.
- Ensure that the different components and subsystems interact as designed.
- Validate that the SoC meets the overall system requirements and specifications.

ASIC Verification Flow



ASIC Verification Flow

The functional verification process allows verification engineers in finding bugs, checking for RTL correctness based on the design specification. The first step in the verification process is to prepare a verification plan which is tightly coupled with the design specification that involves what all features need to be tested and techniques used to verify the design under test (DUT) such as scoreboard check planning, assertions, and functional coverage writing, etc

Verification Architecture

In the verification architectural phase, engineers decide what all verification components are required.

Verification Plan/ Testplan

The verification plan includes a test plan(list of test cases that target design features), functional coverage planning, module/block assignments to the verification engineers, checker, and assertion planning. The verification plan also involves planning for how verification components can be reused at system/ SOC level verification.

Testbench Development

As a part of testbench development, verification engineers develop testbench components, interface connections with the DUT, VIP integration with a testbench, inter-component connections within testbench (like monitor to scoreboard connection), etc.

Testcase Coding

A constraint-based random or dedicated test case is written for single or multiple features in the design. A test case also kicks off UVM-based sequences to generate required scenarios.

Simulation/ Debug

In this phase, engineers validate whether a specific feature is targetted or not, If not, again test case is modified to target the feature. With the help of a checker/ scoreboard, the error is reported if the desired design does not behave as expected. Using waveform analysis or log prints, the design or verification environment is judged and a bug is reported to the design team if it comes out to be a design issue otherwise, simulation is re-run after fixing the verification component.

Analyze Metrics

Assertions, code, and functional coverage are common metrics that are used as analysis metrics before we close the verification of the design.

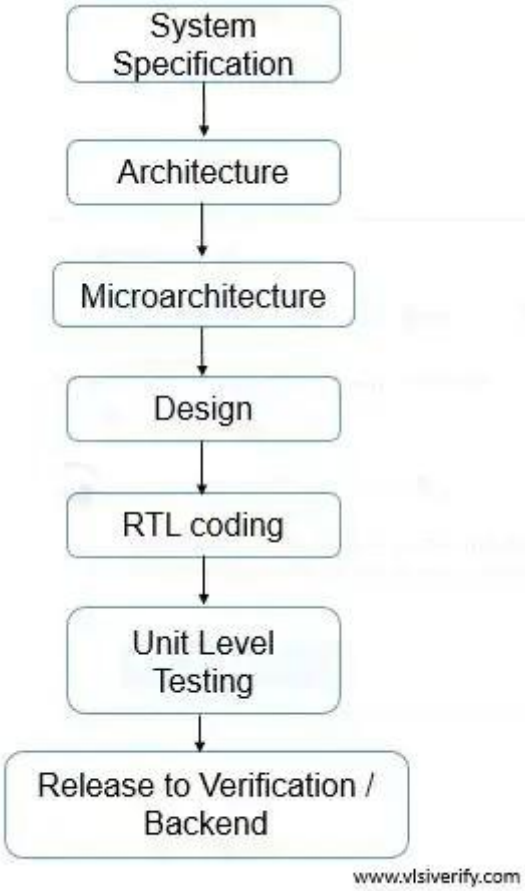
IP and SOC Verification

Due to the increasing complexity of designs, functional verification is done in multiple ways like block/IP verification and System/SoC (System on Chip) verification. Let's see how they are different from each other.

Block/ IP Verification	**System/ SoC Verification**
The focus is on verifying the functionality of individual modules/IPs.	The focus is on verifying the integration of various design IPs in the chip.
The verification involves driving all possible combinations for design testing	The verification involves system-level scenario testing along with software configuration.
All testbench components like the driver, monitor, scoreboard, agent, etc are implemented.	Along with testbench components, co environments (C-based test/ re-using IP level components) are used.

Test cases are mostly limited to dedicated module-level verification	Sometimes, test cases are also used at ATE(Automatic Test Equipment) as part of chip bring up and a few tests for gate-level simulations (GLS)
Detailed coverage is measured and analyzed	Pin level toggle coverage is measured and analyzed. It also includes system-level interrupt functionality verification.
Build/ compile time is less as compared to SoC level verification.	Build/ compile time is much more.

ASIC Design Flow



ASIC Design Flow

System Specification

At first, based on market trends and future requirements what has to be designed is being analyzed and captured as system specifications.

Architecture

As per system specifications, all features that need to be implemented in hardware and firmware/software are decided in this phase. An architectural team decides the data and control flows and can have different approaches to implement the hardware blocks and offload some work to the firmware. An optimized approach is selected considering various trade-offs like clock frequency, speed, power consumption, firmware execution time, etc.

Microarchitecture

In this phase, necessary hardware resources like design blocks, inter-block interface, and their communication protocols within the system are decided, basically it is low-level design discussions.

Design

As a part of this phase, how features will be implemented in the hardware, register configurations, and what all different design components will be used like a finite state machine (FSM), MUX, De-Mux, counters, registers, etc.

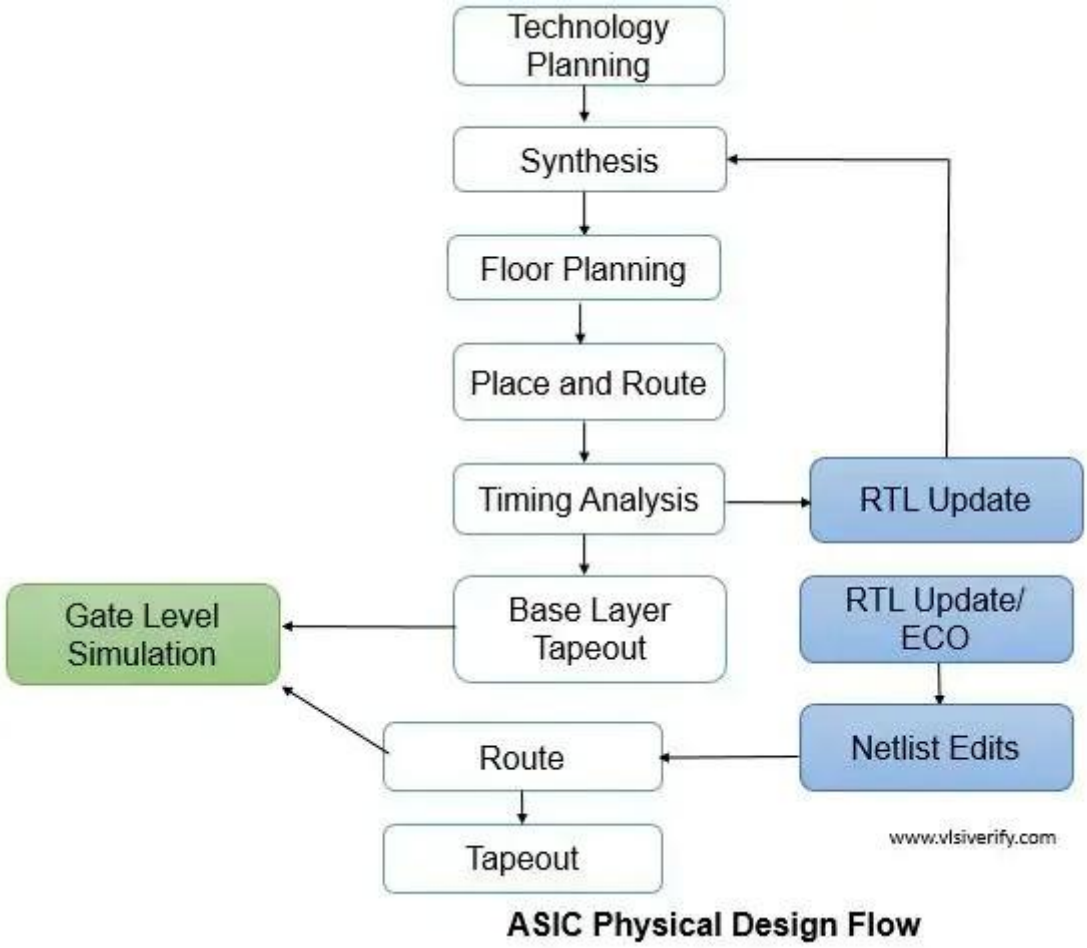
RTL Coding

The actual feature is implemented using hardware description languages (HDL) like Verilog or VHDL. It is necessary to write synthesizable code so that the synthesis tool can properly convert HDL implemented code into the actual gate-level design.

Unit Level Testing

Before releasing the design code to other teams like verification or backend, unit-level testing is performed by the designer to make sure at least the basic functionality of the design works as expected. This helps to capture and resolve basic bugs.

ASIC Physical Design Flow



ASIC Physical Design Flow

Technology Planning

Technology planning is an important step in the physical design flow. The technology libraries are provided by foundries that have information about the type of silicon wafer, standard cells, DRC(Design Rule Check), etc.

Synthesis

The synthesis process does conversion from synthesizable RTL code (higher level of abstraction) into gate-level netlist(lower level abstraction) of specific technology. In this phase, logic optimization, DFT logic insertion, logic equivalence between RTL and gate-level netlist is maintained.

Floorplanning

The floorplanning involves planning the placement of various modules on a chip and net routing in order to reduce congestion, and noise issues. It is one of the most important steps in backend or physical design flow that directly affects chip power consumption, timing, and reliability. For better floorplanning, you should have design and its data flow understanding, IO interactions within blocks, and outside blocks like analog IPs.

Place and Route

Actual standard cell placement in a chip boundary considering optimal consideration taken during floorplanning. The placement efforts are done by the PnR tool with minimal congestion, best timing, and preferences given by the user.

Timing Analysis

Static Timing Analysis (STA) checks for all possible paths in the design for any timing violations. Set up and hold timing violation is being checked as part of static timing analysis. STA is much faster as compared to dynamic timing analysis which is used to validate full design behavior for a given input vector stimulus because as the number of stimulus vectors increases, simulation time also increases in dynamic timing analysis. In brief, DTA checks for functionality, and STA checks for timing.

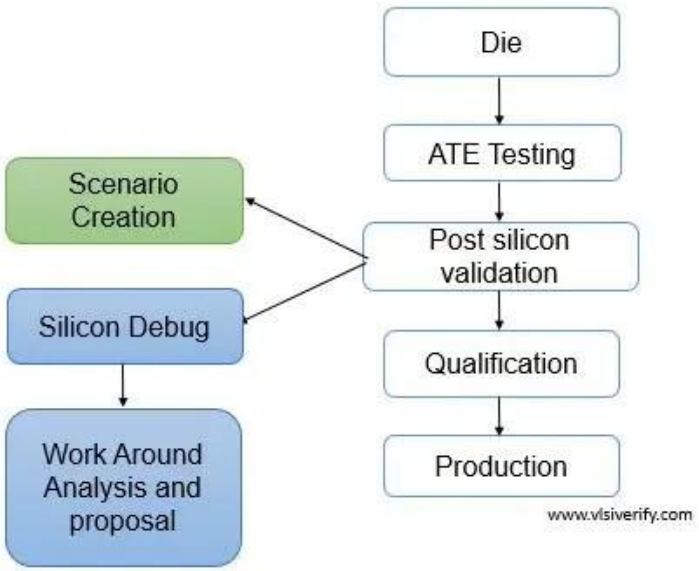
Gate level Simulation (GLS)

RTL simulation is done as functional verification whereas GLS is run when gate-level netlist is ready i.e. RTL simulation is pre-synthesis and GLS is post-synthesis. Usually, GLS is run with unit delay mode and full delay mode. The standard delay file (SDF) contains timing information that is considered as delay mode. Though GLS verification is a dead slow process, it provides benefits like reset and initialization flow check, asynchronous design timing check, X propagation in RTL, etc.

Tapeout

Once physical design verification is done, final design files are sent to the foundry for chip fabrication.

ASIC Post Silicon Flow



ASIC Post Silicon Flow

As a part of post-silicon flow, once the chip/die is ready after fabrication, it is once again tested in a real-time environment before it dispatches to the market. Automatic Test Equipment (ATE) is a device that sends automatic test pattern generation (ATPG) to the device under test (DUT). Thus, post-silicon validation checks for its correctness and performance. ATE testing is much faster as compared to RTL simulation. Once the chip is qualified as per testing norms, it is sent for production to deliver the market need.

DDR protocol?

what type of bugs you found? - system clock frequency of GHz, and 200-400 MHz in FPGA emulation boards, bug in rtl with use of legacy code with extra signals not present in verification platform, interconnects between soc - jtag protocol to flash memory, may be short circuited lines to addresses of flash memory? functionality not matching with specification?

It sounds like you're describing several types of bugs that can arise during FPGA emulation or SoC verification. Here's a breakdown of the categories and specific bug types you may be encountering, based on your input:

---

1. Clock Domain and Frequency Mismatch Bugs

- **System clock frequency (GHz) vs FPGA (200–400 MHz)**:
  Bugs can appear when porting designs from ASIC-targeted RTL to FPGA due to clock domain issues, such as:
    - **Timing violations** (setup/hold failures due to slower FPGA fabric)
    - **Clock domain crossing (CDC) bugs**: improper synchronization between high-speed and low-speed domains
    - **Inconsistent behavior in emulation vs actual silicon**

---

2. Legacy RTL Bugs

- **Extra signals in legacy RTL not present in the verification platform**:
    - **Unconnected inputs or outputs**: May result in undefined behavior or synthesis warnings

    - **Stale signal usage**: Signals no longer driven but still referenced
    - **Dead logic or unreachable states**
    - **Incorrect default values** for signals left uninitialized

---

3. Interconnect/Protocol Issues

- **SoC interconnects (e.g., AXI, AHB)**:
    - **Incorrect addressing or decoding**
    - **Deadlock due to handshake issues**
    - **Protocol violations** (e.g., AXI transaction out-of-order errors)
- **JTAG interface to flash memory**:
    - Possible issues:
        - **Signal contention or short-circuits** (e.g., multiple drivers on same line)
        - **Address line faults** (wrong flash memory region being accessed)
        - **Incorrect JTAG sequence or commands**
        - **Faulty level shifters between SoC and flash memory**

---

4. Functional Mismatches

- **Functionality not matching spec**:
    - Bugs here could be due to:
        - **Incorrect logic implementation**
        - **Wrong parameterization of modules**
        - **Inadequate testbench coverage**
        - **Interface or state machine errors**
        - **Improper reset behavior**

---

5. Potential Electrical Issues in Emulation

- **Short-circuited lines to flash address bus**:
    - Could cause:

- Flash read/write failures
- Unreliable boot or data corruption
- Overheating or board damage in worst cases

Would you like help debugging one specific issue in detail, such as the JTAG-to-flash problem or RTL mismatches? - Yes

Great! Let's narrow it down. Since you mentioned multiple complex issues, could you clarify **which specific problem** you're currently facing or want help debugging first? For example:

- JTAG-to-flash interface not working as expected?
- RTL legacy code introducing unexpected behavior?
- FPGA emulation showing mismatches with ASIC spec?

Let me know the specific issue and a few more details, such as:

- What behavior are you observing?
- Any error messages, waveform mismatches, or failing tests?
- Hardware platform (if applicable)?

This will help me guide you through debugging steps or root cause analysis.

How you verified with cadence vip to check axi3 and ax4?

Verifying AXI3 and AXI4 interfaces using **Cadence VIP (Verification IP)** typically involves integrating the AXI VIP into your UVM testbench and using it either as a **master**, **slave**, or **monitor** to generate or check AXI protocol traffic.

Here's a step-by-step outline of **how AXI3 and AXI4 verification is typically done using Cadence VIP**:

🔧 1. Instantiate AXI VIP in Your UVM Testbench

Cadence provides configurable AXI VIP components (for AXI3, AXI4, or AXI4-Lite).

- Configure the VIP based on protocol version (AXI3 or AXI4).
- Example configuration:

```
axi_cfg = axi_vip_agent_cfg::type_id::create("axi_cfg");
```

axi_cfg.protocol_type = AXI4; // or AXI3

💉 2. Set Up AXI Master or Slave Agent

You use either:

- **AXI Master VIP**: to generate AXI transactions to your DUT
- **AXI Slave VIP**: to receive DUT-generated transactions
- **Monitor**: to passively observe and check protocol correctness

Example instantiation in UVM:

axi_agent = axi_vip_agent::type_id::create("axi_agent");

axi_agent.configure(axi_cfg);

axi_agent.build();

📜 3. Write UVM Sequences

Define stimulus via sequences to generate:

- **Reads/Writes**
- **Burst types (INCR, WRAP, FIXED)**
- **Data widths**
- **ID-based ordering (AXI3-specific)**

Example AXI write sequence:

axi_seq_item = axi_transaction::type_id::create("axi_seq_item");

axi_seq_item.addr = 32'h8000_0000;

axi_seq_item.data = {32'hDEADBEEF};

axi_seq_item.burst_len = 4;

axi_seq_item.cmd = WRITE;

start_item(axi_seq_item);

finish_item(axi_seq_item);

✅ 4. Protocol Checking with Built-in Monitors

Cadence VIP includes **assertions and protocol checks** to catch:

- Handshake violations (AWVALID/AWREADY, etc.)
- Out-of-order or missing responses
- Violations of AXI3/AXI4-specific rules (e.g., ID ordering in AXI3, or single ID queueing in AXI4)

You typically enable assertions like this:

axi_cfg.enable_protocol_checkers = 1;

🔁 5. Compare DUT Behavior to VIP Monitored Transactions

Use:

- Scoreboards
- Functional coverage models
- Protocol log comparison (optional Cadence logs)

You can also use uvm_error/uvm_warning reports generated by VIP to track mismatches.

🧭 6. Switch Between AXI3 and AXI4

Cadence AXI VIP supports both. Key differences you must account for:

- **AXI3**: Multiple outstanding transactions per ID
- **AXI4**: Ordering per ID, narrower valid use of response IDs

Ensure:

- Correct configuration at testbench top
- That the DUT complies with the intended protocol (or handles both)

🧰 Tools Used

- **Xcelium** (simulation)
- **JasperGold** (optional formal check)
- **SimVision** (waveform debug)
- **VIP Configuration GUI** (to auto-generate code for VIP components)

Would you like a sample AXI write/read sequence using Cadence VIP, or integration code for UVM testbench? – Yes

Great! Here's a **simple UVM-based example** of how to integrate **Cadence AXI VIP** and perform a basic **AXI write and read sequence**. This assumes you're using the **Cadence UVM VIP for AXI4**, but you can easily adjust for **AXI3** by changing the configuration.

✅ **Setup Overview**

We'll create:

1. VIP Configuration
2. AXI Master Agent
3. Write and Read Sequences
4. Integration in UVM testbench

---

1. AXI VIP Configuration

```
class my_axi_env extends uvm_env;
  axi_vip_agent agent;
  axi_vip_agent_cfg cfg;

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    cfg = axi_vip_agent_cfg::type_id::create("cfg");
    cfg.protocol_type = axi_protocol_e::AXI4;
    cfg.data_width = 32;
    cfg.addr_width = 32;
    cfg.enable_protocol_checkers = 1;

    agent = axi_vip_agent::type_id::create("agent", this);
```

```
    agent.configure(cfg);
    agent.build();
  endfunction
endclass
```

---

2. Write Sequence

```
class axi_write_seq extends uvm_sequence #(axi_transaction);
  `uvm_object_utils(axi_write_seq)

  function new(string name = "axi_write_seq");
    super.new(name);
  endfunction

  task body();
    axi_transaction tx;
    tx = axi_transaction::type_id::create("tx");

    tx.cmd       = axi_transaction::WRITE;
    tx.addr      = 32'h80000000;
    tx.data      = {32'hDEADBEEF};
    tx.burst_len = 1;
    tx.burst_type= axi_burst_type_e::INCR;
    tx.size      = 2; // 4 bytes (2^2)

    start_item(tx);
    finish_item(tx);
  endtask
endclass
```

---

3. Read Sequence

```
class axi_read_seq extends uvm_sequence #(axi_transaction);
  `uvm_object_utils(axi_read_seq)

  function new(string name = "axi_read_seq");
    super.new(name);
  endfunction

  task body();
    axi_transaction tx;
    tx = axi_transaction::type_id::create("tx");

    tx.cmd       = axi_transaction::READ;
    tx.addr      = 32'h80000000;
    tx.burst_len = 1;
    tx.size      = 2;

    start_item(tx);
    finish_item(tx);
  endtask
endclass
```

---

4. Running the Sequences

In your test class:

```
class my_test extends uvm_test;
  `uvm_component_utils(my_test)
```

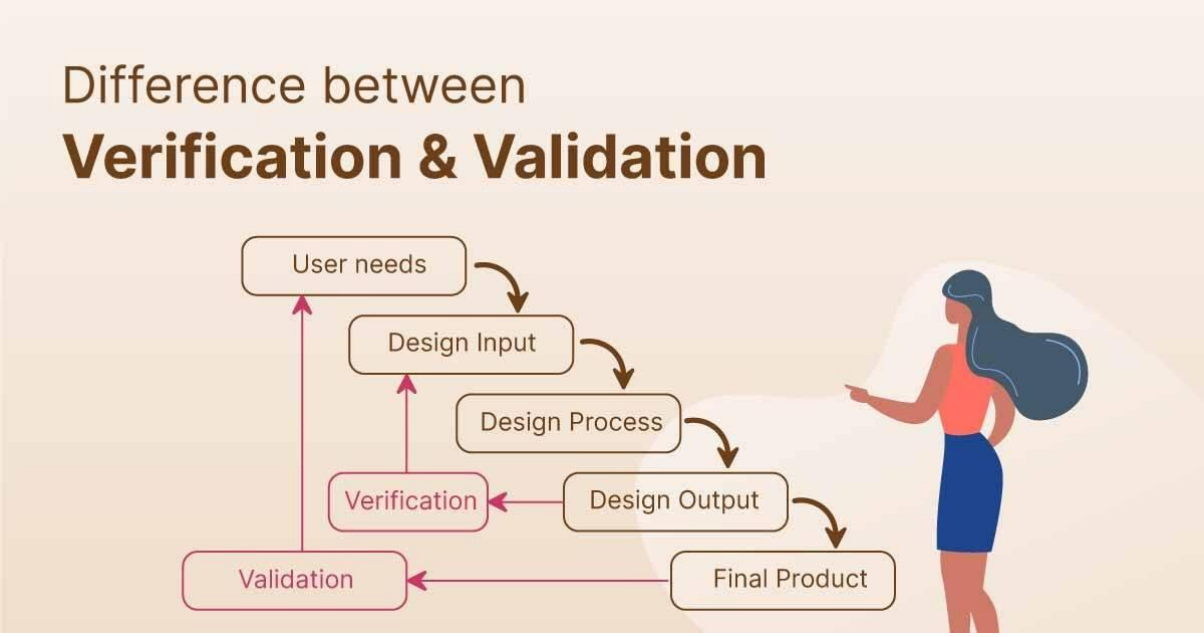
```
  my_axi_env env;

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env = my_axi_env::type_id::create("env", this);
  endfunction

  task run_phase(uvm_phase phase);
    axi_write_seq wr_seq;
    axi_read_seq  rd_seq;

    phase.raise_objection(this);

    wr_seq = axi_write_seq::type_id::create("wr_seq");
    rd_seq = axi_read_seq::type_id::create("rd_seq");

    wr_seq.start(env.agent.sqr);
    rd_seq.start(env.agent.sqr);

    phase.drop_objection(this);
  endtask
endclass
```

🧪 **Results and Debug**

- Use **SimVision** or waveform viewers to observe AXI signals.
- Any violations (e.g., bad handshakes, timing, ordering) will be caught by **Cadence VIP's protocol checker**, and will show up as UVM errors.

Would you like this packaged as a testbench template or want an example adapted for AXI3 instead?

Verification and Validation?



Verification and validation are distinct processes used to ensure the quality of a product, service, or system. Verification confirms that the product is being built correctly, while validation ensures that the correct product is being built to meet user needs.

Here's a more detailed breakdown:

Verification:

- **Focus:** Internal aspects of the product, such as design, code, and documentation.
- **Goal:** To ensure that the product is being built according to specifications and requirements.
- **Methods:** Static analysis, reviews, inspections.
- **Timing:** Occurs throughout the development process, starting from the requirements phase and continuing through the design and coding stages.
- **Question:** "Are we building the product right?".
- **Example:** Verifying that a user registration form includes the required fields (username, email address) based on the design specifications.

Validation:

- **Focus:** External aspects of the product, such as user acceptance and real-world performance.
- **Goal:** To ensure that the product meets user needs and expectations.
- **Methods:** Dynamic testing, user acceptance testing, real-world deployment.
- **Timing:** Typically occurs after the development process is complete, before the product is released.
- **Question:** "Are we building the right product?".
- **Example:** Validating that a user registration form is easy to use, provides clear instructions, and handles errors gracefully.

Key Differences:

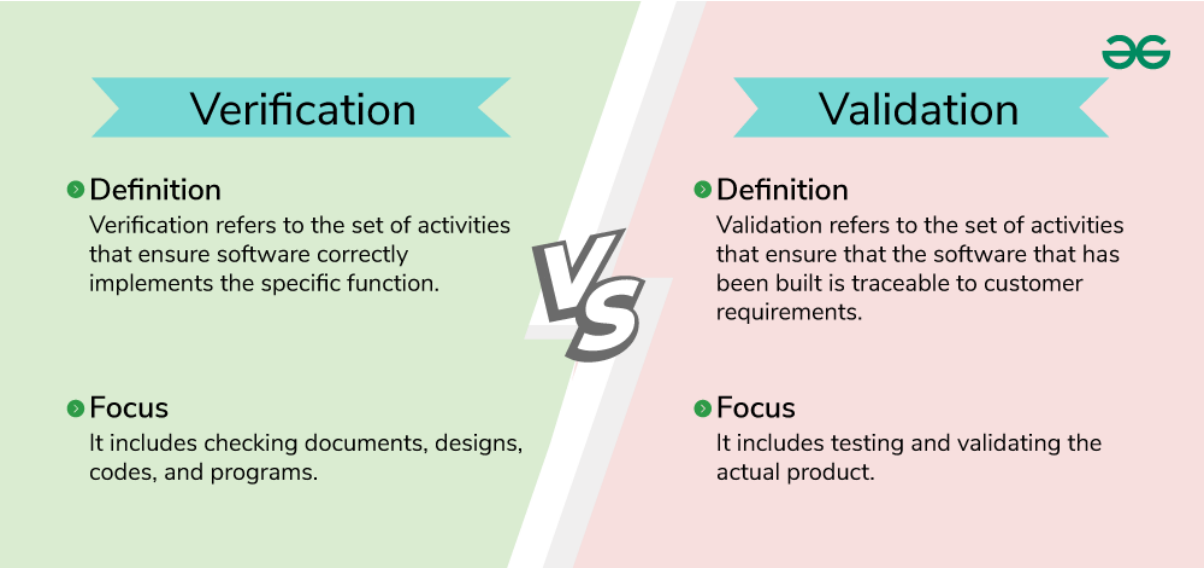
Feature	Verification	Validation
Focus	Internal aspects of the product	External aspects of the product
Purpose	Ensures the product is built correctly	Ensures the product meets user needs
Method	Static analysis, reviews, inspections	Dynamic testing, user acceptance testing
Timing	Throughout the development process	After the development process
Question	"Are we building the product right?"	"Are we building the right product?"

In essence: Verification is about ensuring that the product is being built correctly according to the specifications, while validation is about ensuring that the product meets the user's needs and expectations.

Differences between Verification and Validation

Verification and Validation is the process of investigating whether a software system satisfies specifications and standards and fulfills the required purpose. Verification and Validation both play an important role in developing good software development. Verification helps in examining whether the product is built right according to requirements, while validation helps in examining whether the right product is built to meet user needs.

In this article, we will learning the **Differences between Verification and Validation**.



Differences between Verification and Validation

What is Verification?

Verification is the process of checking that software achieves its goal without any bugs. It is the process to ensure whether the product that is developed is right or not. It verifies whether the developed product fulfills the requirements that we have. Verification is static testing. Verification means **Are we building the product right?**

What is Validation?

Validation is the process of checking whether the **Software Product** is up to the mark or in other words product has high-level requirements. It is the process of checking the validation of the product i.e. it checks what we are developing is the right product. It is validation of the actual and expected products. Validation is dynamic testing. Validation means **Are we building the right product?**

Differences between Verification and Validation

Here is the Differences between Verification and Validation

	Verification	Validation
Definition	Verification refers to the set of activities that ensure software correctly implements the specific function	Validation refers to the set of activities that ensure that the software that has been built is traceable to customer requirements.
Focus	It includes checking documents, designs, codes, and programs.	It includes testing and validating the actual product.
Type of Testing	Verification is the **Static testing**.	Validation is **Dynamic testing.**
Execution	It does *not* include the execution of the code.	It includes the execution of the code.
Methods Used	Methods used in verification are reviews, walkthroughs, inspections and desk-checking.	Methods used in validation are **Black Box Testing**, **White Box Testing** and **Non-Functional testing**.
Purpose	It checks whether the software conforms to specifications or not.	It checks whether the software meets the requirements and expectations of a customer or not.
Bug	It can find the **Bugs** in the early stage of the development.	It can only find the bugs that could not be found by the verification process.

	Verification	Validation
Goal	The goal of verification is application and software architecture and specification.	The goal of validation is an actual product.
Responsibility	Verification is typically performed by the quality assurance (QA) team, focusing on reviewing documents, designs, and code to ensure compliance with specified requirements.	Validation is performed by the testing team, which executes the software in real environments to ensure it meets user expectations and requirements.
Timing	It comes before validation.	It comes after verification.
Human or Computer	It consists of checking of documents/files and is performed by human.	It consists of execution of program and is performed by computer.
Lifecycle	After a valid and complete specification the verification starts.	Validation begins as soon as project starts.
Error Focus	Verification is for prevention of errors.	Validation is for detection of errors.
Another Terminology	Verification is also termed as white box testing or static testing as work product goes through reviews.	Validation can be termed as black box testing or dynamic testing as work product is executed.
Performance	Verification finds about 50 to 60% of the defects.	Validation finds about 20 to 30% of the defects.

	Verification	Validation
Stability	Verification is based on the opinion of reviewer and may change from person to person.	Validation is based on the fact and is often stable.

Real-World Example of Verification vs Validation

- **Verification Example:** Imagine a team is developing a new mobile banking app. During the verification phase, they review the requirements and design documents. They check if all the specified features like fund transfer, account balance check, and transaction history are included and correctly detailed in the design. They also perform peer reviews and inspections to ensure the design aligns with the requirements. This step ensures that the app is being built according to the initial plan and specifications without actually running the app.
- **Validation Example:** In the validation phase, the team starts testing the mobile banking app on actual devices. They check if users can log in, transfer money, and view their transaction history as intended. Testers perform usability tests to ensure the app is user-friendly and functional tests to ensure all features work correctly. They might also involve real users to provide feedback on the app's performance. This phase ensures that the app works as expected and meets user needs in real-world scenarios.

Advantages of Differentiating Verification and Validation

Differentiating between verification and validation in software testing offers several advantages:

1. **Clear Communication:** It ensures that team members understand which aspects of the software development process are focused on checking requirements (verification) and which are focused on ensuring functionality (validation).
2. **Efficiency:** By clearly defining verification as checking documents and designs without executing code, and validation as testing the actual software for functionality and usability, teams avoid redundant efforts and streamline their testing processes.
3. **Minimized Errors:** It reduces the chances of overlooking critical requirements or functionalities during testing, leading to a more thorough evaluation of the software's capabilities.
4. **Cost Savings:** Optimizing resource allocation and focusing efforts on the right testing activities based on whether they fall under verification or validation helps in managing costs effectively.

5. **Client Satisfaction:** Ensuring that software meets or exceeds client and user expectations by conducting both verification and validation processes rigorously improves overall software quality and user satisfaction.
6. **Process Improvement:** By distinguishing between verification and validation, organizations can refine their testing methodologies, identify areas for improvement, and enhance the overall **Software development lifecycle (SDLC).**

In essence, clear differentiation between verification and validation in **Software testing** contributes to a more structured, efficient, and successful software development process.

Conclusion

Verification is a static process focused on reviewing and analyzing documentation and design without running the code. It verify that the software is being built correctly according to specifications. In other-side, **validation** is a dynamic process that involves executing the software to check its functionality, usability, and suitability, verifying the right product is built to meet user needs. Both processes are essential for delivering a high-quality software product.

Verification plan and Test plan?

A verification plan outlines the overall approach and scope for verifying a system or design against its requirements, focusing on the "what" of verification. A test plan, on the other hand, provides detailed instructions on how specific functionalities will be tested to ensure they meet the verification plan's objectives, focusing on the "how". In essence, the verification plan is the high-level strategy, while the test plan is the detailed execution plan.

Verification Plan:

- **Purpose:**

To identify and define the requirements that need to be verified, the success criteria, and the coverage metrics for those requirements.

- **Scope:**

Broadly encompasses all aspects of the system or design to be verified.

- **Focus:**

Determines what needs to be verified and ensures the overall verification process is well-defined.

- **Timing:**

Created early in the design process and updated throughout the development lifecycle.

- **Content:**

May include detailed specifications, test environments, test scenarios, and coverage goals.

Test Plan:

- **Purpose:**

To document specific test cases and procedures that will be used to verify individual features or functionalities.

- **Scope:**

More focused and detailed than a verification plan, concentrating on specific functionalities.

- **Focus:**

Details how each functionality will be tested, including test conditions, expected results, and failure criteria.

- **Timing:**

Created after the design is complete or when a specific feature is ready for testing.

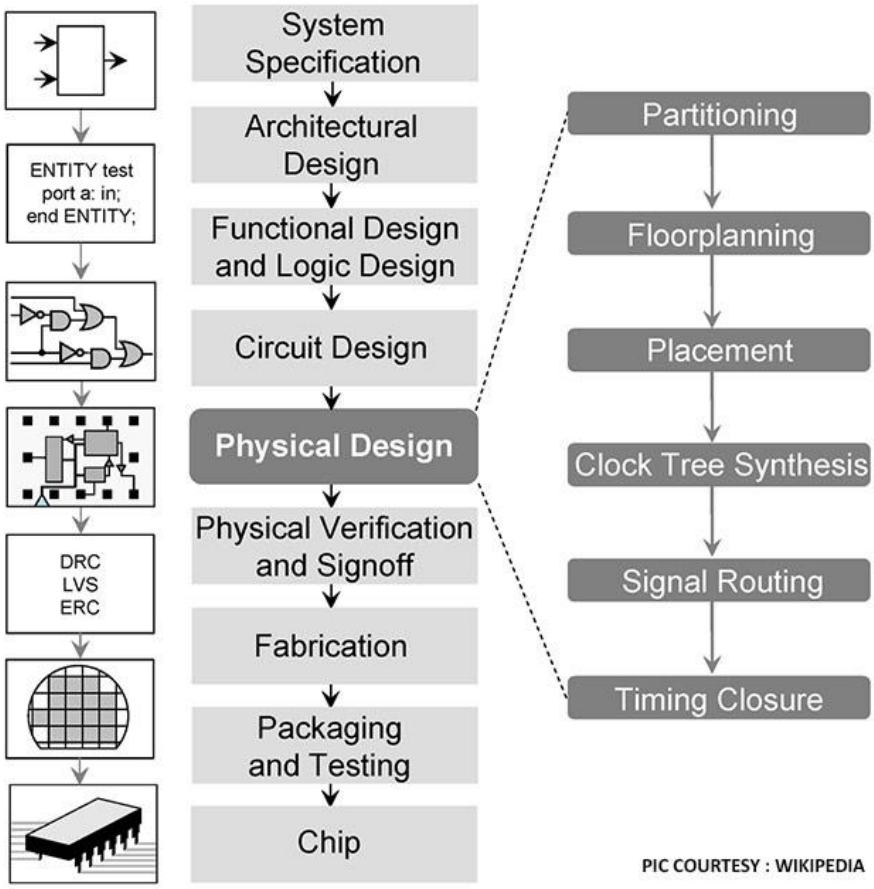
- **Content:**

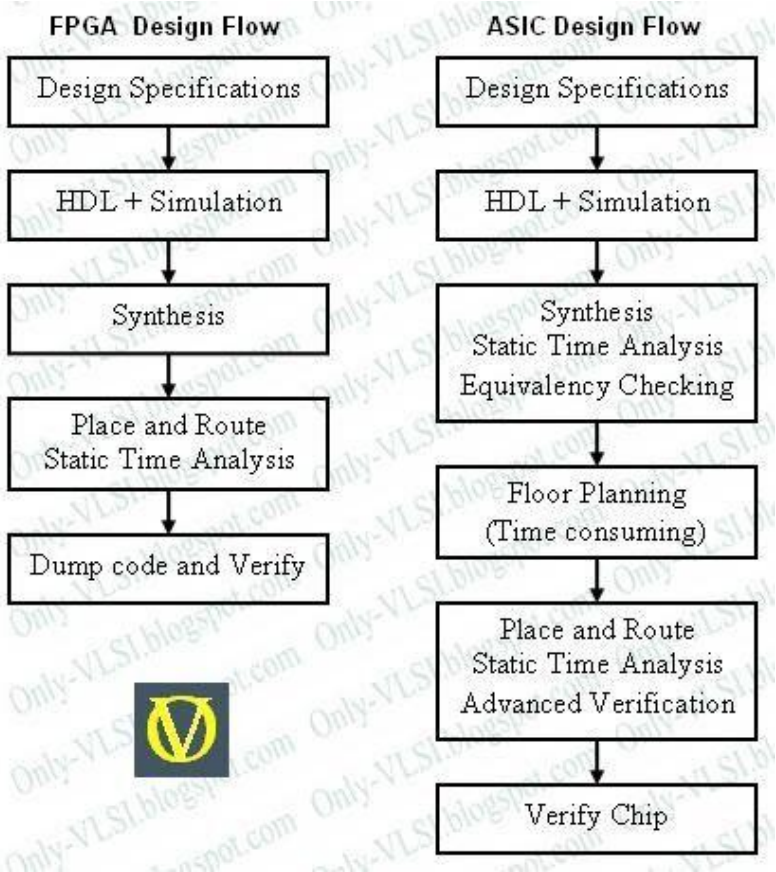
Test cases, expected results, pass/fail criteria, and sometimes include debugging information.

Relationship:

- Test plans are a crucial component of a verification plan.
- The verification plan provides the overall framework for verification, and test plans are used to execute the verification process.
- Test plans should align with and support the objectives outlined in the verification plan.

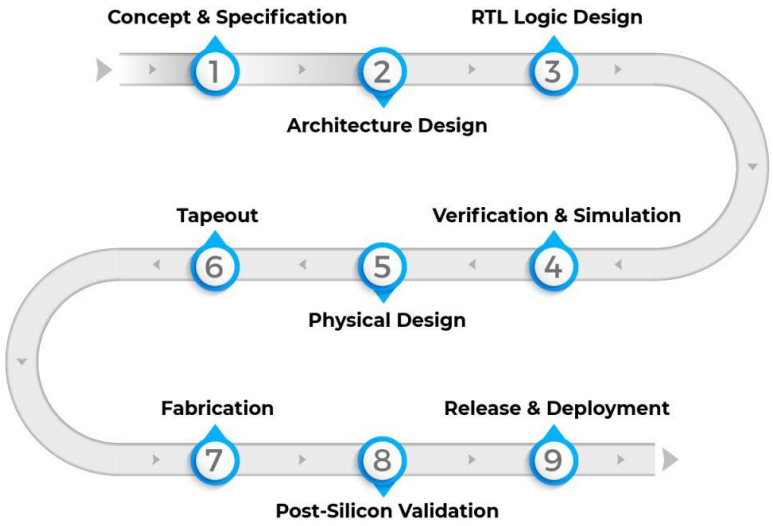
Design flow of fpga and design flow of soc verification?


System Specification
Architectural Design
Functional Design and Logic Design
Circuit Design
Physical Design
Physical Verification and Signoff
Fabrication
Packaging and Testing
Chip
ENTITY test
port a: in;
end ENTITY;
DRC
LVS
ERC
Partitioning
Floorplanning
Placement
Clock Tree Synthesis
Signal Routing
Timing Closure
PIC COURTESY : WIKIPEDIA


FPGA Design Flow
Design Specifications
HDL + Simulation
Synthesis
Place and Route
Static Time Analysis
Dump code and Verify
ASIC Design Flow
Design Specifications
HDL + Simulation
Synthesis
Static Time Analysis
Equivalency Checking
Floor Planning
(Time consuming)
Place and Route
Static Time Analysis
Advanced Verification
Verify Chip

Typical Development Process of Chip

The chip development process, also known as semiconductor or integrated circuit (IC) development, involves designing and manufacturing electronic chips that power a wide range of devices, from simple microcontrollers to advanced processors. Here's an overview of the typical chip development process:



CHIP DEVELOPMENT PROCESS

- **Concept and Specification:**

The chip development process begins with defining the purpose of the chip, its intended functionality, and the target application. Engineers and designers collaborate to create a chip specification, outlining its key features, physical design, performance requirements, power consumption, cost, power, and other essential characteristics.

- **Architecture Design:**

Based on the chip specifications, a high-level architectural design is developed that outlines the major functional blocks and their interconnections. This stage involves selecting appropriate technologies and methodologies to achieve the desired performance. The interfaces and protocols within the chips are also defined in this stage. The results of this stage serve as the foundation of subsequent stages, ensuring the development of an efficient chip with optimum performance.

- **RTL Logic Design:**

The term RTL refers to Register Transfer Level. The objective of digital design is to convert the architectural concept into a behavioral model representing the chip. The logic design phase focuses on creating detailed circuit schematics for each functional block in the chip's

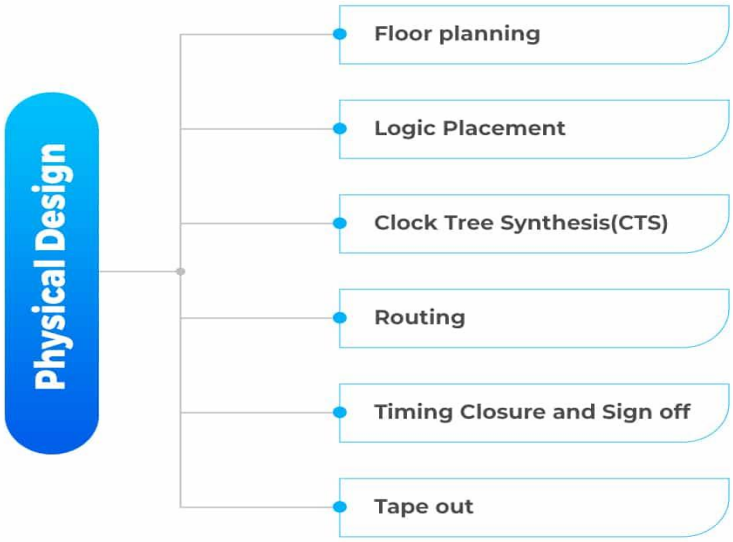
architecture. Hardware description languages (HDLs) like Verilog or VHDL are commonly used to represent the logic and behavior of the circuits. The RTL design specifies how the chip should function in terms of logical operations and data flow between memory blocks.

- **Verification and Simulation:**

Before physically implementing the chip, extensive verification and simulation are conducted to ensure that the design functions correctly and meets the specifications. This involves running various tests and simulations to identify and resolve potential design flaws and bugs. Any design issues found at the verification stage are sent back to the digital design team for corrections until they are fixed.

- **Physical Design:**

The physical design stage involves transforming the logical representation of the chip into a physical layout. The physical design of the chip can be broken down into multiple steps (illustrated below).



1. **Floorplanning:** The process starts with floorplanning, where transistors, interconnects, and other components are placed on the chip's silicon die with the help of Electronic Design Automation (EDA) software.
2. **Placement:** Next, comes placement where all the standard cells in the design are placed and assigned a legal location. A number of optimizations and timing-driven placement algorithms are performed to optimize placements, meet timing constraints and minimize signal delays.
3. **Clock Tree Synthesis:** Clock tree synthesis (CTS) is the stage after placement to make sure that the clock signals are distributed uniformly to all the sequential elements or flops. The objective of this stage is to create a good-quality clock network to meet the timing requirements for the design.

4. **Routing:** Routing is connecting the various blocks in the chips with one another through metal traces. Routing ensures proper electrical connections while optimizing power integrity and signal integrity (PI/ SI).
5. **Timing Closure and Signoff:** After routing, comes the timing closure and sign-off stage, a crucial step in the physical design. It analyzes the performance and functionality of the design by ensuring that all paths meet the timing requirements.
6. **Tape Out:** It is the final stage where the physical design is completely verified, and the design data in graphic data system (GDS II) format is sent to the semiconductor foundry for fabrication.

- **Fabrication:**

Once the physical design is complete, the chip goes into the manufacturing phase. The chip design is sent to a semiconductor foundry, where it undergoes photolithography, etching, doping, and other processes to create the physical chip.

- **Post-Silicon Validation:**

The escaped bugs during the pre-silicon verification stage are captured in the post-silicon validation stage. The prototypes of the chips received from the foundry are tested on reference/ evaluation boards for all functional correctness in a lab setup. The validation checks include many things apart from functional correctness such as compliance with power and performance for targeted use cases, thermal changes, electrical noise margins tolerance, and so on, The tests are performed on the real chips so it runs at the real system speed (in the range of GHz), so this is fast as compared to the pre-silicon verification stage. The overall goal of post-silicon validation is to make sure the Silicon/ ASIC/ SoC/ FPGA are qualified and ready to deploy.

- **Release and Deployment:**

Once the chips pass all testing and quality assurance processes, they are released for integration into various electronic devices or systems.

- **Post-Release Support:**

Manufacturers also provide ongoing support and updates, addressing any issues or improvements that may arise during the chip's lifecycle.

Tools Used in SoC/ FPGA/ ASIC Development

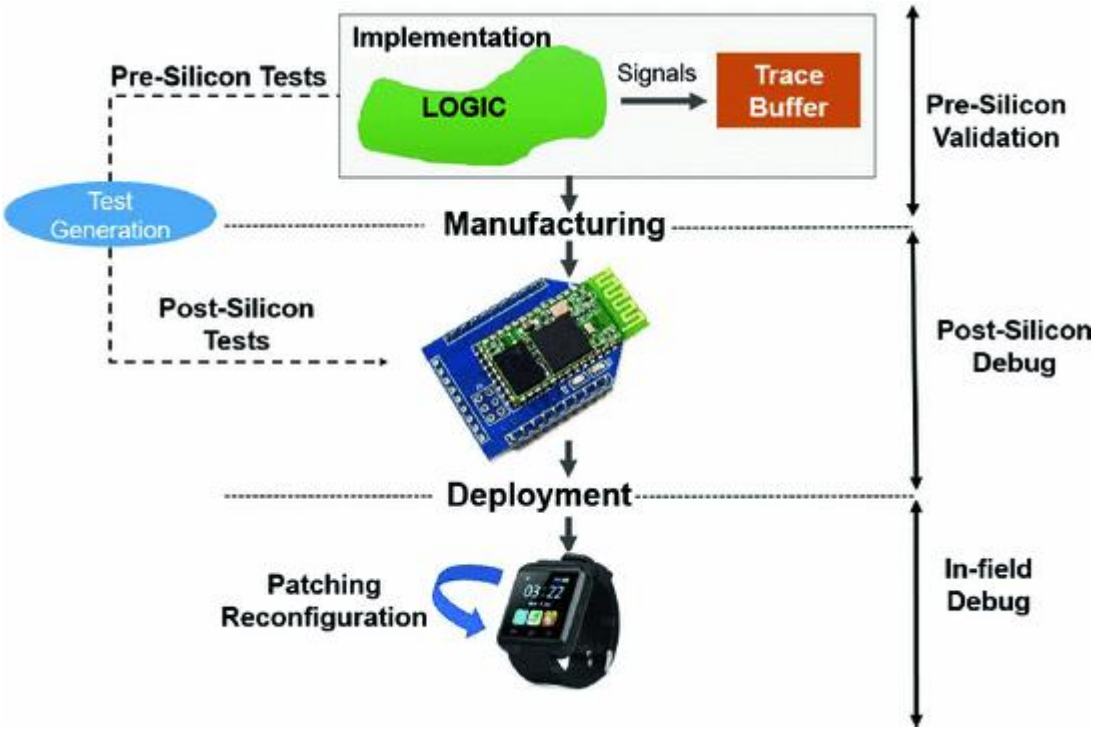
System-on-Chip (SoC), Field-Programmable Gate Array (FPGA), and Application-Specific Integrated Circuit (ASIC) development involve a wide range of tools that help with designing, verifying, and implementing complex integrated circuits. Hardware Description Language (HDL) Editors and Integrated Development Environments (IDEs) enable efficient code writing and simulation. Logic Synthesis, Place and Route, and Static Timing Analysis (STA) tools

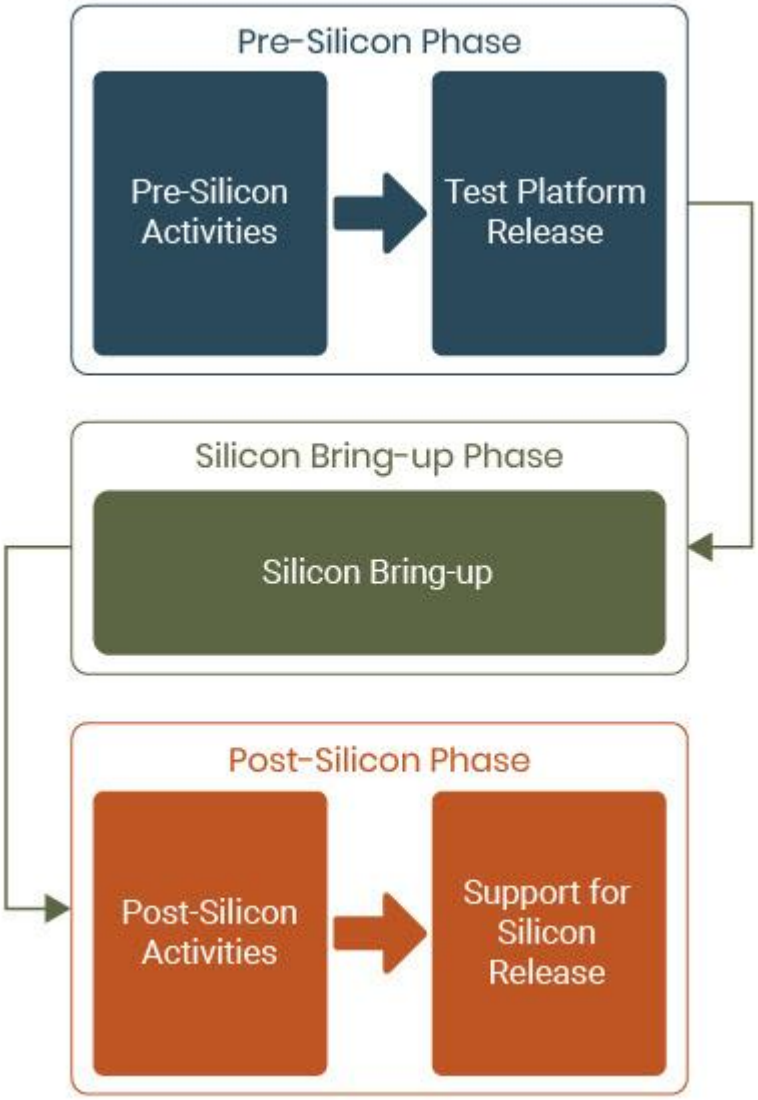
optimize the design process, ensuring optimal performance. Verification and Debugging tools facilitate rigorous testing and troubleshooting of a particular chip that is in the design phase. Additionally, High-Level Synthesis (HLS) and Emulation platforms streamline development, while Power Analysis and Formal Verification tools further enhance efficiency. Many leading companies, such as Cadence, Xilinx/AMD, Intel/Altera, Synopsys, and Mentor, are providing tools for RTL, logic synthesis, place, and route, static timing analysis, physical verification, DFT, simulation, power analysis, and more. With these tools, the journey from concept to production becomes a seamless and transformative experience.

Emerging Trends in Semiconductor Development

The latest trends in semiconductor or chip development are driven by the need for more efficient and specialized processors that can be used in targeted applications. Some of the technologies driving the trends in semiconductor development are 5G, the Internet of Things (IoT), Autonomous vehicles, and Edge Computing. Chip developers are also adopting new technologies in chip development, such as Heterogeneous integration and Artificial intelligence, to develop chips with faster performance. Semiconductor technologies, like 7nm, 5nm, or beyond, are also unlocking greater performance and power efficiency. All of these collectively promise a future of a more connected and intelligent world.

pre-silicon, post-silicon, pre-validation, post-validation?





"Pre-silicon" and "post-silicon" refer to the two main phases of testing and validation in the design of integrated circuits (ICs), specifically System-on-Chip (SoC) designs. Pre-silicon validation occurs before the ICs are manufactured, typically using simulation and emulation tools. Post-silicon validation happens after the ICs are manufactured, involving testing on the actual hardware. The term "validation" in this context often includes aspects of verification and testing.

Here's a more detailed breakdown:

Pre-silicon validation:

- **Focus:**

Functional verification, design analysis, and performance evaluation are done virtually before the ICs are fabricated.

- **Methods:**

Simulation, emulation, and prototyping platforms are used to test the design's behavior under different conditions.

- **Benefits:**

Early bug detection, reduced costs, and faster development time, especially when using emulation and prototyping.

- **Examples:**

Checking the functionality of a memory interface, simulating the behavior of a processor core, or running software on a hardware model.

Post-silicon validation:

- **Focus:**

Testing manufactured ICs in a real-world environment to verify their functionality, performance, and robustness.

- **Methods:**

Running real software, performing functional tests, and conducting stress tests to assess the IC's behavior under different conditions.

- **Benefits:**

Identifying bugs that may have escaped pre-silicon testing and ensuring the IC meets its functional and performance specifications.

- **Examples:**

Running a software application on the chip, checking the timing characteristics of a memory interface, or testing the chip under extreme temperature conditions.

Pre-validation vs. validation:

- **Pre-validation:**

This term can be used in various contexts, but in the IC design world, it often refers to the pre-silicon validation phase itself.

- **Validation:**

This term generally encompasses the entire process of ensuring that a design meets its specifications, and is a broader term that includes both pre- and post-silicon validation.

In summary: Pre-silicon is the virtual phase of testing, while post-silicon is the real-world phase. "Validation" is a broader term that can refer to both, and pre-validation is often used interchangeably with pre-silicon testing.

encoding-decoding errors?

Encoding and decoding errors occur when data is improperly converted between different formats, like text and binary, or when a specific encoding scheme is not used correctly. This can lead to data corruption, loss, or misinterpretation.

Here's a more detailed look at the issue:

What are encoding and decoding?

- **Encoding:**

Transforming data into a specific format, like converting text to a sequence of bytes.

- **Decoding:**

Converting data back from that format, like transforming bytes back into readable text.

Why do errors happen?

- **Mismatch in encoding/decoding methods:**

Using the wrong encoding scheme or method can cause errors when converting data.

- **Incorrect handling of special characters:**

Some encoding schemes might not handle special characters or accented letters correctly, leading to errors.

- **Incomplete data:**

If data is not fully encoded or decoded, it can lead to errors or missing values.

Examples of encoding/decoding errors:

- **UnicodeDecodeError:**

Occurs when the decoding process cannot interpret a sequence of bytes using the specified encoding.

- **UnicodeEncodeError:**

Occurs when a Unicode character cannot be encoded using the specified encoding.

- **"An asn.1 encoding/decoding error occurred":**

This error can happen when importing a certificate in DCM due to an incorrect file format or unsupported version of a standard.

How to address encoding/decoding errors:

- **Identify the encoding scheme:**

Determine the original encoding of the data to use the correct decoder.

- **Choose the right encoding:**

Select the appropriate encoding for the type of data and the desired format.

- **Use appropriate error handling:**

Implement error handling mechanisms to gracefully deal with potential errors during encoding or decoding.

- **Validate data integrity:**

Check the integrity of the data after encoding and decoding to ensure it is correct.

By understanding the fundamentals of encoding and decoding, and by being aware of potential errors, you can effectively handle data in various formats and prevent data loss or corruption.

connectivity issue between 2 modules?

A connectivity issue between two modules can stem from various factors, including physical connections, software settings, and network problems. To troubleshoot, check the physical connections, ensure proper software configurations, and verify network settings.

Troubleshooting Steps:

1. **1. Physical Connections:**
    - **Cable Issues:** If the modules are connected via cables, ensure the cables are securely plugged in and not damaged. Try a different cable to rule out a faulty cable.
    - **Antennas:** If the modules use antennas for wireless communication, make sure they are properly connected and mounted in a location that minimizes interference, such as away from metal objects.
    - **Power Supply:** Verify that both modules are receiving adequate power and that the power connections are secure.
2. **2. Software Configuration:**

- **Network Settings:** If the modules are on a network, ensure they have the correct IP addresses, subnet masks, and gateway settings. Check if firewalls or other network restrictions are blocking communication between the modules.
- **Module Configuration:** Verify that the modules are configured correctly, such as their mode of operation (e.g., access point or station mode) and communication protocols.
- **Firmware:** Ensure that the modules have the latest firmware, as older versions might have compatibility issues or bugs that affect connectivity.

3. **3. Network Problems:**
    - **Router Issues:** If the modules are connected to a router, check for any issues with the router itself, such as a reboot or a temporary outage.
    - **Network Congestion:** Excessive network traffic or other network-related problems can affect module communication.
    - **Interference:** Wireless communication can be affected by interference from other devices. Try moving the modules to a different location or try a different wireless frequency.

Additional Tips:

- **Check Documentation:**

Refer to the documentation for both modules to ensure that they are configured and used correctly, including any specific software or hardware requirements.

- **Use Network Tools:**

Tools like ping or traceroute can help identify network connectivity problems.

- **Restart Modules:**

If the modules are not functioning correctly, try restarting them to resolve temporary glitches.

When uvm gets hanged?

A UVM simulation can get stuck or "hang" due to a variety of reasons, often related to how components interact and manage objections during different phases of the simulation. Common causes include improper handling of objections, excessive delays in drivers or sequences, and potential issues with the testbench's run phase or shutdown phase.

Here's a more detailed breakdown of the common reasons why a UVM simulation might hang:

1. Objection Management Issues:

- **Objections not dropped:**

If a component raises an objection but forgets to drop it when it's done with a particular phase, the simulation will stall as that phase cannot proceed.

- **Objections dropped prematurely:**

If a component prematurely drops an objection (e.g., before it's fully finished with a task), other components might not be able to complete their work, leading to a deadlock.

- **Multiple objections raised by different components:**

If multiple components are raising objections to the same phase but aren't coordinating their work, it can lead to a situation where no one is dropping the objections and the simulation hangs.

- **Incorrect phase objection handling:**

The testbench might be object to the wrong phase, or not object at all, preventing other components from completing their work.

2. Delays in Drivers and Sequences:

- **Incorrect usage of delays:**

Using #(time) delays in drivers or sequences can be problematic if not carefully managed, potentially causing the simulation to wait indefinitely for a clock cycle that may never come.

- **Long delays causing timeouts:**

If a sequence introduces a very long delay, and the simulation's default timeout expires before the delay finishes, the simulation might terminate prematurely.

3. Run Phase Issues:

- **Stuck in run_phase:**

If a component gets stuck within its run_phase, it might never drop its objections, causing the simulation to wait indefinitely.

- **Infinite loops in run_phase:**

A component might have an infinite loop within its run_phase, preventing it from completing its work and dropping objections.

4. Shutdown Phase Issues:

- **Monitor not dropping objections:**

If a monitor is not dropping objections during the shutdown phase, it can prevent the simulation from terminating.

- **Delayed response from DUT:**

If the device under test (DUT) doesn't respond to the final stimulus in a timely manner, a monitor might continue to object, preventing the shutdown phase from completing.

5. Other Potential Issues:

- **Large and complex testbenches:**

Large testbenches with many VIPs, custom drivers, and monitors can increase the complexity of objection handling, making it harder to debug hangs.

- **Tool-specific issues:**

Some simulation tools have known bugs or limitations that can cause UVM simulations to hang.

Debugging Hangs:

- **UVM debug tools:**

Use UVM's debug tools (like uvm_debug or UVM_INFO messages) to identify the component that is hanging and the phase it's stuck in.

- **Print statements:**

Add print statements to the code to track the flow of execution, especially in areas where objections are being raised and dropped.

- **Step through the simulation:**

Use the debugger to step through the simulation code line by line to identify the exact point where the hang occurs.

- **Check the simulation log:**

Examine the simulation log for error messages or warnings that might provide clues about the cause of the hang.

- **Review objection counts:**

Use the uvm_phase::num_objections() method to track the number of objections raised and dropped by each component.

- **Use the heartbeat mechanism:**

If you can't pinpoint the exact cause of the hang, you can set up a heartbeat mechanism to detect if the simulation is progressing.

- **Simulate with a reduced testbench:**

Try simulating a smaller version of the testbench to isolate the problem.

messages or frame format, buffer overflow, boundary conditions errors?

In communication systems and computer programming, both frame format errors and boundary condition errors can cause problems. Frame format errors occur when a message doesn't adhere to the expected structure, leading to misinterpretation or corruption of data, as explained by Medium and GeeksforGeeks. Boundary condition errors, on the other hand, occur when a program encounters unexpected values at the edges of valid input ranges, potentially causing crashes or incorrect results, as discussed by LabEx and SpringerLink.

Frame Format Errors:

- **CAN Bus:**

In protocols like CAN (Controller Area Network), a frame format error, also called a "form error," occurs when a node detects a violation of the fundamental CAN frame encoding. For example, if a node starts transmitting a new frame before the previous frame's end sequence completes, a form error is detected.

- **Error Frames:**

When an error, including a frame format error, is detected, a special "error frame" is transmitted. This frame violates the CAN message formatting rules, causing other nodes to also send error frames, which effectively stops further communication.

- **Error Detection:**

Error detection mechanisms, like Cyclic Redundancy Check (CRC), are used to identify and correct errors in data frames, as explained by GeeksforGeeks.

Boundary Condition Errors:

- **Definition:**

Boundary conditions are critical points in code where input values can cause unexpected behavior or errors. They often occur at the edges of valid input ranges, such as array limits or numeric type boundaries.

- **Examples:**
    - **Array Out-of-Bounds:** Accessing an array element beyond its defined limits can lead to crashes or unpredictable behavior.

- **Numeric Overflow/Underflow:** Exceeding the maximum or minimum value of a data type can cause errors.

- **Handling:**

Boundary conditions should be carefully checked in code to prevent unexpected errors, as suggested by LabEx.

In essence, frame format errors are about the structure of data being transmitted, while boundary condition errors are about the values of data within a program's logic.

CAN Protocol: Error Detection And Fault Confinement

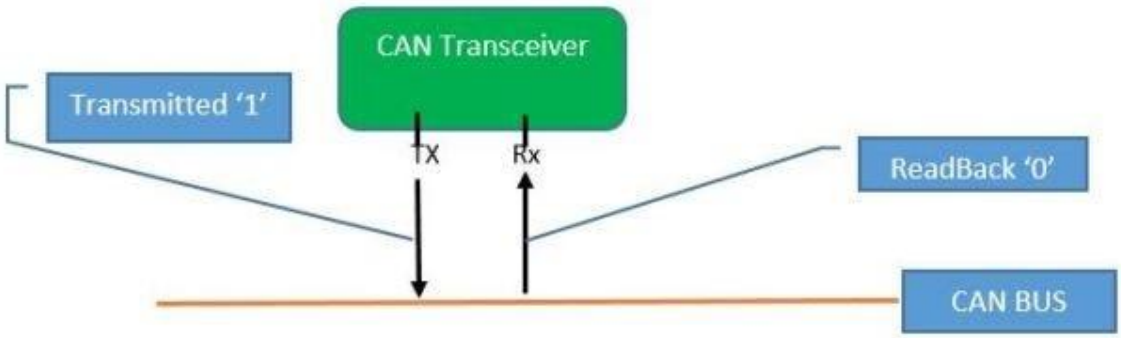
Error Detection

There are five types of CAN error can introduce on CAN-BUS by receiver and transmitter node-

1. Bit error
2. Stuff error
3. CRC error
4. Acknowledgment error
5. Form Error

Bit Error(Introduce By Transmitter)

As I discussed in Bit Monitoring chapter, Every transmitter node reads back its transmitted bit from CAN line along with other ECUs But if it will not get read bit same as it transmitted then transmitted node stop the further transmission and introduce an error frame on CAN line. This error is known as **bit error.**



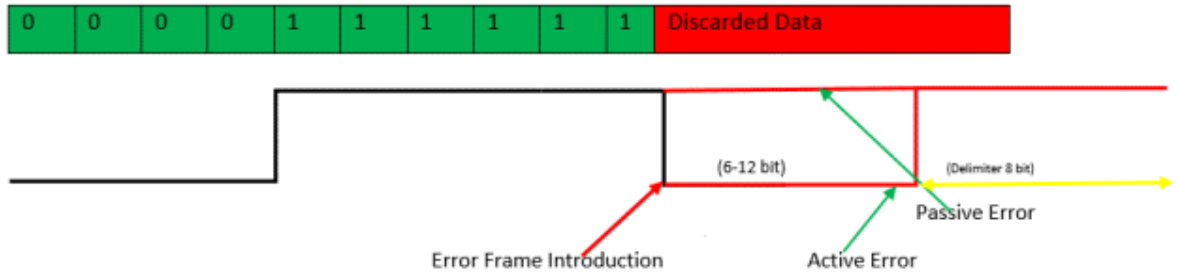
Bit Error Situation

Note:

1. If bit mismatch detected during Arbitration field or ACK then transmitter neither looks it as bit error nor introduce any error frame because as per the CAN standard these fields has defined functionality.
2. CRC Delimeter, ACK and EOF are also fixed length fields in CAN frame so it will also not be the part of bit error frame execution if more than five consecutive bits found.

Stuff Error(Introduce By Receiver)

As I discussed in Bit Stuffing chapter, More than 5 consecutive bits of the same polarity in CAN frame between the start of Frame(SOF) to CRC field is considered as a faulty frame on CAN Bus and it signaled as **stuff error** on CAN line.



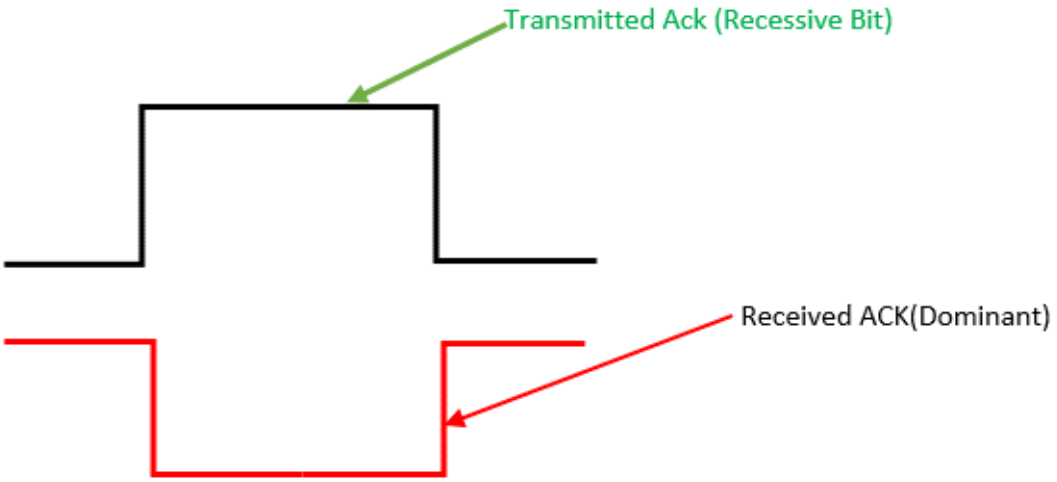
Stuff error Frame Introduction after 5 consecutive recessive bits

CRC Error(Introduce By Receiver)

The transmitter transmits CRC of transmitted data at CRC field of CAN frame and receivers also calculate CRC on received data. If the receiver found Calculated CRC is different from received CRC at CRC field then receiver signaled it as **CRC error** and introduce an error frame on CAN line.

ACK Error(Introduce By Transmitter)

After Transmission of CRC field of CAN frame Transmitter send ACK (a recessive bit) and receiver makes it dominant as a part of acknowledgment to the transmitter. During readback, transmitter found the dominant bit and consider it as receiver acknowledgment and if it reads a recessive bit then transmitter signaled it as ACK error and introduce an error frame.



Receiver Acked to transmitter By recessive to Dominant transition of ACK bit

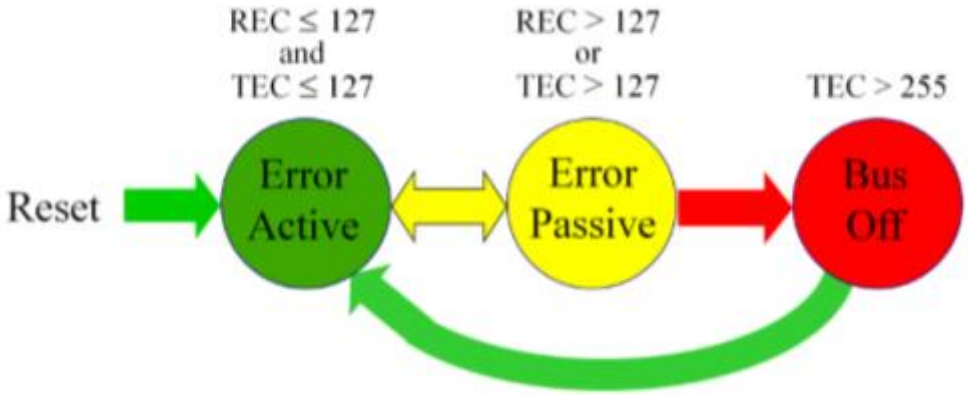
Form Error(Introduce By Receiver)

As per CAN Frame Format, there are some fields of fixed length and format like CRC Delimeter, ACK Delimeter, EOF ,InterFrame Spaceand if it is detected corrupted at receiver side then it signaled as Form Error and Node will introduce an error frame on CAN line.

Error Confinement Mechanism

In CAN network Every node is very honest. if any node transmits/receive continuous faulty frame then it will disconnect itself from CAN network after a threshold limit if fault not recovered. To decide this threshold every Node has two error counters: Transmit error Counter(TEC) and Receive Error Counter(REC) to keep the track of error generated because of above-discussed situations. On the basis of these counter, Nodes are categories in three error state-

1. Active Error state(When TEC<=127 and REC<=127)
2. Passive Error state(When TEC>127 or REC>127)
3. Bus-Off State(When TEC>255)



CAN Error States

Active Error State

A CAN Node enters into Active Error State when TEC<=127 and REC<=127 In This State Node can participate in all CAN activity like –

A node can Transmit all type of Frames like Data Frame, Remote Frame, Overload Frame, and **active Error Frame(6–12 Dominant bit)**

Passive Error State

A CAN Node enters into Passive Error State when TEC<=127 or REC<=127. This state still has a chance to come back in error-free state.In This State a Node still can participate in all CAN activity like –

1. A node can Transmit all type of Frames like Data Frame, Remote Frame, Overload Frame, but **Passive Error Frame**(**6–12 Recessive bit)**
2. A node can come back to in error active state if it starts transmission of the correct frames and its counter value comes under 127.

Bus-Off State

A CAN Node enters into Bus-Off State When TEC>255 . In this state CAN node will be removed from CAN network and now it will no longer available on CAN network to participate in any CAN activity.

Please follow next articles for more detail about CAN Bus Arbitration and Message Identification: https://medium.com/@mohammednumeir13/can-protocol-bus-arbitration-and-message-identification-ac52e80be1a1

Buffer Overflow

It is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations. This is a special case of the violation of memory safety.

Buffer overflows can be triggered by inputs that are designed to execute code, or alter the way the program operates. This may result in erratic program behavior, including memory access errors, incorrect results, a crash, or a breach of system security. Thus, they are the basis of many software vulnerabilities and can be maliciously exploited.

Programming languages commonly associated with buffer overflows include C and C++, which provide no built-in protection against accessing or overwriting data in any part of memory and do not automatically check that data written to an array (the built-in buffer type) is within the boundaries of that array. Bounds checking can prevent buffer overflows.

A buffer overflow occurs when data written to a buffer also corrupts data values in memory addresses adjacent to the destination buffer due to insufficient bounds checking. This can occur when copying data from one buffer to another without first checking that the data fits within the destination buffer.

In the following example, a program has two data items which are adjacent in memory: an 8-byte-long string buffer, A, and a two-byte big-endian integer, B.

char A[8] = “”;

unsigned short B = 1979;

Initially, A contains nothing but zero bytes, and B contains the number 1979.

variable name	**A**								**B**	
value	[null string]								1979	
hex value	00	00	00	00	00	00	00	00	07	BB

Now, the program attempts to store the null-terminated string “excessive” with ASCII encoding in the A buffer.

strcpy(A, “excessive”);

“excessive” is 9 characters long and encodes to 10 bytes including the terminator, but A can take only 8 bytes. By failing to check the length of the string, it also overwrites the value of B:

variable name	**A**								**B**	
value	‘e’								‘x’ ‘c’	25856
hex	65								78 63	65 00

B’s value has now been inadvertently replaced by a number formed from part of the character string. In this example “e” followed by a zero byte would become 25856.

Writing data past the end of allocated memory can sometimes be detected by the operating system to generate a segmentation fault error that terminates the process.

The techniques to exploit a buffer overflow vulnerability vary by architecture, by operating system and by memory region. For example, exploitation on the heap (used for dynamically allocated memory), differs markedly from exploitation on the call stack.

Stack-based exploitation

A technically inclined user may exploit stack-based buffer overflows to manipulate the program to their advantage in one of several ways:

- by overwriting a local variable that is near the buffer in memory on the stack to change the behavior of the program – which may benefit the attacker.
- by overwriting the return address in a stack frame. Once the function returns, execution will resume at the return address as specified by the attacker, usually a user-input filled buffer.
- by overwriting a function pointer or exception handler, which is subsequently executed
- by overwriting a parameter of a different stack frame or a non-local address pointed to in the current stack context

With a method called “trampolining”, if the address of the user-supplied data is unknown, but the location is stored in a register, then the return address can be overwritten with the address of an opcode which will cause execution to jump to the user supplied data. If the location is stored in a register R, then a jump to the location containing the opcode for a jump R, call R or similar instruction, will cause execution of user-supplied data. The locations of suitable opcodes, or bytes in memory, can be found in DLLs or in the executable itself. However the address of the opcode typically cannot contain any null characters and the locations of these opcodes can vary between applications and versions of the operating system. The Metasploit Project, for example, maintains a database of suitable opcodes, though listing only those found in the Windows operating-system.

Stack-based buffer overflows are not to be confused with stack overflows. Also note that these vulnerabilities are usually discovered through the use of a fuzzer.

Heap-based exploitation

A buffer overflow occurring in the heap data area is referred to as a heap overflow and is exploitable in a manner different from that of stack-based overflows. Memory on the heap is dynamically allocated by the application at run-time and typically contains program data. Exploitation is performed by corrupting this data in specific ways to cause the application to overwrite internal structures such as linked list pointers. The canonical heap overflow technique overwrites dynamic memory allocation linkage (such as malloc meta data) and uses the resulting pointer exchange to overwrite a program function pointer. Microsoft’s GDI+ vulnerability in handling JPEGs is an example of the danger a heap overflow can present.

Barriers to exploitation

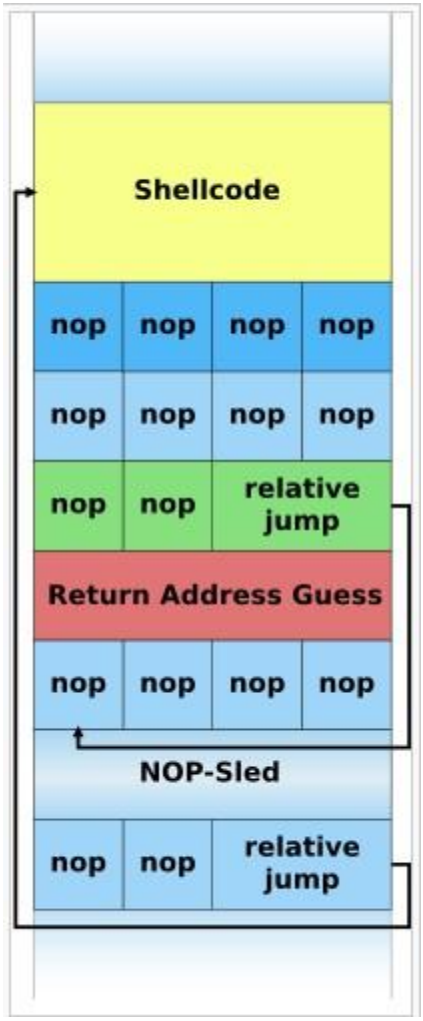
Manipulation of the buffer, which occurs before it is read or executed, may lead to the failure of an exploitation attempt. These manipulations can mitigate the threat of exploitation, but may not make it impossible. Manipulations could include conversion to upper or lower case, removal of metacharacters and filtering out of non-alphanumeric strings. However, techniques exist to bypass these filters and manipulations; alphanumeric

code, polymorphic code, self-modifying code and return-to-libc attacks. The same methods can be used to avoid detection by intrusion detection systems. In some cases, including where code is converted into unicode, the threat of the vulnerability have been misrepresented by the disclosers as only Denial of Service when in fact the remote execution of arbitrary code is possible.

Practicalities of exploitation

In real-world exploits there are a variety of challenges which need to be overcome for exploits to operate reliably. These factors include null bytes in addresses, variability in the location of shellcode, differences between environments and various counter-measures in operation.

NOP sled technique



A NOP-sled is the oldest and most widely known technique for successfully exploiting a stack buffer overflow. It solves the problem of finding the exact address of the buffer by effectively increasing the size of the target area. To do this, much larger sections of the stack are corrupted with the no-op machine instruction. At the end of the attacker-supplied data, after the no-op instructions, the attacker places an instruction to perform a relative jump to the

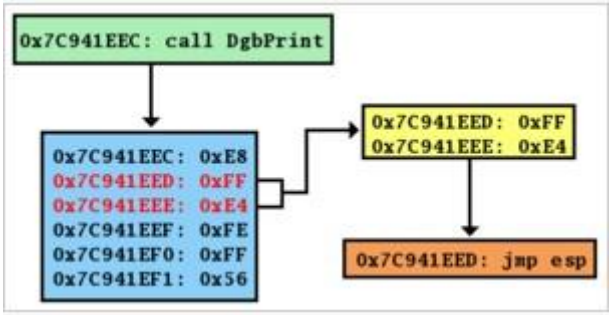
top of the buffer where the shellcode is located. This collection of no-ops is referred to as the “NOP-sled” because if the return address is overwritten with any address within the no-op region of the buffer it will “slide” down the no-ops until it is redirected to the actual malicious code by the jump at the end. This technique requires the attacker to guess where on the stack the NOP-sled is instead of the comparatively small shellcode.

Because of the popularity of this technique, many vendors of intrusion prevention systems will search for this pattern of no-op machine instructions in an attempt to detect shellcode in use. It is important to note that a NOP-sled does not necessarily contain only traditional no-op machine instructions; any instruction that does not corrupt the machine state to a point where the shellcode will not run can be used in place of the hardware assisted no-op. As a result it has become common practice for exploit writers to compose the no-op sled with randomly chosen instructions which will have no real effect on the shellcode execution.

While this method greatly improves the chances that an attack will be successful, it is not without problems. Exploits using this technique still must rely on some amount of luck that they will guess offsets on the stack that are within the NOP-sled region. An incorrect guess will usually result in the target program crashing and could alert the system administrator to the attacker’s activities. Another problem is that the NOP-sled requires a much larger amount of memory in which to hold a NOP-sled large enough to be of any use. This can be a problem when the allocated size of the affected buffer is too small and the current depth of the stack is shallow (i.e. there is not much space from the end of the current stack frame to the start of the stack). Despite its problems, the NOP-sled is often the only method that will work for a given platform, environment, or situation; as such it is still an important technique.

The jump to address stored in a register technique

The “jump to register” technique allows for reliable exploitation of stack buffer overflows without the need for extra room for a NOP-sled and without having to guess stack offsets. The strategy is to overwrite the return pointer with something that will cause the program to jump to a known pointer stored within a register which points to the controlled buffer and thus the shellcode. For example, if register A contains a pointer to the start of a buffer then any jump or call taking that register as an operand can be used to gain control of the flow of execution.



An instruction from ntdll.dll to call the DbgPrint() routine contains the i386 machine opcode for jmp esp.

In practice a program may not intentionally contain instructions to jump to a particular register. The traditional solution is to find an unintentional instance of a suitable opcode at a fixed location somewhere within the program memory. In figure E on the left you can see an example of such an unintentional instance of the i386 jmp esp instruction. The opcode for this instruction is FF E4. This two-byte sequence can be found at a one-byte offset from the start of the instruction call DbgPrint at address 0x7C941EED. If an attacker overwrites the program return address with this address the program will first jump to 0x7C941EED, interpret the opcode FF E4 as the jmp esp instruction, and will then jump to the top of the stack and execute the attacker's code.

When this technique is possible the severity of the vulnerability increases considerably. This is because exploitation will work reliably enough to automate an attack with a virtual guarantee of success when it is run. For this reason, this is the technique most commonly used in Internet worms that exploit stack buffer overflow vulnerabilities.

This method also allows shellcode to be placed after the overwritten return address on the Windows platform. Since executables are mostly based at address 0x00400000 and x86 is a Little Endian architecture, the last byte of the return address must be a null, which terminates the buffer copy and nothing is written beyond that. This limits the size of the shellcode to the size of the buffer, which may be overly restrictive. DLLs are located in high memory (above 0x01000000) and so have addresses containing no null bytes, so this method can remove null bytes (or other disallowed characters) from the overwritten return address. Used in this way, the method is often referred to as "DLL Trampolining".

Protective countermeasures

Various techniques have been used to detect or prevent buffer overflows, with various tradeoffs. The most reliable way to avoid or prevent buffer overflows is to use automatic protection at the language level. This sort of protection, however, cannot be applied to legacy code, and often technical, business, or cultural constraints call for a vulnerable language. The following sections describe the choices and implementations available.

Choice of programming language – The choice of programming language can have a profound effect on the occurrence of buffer overflows. As of 2008, among the most popular languages are C and its derivative, C++, with a vast body of software having been written in these languages. C provides no built-in protection against accessing or overwriting data in any part of memory; more specifically, it does not check that data written to a buffer is within the boundaries of that buffer. The standard C++ libraries provide many ways of safely buffering data, and C++'s Standard Template Library (STL) provides containers that can optionally perform bounds checking if the programmer explicitly calls for checks while accessing data. For example, a vector's member function at() performs a bounds check and

throws an out_of_range exception if the bounds check fails. However, C++ behaves just like C if the bounds check is not explicitly called. Techniques to avoid buffer overflows also exist for C.

Many other programming languages provide runtime checking and in some cases even compile-time checking which might send a warning or raise an exception when C or C++ would overwrite data and continue to execute further instructions until erroneous results are obtained which might or might not cause the program to crash. Examples of such languages include Ada, Eiffel, Lisp, Modula-2, Smalltalk, OCaml and such C-derivatives as Cyclone, Rust and D. The Java and .NET Framework bytecode environments also require bounds checking on all arrays. Nearly every interpreted language will protect against buffer overflows, signaling a well-defined error condition. Often where a language provides enough type information to do bounds checking an option is provided to enable or disable it. Static code analysis can remove many dynamic bound and type checks, but poor implementations and awkward cases can significantly decrease performance. Software engineers must carefully consider the tradeoffs of safety versus performance costs when deciding which language and compiler setting to use.

Use of safe libraries – The problem of buffer overflows is common in the C and C++ languages because they expose low level representational details of buffers as containers for data types. Buffer overflows must thus be avoided by maintaining a high degree of correctness in code which performs buffer management. It has also long been recommended to avoid standard library functions which are not bounds checked, such as gets, scanf and strcpy. The Morris worm exploited a gets call in fingerd.

Well-written and tested abstract data type libraries which centralize and automatically perform buffer management, including bounds checking, can reduce the occurrence and impact of buffer overflows. The two main building-block data types in these languages in which buffer overflows commonly occur are strings and arrays; thus, libraries preventing buffer overflows in these data types can provide the vast majority of the necessary coverage. Still, failure to use these safe libraries correctly can result in buffer overflows and other vulnerabilities; and naturally, any bug in the library itself is a potential vulnerability. “Safe” library implementations include “The Better String Library”, Vstr and Erwin. The OpenBSD operating system’s C library provides the strlcpy and strlcat functions, but these are more limited than full safe library implementations.

In September 2007, Technical Report 24731, prepared by the C standards committee, was published; it specifies a set of functions which are based on the standard C library’s string and I/O functions, with additional buffer-size parameters. However, the efficacy of these functions for the purpose of reducing buffer overflows is disputable; it requires programmer intervention on a per function call basis that is equivalent to intervention that could make the analogous older standard library functions buffer overflow safe.

Buffer overflow protection – Buffer overflow protection is used to detect the most common buffer overflows by checking that the stack has not been altered when a function returns. If it has been altered, the program exits with a segmentation fault. Three such systems are Libsafe, and the StackGuard and ProPolice gcc patches.

Microsoft's implementation of Data Execution Prevention (DEP) mode explicitly protects the pointer to the Structured Exception Handler (SEH) from being overwritten.

Stronger stack protection is possible by splitting the stack in two: one for data and one for function returns. This split is present in the Forth language, though it was not a security-based design decision. Regardless, this is not a complete solution to buffer overflows, as sensitive data other than the return address may still be overwritten.

Pointer protection – Buffer overflows work by manipulating pointers (including stored addresses). PointGuard was proposed as a compiler-extension to prevent attackers from being able to reliably manipulate pointers and addresses. The approach works by having the compiler add code to automatically XOR-encode pointers before and after they are used. Because the attacker (theoretically) does not know what value will be used to encode/decode the pointer, he cannot predict what it will point to if he overwrites it with a new value. PointGuard was never released, but Microsoft implemented a similar approach beginning in Windows XP SP2 and Windows Server 2003 SP1. Rather than implement pointer protection as an automatic feature, Microsoft added an API routine that can be called at the discretion of the programmer. This allows for better performance (because it is not used all of the time), but places the burden on the programmer to know when it is necessary.

Because XOR is linear, an attacker may be able to manipulate an encoded pointer by overwriting only the lower bytes of an address. This can allow an attack to succeed if the attacker is able to attempt the exploit multiple times or is able to complete an attack by causing a pointer to point to one of several locations (such as any location within a NOP sled). Microsoft added a random rotation to their encoding scheme to address this weakness to partial overwrites.

Executable space protection – Executable space protection is an approach to buffer overflow protection which prevents execution of code on the stack or the heap. An attacker may use buffer overflows to insert arbitrary code into the memory of a program, but with executable space protection, any attempt to execute that code will cause an exception.

Some CPUs support a feature called NX ("No eXecute") or XD ("eXecute Disabled") bit, which in conjunction with software, can be used to mark pages of data (such as those containing the stack and the heap) as readable and writeable but not executable.

Some Unix operating systems (e.g. OpenBSD, OS X) ship with executable space protection (e.g. W^X). Some optional packages includes PaX, Exec Shield and Openwall. Newer variants

of Microsoft Windows also support executable space protection, called Data Execution Prevention. Proprietary add-ons includes BufferShield and StackDefender

Executable space protection does not generally protect against return-to-libc attacks, or any other attack which does not rely on the execution of the attackers code. However, on 64-bit systems using ASLR, as described below, executable space protection makes it far more difficult to execute such attacks.

Address space layout randomization – Address space layout randomization (ASLR) is a computer security feature which involves arranging the positions of key data areas, usually including the base of the executable and position of libraries, heap, and stack, randomly in a process' address space.

Randomization of the virtual memory addresses at which functions and variables can be found can make exploitation of a buffer overflow more difficult, but not impossible. It also forces the attacker to tailor the exploitation attempt to the individual system, which foils the attempts of internet worms. A similar but less effective method is to rebase processes and libraries in the virtual address space.

Deep packet inspection – The use of deep packet inspection (DPI) can detect, at the network perimeter, very basic remote attempts to exploit buffer overflows by use of attack signatures and heuristics. These are able to block packets which have the signature of a known attack, or if a long series of No-Operation instructions (known as a nop-sled) is detected, these were once used when the location of the exploit's payload is slightly variable. Packet scanning is not an effective method since it can only prevent known attacks and there are many ways that a 'nop-sled' can be encoded. Shellcode used by attackers can be made alphanumeric, metamorphic, or self-modifying to evade detection by heuristic packet scanners and intrusion detection systems.

Parity bit, CRC, ECC?

Parity bits, Cyclic Redundancy Check (CRC), and Error Correction Codes (ECC) are all techniques used to detect and correct errors in data transmission or storage, but they operate at different levels and offer different error-handling capabilities.

Parity Bits:

- A simple error detection technique where an extra bit (the parity bit) is added to a block of data.
- The parity bit is set to ensure that the total number of 1s in the block (including the parity bit) is either even (even parity) or odd (odd parity).
- Useful for detecting single-bit errors, but can't correct them or detect multiple-bit errors, according to a Quora post [4, 5].

Cyclic Redundancy Check (CRC):

- A more robust error detection technique that uses a mathematical algorithm to generate a checksum based on the data.
- This checksum is appended to the data and transmitted along with it.
- The receiver recalculates the checksum and compares it with the received checksum to verify data integrity.
- CRC can detect a wider range of errors than parity bits, including burst errors, but cannot correct them.

Error Correction Codes (ECC):

- ECC includes a set of algorithms used to detect and correct errors in data transmission or storage.
- ECC adds extra bits (redundancy) to the data, allowing the receiver to not only detect errors but also correct them.
- ECC offers a higher level of reliability and is used in various applications, including memory chips and storage devices.

jasper gold usage, spyglass usage, questasim usage, vcs usage, Verdi usage, ncsim, simvision?

JasperGold is a formal verification platform primarily used in the semiconductor industry for verifying the functionality and reliability of electronic designs. It's used across various verification tasks, including block verification, integration, equivalence checking, security verification, and coverage closure, offering a powerful alternative to traditional simulation-based methods.

Key Areas of JasperGold Usage:

- **Block Verification:** Ensuring individual design blocks function as expected.
- **Integration Verification:** Verifying the interaction between different design blocks.
- **Equivalence Checking:** Confirming that different design models or implementations are functionally equivalent.
- **Security Verification:** Checking for vulnerabilities and ensuring security protocols are followed.
- **Coverage Closure:** Ensuring sufficient verification coverage to identify potential issues.

- **Functional Safety Verification:** Specifically, the Jasper FSV app utilizes formal verification techniques to ensure safety standards are met.
- **Low-Power Verification:** The Jasper LPV app helps verify low-power design aspects and power intents.
- **Equivalence Checking:** The Jasper Sequential Equivalence Checking (SEC) App verifies the behavioral equivalence of two RTL models, especially useful when optimizing for power or performance.
- **Control and Status Register Verification:** The Jasper CSR App automates the verification of control and status registers in designs.
- **Deep Bug Hunting:** JasperGold is used to find bugs deep within the design, even in small designs.
- **Design Signoff:** Increasingly, designs are signed off using formal verification with JasperGold, providing greater assurance of correctness.

Advantages of using JasperGold:

- **Exhaustive Verification:** Formal verification methods like those in JasperGold can provide exhaustive analysis, ensuring all possible scenarios are considered.
- **Early Bug Detection:** Identifying bugs early in the design process can save time and resources.
- **Increased Confidence:** Formal verification can provide a higher level of confidence in the correctness of the design compared to simulation-based approaches.
- **Scalability:** JasperGold can be scaled up to handle large and complex designs.
- **User-Friendly Interface:** JasperGold offers a user-friendly interface, including debug tools like Visualize and QuietTrace, which simplify the verification process.
- **Automated Features:** JasperGold provides automated features, such as automatic coverage generation and bug detection, reducing the manual effort required.

In essence, JasperGold is a powerful formal verification platform that helps engineers ensure the correctness and reliability of electronic designs, from individual blocks to entire systems, and is increasingly being used for design signoff and verification tasks across the semiconductor industry.

Lint in VLSI using Spyglass

Linting in VLSI is the process of checking the program code (static code analysis) against a set of design rules and generating a report with all details of violations.

After the compilation and elaboration step, the design will be free of syntax errors. However, still all the design rules need not be satisfied. Lint is now the solution to this. In order to have our design to be completely synthesizable, correcting lint errors are essential.

Synopsys Spyglass Tool is a popular industry standard tool used to perform Linting on Verilog HDL code. Cadence HAL is one another such tool.

The lint report may be large and will consist of various criterions of design violations. However, let us look at some of the major violations:

1. Combinational Loops
2. Inferred Latches
3. Multi-driven signals
4. Width Mismatches
5. Undriven Inputs

1. Combinational Loops

Combinational loops are strictly not allowed in Verilog. Example of a combinational loop snippet:

```
always @ *
a = a & b;
```

This computes the AND operation of a with b and sends it back to a. This will lead to an infinite loop, completely occurring at zero simulation time.

Incase, we really need to perform this operation, we need to use non-blocking assignments.

```
always @ (posedge clock)

a <= a & b;
```

This ensures that a AND b is computed first, and then updated to a only in the next clock cycle.
Each clock cycle, a AND b is computed and the result is updated in a in the following clock cycle.

2. Inferred Latches

Inferred latches is a major issue that can possibly cause the design to malfunction.

This happens when the Verilog code does not account for the value held by a variable at certain times.

It occurs due to incomplete usage of certain constructs.

Example:

1. If condition without else block

```
always @ (posedge clk)

if (a > b)

c <= a;
```

If this is all the given code, how do we know what happens to c when a > b condition fails?

This is called inferred latches when the tool infers that under such circumstances, the previous value of the variable is maintained.

To avoid the tool making such an inference, it is always desirable for the designer to explicitly mention what happens in such a case. Even if the value is maintained, it can be explicitly coded as shown:

```
always @ (posedge clk)

if (a > b)

c <= a;

else

c <= c;
```

This will ensure that the latches are not 'inferred' by the tool.

3. Multi-Driven Signals

Multi-driven signals are also forbidden in Verilog. When the same signal is driven with multiple values at the same time instant, it becomes multi-driven and causes undesired outputs to occur, usually in simulations it goes to a don't care (x) state.

Example:

```
always @ (posedge clk)
```

```
if (a > b)
c <= a;
c <= b;
else
c <= c;
```

As can be seen above, the same signal c is simultaneously trying to obtain the value of a as well as b. This is not allowed. At a particular clock cycle, we can drive the signal c with only one value, either that of a or that of b.

4. Width Mismatches

This is one of the more common issues that can be discovered only by lint. Mismatches in the width of signals being used on LHS as well as RHS.

As a simple example:

```
wire [3:0] a;
wire [7:0] b;
assign a = b;
```

This will cause a width mismatch. The width of signal a is 4-bits while the width of b is 8-bits.

When you assign b to a, a will be able to store only the least significant 4-bits of b, ie., b[3:0].

The remaining 4-bits of b will be lost.

Hence to avoid potential data loss, it is advised to ensure that the signals used in the LHS as well as RHS of an expression is of the same width.

5. Undriven Inputs

This situation occurs when one or more inputs used in a module is not being driven externally.

If the input is not being driven, then the design may not work as expected.

The above points are just few examples of violations flagged by lint that need to be noted. Many others can be added to this lint depending on the design requirement.

Tcl Shell Commands using Spyglass

The above image shows the Spyglass GUI Window, but we can utilize the tool more easily using Tcl scripts.

Let us now see how we can invoke the Spyglass Tool using Tcl commands.

1. Create a new project

new_project command is used to create a new project.

Usage: *new_project* project_name

2. Read required files

read_file command is used to read the design files. It comes along with a -type argument.

Consider you are reading a single file example_design.v

Usage: *read_file* -type verilog example_design.v

(Note: Ensure that you mention the full path of example_design.v if not in current folder)

To read a heirarchy of files specified in a filelist, we use:

read_file -type sourcelist filelist

We might need to add waivers to waive off certain violations. For waivers mentioned in a TCL file, the command to read that would be:

read_file -type awl waivers.tcl

The waivers.tcl file would look something like this:

#whatever rule you want to waive is mentioned in brackets, along with msg and optional comment

waive -rule {W287b} - msg { } - comment {waivable}

waive -rule {W415a} - msg { } - comment {waivable}

Sometimes you might have some design constraints like setting clock and reset values, and you might need to read those as well.

Such design constraints are saved in a .sgdc file and are read using:

read_file -type sgdc constraints.sgdc

Once the desired files are read, proceed with the next step.

3. Set desired options

set_option command is used to set custom options for the run. Example:

For a hierarchy of files, we can set the top module using: *set_option* top topmodule

There are a number of arguments available for the *set_option* command that can be explored using the Spyglass userguide.

Another usage: *set_option* handlememory true can be used to handle memories in an optimized manner.

4. Perform the desired run

run_goal command to perform the desired run.

In order to perform lint, we use the command like this:

run_goal lint/lint_rtl

5. Save and Exit

At the end, use the following command to save the project before exiting:

exit -save

These commands can be saved in a Tcl file, and should be enough to invoke the tool and run the selected goal (lint) on the desired files.

The spyglass reports will be in the following path:

'your design path'/lint/lint_rtl/spyglass_reports

The moresimple.rpt will contain a summary report of all lint violations.

With this, we conclude the topic on Lint and its analysis using Spyglass Tool.

Refer the Spyglass tool documents for more information.

References:

Spyglass Lint User Guide

Questasim

QuestaSim is a functional simulator used for verifying designs written in VHDL, Verilog, SystemVerilog, and SystemC. It's part of the Mentor Graphics' Questa Advanced Functional Verification Platform and offers advanced verification features like coverage databases and assertions. To use QuestaSim, you can either use the GUI or command-line commands.

Basic Usage:

1. **Launch QuestaSim:** Start the simulator using commands like vsim & (for Linux).
2. **Set up the project:** Use commands like do msim_setup.tcl to configure the simulation environment.
3. **Compile the design:** Compile your VHDL, Verilog, or SystemVerilog files using commands like vcom -work work <design_file>.vhd.
4. **Run the simulation:** Use commands like ld run -all to start the simulation.
5. **Analyze results:** Check the simulation logs for successful completion messages.
6. **View waveforms:** Add signals to the waveform window using the Object Browser and then add them as waveforms.

Command-Line Commands:

- vlib: Creates a library.
- vmap: Maps a library to a directory.
- vcom: Compiles VHDL or Verilog HDL files.
- vsim: Starts the simulator.
- ld run -all: Runs the simulation.
- do <script_name>.do: Executes a QuestaSim macro script.

GUI Usage:

- **Compile:** Use the Compile menu to compile design and testbench files.
- **Set libraries:** Select the work library in the Compile HDL Source Files dialog box.
- **Add signals to waveforms:** Use the Object Browser to select signals and add them to the waveform window.
- **Run the simulation:** Use the Simulate menu to start the simulation.

Key Features:

- **Advanced Verification Features:** Supports coverage databases, coverage-driven verification, assertions, and SystemVerilog constrained-random functionality.
- **Performance and Capacity:** Offers industry-leading performance and capacity through optimization algorithms.
- **Testbench Automation:** Supports comprehensive testbench automation solutions for generating complex input stimuli.

- **Debugging:** Provides advanced debugging capabilities.
- **Timing Simulation:** Can perform timing simulations after a functional simulation.

VCS:

Synopsys VCS (Verification Compiler Suite) is a powerful functional verification tool used for simulating and debugging digital designs, particularly those written in Verilog and SystemVerilog. It's known for its high performance and comprehensive features, including coverage analysis and advanced debug capabilities. VCS is a key component of Synopsys's broader verification ecosystem, which also includes tools like Verdi for debugging and Formality for formal verification.

Key Features and Usage of Synopsys VCS:

- **High-Performance Simulation:**

VCS is designed to provide fast simulation speeds for complex designs, allowing for efficient verification and debugging.

- **SystemVerilog Support:**

VCS supports the latest SystemVerilog language features, enabling verification of complex designs.

- **Advanced Debugging:**

VCS integrates with Verdi, a leading debug tool, to provide a comprehensive debugging environment.

- **Coverage Analysis:**

VCS offers built-in coverage analysis capabilities, allowing users to track the execution of their designs and identify areas where further testing is needed.

- **Mixed-HDL and Mixed-Signal Simulation:**

VCS supports simulation of mixed-HDL designs (Verilog and VHDL) and even mixed-signal designs (functional and analog).

- **Native Testbench (NTB) Support:**

VCS supports NTB, which allows users to create testbenches using the same language as the design, improving consistency and reducing setup time.

- **Integration with Other Tools:**

VCS integrates with other Synopsys tools, such as Vera for testbench automation and CoCentric for SystemC simulation.

- **Verification Planning:**

VCS provides tools for verification planning, helping users define verification goals and strategies.

- **Coverage Closure:**

VCS helps users close coverage gaps by identifying and addressing uncovered areas in their designs.

How VCS is Used:

1. **1. Compilation:**

VCS is used to compile Verilog or SystemVerilog code into an executable simulator.

2. **2. Simulation:**

The executable simulator is then used to run the simulation, allowing users to observe the behavior of their design under different conditions.

3. **3. Debugging:**

VCS provides tools for debugging, such as waveform viewing and breakpoint setting, to help users identify and fix errors in their design.

4. **4. Coverage Analysis:**

VCS can be used to analyze the coverage of their simulation, ensuring that all parts of the design are being tested.

5. **5. Testbench Creation:**

VCS supports the creation of testbenches, which are used to provide stimulus to the design and monitor its output.

6. **6. Integration with other Tools:**

VCS can be integrated with other Synopsys tools, such as Verdi, to provide a complete verification solution.

ncsim

NCSim is a simulation tool, specifically the Cadence NC Sim simulator, used for simulating digital designs and systems. It's a powerful tool for verifying the functionality of designs written in languages like Verilog, VHDL, and SystemC.

Here's a more detailed explanation of its usage:

Purpose and Functionality:

- **Simulation Engine:**

NCSim is the core simulation engine that runs simulations of digital designs.

- **Unified Simulation:**

It supports various design languages (Verilog, VHDL, SystemC) and can be used for both functional and timing simulations.

- **Debugging and Verification:**

NCSim provides a comprehensive set of debugging features, similar to ModelSim's vsim, allowing users to step through simulations, examine signals, and identify issues.

- **Snapshot Images:**

NCSim works with snapshot images generated by the NC Elaborator, a tool that compiles and links Verilog, VHDL, and SystemC libraries.

Key Components and Tools:

- **NC Sim:**

The main simulation engine itself, used to load and run simulations.

- **NC Elaborator (ncelab):**

Compiles and links Verilog, VHDL, and SystemC libraries, creating a snapshot image for simulation.

- **Irun:**

A tool for running simulations with a single-step invocation, automatically calling other tools like ncvlog, ncvhdl, and ncsc.

- **Sim Vision:**

A standalone graphical waveform viewer and netlist tracer, similar to Novas Software's Debussy.

Usage in Context:

1. **1. Design and Verification:**

NCSim is used extensively in the process of designing and verifying digital circuits and systems.

2. **2. FPGA Development:**

It's also used in FPGA development, where users simulate designs before deploying them on the hardware.

3. **3. System Verilog Functional Coverage:**

NCSim can be used to analyze functional coverage in System Verilog designs.

4. **4. Mixed-Signal Design:**

It can be used for simulating designs that include both digital and analog components.

Simvision

SimVision is a waveform viewer and simulator debugging tool, primarily used with digital design simulation environments. It allows users to view and analyze simulation waveforms, identify timing issues, and debug circuits by tracing signals and exploring design hierarchies. SimVision also offers features for debugging mixed-signal designs and supports various simulation formats.

Key Features and Use Cases:

- **Waveform Viewing:**

SimVision displays simulation waveforms, enabling users to observe signal transitions and values over time.

- **Timing Analysis:**

It helps identify circuit delays and timing issues by visualizing signal timings and potential bottlenecks.

- **Debugging:**

SimVision provides tools for tracing signals through the design, exploring hierarchies, and debugging errors by inspecting signal values and conditions.

- **Mixed-Signal Debugging:**

It includes features specifically for debugging designs that combine digital and analog components, such as tracing currents through hierarchical designs and analyzing mixed nets.

- **Design Hierarchy Navigation:**

SimVision allows users to navigate the design hierarchy, select specific signals or instances, and view waveforms or schematic representations.

- **Signal Tracing:**

Users can trace signals back to their drivers, identify the causes of signal behavior, and pinpoint errors in the design.

- **Source Code Debugging:**

SimVision integrates with source code debuggers, allowing users to inspect the code associated with a signal or instance and understand the logic behind its behavior.

- **Data Type Debugging:**

SimVision supports debugging fixed-point data types and other complex data types.

- **Class Debugging:**

SimVision allows users to view and debug class instances within the simulation, providing insights into object-oriented design behavior.

- **UVM Debugging:**

SimVision supports debugging Universal Verification Methodology (UVM) testbenches, allowing users to analyze class transactions and other UVM-specific debug features.

- **Frequency Analysis:**

SimVision can display the frequency of a signal in the waveform window, aiding in verifying clock and oscillator behavior.

- **Expression Calculator:**

SimVision's expression calculator allows users to perform boolean and other operations on signals, enabling more complex analysis.

- **Mnemonic Mapping:**

SimVision supports mapping signal values to meaningful strings (mnemonic maps), making it easier to understand signal behaviors.

- **Simulation Database:**

SimVision interacts with simulation databases (e.g., .trn files), enabling users to load simulation results and analyze them.

Common Tasks:

- **Loading Simulation Results:** Loading simulation waveforms from files (e.g., .trn files).
- **Selecting Signals:** Choosing which signals to display in the waveform window.
- **Navigating Hierarchy:** Exploring the design hierarchy to locate specific signals or instances.
- **Setting Time Markers:** Setting time markers to focus on specific simulation segments or events.
- **Using the Design Browser:** Using the design browser to browse the design hierarchy and select signals.

- **Using the Source Browser:** Using the source browser to view and debug source code.
- **Creating Expressions:** Using the expression calculator to perform signal calculations and create custom views.
- **Saving and Loading Waveforms:** Saving waveform configurations for later reuse or loading saved configurations.
- **Using the Command Line:** Using SimVision's command-line interface for more advanced debugging tasks.

In summary, SimVision is a powerful tool for debugging digital designs and mixed-signal simulations, offering a wide range of features for waveform analysis, signal tracing, and design exploration.

Verdi

Synopsys Verdi is a debug and verification platform used for finding errors and analyzing designs, particularly in the context of digital hardware development. It's known for its ability to help engineers quickly identify and fix bugs, analyze simulation results, and understand complex design behaviors. Verdi also integrates with other Synopsys tools, like VCS simulator, to provide a unified environment for verification tasks.

Here's a more detailed look at how Verdi is used:

1. Debugging and Verification:

- Verdi helps find and fix errors in designs by providing tools to analyze waveforms, trace signals, and examine the execution flow.
- It can be used in various stages of the design process, from initial verification to detailed debug.
- AI-powered features in Verdi can automate some debugging steps and assist in complex design environments, says Synopsys.

2. Coverage Analysis:

- Verdi can be used to view, analyze, and merge coverage data from various Synopsys tools, says Synopsys.
- This allows engineers to track the completeness of their verification efforts and identify areas where further work is needed.
- Verdi can also integrate with coverage tools like VC Formal to analyze coverage from different sources, according to Synopsys.

3. Verification Planning and Test Execution:

- Verdi can be used to plan and execute simulation regressions, says Synopsys.
- It can also help manage and analyze the results of those regressions.
- Verdi's integration with tools like VCS allows for seamless test execution and result analysis.

4. Design Understanding:

- Verdi can be used to understand the structure and behavior of a design, including tracing signals, identifying drivers and loads, and exploring the source code.
- It can also help in understanding complex designs by providing tools for filtering, searching, and navigating the design hierarchy, says Synopsys.

5. AI-Powered Debugging:

- Verdi includes AI-based tools like Regression Debug Automation (RDA) to automate the process of finding root causes of failures in the design and testbench, according to Synopsys.
- This helps to reduce the time and effort required for debugging, especially in complex and large designs, says Synopsys.

formal verification?

Getting Started With Formal Verification

Need for Higher level of Validation

As design complexity is increasing, the goal of 100% functional coverage becomes harder to achieve even after using constrained random stimulus and directed scenarios, therefore there is a need to adopt new methods of validation.

The verification engineering team tries to break down the RTL design into pieces and analyse them – independently and then in sync. This takes time! Design verification takes almost 80% of the overall time spent on chip development and verification engineers lose hair, sleep, peace, patience and weekends on tight deadlines to deliver the quality of verification.

Each miss from the verification team can cost a potential revenue loss of millions. The bugs can result in rolling back all the produced silicon from the market. One example was when Samsung rolled back its Galaxy Note 7. Turns out the device was failing due to battery short circuit, which was missed during the verification of the design. RTL designing and verification is a tricky task because we can’t give an update or version upgrade like what happens in the software industry and we can’t translate our design to silicon for each update in design. Hence, we need to verify and thoroughly analyse the design before actually

sending the design to foundry. As we can rightly guess, this is a very expensive process both in terms of time and money.

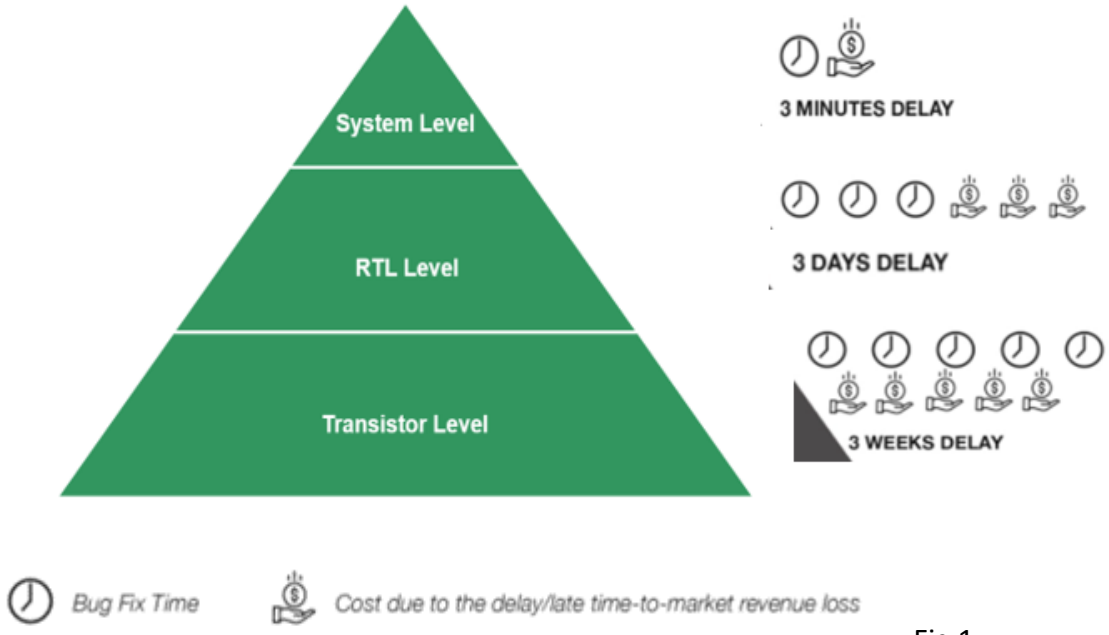


Fig 1

The figure above shows how the delay in the making of a chip affects the cost. This shows that as many bugs as possible needs to be removed in the early stages and new design errors should not be introduced while refining the design.

Introduction to Formal Verification

Formal verification uses mathematical models/methods to prove or disprove the correctness of the system's design with respect to formal specifications expressed as properties to verify the design thoroughly. It covers all the possible values a design variable can take and hence generating all the possible scenarios and stimulus for the design. We can theoretically achieve 100% coverage, which is a huge boost to the confidence of the design.

Formal verification is a technique used in different stages in ASIC project life cycle like front end verification, Logic Synthesis, Post Routing Checks and also for ECOs. But when we delve deeper, the formal verification used for verifying RTLs is entirely different from others.

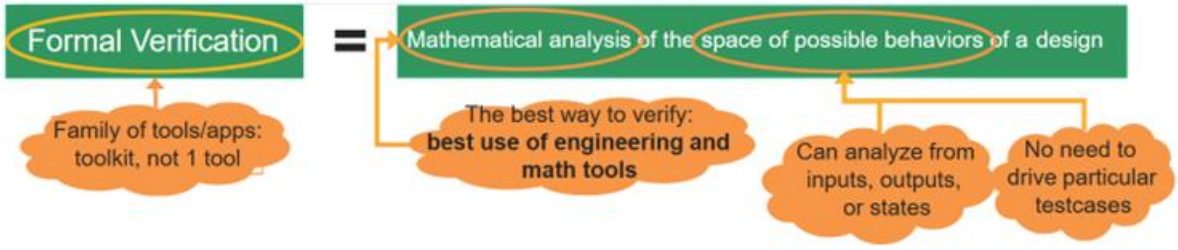


Fig 2

Formal Verification Techniques:

The following figure illustrates the various formal verification techniques:

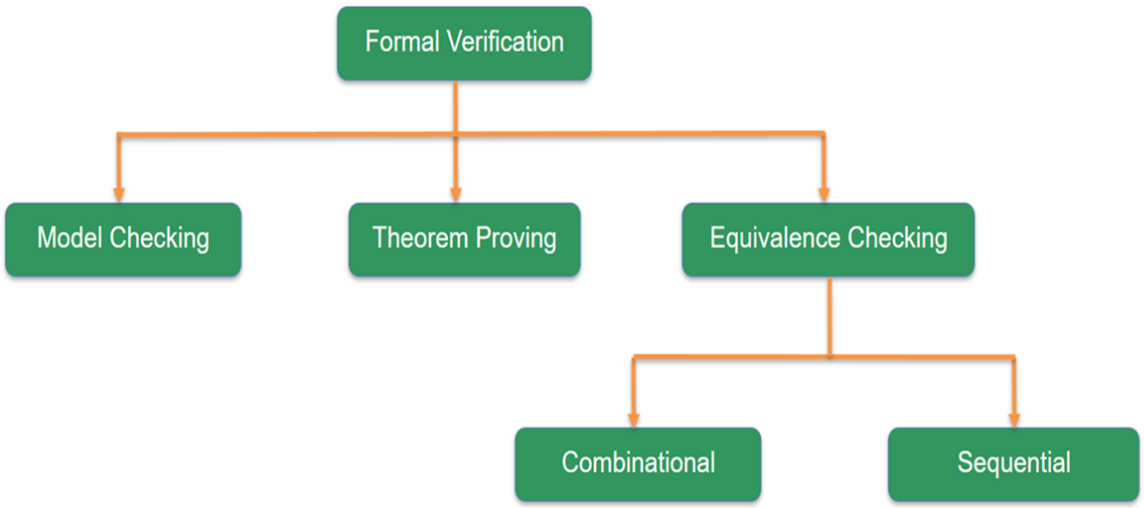


Fig 3

What is Formal Property Verification (FPV)?

What is formal property verification? A natural language such as English allows us to interpret the term formal property verification in two ways, namely:

- Verification of formal properties, or
- Formal methods for property verification(widely used interpretation by formal engineers)

FPV is the simplest form of formal verification; it is a method to prove the correctness of a design or show root cause of an error by rigorous mathematical procedures. This does not mean that the user must be a mathematician. It does not require test benches or stimuli and turnaround time is very less.

How FPV is Done?

In practice, there are two ways in which property verification is done today. These are static Assertion-based Verification (ABV) and dynamic Assertion-based Verification (ABV). In both forms, formal properties specify the correct requirements of the design, and the goal is to check whether a given implementation satisfies the properties. Static ABV techniques formally verify whether all possible behaviors of the design satisfy the given properties. Dynamic ABV is a simulation-based approach, where the properties are checked over a simulation run – the verification is thereby confined to only those behaviors that are encountered during the simulation.

Property checking can be done by using either using property languages (for example, ITL Interval Language) or Assertion languages (SVA, PSL, etc.). SVA is the assertions subset of the

System Verilog language. Assertions or properties are primarily used to validate the behavior of a design and can be checked statically by property checker tool and proves whether or not a design meets its specifications. A holding SVA/ITL/PSL means that the assertion/property has been formally and exhaustively checked and it holds in all possible traces of the design. A failing SVA/ITL/PSL means that a counterexample was found that represents a violation of the intended design behavior.

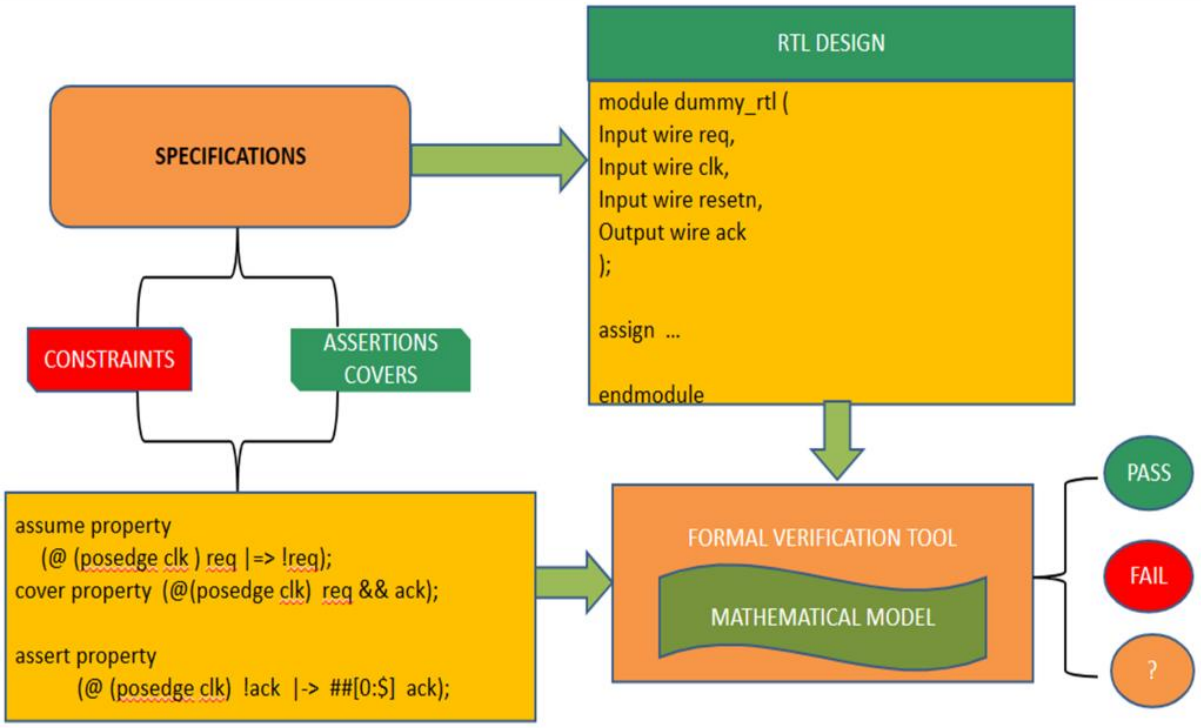


Fig 4: High level Architecture view for formal property verification

Formal verification tool automatically uses its magic to generate the stimulus and implicitly covers all the cases. The only requirement of Formal tool is to give it the RTL design and a formal description of the specifications in form of PROPERTIES for covering all the input and output combinations exhaustively. Basically Formal Verification works on the principle of **"failing to fail"** to prove the design's correctness. It generates all the possible stimulus and tries to fail our check. When it fails to fail – it states our design is correct else it stop immediately once a failure is found.

Let's understand this through an example: We need to verify a DUT. We have two options:

- We can generate possible input and output combinations on our own and write very specific test patterns to validate that the input we provided generates the expected

output. This is known as Directed Testing of the design of Directed Verification. We use SV, UVM test-benches to write specific testcases to check the DUT.

- The other option is – a tool generates all the possible inputs possible for our design and we only describe the behavior or relationship of input with the output. This description of behavior is done using SVA(system Verilog Assertions).

What type of designs can be easily verified using FPV?

Fig 5

Key Differences between Simulation and Formal:

	Formal	Simulation
Scope	Formal Proves absence of Bugs	Simulation can only detect Bugs
Inputs	User specifies only illegal stimulus	User creates given stimulus set
Testbench	TB is a set of properties connected to design	TB is a complicated wrapper around design
Assertions	Checks correctness for any possible input stimuli	Checks correctness for given input stimuli
Assumptions	Restricts the set of stimuli for correctness checking	Restricts the set of stimuli for correctness checking
Cover	Checks whether there is any feasible stimuli is covering the property	Checks whether the driven input stimuli is covering the property

Fig 6

What are Formal Engines?

Just like the engines of a car or any other vehicle. Formal engine is the actual heart and soul of the Formal Tool. It is the driving force. It determines how exhaustively the tool is verifying. It contains the algorithm which mathematically proves the correctness of our design.

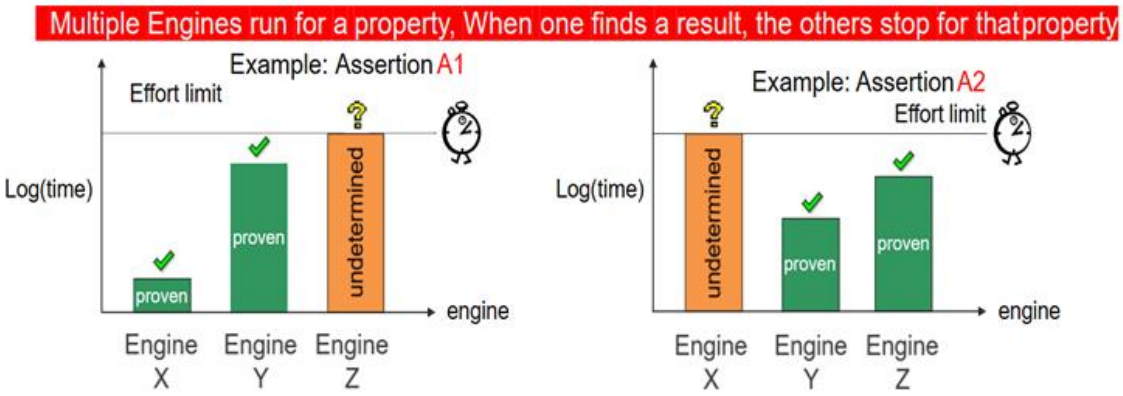


Fig 7: Graphical representation of various engine's performance

What are the EDA tools and supported apps available?

There are many tools in the industry which uses formal methodology to achieve a particular purpose.

- **Synopsys:**Synopsys provides VC Formal tool which covers a wide range of formal applications such as assertion-based verification, connectivity verification, sequential verification, etc. There is VC LP which is mainly used to verify formally the low poour design intent.
- **Siemens/Mentor Graphics**: Siemens/Mentor Graphics Questa tool also provides formal verification specific applications such as Questa Connectivity check, Questa post silicon debug, Questa property check, Questa Register check, etc.
- **JasperGold from Cadence:** Cadence JasperGold tool supports many 'Apps' through which certain formal verification tasks can be performed very easily. Along with that, for standard interfaces, assertions/properties can be generated automatically or readily available as Assertion Based Verification IPs (ABVIP).
    - Formal Property Verification (FPV)
    - Unreachability Analysis (UNR)
    - Pin connectivity verification (CONN)
    - Configuration register verification (CSR)
    - AMBA Interface protocol compliance verification using ABVIPs (FPV)
    - Formal Coverage (COV)
    - Formal Superlint/ Auto Formal Lint (AFL)
    - Sequential RTL equivalence (SEC)
    - Security Path Verification (SPV)
    - Functional Safety Verification (FSV)

Fig 8: View of Supported JasperGold apps

FPV Flow, Environment and Setup

Formal verification tools try to prove design correctness by analysing the space of possible behaviors of a design with static analysis algorithms, without simulating the design behavior over time. Therefore, formal techniques do not need a traditional testbench with dynamic tests/stimuli. Before starting formal verification, we must specify design intent, usually in the form of properties or assertions. This gives the tool a formal basis to reason about the design, and to identify violations that signify problems or bugs.

Formal tool doesn't require a big SV/UVM based testbench. The tool creates a mathematical model for both the design and the specified assertions and try to prove one model against the other. In the process, it automatically checks whether the defined assertion is valid for all the legal possible stimuli defined by the set of user constraints.

Formal tool requires the clocks and resets to be specified before the actual formal analysis/verification begins. Formal tool can infer clocks based on how they are used in the design. But specifying clocks explicitly would help the tool analyse the sequential behavior of the design accurately. Also specifying the reset would help the tool apply reset for a few cycles and reset the design to a known state. And the other way of getting to a known state is to load waveforms from the simulation. This would help in reducing the non-deterministic values on non-resettable flops for formal analysis. More the non-deterministic values, the formal analysis would be more pessimistic. It would be good to reduce the state space by initializing the design using waveforms.

Sometimes we may need control over the state space. Legalize our state space by specifying constraints. Constraints direct the formal tool to choose values that follow the constraints. If we don't specify the constraints, tool tries to mathematically prove the assertion for all the possible stimuli which may not be the desired behavior.

Formal verification tools ensure faster run times for comparatively small designs. However, adding assertions can take some time before doing actual verification. Also, time taken for establishing a given proof (assertions) can increase exponentially with the size of the design under consideration. We can direct the tool on what engine it should use and how much time it needs to spend on proving assertions, etc.

FPV Flow and Setup

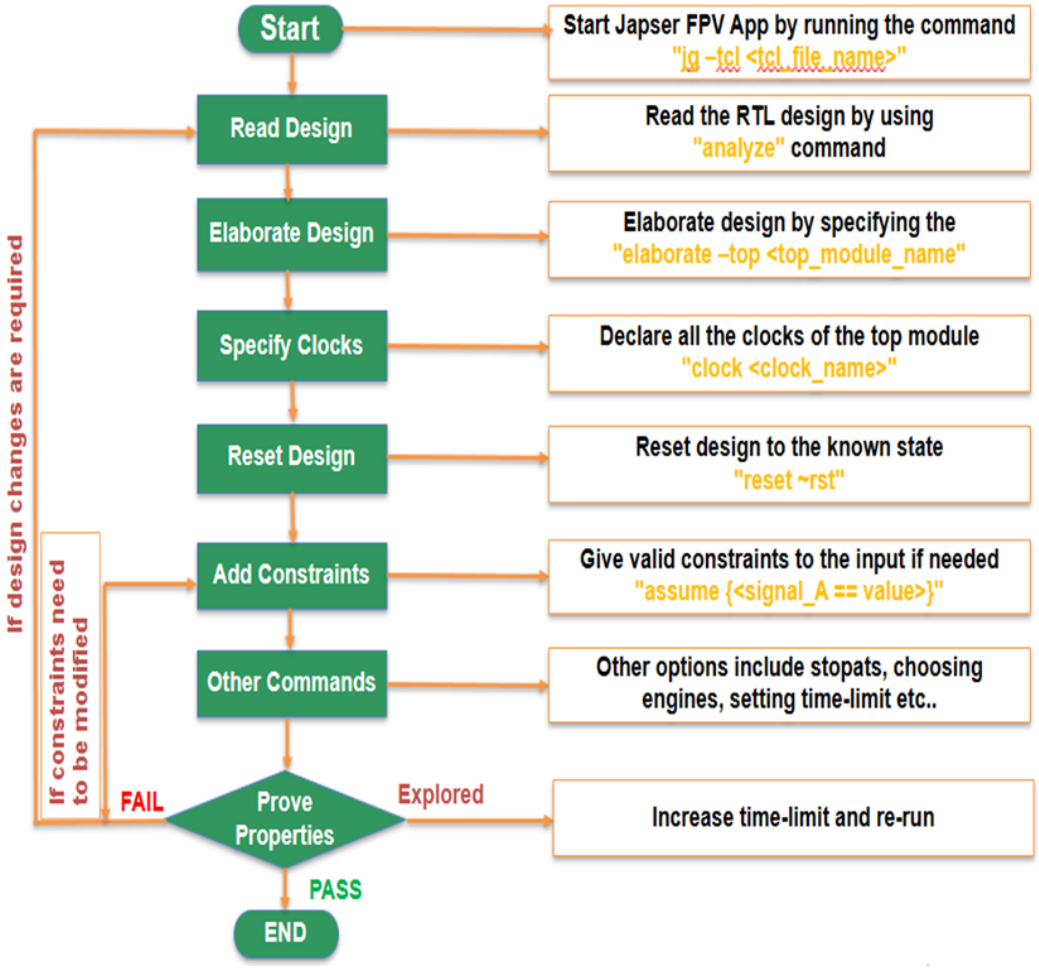


Fig 9: High Level Flowchart for formal property verification

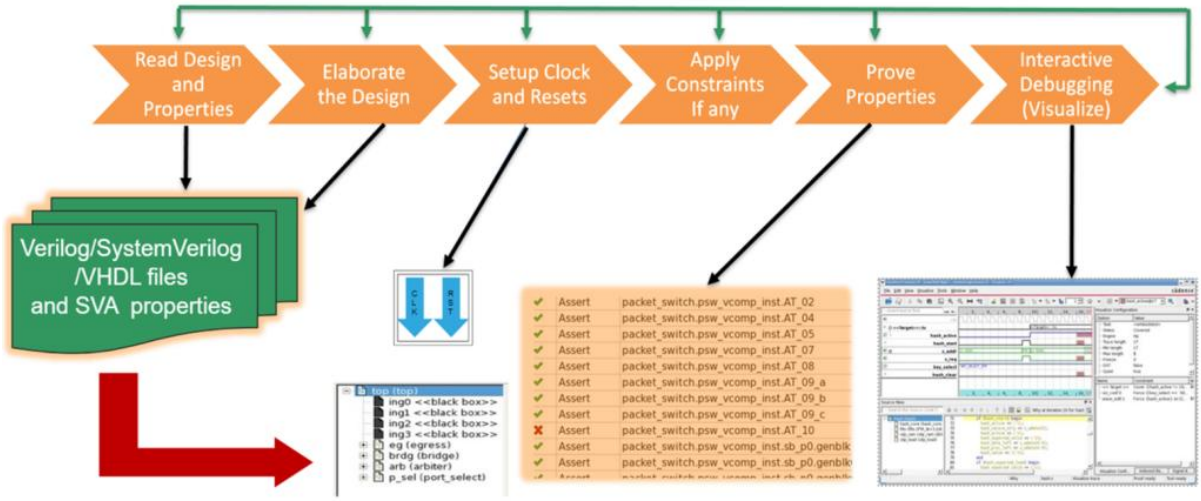


Fig 10

Sample TCL File

A sample TCL file.

```
# Read design
analyze -sv09 \
  +libext+.v+.sv+.svp+.psl \
  /proj/Phoenix/RevCtl/src/tapeout_r0.2_with_DTCP/xcore/XCORE.v \
  -y /proj/Phoenix/RevCtl/src/tapeout_r0.2_with_DTCP/accelerator \
  +incdir+/proj/Phoenix/RevCtl/src/tapeout_r0.2_with_DTCP/xcore \
analyze -sv09 \
  -psl /nfs/cadwa/pravind2/active/iev_evaluation/phoenix_issue/vunit.psl \

# Elaborate design
elaborate -top XCORE -disable_auto_bbox

# Specify clocks
clock PLL_CK
clock HCLK_IN -factor 2

# reset design using expressions or reset design using waves
reset -shm /proj/Phoenix/pwa/ssundar/misr/misr_code_and_DMA_i-0-0.trn -time 500 -time_scale ns -hier_path simenv.MAKO.XCORE

# Specify Constraints if needed
assume  { RESET_B == 1 }
assume  { IRESETB == 1 }

# setting engine
set_engine_mode TRI

# setting the time limit
set_prove_time_limit 72h
set_prove_per_property_max_time_limit 1h

# disabling and enabling the assertions
assert -disable {^.*$} -regexp
assert -enable XCORE.asrt_inst.my_check1

# proving the assertions
prove -asserts -all
```

Fig 11

Advantages of FPV over Simulation approach

- Verification is mostly fully automatic except for initial set-up, property coding
- Verification can be started as soon as RTL is ready. It need not wait for the readiness of complete RTL. It can start the analysis even with basic functionalities:

    - Reset Analysis
    - Sanity checks, etc.
- As testbench is not required, stimulus generation will be automatically managed by the FA tool with or without constraints around it
- The prove will be extremely fast compared to dynamic simulations.
- Simulations usually takes more time to detect the corner cases, FSM deadlock conditions. Sometimes it need additional metrics (coverage data) to identify the corners. But FA tool can easily find and reach those cases in quick time (fraction of minutes)
- Both black box and white box approach can be used
- There is no specific testbench required to drive stimulus to the DUT. Thus, formal can be applied to the designs in very early phases of the project.
- Gives a higher level of confidence than simulation alone. Formal should be considered as complementary to simulation rather than replacing simulations entirely.
- Can replace millions of simulation cycles.
- Allows a more concrete mapping of specifications/design-intent to the actual design.

Limitations of Formal Verification:

Even though formal verification has many advantages it does have certain disadvantages:

- As the design size grows, the state-space increases exponentially. So, when compared to a design with 2048 flops, a design with 2049 flops will have an exponential increase in the state space. Hence as the design size increases, quickly we reach an upper limit where the machine cannot handle the size of the design or the time taken to explore the state space starts to reach a point of diminishing returns. This is called the problem of state-space explosion.
- FA is not suitable for data paths functions except for simple functions. This is true especially for algorithmic blocks
- FA is good for verifying a system but not for validating the system (As Validation usually can be done only dynamically and formal verification is a static process).
- FA can only handle RTL logic. Any analog block will get black boxed or can be removed from the DUT during FA

Conclusion:

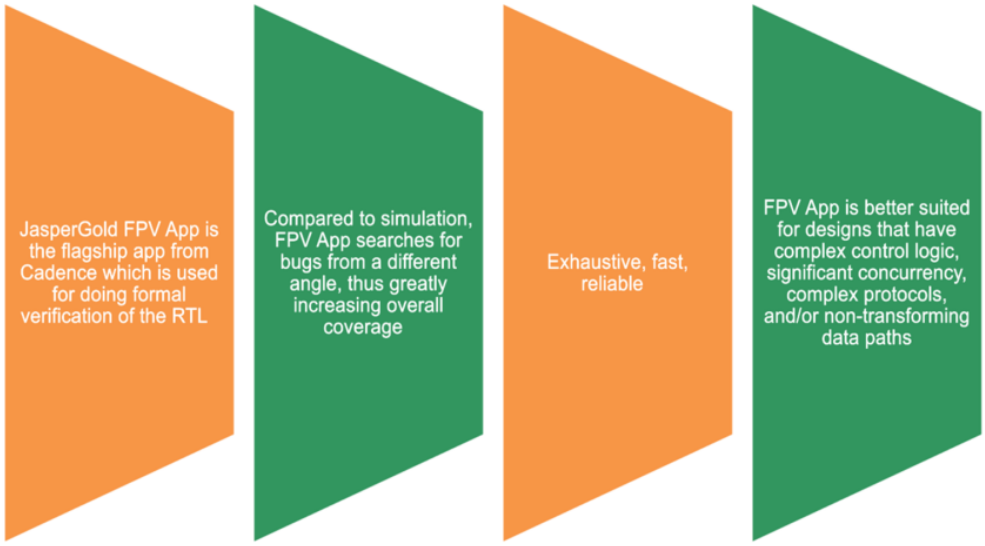


Fig 12

Tools comparison from cadence, synopsys, mentorgraphics (siemens eda):

Cadence, Synopsys, and Siemens EDA (formerly Mentor Graphics) are the major players in the Electronic Design Automation (EDA) software market, each offering a suite of tools for IC design. Synopsys and Cadence are generally considered market leaders, with Synopsys slightly ahead in revenue. Siemens EDA offers a strong suite of tools, particularly value-oriented for smaller teams or less complex projects.

Here's a more detailed comparison:

Synopsys:

- **Fusion Design Platform:** Integrates tools for synthesis, verification, and physical design.
- **Design Compiler:** Synthesizes RTL code into optimized gate-level netlists, including power optimization.
- **IC Compiler II:** Offers concurrent placement, routing, and clock tree synthesis, optimized for advanced process nodes.
- **PrimeTime:** Industry-standard tool for static timing analysis (STA).
- **VCS:** High-performance logic simulation with support for SystemVerilog and UVM.
- **Strengths:** Broad portfolio, strong market leadership, recognized as a leader in synthesis and verification.
- **Weaknesses:** Can be more expensive and require a larger team.

Cadence:

- **Genus Synthesis Solution:** Tailored for large and complex designs, with emphasis on early PPA optimization.
- **Virtuoso:** Used for schematic entry, physical design, and verification, with strong support for custom ICs and analog/RF designs.
- **Assura:** Physical verification tool known for its extensive capabilities and rule decks.
- **Simulations:** Offers tools like JasperGold for verification.
- **Strengths:** Strong in analog and RF design, good tool for custom ICs, well-regarded for its physical verification capabilities.
- **Weaknesses:** May not be as dominant in all areas as Synopsys.

Siemens EDA (Mentor Graphics):

- **Value-Oriented:** Often a more cost-effective option, especially for smaller teams or less complex projects.
- **QuestaSim:** Popular simulator supporting multiple HDLs.
- **Calibre:** Physical verification tool known for its extensive commands and rule decks.
- **Tessent:** Design-for-test (DFT) solutions.
- **Strengths:** Provides a comprehensive suite of tools, good for FPGA designs.
- **Weaknesses:** May not have the same dominance in all areas compared to Synopsys and Cadence, and support infrastructure might be less extensive.

Key Differences and Considerations:

- **Market Share:**

Synopsys and Cadence are the dominant players, with Synopsys slightly ahead in revenue.

- **Cost:**

Siemens EDA can be a more budget-friendly option, while Synopsys and Cadence can be more expensive.

- **Design Flow:**

Synopsys emphasizes an integrated design flow with its Fusion Design Platform, while Cadence has a broader portfolio of tools, particularly in custom IC and analog/RF.

- **Verification:**

All three offer strong verification tools, but Synopsys VCS, Cadence JasperGold, and Siemens EDA QuestaSim are prominent options.

- **Specific Strengths:**

Cadence is strong in custom IC and analog/RF design, while Siemens EDA excels in FPGAs.

In conclusion: Choosing the right EDA tools depends on the specific project requirements, team size, budget, and design focus. Synopsys and Cadence offer broad, comprehensive solutions, while Siemens EDA provides a cost-effective alternative for smaller teams or less complex designs.

Waveform viewers from cadence, synopsys, siemens:

Cadence, Synopsys, and Siemens (formerly Mentor Graphics) all offer waveform viewers as part of their Electronic Design Automation (EDA) tool suites. Cadence's primary waveform viewer is SimVision, while Synopsys uses Verdi and Custom WaveView, and Siemens (ModelSim) has a built-in waveform viewer.

Detailed Breakdown:

- **Cadence:**

Cadence's primary waveform viewer is SimVision. It's a powerful tool for debugging and analyzing complex digital designs, as well as mixed-signal designs. SimVision integrates with Cadence's verification environment and allows users to probe signals, analyze waveforms, and debug their designs.

- **Synopsys:**

Synopsys offers several waveform viewers, with Verdi being the primary debugging tool for functional verification. It's known for its advanced debugging capabilities, including waveform analysis, assertion checking, and fault tracing. Synopsys also uses Custom WaveView for analog and mixed-signal designs, allowing users to analyze waveforms in both time and frequency domains.

- **Siemens EDA (formerly Mentor Graphics):**

Siemens' ModelSim simulator includes a built-in waveform viewer, allowing users to visualize signals during simulation. ModelSim is a popular choice for HDL simulation and verification, and its waveform viewer supports VHDL, Verilog, and SystemVerilog designs.

Key Considerations:

- **Tool Integration:**

Waveform viewers are typically integrated with the EDA tools' overall design flow. This allows seamless debugging and analysis throughout the design process.

- **Functionality:**

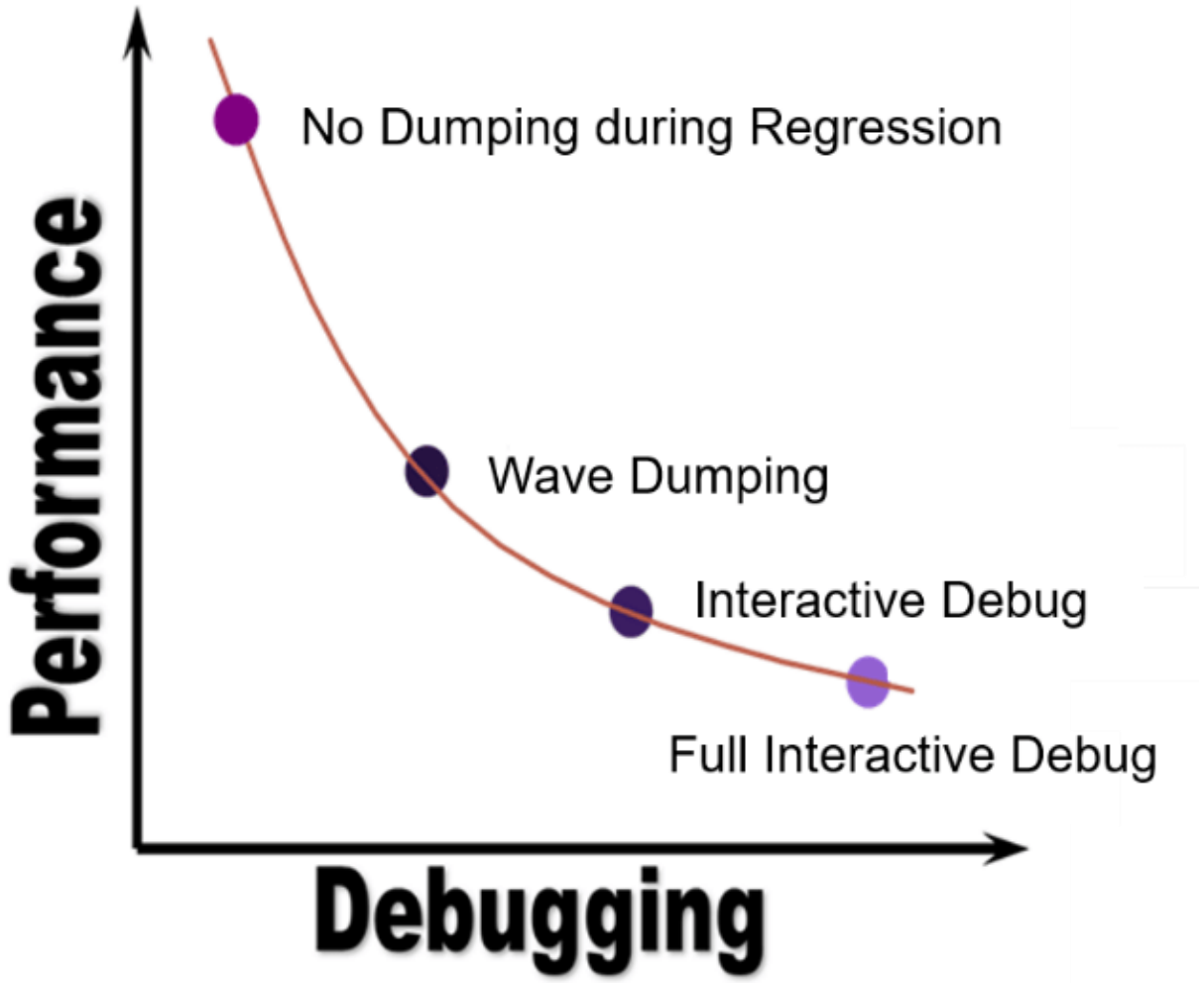
Waveform viewers offer various features, including signal probing, waveform analysis, assertion checking, and debugging.

- **Target Designs:**

Some waveform viewers are specifically designed for digital designs, while others are better suited for analog or mixed-signal designs.

For a more comprehensive understanding, it's recommended to consult the documentation of each tool and explore its specific features and capabilities.

How to increase debugging effort?



`uvm_field_* macros that were used between *_begin and *_end utility macros are basically called field macros since they operate on class properties and provide automatic implementations of core methods like copy, compare and print. This helps the user save some time from implementing custom do_copy, do_compare and do_print functions for each and every class. However, since these macros are expanded into general code, it may impact simulation performance and are generally not recommended.

Usage Example

```
class ABC extends uvm_object;
        rand bit [15:0]         m_addr;
        rand bit [15:0]         m_data;

        `uvm_object_utils_begin(ABC)
                `uvm_field_int(m_addr, UVM_DEFAULT)
                `uvm_field_int(m_data, UVM_DEFAULT)
        `uvm_object_utils_end

        function new(string name = "ABC");
                super.new(name);
        endfunction
endclass
```

The `uvm_field_* corresponding to the data type of each variable should be used. For example, variables of type int, bit, byte should use `uvm_field_int, while variables of type string should use `uvm_field_string and so on. The macro accepts atleast two arguments: *ARG* and *FLAG*.

```
uvm_report_info (get_type_name (), $sformatf ("Debug level message"), UVM_DEBUG);
uvm_report_warning (get_type_name (), $sformatf ("Warning level message"));
uvm_report_error (get_type_name (), $sformatf ("Error level message"));
uvm_report_fatal (get_type_name (), $sformatf ("Fatal level message"));
`uvm_info (get_type_name (), $sformatf ("[Driver] Debug level message"), UVM_DEBUG)
```

`uvm_warning (get_type_name (), $sformatf ("[Driver] Warning level message"))

`uvm_error (get_type_name (), $sformatf ("[Driver] Error level message"))

`uvm_fatal (get_type_name (), $sformatf ("[Driver] Fatal level message"))

There are four basic reporting functions that can be used with different verbosity levels.

uvm_report_* ("TAG", $sformatf ("[Enter the display message]"), VERBOSITY_LEVEL);

where * can be either **info, error, warning, fatal**. UVM has six levels of verbosity with each one represented by an integer.

```
typedef enum {
  UVM_NONE   = 0,
  UVM_LOW    = 100,
  UVM_MEDIUM = 200,
  UVM_HIGH   = 300,
  UVM_FULL   = 400,
  UVM_DEBUG  = 500
} uvm_verbosity;
```

Note that the VERBOSITY_LEVEL is only required for uvm_report_info. Usage of uvm_report_fatal will exit the simulation.

clock and reset testcases?

Introduction

Writing a testbench is as complex as writing the RTL code itself. These days ASICs are getting more and more complex and thus

verifying these complex ASIC has become a challenge. Typically 60-70% of time needed for any ASIC is spent on verification/validation/testing. Even though the above facts are well known to most ASIC engineers, still engineers think that there is no glory in verification.

I have picked up some examples from the VLSI classes that I used to teach during 1999-2001, when I was in Chennai. Please feel free to give your feedback on how to improve the tutorial below.

Before you Start

For writing testbenches it is important to have the design specification of "design under test" or simply DUT. Specs need to be understood clearly and a test plan, which basically documents the test bench architecture and the test scenarios (test cases) in detail, needs to be made.

Example - Counter

Let's assume that we have to verify a simple 4-bit up counter, which increments its count whenever enable is high, and resets to zero when reset is asserted high. Reset is synchronous to clock.

Code for Counter

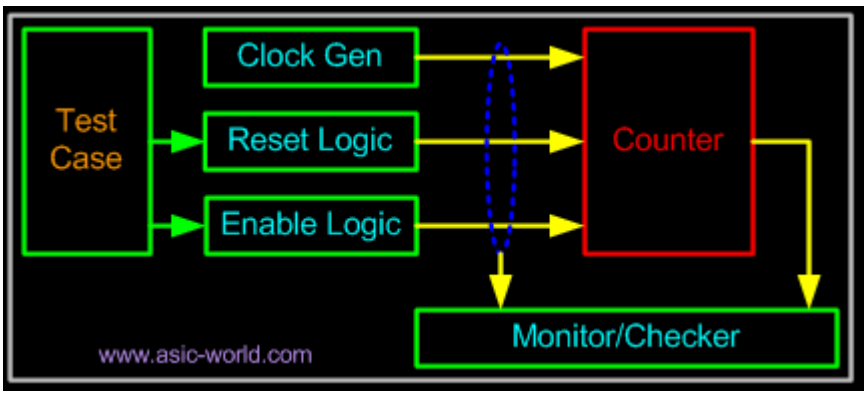
```
1 //-------------------------------------------------
2 // Design Name : counter
3 // File Name   : counter.v
4 // Function    : 4 bit up counter
5 // Coder       : Deepak
```

```
 6 //------------------------------------------------------
 7 module counter (clk, reset, enable, count);
 8 input clk, reset, enable;
 9 output [3:0] count;
10 reg [3:0] count;
11
12 always @ (posedge clk)
13 if (reset == 1'b1) begin
14   count <= 0;
15 end else if ( enable == 1'b1) begin
16   count <= count + 1;
17 end
18
19 endmodule
```

You could download file counter.v here

Test Plan

We will write a self-checking test bench, but we will do this in steps to help you understand the concept of writing automated test benches. Our testbench environment will look something like the figure below.



DUT is instantiated in the testbench, and the testbench will contain a clock generator, reset generator, enable logic generator and compare logic, which basically calculates the expected count value of counter and compares it with the output of counter.

Test Cases

- Reset Test : We can start with reset de-asserted, followed by asserting reset for few clock ticks and deasserting the reset, See if counter sets its output to zero.
- Enable Test : Assert/deassert enable after reset is applied.
- Random Assert/deassert of enable and reset.

We can add some more test cases; but we are not here to test the counter, rather to learn how to write test benches.

Clock and reset test cases are crucial for verifying the proper functionality and stability of digital circuits. They ensure that the circuit operates correctly under various clock frequencies and reset conditions, including different reset timings and sequencing.

Clock Test Cases:

- **Clock Frequency:**

Test the circuit with different clock frequencies to ensure it can handle varying speeds and that the timing constraints are met.

- **Clock Skew:**

Simulate clock skew, the temporal difference between the arrival of the same clock edge at different parts of the chip, to verify the design's robustness.

- **Clock Domain Crossing (CDC):**

If the design has multiple clock domains, test cases are needed to ensure that data transfer between different clock domains is correct and no glitches or metastability issues occur.

- **Clock Gating:**

Test cases should verify that clock gating logic, which controls clock signals, works correctly and doesn't introduce any timing issues or logic errors.

- **Edge Sensitivity:**

Test whether the design responds correctly to rising or falling edges of the clock signal.

Reset Test Cases:

- **Reset Timing:**

Test the circuit by applying reset signals at different times and for different durations to verify that the reset logic works correctly and the circuit initializes properly.

- **Reset Sequencing:**

If the design has multiple reset signals or resets are applied in sequence, test cases are needed to ensure that the reset order is followed correctly and that the circuit initializes in the expected state.

- **Reset Removal:**

Verify that the circuit functions correctly after the reset signal is removed and the circuit starts operating normally.

- **Reset Persistence:**

Test cases should ensure that the reset signal is held for long enough to allow the circuit to initialize correctly and that it is not prematurely removed.

- **Asynchronous Reset:**

If the reset signal is asynchronous (not synchronized with the clock), test cases are needed to verify that it doesn't introduce metastability or glitches.

General Considerations:

- **Testbench:**

A well-designed testbench is crucial for generating the necessary stimuli and verifying the outputs of the design under test.

- **Coverage:**

Test cases should aim to cover a wide range of scenarios, including normal operation, edge cases, and potential error conditions.

- **Coverage metrics:**

Coverage metrics can be used to assess the completeness and thoroughness of the test suite.

By implementing comprehensive clock and reset test cases, you can ensure that your digital design is robust, reliable, and meets the required functional and timing specifications.

No response of messages

When someone doesn't respond to your messages, it could be due to a variety of reasons, ranging from being busy to simply being disengaged. It's important to consider the context and your relationship with the person before jumping to conclusions.

Here's a breakdown of possible reasons and how to handle them:

Possible Reasons:

- **Busyness:**

They might be genuinely busy with work, personal commitments, or other obligations that prevent them from replying immediately.

- **Distraction:**

They could have been distracted by something else and simply forgot to respond.

- **Lack of Interest:**

Sometimes, a lack of response can indicate disengagement or disinterest in the conversation.

- **Technological Issues:**

Their phone might be off, they might be in a dead zone, or their phone's battery might be dead.

- **Emotional Factors:**

They might be feeling overwhelmed, anxious, or have underlying anxieties that make them bad at replying.

- **Uncertainty:**

They might not know what to say or how to respond.

- **Personal Choice:**

They might simply choose not to respond, perhaps due to feeling taken for granted or wanting to maintain their privacy.

How to Handle No Response:

- **Wait and see:**

Give them a reasonable amount of time to respond before reaching out again.

- **Consider the context:**

If it's a serious matter, you might want to reach out through a different channel, like a phone call or email.

- **Don't assume the worst:**

It's easy to get upset or offended, but try to remember that there could be a valid reason for their silence.

- **Try a follow-up:**

A polite follow-up message can gently nudge them to respond without being overly pushy.

- **Consider your relationship:**

If it's a close friend or partner, you might want to reach out and inquire if they're okay.

- **Move on:**

If you've given them a reasonable amount of time and they haven't responded, you might need to accept that they're not interested in communicating with you.

In summary: No response can be a complex issue with multiple potential causes. By considering the context, being patient, and avoiding assumptions, you can navigate the situation more effectively.

A "no response" message in electronics, particularly in communication systems like CAN or LIN buses, indicates a failure in communication between devices. This can occur due to various reasons, including network errors, hardware malfunctions, or issues with communication protocols. The message often specifies the device or component that is not responding.

Possible Causes and Troubleshooting:

- **Network Errors:**
    - **Transmission Errors:** CAN and LIN buses can experience transmission errors due to various factors like bit rate mismatches, checksum failures, or framing errors.
    - **No Node Responding:** If a subscriber message is sent and no nodes respond, a transmission error is logged.
    - **Bus Overload:** Excessive traffic or long cable lengths can overload the bus, leading to communication failures.
- **Hardware Issues:**
    - **Faulty Components:** A faulty module or device on the bus can prevent it from responding to messages.

    - **Power Supply Problems:** Inadequate power supply to a device can also lead to communication failures.
    - **Physical Connections:** Loose or damaged wires or connectors can interrupt communication.
- **Protocol Issues:**
    - **Incorrect Configuration:** If the communication parameters (e.g., bit rate, ID) are not configured correctly, devices may not be able to communicate.
    - **Software Errors:** Software errors in the nodes or communication software can prevent them from responding to messages.

Troubleshooting Steps:

1. **Verify the Bus Termination:** Ensure the bus is properly terminated at both ends.
2. **Check Device Power:** Ensure all devices on the bus have sufficient power.
3. **Inspect Connections:** Examine all wires and connectors for damage or loose connections.
4. **Test with a Known Good Device:** Try replacing the non-responding device with a known working device to isolate the problem.
5. **Check Communication Parameters:** Verify that the communication parameters (e.g., bit rate, ID) are configured correctly.
6. **Check for Software Errors:** If the problem is software-related, try updating or troubleshooting the software.
7. **Use Diagnostic Tools:** Utilize CAN bus analyzers or other diagnostic tools to monitor the bus for errors and identify the source of the problem.

take 5 print outs of the updated or arm resume?

Digital circuits, clock domain crossing?

Clock Domain Crossing (CDC) in digital circuits refers to the process of transferring data between different clock domains, each operating under a different clock signal. This can cause issues like metastability and data loss, as the timing relationships between the clocks

may not be predictable. Synchronization techniques, like synchronizers and handshaking, are used to mitigate these problems.

Here's a more detailed explanation:

What is a Clock Domain?

A clock domain is a portion of a digital system that operates under the control of a specific clock signal. This clock dictates the timing of operations within that domain, including how data is sampled and shifted.

Why is CDC a Problem?

When data needs to be moved between different clock domains, it can lead to several issues:

- **Metastability:**

If a signal from one clock domain is sampled by a flip-flop in another domain at an uncertain time relative to the destination clock edge, it can enter an unstable metastable state, says ChipEdge VLSI Training Company.

- **Data Loss:**

If the incoming data is not held stable for long enough for the destination clock to sample it correctly, data can be lost or corrupted.

- **Data Incoherency:**

If the data is not properly synchronized between the domains, it can lead to inconsistent or incorrect data interpretation in the destination domain.

How to Handle CDC

Several techniques are used to manage CDC and reduce the risks associated with it:

- **Synchronizers:**

These circuits, often using multiple flip-flops, resample the incoming data in the destination clock domain, reducing the probability of metastability.

- **Handshaking:**

This method involves using signals to indicate when data is valid and has been received, ensuring that the data is properly transferred and synchronized.

- **FIFO Buffers:**

First-In, First-Out (FIFO) buffers can be used to store data between the domains, allowing the source and destination clocks to operate at different speeds.

- **Gray Coding:**

This technique uses a specific code that minimizes the number of bit changes between adjacent values, reducing the chance of metastability.

In summary, CDC is a crucial consideration in digital design, especially when dealing with complex systems and multiple clock domains. By understanding the potential problems and employing appropriate synchronization techniques, engineers can ensure the reliable and accurate transfer of data between these domains.

Computer architecture? and operating system?

Computer architecture and operating systems are fundamentally intertwined. Computer architecture defines the hardware components and their organization, while the operating system provides the software layer that manages these components and allows users to interact with the hardware. They work together to enable the execution of programs and the efficient operation of a computer system.

Computer Architecture:

- **Hardware Components:**

This includes the CPU, memory (RAM, ROM), storage devices (hard drives, SSDs), input/output devices (keyboard, mouse, display), and other physical parts of the computer.

- **Instruction Set Architecture (ISA):**

Defines the set of instructions the CPU understands and can execute. This determines the type of programming languages and software that can run on the system.

- **Memory Organization:**

How memory is organized and accessed, including memory hierarchy (cache, RAM, storage), virtual memory, and memory management techniques.

- **Input/Output (I/O) Systems:**

Handles communication between the computer and external devices, such as keyboard, mouse, and network interfaces.

Operating System (OS):

- **Resource Management:** Manages CPU time, memory allocation, file system access, and other resources to ensure efficient utilization and prevent conflicts.
- **User Interface:** Provides an interface (GUI or CLI) for users to interact with the computer and execute applications.

- **Process Management:** Handles the creation, execution, and termination of processes (programs).
- **File System:** Manages the organization and storage of files and directories on storage devices.
- **Network Management:** Handles communication with other devices on a network.

Relationship between Architecture and OS:

- **Hardware Dependence:**

The OS is designed to work with the specific hardware architecture of the computer. Changes in architecture can necessitate changes in the OS.

- **Abstraction:**

The OS provides an abstract layer that simplifies programming and allows developers to write applications without needing to directly control the hardware.

- **Performance:**

The efficiency of the OS and its interaction with the hardware directly impacts the overall performance of the system.

Examples:

- **Microkernel vs. Monolithic Kernel:**

These are different approaches to OS architecture, with microkernels offering better modularity and security, but potentially at the cost of performance.

- **Virtual Memory:**

The OS uses virtual memory to provide the illusion of a larger memory space than physically available, allowing more programs to run concurrently.

- **Interrupts:**

Hardware interrupts allow devices to signal the CPU when they need attention, which the OS handles to ensure efficient resource management.

cache and pipelined architecture?

Cache memory and pipelined architecture are two distinct concepts in computer architecture that both aim to improve system performance. Cache memory is a small, high-speed memory that stores frequently accessed data to reduce access time from slower main memory. Pipelining, on the other hand, is a technique that breaks down instruction

execution into stages, allowing multiple instructions to be in different stages of completion simultaneously.

Cache Memory:

- **Purpose:**

To store frequently accessed data and instructions, making them readily available to the CPU.

- **Mechanism:**

When the CPU needs data, it first checks the cache. If the data is found (a "cache hit"), it's retrieved quickly. If not (a "cache miss"), the CPU accesses main memory, which is slower.

- **Types:**

CPUs often use multiple levels of cache (L1, L2, L3), with L1 being the fastest and smallest, and L3 being the slowest and largest.

- **Benefit:**

Significantly reduces the time it takes to access data, leading to faster program execution.

Pipelined Architecture:

- **Purpose:**

To improve instruction execution throughput by overlapping the execution of different instructions.

- **Mechanism:**

The CPU is divided into multiple pipeline stages (e.g., fetch, decode, execute, write-back). Each stage performs a specific part of the instruction execution.

- **Benefit:**

Allows multiple instructions to be in different stages of execution concurrently, resulting in a higher rate of instruction completion.

- **Analogy:**

Similar to an assembly line, where different parts of a product are assembled at different stages by different workers.

Key Differences:

- **Purpose:**

Cache focuses on reducing memory access time, while pipelining focuses on improving instruction throughput.

- **Mechanism:**

Cache is a memory storage mechanism, while pipelining is an execution technique.

- **Impact:**

Cache improves individual instruction execution speed, while pipelining improves the overall rate of instruction completion.

Relationship:

- Cache can be pipelined, meaning the process of accessing data within the cache can be broken down into stages, further reducing the time it takes to access the data.
- Pipelining can be used in conjunction with cache to achieve even greater performance gains.

Semaphore, Mailbox, Queues, Tlm_fifos?

Semaphores, mailboxes, queues, and TLM FIFOs are all concepts related to managing communication and synchronization in concurrent or multi-threaded environments. Semaphores provide a way to control access to shared resources, mailboxes offer a buffered communication channel between two processes, queues are data structures that store items in a specific order (FIFO), and TLM FIFOs are specialized FIFOs used in Transaction-Level Modeling to buffer data between different parts of a simulation.

Semaphores:

- Semaphores are signaling mechanisms used to control access to shared resources.
- They allow threads to request tokens (e.g., semaphore.acquire()) and release them (e.g., semaphore.release()) to ensure that a specific number of threads can access a resource simultaneously.
- Semaphores can be used to implement mutual exclusion (locking a resource for exclusive access) or to control the number of concurrent users.

Mailboxes:

- A mailbox is a communication channel between two processes, acting as a buffered FIFO.
- The sender places data into the mailbox (e.g., using a put() method), and the receiver retrieves it (e.g., using a get() method).
- Mailboxes are often used in conjunction with semaphores to manage access to the queue and ensure that the receiver doesn't read from an empty mailbox.

Queues:

- Queues are a fundamental data structure that follows the First-In, First-Out (FIFO) principle.
- Elements are added to the back of the queue (enqueue) and removed from the front of the queue (dequeue).
- Queues are widely used in various applications, such as operating system scheduling, printing spooling, and data buffers.

TLM FIFOs:

- TLM FIFOs are specialized FIFOs used in Transaction-Level Modeling (TLM) to buffer data between different parts of a simulation.
- They are designed to handle data transfer between components operating at different speeds, allowing one component to send data independently of the other.
- TLM FIFOs provide blocking methods (get, put, peek) and non-blocking methods (try_put, try_get, try_peek) for interacting with the FIFO, says Siemens Verification Academy.
- They are commonly used in UVM (Universal Verification Methodology) verification environments.

Memory management in operating system?

Memory management in an operating system (OS) is the process of controlling and coordinating how computer memory is used to ensure that programs and the OS itself have the memory they need to run efficiently. It involves allocating memory to processes, keeping track of memory usage, and deallocating memory when it's no longer needed.

Here's a more detailed breakdown:

Key Functions:

- **Allocation and Deallocation:**

The OS determines which processes receive memory, how much they get, and when they get it. It also tracks when memory is freed and updates the status accordingly.

- **Memory Tracking:**

The OS keeps track of every memory location, whether it's currently allocated to a process or free.

- **Process Synchronization:**

When multiple processes share the same memory space, the OS uses synchronization mechanisms to ensure data consistency and prevent conflicts.

- **Memory Protection:**

The OS isolates processes from each other, preventing one process from interfering with the memory of another.

- **Logical and Physical Organization:**

The OS manages both the logical view of memory (how processes perceive it) and the physical organization of memory (how it's actually stored).

Common Techniques:

- **Contiguous Allocation:**

Processes are allocated memory as a single, continuous block.

- **Partitioning:**

Memory is divided into fixed-size or variable-size partitions, and processes are assigned to these partitions.

- **Paging:**

Memory is divided into fixed-size blocks called pages, and processes are allocated memory in terms of these pages.

- **Segmentation:**

Memory is divided into variable-size segments, which can be of different sizes, and processes are allocated memory in terms of these segments.

- **Swapping:**

Processes that are not currently running can be swapped out to disk and swapped back in when needed, allowing more processes to fit in memory at once.

- **Virtual Memory:**

A technique that allows processes to use more memory than is physically available by using disk space as an extension of RAM.

- **Memory Management Unit (MMU):**

A hardware component that translates logical addresses (used by processes) into physical addresses (used by the hardware).

Importance of Memory Management:

- **Efficiency:**

It ensures that memory is used efficiently, minimizing wasted space and maximizing resource utilization.

- **Stability:**

It prevents processes from interfering with each other, contributing to the overall stability of the system.

- **Performance:**

It allows more processes to run concurrently, leading to faster response times and improved system performance.

- **Security:**

It helps to protect system resources by isolating processes from each other.

In essence, memory management is a critical function of the operating system that enables efficient and reliable operation of a computer system.

Pages, and virtual memory management?

IPC - interprocess communication?

explain college project???

fsm design for counters?

program pattern recognition?

stack and heap?

Cache coherence - MESI Protocol?

Cache Memory

Cache memory is a special type of high-speed memory located close to the CPU in a computer. It stores frequently used data and instructions, So that the CPU can access them quickly, improving the overall speed and efficiency of the computer.

It is a faster and smaller segment of memory whose access time is as close as registers. In a hierarchy of memory, cache memory has access time lesser than primary memory. Generally, cache memory is used as a buffer.

In this article we will see cache memory in detail.

What is Cache Memory?

Data in primary memory can be accessed faster than secondary memory but still, access times of primary memory are generally in a few microseconds, whereas the CPU is capable of performing operations in nanoseconds. Due to the time lag between accessing data and acting on data performance of the system decreases as the CPU is not utilized properly, it may remain idle for some time. In order to minimize this time gap new segment of memory is Introduced known as Cache Memory.

It is based on principle of locality of reference, which refers to the observation that program tries to access a relatively small portion of their address space at any given time, and repeatedly tries to access some portion of the memory. For ex: In fees department of your college, transactions are accessed frequently to check on the dues.

Key Features of Cache Memory

1. **Speed:** Faster than the main memory (RAM), which helps the CPU retrieve data more quickly.
2. **Proximity:** Located very close to the CPU, often on the CPU chip itself, reducing data access time.
3. **Function:** Temporarily holds data and instructions that the CPU is likely to use again soon, minimizing the need to access the slower main memory.

Role of Cache Memory

The role of cache memory is explained below,

- Cache memory plays a crucial role in computer systems.
- It provide faster access.
- It acts buffer between CPU and main memory(RAM).
- Primary role of it is to reduce average time taken to access data, thereby improving overall system performance.

Benefits of Cache Memory

Various benefits of the cache memory are,

1. **Faster access**: Faster than main memory. It resides closer to CPU , typically on same chip or in close proximity. Cache stores subset of data and instruction.
2. **Reducing memory latency**: Memory access latency refers to time taken for processes to retrieve data from memory. Caches are designed to exploit principle of locality.
3. **Lowering bus traffic**: Accessing data from main memory involves transferring it over system bus. Bus is shared resource and excessive traffic can lead to congestion and slower data transfers. By utilizing cache memory , processor can reduce frequency of accessing main memory resulting in less bus traffic and improves system efficiency.
4. **Increasing effective CPU utilization**: Cache memory allows CPU to operate at a higher effective speed. CPU can spend more time executing instruction rather than waiting for memory access. This leads to better utilization of CPU's processing capabilities and higher overall system performance.
5. **Enhancing system scalability**: Cache memory helps improve system scalability by reducing impact of memory latency on overall system performance.

Working of Cache Memory

In order to understand the working of cache we must understand few points:

- Cache memory is faster, they can be accessed very fast
- Cache memory is smaller, a large amount of data cannot be stored

Whenever CPU needs any data it searches for corresponding data in the cache (fast process) if data is found, it processes the data according to instructions, however, if data is not found in the cache CPU search for that data in primary memory(slower process) and loads it into the cache. This ensures frequently accessed data are always found in the cache and hence minimizes the time required to access the data.

How does Cache Memory Improve CPU Performance?

Cache memory improves CPU performance by reducing the time it takes for the CPU to access data. By storing frequently accessed data closer to the CPU, it minimizes the need for the CPU to fetch data from the slower main memory.

What is a Cache Hit and a Cache Miss?

Cache Hit: When the CPU finds the required data in the cache memory, allowing for quick access.On searching in the cache if data is found, a cache hit has occurred.

Cache Miss: When the required data is not found in the cache, forcing the CPU to retrieve it from the slower main memory.On searching in the cache if data is not found, a cache miss has occurred

*Performance of cache is measured by the number of cache hits to the number of searches. This parameter of measuring performance is known as the **Hit Ratio.***

Hit ratio=(Number of cache hits)/(Number of searches)

Types of Cache Memory

1. **L1 or Level 1 Cache:** It is the first level of cache memory that is present inside the processor. It is present in a small amount inside every core of the processor separately. The size of this memory ranges from 2KB to 64 KB.
2. **L2 or Level 2 Cache:** It is the second level of cache memory that may present inside or outside the CPU. If not present inside the core, It can be shared between two cores depending upon the architecture and is connected to a processor with the high-speed bus. The size of memory ranges from 256 KB to 512 KB.
3. **L3 or Level 3 Cache:** It is the third level of cache memory that is present outside the CPU and is shared by all the cores of the CPU. Some high processors may have this cache. This cache is used to increase the performance of the L2 and L1 cache. The size of this memory ranges from 1 MB to 8MB.

Difference Between Cache and RAM

Although Cache and RAM both are used to increase the performance of the system there exists a lot of differences in which they operate to increase the efficiency of the system.

Feature	Cache Memory	RAM (Random Access Memory)
Location	Located close to the CPU.	Connected to the CPU via the memory bus.
Purpose	Stores frequently accessed data and instructions.	Serves as the main working memory for the CPU.
Speed	Very fast, with access times in nanoseconds.	Fast, but slower than cache memory, with access times in tens of nanoseconds.
Size	Smaller in size, typically measured in kilobytes (KB) to a few megabytes (MB).	Larger in size, ranging from gigabytes (GB) to terabytes (TB).

Feature	Cache Memory	RAM (Random Access Memory)
Type of Memory	Uses SRAM (Static RAM), which is faster but more expensive.	Uses DRAM (Dynamic RAM), which is slower but more cost-effective.
Accessibility	Extremely fast access times due to proximity to the CPU.	Slightly slower access times compared to cache memory.
Cost	More expensive per unit of memory due to its speed and proximity to the CPU.	Less expensive per unit of memory compared to cache memory.
Hierarchy	Typically organized into multiple levels (L1, L2, L3), with each level increasing in size and latency.	Single level, serving as the primary working memory for the CPU.
Usage	Acts as a buffer between the CPU and main memory (RAM), speeding up data access.	Used for storing data and instructions currently being processed by the CPU.
Capacity	Limited capacity due to its small size and high-speed nature.	Larger capacity, providing ample storage space for running applications and processes.

Conclusion

In conclusion, cache memory plays an important role in enhancing the speed and efficiency of computer systems. By storing frequently accessed data and instructions close to the CPU, cache memory minimizes the time required for the CPU to access information, thereby reducing latency and improving overall system performance.

Cache Coherence Protocols in Multiprocessor System

Prerequisite – Cache Memory
In multiprocessor system where many processes needs a copy of same memory block, the maintenance of consistency among these copies raises a problem referred to as **Cache**

Coherence Problem.
This occurs mainly due to these causes:-

- Sharing of writable data.
- Process migration.
- Inconsistency due to I/O.

Basic Schemes in Enforcing Coherence Protocol

The coherence problem for multiprocessor, the I/O although similar in origin, has different characteristics that affect the appropriate solution. In a coherent multiprocessor, the caches provides both migration & replication of shared data items.

Coherent caches also provide migration, since a data item can moved to local cache and used there in a transparent fashion.

This migration reduces both the latency to access a shared data items that allocated remotely and bandwidth demand on shared memory. Multiprocessor adopt a hardware solution by introducing a protocol to maintain coherent caches. The protocol to maintain coherent for multiple processor are called coherence protocol.

There are two classes of protocols in use each of which uses different techniques to track the sharing status.

1)Directory based

2)snooping

1)Directory based

The sharing status of a particular block of physical memory is kept in one location called the directory. There are different types of directory based cache coherence.

Associated with the memory or some other single serialization point, such as the outermost cache in a multicore.

2)snooping

Rather than keeping a state of sharing in a single directory, every cache that has of copy of data from a block of physical memory could track the sharing of the status of the block.

Snooping can also be used as the coherence protocol for a multichip multiprocessor and some design support a snooping protocol on top of a directory protocol with each multicore.

Cache Coherence Protocols:
These are explained as following below:

1. MSI Protocol:

This is a basic cache coherence protocol used in multiprocessor system. The letters of protocol name identify possible states in which a cache can be. So, for MSI each block can have one of the following possible states:

- **Modified –**
  The block has been modified in cache, i.e., the data in the cache is inconsistent with the backing store (memory). So, a cache with a block in "M" state has responsibility to write the block to backing store when it is evicted.
- **Shared –**
  This block is not modified and is present in at least one cache. The cache can evict the data without writing it to backing store.
- **Invalid –**
  This block is invalid and must be fetched from memory or from another cache if it is to be stored in this cache.

2. MOSI Protocol:

This protocol is an extension of MSI protocol. It adds the following state in MSI protocol:

- **Owned –**
  It indicates that the present processor owns this block and will service requests from other processors for the block.

3. MESI Protocol –

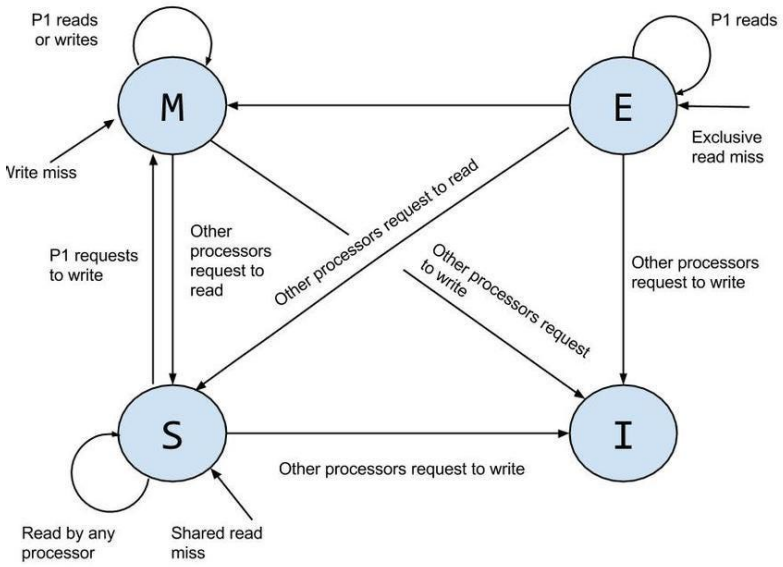
It is the most widely used cache coherence protocol. Every cache line is marked with one the following states:

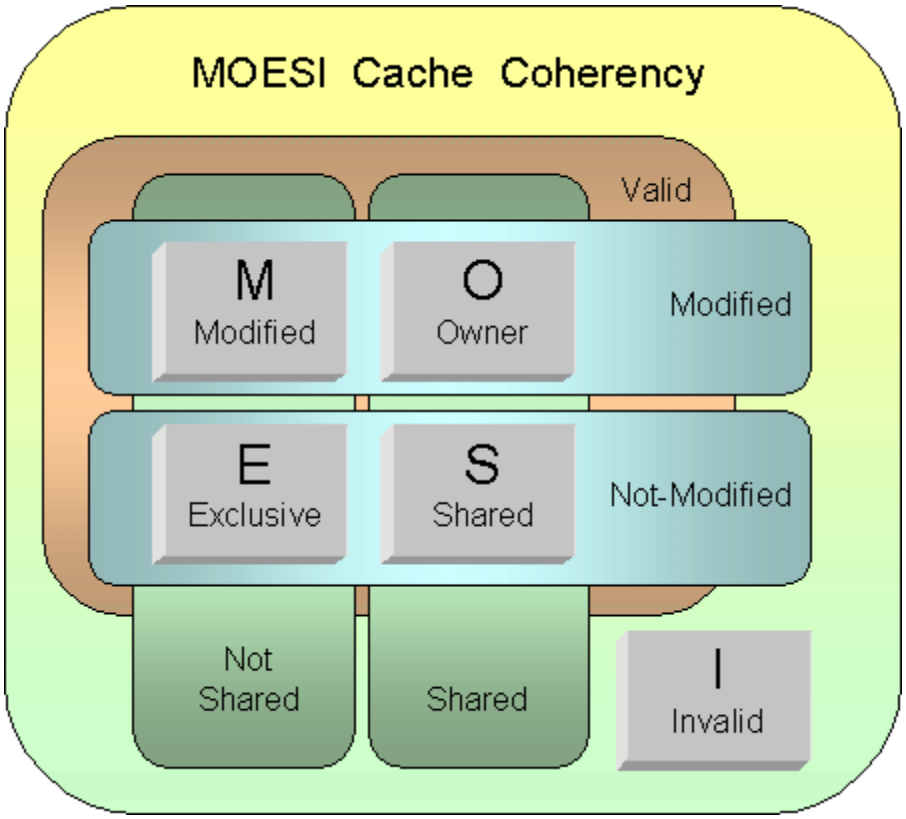
- **Modified –**
  This indicates that the cache line is present in current cache only and is dirty i.e its value is different from the main memory. The cache is required to write the data back to main memory in future, before permitting any other read of invalid main memory state.
- **Exclusive –**
  This indicates that the cache line is present in current cache only and is clean i.e its value matches the main memory value.
- **Shared –**
  It indicates that this cache line may be stored in other caches of the machine.
- **Invalid –**
  It indicates that this cache line is invalid.

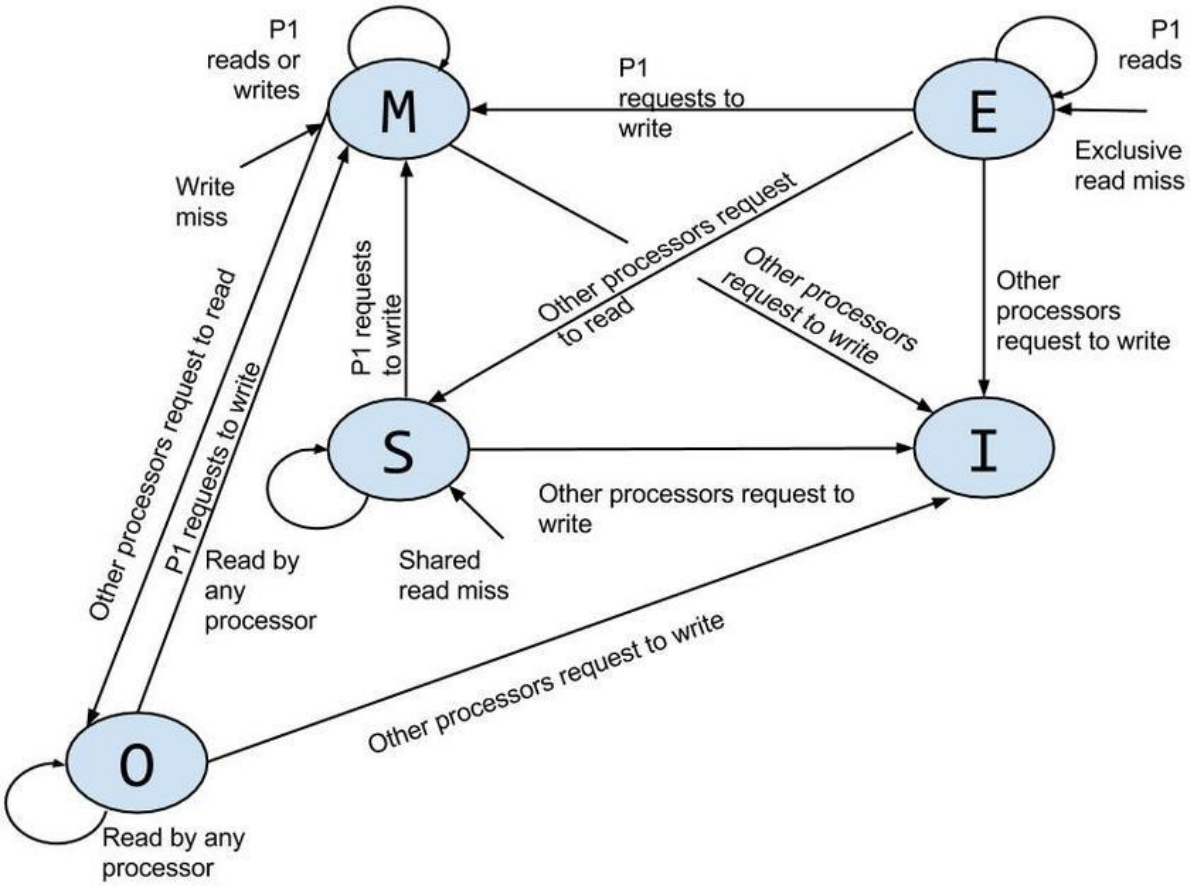
4. MOESI Protocol:

This is a full cache coherence protocol that encompasses all of the possible states commonly used in other protocols. Each cache line is in one of the following states:

- **Modified –**
  A cache line in this state holds the most recent, correct copy of the data while the copy in the main memory is incorrect and no other processor holds a copy.
- **Owned –**
  A cache line in this state holds the most recent, correct copy of the data. It is similar to shared state in that other processors can hold a copy of most recent, correct data and unlike shared state however, copy in main memory can be incorrect. Only one processor can hold the data in owned state while all other processors must hold the data in shared state.
- **Exclusive –**
  A cache line in this state holds the most recent, correct copy of the data. The main memory copy is also most recent, correct copy of data while no other holds a copy of data.
- **Shared –**
  A cache line in this state holds the most recent, correct copy of the data. Other processors in system may hold copies of data in shared state as well. The main memory copy is also the most recent, correct copy of the data, if no other processor holds it in owned state.
- **Invalid –**
  A cache line in this state does not hold a valid copy of data. Valid copies of data can be either in main memory or another processor cache.




MOESI Cache Coherency
Valid
M
Modified
O
Owner
Modified
E
Exclusive
S
Shared
Not-Modified
Not
Shared
Shared
I
Invalid


P1
reads or
writes
M
P1
requests to
write
E
P1
reads
Exclusive
read miss
Write
miss
Other processors request
to read
Other processors
request to write
Other
processors
request to write
P1 requests
to write
Other processors request to read
P1 requests to write
S
I
Other processors request to
write
Read by
any
processor
Shared
read miss
Other processors request to write
O
Read by any
processor

Structure of a processor, pipelining, and cache coherence based questions?

Here's a breakdown of processor structure, pipelining, and cache coherence, including common questions and explanations:

Processor Structure

- **Core Components:**

Processors consist of several key units like the Control Unit (CU), Arithmetic Logic Unit (ALU), registers, and memory access mechanisms.

- **Instruction Cycle:**

The processor follows an instruction cycle, typically involving fetching an instruction, decoding it, executing it, and writing back results.

- **Registers:**

Registers are small, high-speed memory units within the CPU used to store frequently accessed data and instructions.

- **Memory Hierarchy:**

Processors use a memory hierarchy (registers, caches, main memory) to balance speed and capacity.

Pipelining

- **Overlapping Execution:**

Pipelining is a technique where multiple instructions are executed simultaneously, overlapping their stages (fetch, decode, execute, etc.).

- **Improved Throughput:**

Pipelining significantly increases throughput by keeping different stages of the pipeline busy.

- **Hazards:**

Pipelining can be affected by hazards (data, control, or structural), which can stall the pipeline.

Cache Coherence

- **Maintaining Consistency:**

Cache coherence is essential in multi-core systems to ensure that all processors see a consistent view of memory, even with multiple caches.

- **Multiple Caches:**

In multi-core systems, each core might have its own private cache.

- **Snooping Protocol:**

One approach to cache coherence is a snooping protocol, where caches listen for bus transactions and update their internal state accordingly.

- **MESI Protocol:**

The MESI (Modified, Exclusive, Shared, Invalid) protocol is a common cache coherence protocol that uses different states to manage cached data.

- **Directory-Based Protocol:**

Another approach is directory-based protocols, where a central directory keeps track of which cache contains which memory block.

Sample Questions & Answers

Q1: What is pipelining and how does it improve performance?

A: Pipelining is a technique where multiple instructions are executed concurrently by overlapping their stages. It improves performance by increasing throughput, as different parts of multiple instructions are processed at the same time.

Q2: Explain the different states in the MESI protocol.

A: The MESI protocol defines the following states for a cache block: Modified (the cache holds a copy that has been modified), Exclusive (the cache has the only copy), Shared (multiple caches have a copy), and Invalid (the cache block is not valid).

Q3: What are the two main approaches to cache coherence?

A: The two main approaches to cache coherence are snooping protocols and directory-based protocols. Snooping protocols involve caches listening for bus transactions, while directory-based protocols use a central directory to track cache ownership.

Q4: What are some of the common hazards in pipelining?

A: Common hazards in pipelining include data hazards (when an instruction needs data that is not yet available from a previous instruction), control hazards (when branching instructions disrupt the flow of instructions), and structural hazards (when the hardware cannot support two instructions simultaneously).

Q5: Explain the difference between a cache hit and a cache miss.

A: A cache hit occurs when the requested data is found in the cache, while a cache miss occurs when the data is not in the cache and must be retrieved from main memory.

The MIPS (Million Instructions Per Second) formula is: MIPS = (Clock Speed in Hz * Instructions per Cycle) / 1,000,000. Alternatively, MIPS can be calculated as: MIPS = (Number of Instructions / Execution Time) * 10^6.

What is MIPS(Million of Instructions Per Second)?

MIPS is a measure of a processor's speed, representing the number of instructions a CPU can process in one second. It's used to indicate how well a computer performs and how much work it can do. A computer that can process 12,000 MIPS should outperform one that processes 10,000 MIPS.

The MIPS estimation has been utilized by computer producers like IBM to degree the "cost of computing." The value of computers is decided in MIPS per dollar. Interests, the esteem of computers in MIPS per dollar has consistently multiplied on a yearly premise for a final couple of decades. MIPS is a simplified metric and doesn't account for other factors that affect real-world performance, such as instruction complexity, I/O speed, or processor architecture.

How to Calculate MIPS

Required inputs for calculating MIPS are the

- Number of instructions per second can be performed by a processor
- CPU processor speed (cycles per second), Ex (1 GHz, 2 GHz).
- CPI (average clock cycles per instruction).
- Execution time.

Method 1: If Number of Instructions and Execution Time is Given.

Step 1: Perform the Divide operation between no. of instructions and Execution time and store the value (Let X) in a variable.

Step 2: Perform the Divide operation between that variable (X) and 1 million for finding millions of instructions per second. **Example:**

if a computer completed 2 million instructions in 0.10 seconds .
X = 2 million/0.10 = 20 million.
No of MIPS = X/1 million (as the name tells about it)
No of MIPS=20 million/1 million =20.

Method 2: If the Number of Cycles per Second (CPU) and the Number of Cycles per Instruction (CPI) are Given.

Step 1: Perform the Divide operation between the number of cycles per second (CPU) and the number of cycles per instruction (CPI) and store the value (X) in a variable.

Step 2: Perform a Divide operation between that variable and 1 million for finding millions of instructions per second. **Example:**

If a computer with a CPU of 400 megahertz had a CPI of 2.
IPS(Instructions per second)=Clock Speed/CPI
X = 400×106/2
X =200×106instructions per second
For million instruction per second we divide by one million.
MIPS=200×106/106=200MIPS.

Advantages of MIPS:

- It is easy to understand and measure
- It helps in the calculation of CPU processor speed (cycles per second), CPI (average clock cycles per instruction) and Execution time.
- It handles when the amount of work is large.

Disadvantages of MIPS:

- It may not reflect real execution, since simple instructions do way better.
- It is an older, obsolete measure of a computer's speed and power.

What is MIPS and Mflops?

MIPS—"million instructions per second": not useful due to variations in instruction length, implementation, etc. Mflop/s—"million floating-point operations per second".

What are the three types of MIPS instructions?

In MIPS there are only 3 ways to format instructions. They are the R-format (register), the I-format (immediate), and the J-format (jump).

Is MIPS RISC or CISC?

The MIPS processor, designed in 1984 by researchers at Stanford University, is a RISC (Reduced Instruction Set Computer) processor.

Computer Organization and Architecture | Pipelining | Set 1 (Execution, Stages and Throughput)

Pipelining is a technique used in modern processors to improve performance by executing multiple instructions simultaneously. It breaks down the execution of instructions into several stages, where each stage completes a part of the instruction. These stages can overlap, allowing the processor to work on different instructions at various stages of completion, similar to an assembly line in manufacturing.

In this article, you will get a detailed overview of Pipeline in Computer Organization and Architecture.

Table of Content

- What is Pipelining?
- What is Throughout?
- What is Latenecy?
- Advantages of Pipelining
- Disadvantages of Pipelining

What is Pipelining?

Pipelining is an arrangement of the CPU's hardware components to raise the CPU's general performance. In a pipelined processor, procedures called 'stages' are accomplished in parallel, and the execution of more than one line of instruction occurs. Now let us look at a real-life example that should operate based on the pipelined operation concept. Consider a water bottle packaging plant. For this case, let there be 3 processes that a bottle should go through, ensing the bottle(I), Filling water in the bottle(F), Sealing the bottle(S).

It will be helpful for us to label these stages as stage 1, stage 2, and stage 3. Let each stage take 1 minute to complete its operation. Now, in a non-pipelined operation, a bottle is first inserted in the plant, and after 1 minute it is moved to stage 2 where water is filled. Now, in stage 1 nothing is happening. Likewise, when the bottle is in stage 3 both stage 1 and stage 2 are inactive. But in pipelined operation, when the bottle is in stage 2, the bottle in stage 1 can be reloaded. In the same way, during the bottle 3 there could be one bottle in the 1st

and 2nd stage accordingly. Therefore at the end of stage 3, we receive a new bottle for every minute. Hence, the average time taken to manufacture 1 bottle is:

Therefore, the average time intervals of manufacturing each bottle is:

Without pipelining = 9/3 minutes = 3m

I F S | | | | | |
| | | I F S | | |
| | | | | | I F S (9 minutes)

With pipelining = 5/3 minutes = 1.67m

I F S | |
| I F S |
| | I F S (5 minutes)

Thus, pipelined operation increases the efficiency of a system.

Design of a basic Pipeline

- In a pipelined processor, a pipeline has two ends, the input end and the output end. Between these ends, there are multiple stages/segments such that the output of one stage is connected to the input of the next stage and each stage performs a specific operation.
- Interface registers are used to hold the intermediate output between two stages. These interface registers are also called latch or buffer.
- All the stages in the pipeline along with the interface registers are controlled by a common clock.

Execution in a pipelined processor Execution sequence of instructions in a pipelined processor can be visualized using a space-time diagram. For example, consider a processor having 4 stages and let there be 2 instructions to be executed. We can visualize the execution sequence through the following space-time diagrams:

Non-Overlapped Execution

Stage / Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
S1	I1				I2			
S2		I1				I2		

Stage / Cycle	1	2	3	4	5	6	7	8
S3			I1				I2	
S4				I1				I2

Total time = 8 Cycle

Overlapped Execution

Stage / Cycle	1	2	3	4	5
S1	I1	I2			
S2		I1	I2		
S3			I1	I2	
S4				I1	I2

Total time = 5 Cycle **Pipeline Stages** RISC processor has 5 stage instruction pipeline to execute all the instructions in the RISC instruction set. Following are the 5 stages of the RISC pipeline with their respective operations:

- **Stage 1 (Instruction Fetch):** In this stage the CPU fetches the instructions from the address present in the memory location whose value is stored in the program counter.
- **Stage 2 (Instruction Decode):** In this stage, the instruction is decoded and register file is accessed to obtain the values of registers used in the instruction.
- **Stage 3 (Instruction Execute):** In this stage some of activities are done such as ALU operations.
- **Stage 4 (Memory Access):** In this stage, memory operands are read and written from/to the memory that is present in the instruction.
- **Stage 5 (Write Back):** In this stage, computed/fetched value is written back to the register present in the instructions.

Performance of a pipelined processor Consider a ‘k’ segment pipeline with clock cycle time as ‘Tp’. Let there be ‘n’ tasks to be completed in the pipelined processor. Now, the first instruction is going to take ‘k’ cycles to come out of the pipeline but the other ‘n – 1’ instructions will take only ‘1’ cycle each, i.e, a total of ‘n – 1’ cycles. So, time taken to execute ‘n’ instructions in a pipelined processor:

ETpipeline = k + n – 1 cycles
= (k + n – 1) Tp

In the same case, for a non-pipelined processor, the execution time of ‘n’ instructions will be:

ETnon-pipeline = n * k * Tp

So, speedup (S) of the pipelined processor over the non-pipelined processor, when ‘n’ tasks are executed on the same processor is:

S = Performance of non-pipelined processor /
Performance of pipelined processor

As the performance of a processor is inversely proportional to the execution time, we have,

S = ETnon-pipeline / ETpipeline
=> S = [n * k * Tp] / [(k + n – 1) * Tp]
S = [n * k] / [k + n – 1]

When the number of tasks ‘n’ is significantly larger than k, that is, n >> k

S = n * k / n
S = k

where ‘k’ are the number of stages in the pipeline. Also, **Efficiency** = Given speed up / Max speed up = S / Smax We know that Smax = k So, **Efficiency** = S / k **Throughput** = Number of instructions / Total time to complete the instructions So, **Throughput** = n / (k + n – 1) * Tp Note: The cycles per instruction (CPI) value of an ideal pipelined processor is 1 Please see **Set 2** for Dependencies and Data Hazard and **Set 3** for Types of pipeline and Stalling.

Performance of pipeline is measured using two main metrices as Throughput and latency.

What is Throughout?

- It measure number of instruction completed per unit time.
- It represents overall processing speed of pipeline.
- Higher throughput indicate processing speed of pipeline.
- Calculated as, throughput= number of instruction executed/ execution time.

- It can be affected by pipeline length, clock frequency. efficiency of instruction execution and presence of pipeline hazards or stalls.

What is Latenecy?

- It measure time taken for a single instruction to complete its execution.
- It represents delay or time it takes for an instruction to pass through pipeline stages.
- Lower latency indicates better performance .
- It is calculated as, Latency= Execution time/ Number of instruction executed.
- It in influenced by pipeline length, depth, clock cycle time, instruction dependencies and pipeline hazards.

Advantages of Pipelining

- **Increased Throughput:** Pipelining enhance the throughput capacity of a CPU and enables a number of instruction to be processed at the same time at different stages. This leads to the improvement of the amount of instructions accomplished in a given period of time, thus improving the efficiency of the processor.
- **Improved CPU Utilization:** From superimposing of instructions, pipelining helps to ensure that different sections of the CPU are useful. This gives no time for idling of the various segments of the pipeline and optimally utilizes hardware resources.
- **Higher Instruction Throughput:** Pipelining occurring because when one particular instruction is in the execution stage it is possible for other instructions to be at varying stages of fetch, decode, execute, memory access, and write-back. In this manner there is concurrent processing going on and the CPU is able to process more number of instructions in a given time frame than in non pipelined processors.
- **Better Performance for Repeated Tasks:** Pipelining is particularly effective when all the tasks are accompanied by repetitive instructions, because the use of the pipeline shortens the amount of time each task takes to complete.
- **Scalability:** Pipelining is RSVP implemented in different types of processors hence it is scalable from simple CPU's to an advanced multi-core processor.

Disadvantages of Pipelining

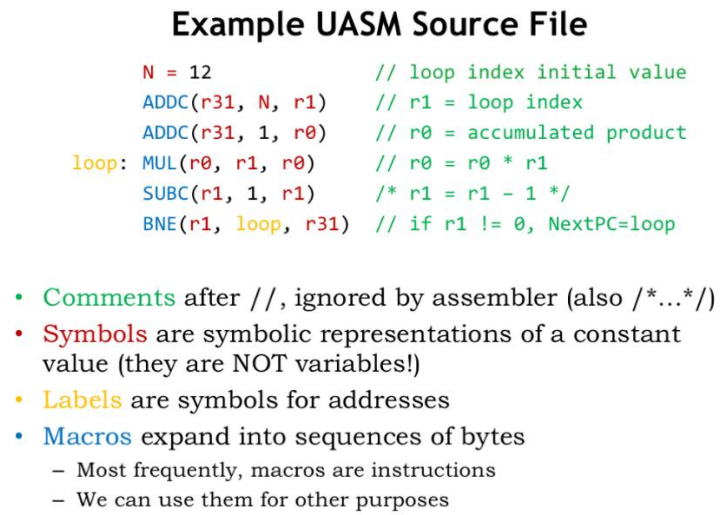
- **Pipeline Hazards:** Pipelining may result to data hazards whereby instructions depends on other instructions; control hazards, which arise due to branch instructions; and structural hazards whereby there are inadequate hardware facilities. Some of these hazards may lead to delays hence tough strategies to manage them to ensure progress is made.

- **Increased Complexity:** Pipelining enhances the complexity of processor design as well as its application as compared to non-pipelined structures. Pipelining stages management, dealing with the risks and correct instruction sequence contribute to the design and control considerations.
- **Stall Cycles:** When risks are present, pipeline stalls or bubbles can be brought about, and this produces idle times in certain stages in the pipeline. These stalls can actually remove some of the cycles acquired by pipelining, thus reducing the latter's efficiency.
- **Instruction Latency:** While pipelining increases the throughput of instructions the delay of each instruction may not necessarily be reduced. Every instruction must still go through all the pipeline stages and the time it takes for a single instruction to execute can neither reduce nor decrease significantly due to overheads.
- **Hardware Overhead:** It increases the complexity in designing the pipelining due to the presence of pipeline registers and the control logic used in managing the pipe stages and the data. This not only increases the cost of the wares but also forces integration of more complicated, and thus costly, hardware.

Conclusion

Pipelining is one of the most essential concepts and it improves CPU's capability to process several instructions at the same time across various stages. It increases immensely the system's throughput and overall efficiency by effectively determining the optimum use of hardware. On its own it enhances the processing speed but handling of pipeline hazards is critical for enhancing efficiency. It is thus crucial for any architect developing systems that will support HPC to have a war chest of efficient pipelining strategies that they can implement.

Assembly language:



An instruction set in assembly code is the collection of all the commands (instructions) that a processor can understand and execute. Assembly language uses mnemonics (short, memorable codes) to represent these instructions, making them more human-readable than the binary code the processor actually uses. Each processor architecture has its own unique instruction set.

Here's a more detailed explanation:

1. What is an Instruction Set?

- An instruction set defines the vocabulary of commands that a CPU can understand.
- It essentially outlines the operations the CPU can perform, such as arithmetic operations, data movement, and control flow (jumps, calls, etc.).
- The instruction set is a fundamental part of a CPU's architecture and determines the type of programming it supports.

2. Assembly Language and Instruction Sets

- Assembly language uses mnemonics (e.g., ADD, MOV, JMP) to represent instructions in the instruction set.
- For example, ADD R1, R2 might represent the instruction to add the contents of register R2 to the contents of register R1, storing the result in R1.
- Assemblers translate assembly code into the machine language (binary code) that the CPU can directly execute.

3. Key Concepts Related to Assembly Instructions

- **Opcode:**

The part of the instruction that specifies the operation to be performed (e.g., ADD, MOV).

- **Operand(s):**

The data or memory addresses that the instruction operates on. For example, in ADD R1, R2, R1 and R2 are operands.

- **Addressing Modes:**

Different ways of specifying the location of data, such as using register numbers, memory addresses, or immediate values.

4. Examples of Instruction Sets

- **x86:** A widely used instruction set architecture found in many personal computers.
- **ARM:** Another popular ISA, commonly used in mobile devices and embedded systems.

- **RISC-V:** A newer, open-source ISA gaining popularity.

5. Importance of Understanding Instruction Sets

- Understanding the instruction set of a processor is essential for writing efficient assembly language programs.
- It allows programmers to interact directly with the hardware and optimize code for specific performance characteristics.
- Knowledge of assembly language and instruction sets is helpful for understanding how high-level languages (like C or Java) are translated into machine code and executed.

The ARM instruction set, used in ARM assembly language, is a collection of commands the ARM processor can execute, allowing for low-level control over the machine. It's comprised of various instructions for data processing, memory access, and control flow. ARM assembly code uses mnemonics (abbreviations) to represent these instructions, making it easier for humans to understand and write.

Key Components of ARM Assembly:

- **Instructions:**

The fundamental building blocks of ARM assembly code, each representing a specific operation the processor can perform.

- **Mnemonics:**

Symbolic codes used to represent instructions, such as ADD, SUB, LDR, STR, etc.

- **Operands:**

The data that the instructions operate on, which can include registers, memory addresses, or immediate values.

- **Registers:**

Internal storage locations within the processor used to hold data and addresses.

- **Memory:**

The main storage space of the system where data and instructions are stored.

- **Assembler:**

A program that translates ARM assembly code into machine code (binary instructions the processor can directly execute).

Example Instructions:

- **Data Processing:**
    - ADD: Adds two operands and stores the result in a destination register.
    - SUB: Subtracts one operand from another and stores the result.
    - MUL: Multiplies two operands.
    - MOV: Copies data from one location (register or memory) to another.
- **Data Movement:**
    - LDR: Loads data from memory into a register.
    - STR: Stores data from a register into memory.
    - SWP: Swaps data between a register and a memory location.
- **Control Flow:**
    - B: Unconditional branch to a specified address.
    - BL: Branch with link (calls a subroutine).
    - BEQ: Conditional branch if equal to zero.
    - BNE: Conditional branch if not equal to zero.
    - CMP: Compares two values and sets condition flags.

ARM Instruction Set Architectures (ISAs):

ARM supports different ISAs, including:

- **A32 (ARM32):**

The traditional 32-bit instruction set, commonly found in older ARM processors.

- **T32 (Thumb):**

A 16-bit and 32-bit instruction set, optimized for code density and faster execution.

- **A64 (ARM64):**

A 64-bit instruction set for modern ARM processors, offering expanded register usage and more complex instructions.

Additional Resources:

- **ARM Developer:** Provides extensive documentation and resources on ARM architecture and assembly language.
- **ARM Architecture Reference Manual (ARM ARM):** The definitive guide to the ARM instruction set.
- **Instruction Set Exploration Tools:** Online tools for browsing and searching the ARM instruction set.

x86:

The x86 instruction set is a large and complex collection of instructions used by x86-compatible processors. It includes instructions for data movement, arithmetic operations, logical operations, control flow, and more. The set has been extended over time, introducing new registers and functionalities.

Key areas of the x86 instruction set:

- **Data Movement:**

Instructions like mov are used to copy data between registers and memory, and between memory locations. Other data movement instructions include push (to put data on the stack) and pop (to retrieve data from the stack).

- **Arithmetic Operations:**

Instructions such as add, sub, inc, and dec perform basic arithmetic calculations.

- **Logical Operations:**

Instructions like and, or, and xor perform bitwise logical operations.

- **Control Flow:**

Instructions like jmp, call, ret, and conditional jump instructions are used to control the execution flow of a program.

- **String Instructions:**

Instructions like movs, stos, lodsw, scas are used for string manipulation, allowing for efficient copying, filling, and scanning of memory blocks.

- **Floating-Point Instructions:**

The x87 FPU (Floating Point Unit) has its own set of instructions for working with floating-point numbers.

- **SIMD (Single Instruction, Multiple Data) Instructions:**

Modern x86 processors have SIMD extensions, which allow for performing the same operation on multiple data elements simultaneously, improving performance for tasks like image processing and video encoding.

- **Bit Manipulation:**

x86 also includes instructions for manipulating individual bits within registers and memory.

- **System Calls:**

Instructions like syscall are used to interact with the operating system kernel, providing access to system resources.

x86-64 (64-bit x86):

The x86-64 instruction set is a 64-bit extension of the original x86 architecture, offering wider registers, larger address spaces, and new instructions. It's the dominant architecture for modern computing.

In Summary:

The x86 instruction set is a vast and evolving collection of instructions that powers a wide variety of computers and software. It includes instructions for everything from moving data to executing complex operations and interacting with the operating system. The x86-64 extension provides a 64-bit architecture that is widely used in modern systems.

RISC:

RISC assembly code utilizes a streamlined set of instructions focusing on simplicity and speed. Instructions often involve a destination register and two source operands. Common instructions include add, addi, sub, subi, load (lw, lb, lh), store (sw, sb, sh), and branch instructions (beq, bne, blt, etc.).

Key Features of RISC Assembly:

- **Simplicity:**

RISC architectures prioritize simple, fast instructions to improve performance.

- **Instruction Types:**
    - **Arithmetic:** add, addi, sub, subi
    - **Load/Store:** lw (load word), sw (store word), lb (load byte), sb (store byte), lh (load halfword), sh (store halfword)
    - **Branch:** beq (branch if equal), bne (branch if not equal), blt (branch if less than), bltu (branch if less than unsigned), bge (branch if greater than or equal), bgeu (branch if greater than or equal unsigned)

- **Registers:**

RISC processors use a set of registers for storing data and addresses. Common registers include x0 (zero register), sp (stack pointer), and various general-purpose registers.

- **Addressing Modes:**

Instructions can use different addressing modes, such as immediate addressing (using a constant value directly in the instruction) and register addressing (using the contents of a register as an operand).

Examples:

- add s0, s1, s2:

Adds the values in registers s1 and s2 and stores the result in register s0.

- lw t0, 0(t1):

Loads a word (4 bytes) from the memory location pointed to by register t1 and stores it in register t0.

- beq t0, t1, label:

Branches to the instruction labeled label if the values in registers t0 and t1 are equal.

Note: The specific instructions and their syntax may vary depending on the particular RISC architecture (e.g., RISC-V, MIPS).

Types of RISC V:

RISC V has lots of options. RV32I and RV64I are base instruction sets, the 32-bit and 64-bit instruction sets, respectively. There is also a 128 bit version, RV128I, and, a smaller 32-bit version RV32E, which has only 16 general purpose registers, whereas the others have 32 registers. For all instruction sets, the base set are 32 bit instructions.

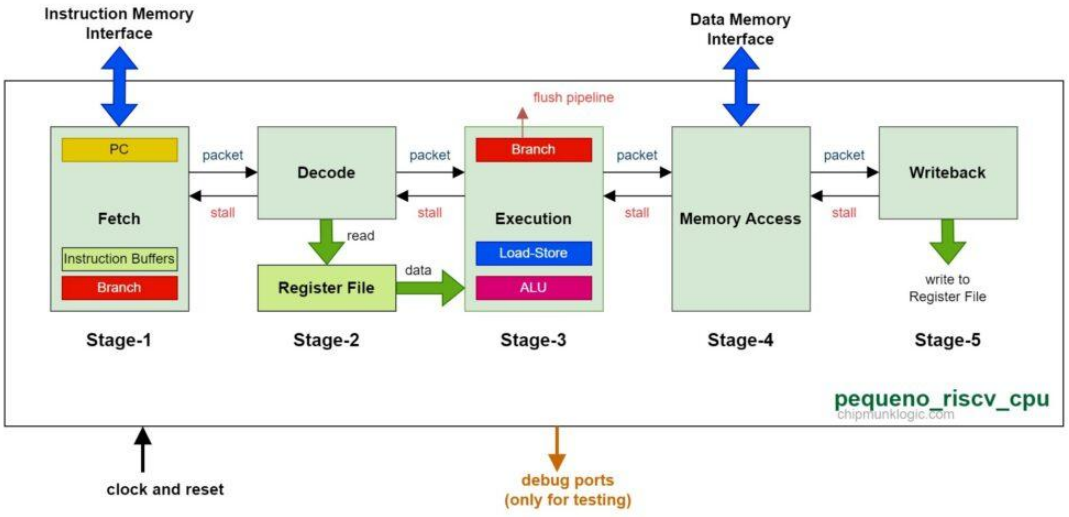
To the base instructions sets, they add optional extensions, e.g. for floating point, multiply and divide, etc.., so like RV32IMAC is based on RV32I with the M, A, and C extensions.

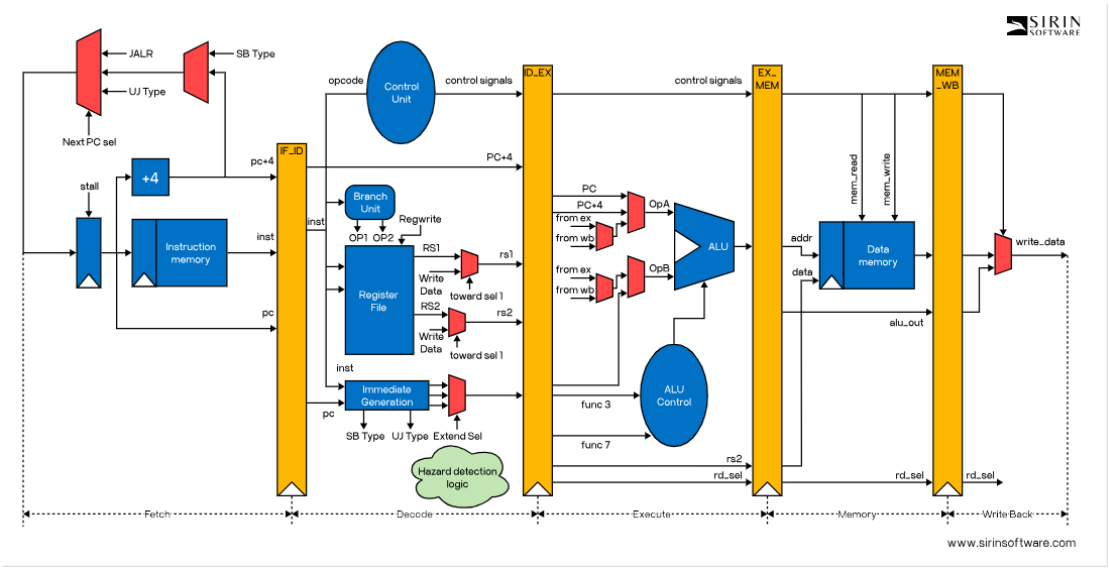
The main difference between RV32I, RV64I, and RV128I is the width of the registers, 32, 64, or 128 bits in width — along with an expected increase in address space from larger widths. All instruction sets use 32-bit instructions, and with the compressed instruction option RVC, add 16-bit instructions to the mix. No instructions of larger size are defined as standard yet, but there's opcode space for larger instruction, e.g. 48-bit, 64-bit and beyond that can be added to RV32 or RV64 (or RV128).

There are some opcode differences between RV32 and RV64 as well. The larger widths add a few instructions to deal with the larger sizes. RV64I, for example, uses the same opcode as

RV32I for add, but means 64-bit add instead of 32-bit add, while there is a new opcode in RV64I for doing 32-bit only add (addw). RV64I uses the same opcodes as RV32I for load and stores (for data of byte, half, and 32-bit word sizes), though includes new opcodes for loading/storing 64-bit data.

Otherwise one benefit of RV64 over RV32 is being able to address large address space than 4GB, which also applies to x86 vs. x64. Since we can run out of space for data with a 32-bit address space, having a larger address space and being able to do 64-bit computations in 64-bit registers is the gist of why the processor is called a 64-bit processor (despite sharing the instruction size of 32 bits RV32) (this is also largely also true for x86 vs. x64, in that there is a large amount of similarity between them, such as variable length instructions ranging from 1 byte to 10 or so bytes.)

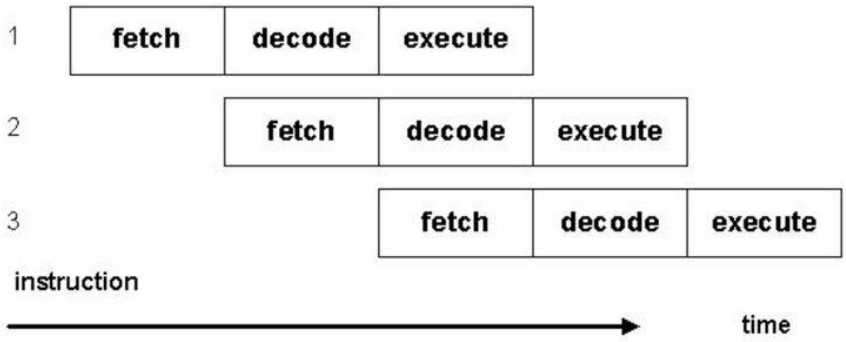




The 3-stage pipeline

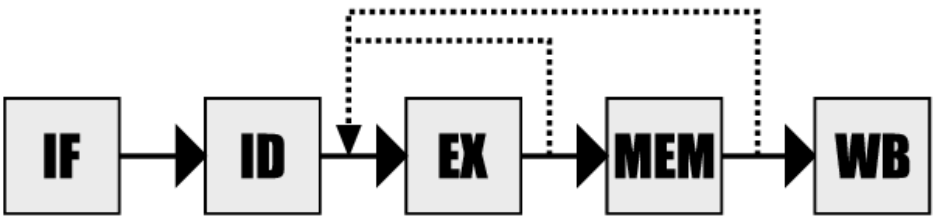
ARM processors up to the ARM7 employ a simple 3-stage pipeline with the following pipeline stages

1. ***Fetch***
2. ***Decode***
3. ***Execute***



5

5-stage pipeline:



IF: Instruction fetch
ID: Instruction decode and register read
EX: Execute, address generation
MEM: Memory access
WB: Write back to registers

What???

8-stage pipeline:

An 8-stage pipeline is a method of speeding up instruction execution in a CPU by breaking down the process into eight distinct stages. This allows multiple instructions to be processed concurrently, increasing throughput (number of instructions completed per unit of time) while reducing individual instruction latency. The pipeline stages typically include instruction fetch, decode, execute, and memory access, with variations depending on the specific architecture.

Here's a more detailed breakdown of an 8-stage pipeline:

Stages:

1. **Instruction Fetch (IF):** Retrieves the next instruction from memory.
2. **Instruction Decode (OF):** Decodes the fetched instruction to determine what operation to perform.
3. **Execute (EX):** Performs the operation specified by the decoded instruction.
4. **Access Memory 0 (M0):** Accesses memory, potentially for data read or write, in the first memory access stage.
5. **Access Memory 1 (M1):** Another memory access stage, possibly used for more complex memory operations.
6. **Access Memory 2 (M2):** Further memory access stage, if needed.
7. **Access Memory 3 (M3):** Final memory access stage.
8. **Writeback (WB):** Writes the results of the operation to the register file.

Benefits of Pipelining:

- **Increased Throughput:**

By overlapping the execution of multiple instructions, pipelining significantly increases the number of instructions completed per unit of time.

- **Reduced Latency:**

While the execution of a single instruction is slightly longer, the overall latency of processing a sequence of instructions is significantly reduced.

- **Improved Resource Utilization:**

Different pipeline stages can utilize different resources, such as memory and registers, concurrently, leading to better resource utilization.

Considerations:

- **Hazards:**

Pipelining can introduce hazards (e.g., data hazards, control hazards) that can disrupt the flow of instructions. Techniques like forwarding and branch prediction are used to mitigate these hazards.

- **Complexity:**

Implementing and managing a pipelined processor is more complex than a non-pipelined one.

In Summary:

The 8-stage pipeline is a key technique for building fast CPUs, allowing for the concurrent processing of multiple instructions and achieving significant performance improvements.

explain interrupt handling, and various scenarios?

What are Interrupts and How Interrupt Handling is Done in Modern Operating Systems

Operating SystemHardwareSoftware & Coding

Interrupts are generally called signals which are generated by the software or hardware when a particular event or process requires immediate attention. So, the signal informs the processor about a high priority and urgent information demand causing an interruption in the current working process.

Thus, whenever an interruption occurs the processor finishes the current instruction execution and starts the execution of the interrupt known as interrupt handling. Moreover, for every interrupt handling to occur there is an Interrupt service routine (ISR) or interrupt handler.

Interrupt handling in modern operating systems

In several operating systems such as Linux. mac or windows Interrupt handling is divided into two parts −

- First-level interrupt handlers (FLIH), also known as hard interrupt handlers or fast interrupt handlers.
- Second-level interrupt handlers (SLIH), also known as slow interrupt handlers or soft interrupt handlers.

The uses and properties of FLIH and SLIH are given below −

FLIH

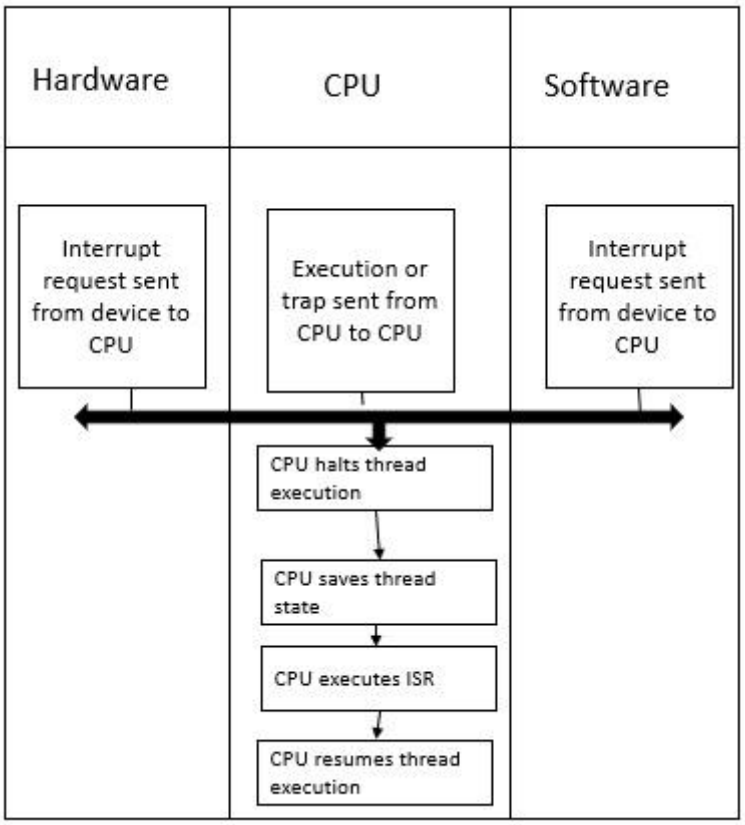
- These include platform specific interrupt handling.
- It causes jitter in the execution of the process.
- It also masks interrupts.
- FLIH is known as the upper half in Linux.

SLIH

- It completes long interrupt processing tasks similar to the process or event.
- Have a dedicated thread for every single handler.
- It has a long-lived execution time
- SLIH is known as the Lower half or bottom half in Linux.

The interrupt handling mechanism of an operating system accepts a number which is an address and then selects what specific action to be taken which is already mentioned in the interrupt service routine. In most architecture, the address is stored in a table known as a vector table.

Now let us see the interrupt handling by OS scheme in pictorial representation.



programs in c, fsm design?

Program block vs. module block?

In SystemVerilog, a module is a fundamental building block for hardware design, used to describe functionality and can be synthesized into hardware. A program block, on the other hand, is primarily used for verification purposes and is not intended to be synthesized. Modules can contain various elements like always, initial, and assign statements, while programs are typically used within the testbench to drive the simulation and check the behavior of the design.

Here's a more detailed breakdown of the key differences:

1. Purpose:

- **Module:** Describes hardware behavior and can be synthesized into a physical chip.
- **Program:** Primarily used for verification and driving simulation, not intended for synthesis.

2. Synthesis:

- **Module:** Can be synthesized into hardware.
- **Program:** Not intended for synthesis and is typically not included in the final hardware design.

3. Hierarchy:

- **Module:** Can contain other modules, interfaces, and program blocks.
- **Program:** Can be nested within modules or interfaces, but cannot contain modules or other programs.

4. Event Regions:

- **Module:** initial blocks are executed in the active region of the simulation.
- **Program:** initial blocks are typically executed in the reactive region of the simulation.

5. Restrictions:

- **Module:** Can contain always and initial blocks, as well as assign statements.
- **Program:** Cannot contain always blocks, User-Defined Primitives (UDPs), or other nested programs. It primarily uses initial blocks.

6. Race Conditions:

- **Module:** Can lead to race conditions in testbenches if not handled carefully.
- **Program:** Designed to help avoid race conditions by executing in the reactive region.

7. Variables:

- **Module:** Can use both blocking and non-blocking assignments.
- **Program:** Variables are typically assigned using blocking assignments.

8. Function/Task Calls:

- **Module:** Can call tasks and functions defined within other modules or programs.
- **Program:** Cannot call tasks or functions within a module, but it can call tasks and functions within other modules or programs.

In summary:

- **Modules**: are for hardware design and can be synthesized.
- **Programs**: are for verification and driving simulation, and are not synthesized.

For a more helpful discussion on the nuances of using modules and program blocks in SystemVerilog, you can check out the Verification Academy forums and the Project VeriPage website.

Finite State Machines (FSMs) are computational models that transition between a finite number of states based on input events. They are useful for modeling systems with discrete states and transitions. Here's how to implement an FSM in C:

- **Define States:**
    - Use an enum to represent the possible states of the FSM.

C

```
typedef enum {
    STATE_A,
    STATE_B,
    STATE_C
} State;
```

- **Define Events:**
    - Use an enum to represent the events that trigger state transitions.

C

```
typedef enum {
    EVENT_X,
    EVENT_Y,
    EVENT_Z
} Event;
```

- **Implement the State Transition Logic:**
    - Create a function that takes the current state and an event as input and returns the next state.
    - Use a switch statement to handle different state and event combinations.

C

```
State transition(State current_state, Event event) {
    switch (current_state) {
        case STATE_A:
            if (event == EVENT_X) {
                return STATE_B;
            } else if (event == EVENT_Y) {
                return STATE_C;
            }
            break;
        case STATE_B:
            if (event == EVENT_Z) {
                return STATE_A;
            }
            break;
        case STATE_C:
            if (event == EVENT_X) {
              return STATE_A;
            }
            break;
    }
    return current_state; // Default: stay in the same state
}
```

- **Implement the Main FSM Loop:**
    - Initialize the FSM to a starting state.
    - Continuously monitor for events.
    - When an event occurs, call the transition function to get the next state.

- Update the current state.

C

```
int main() {
    State current_state = STATE_A;
    Event event;

    while (1) {
        // Get the next event (e.g., from user input or sensor reading)
        // event = get_next_event();

        current_state = transition(current_state, event);

        // Perform actions based on the current state
        // perform_state_action(current_state);
    }

    return 0;
}
```

Here is an example of a finite state machine (FSM) implemented in C, modeling a simple traffic light:

C

```
#include <stdio.h>

// Define the states of the traffic light
typedef enum {
    RED,
    YELLOW,
    GREEN
} TrafficLightState;

// Function to simulate the traffic light behavior
void trafficLight(TrafficLightState *currentState) {
    switch (*currentState) {
        case RED:
```

```
            printf("RED\n");
            *currentState = GREEN;
            break;
        case YELLOW:
            printf("YELLOW\n");
            *currentState = RED;
            break;
        case GREEN:
            printf("GREEN\n");
            *currentState = YELLOW;
            break;
    }
}

int main() {
    TrafficLightState currentState = RED; // Initial state

    // Simulate the traffic light for a few cycles
    for (int i = 0; i < 6; i++) {
        trafficLight(&currentState);
    }

    return 0;
}
```

In this example:

- The TrafficLightState enum defines the possible states of the traffic light: RED, YELLOW, and GREEN.
- The trafficLight function takes a pointer to the current state as input. It uses a switch statement to determine the next state based on the current state, and prints the current state.
- The main function initializes the traffic light to the RED state and then simulates the traffic light for 6 cycles, printing the state at each step.

hr questions?

Interview questions [3]

Question 1

What will be the last line of code in a UVM testcase?

1 Answer

Question 2

What are the 1st two lines of codes in a typical UVM test?

1 Answer

Question 3

How will you initiate a verification?

Answer question

some puzzle questions? do from your uvm book?

polymorphism parent p1, child p2, then p1 = p2 or p2 = p1?

Parent p1;

p1 = new();

child p2;

p2 = new();

p1 = p2 is allowed!!!!!!!!!!!!

Interview questions [1]

Question 1

A data vs clock path is given and there is latency in one of them how would you current it

Either reduce the latency or include that delay in path of another one.

Write a logic to find a maximum number among the three given numbers.

Microprocessor Interrupts C programming ARM architecture Amba

1. The interview showed a complex SVA and asked me to explain the functionality of the assertion. 2. A C function which reads the value from a specific address. 3. A question about Functional coverage 4. Questions about coverage (code and functional) 5. UVM factory, config_db, 6. Formal verification 7. Processor based verification. Basically they were asking how you verify a Subsystem using C/C++ 8. AHB-AXI bus

Systemverilog assertions and constraints questions

revise from your uvm book?

cortex-x/a/r/m?

differences?

silicon ip multimedia:

Product FamilyClear All | X

Arm **Immortalis** Graphics Processors

Highest graphics performance for flagship gaming experiences

Mali Graphics Processors

Higher graphics performance combined with power efficiency

Mali Image Signal Processors

Advanced ISPs for human display and computer vision applications

silicon-ip-cpu:

Processor FamilyClear All | X

Cortex-A

Supreme performance at **optimal power**

Cortex-M

Powering the **most energy-efficient embedded devices**

Cortex-R

Robust **real-time** performance

Ethos - NPUs

Highest performance for **machine learning** inference

Neoverse

Scalable and flexible for **cloud to edge** infrastructure

SecurCore

Powerful solutions for **physical security** applications

Embedded device vs. SoC:

An embedded device is a computer system designed for a specific function, often built into another device or product. A System on a Chip (SoC) is an integrated circuit that combines all or most of the components of a system onto a single chip. Essentially, an embedded device can utilize an SoC as its core processing unit.

Here's a more detailed breakdown:

Embedded Device:

- **Definition:**

An embedded system is a computer that is designed to be integrated into another device.

- **Purpose:**

These systems perform a specific task, and are typically not general-purpose computers.

- **Examples:**

Refrigerators, washing machines, smartphones, cameras, and more.

- **Functionality:**

Embedded systems often include a microcontroller (MCU) or a processor (CPU) and other peripherals like memory, sensors, and communication interfaces.

System on a Chip (SoC):

- **Definition:**

An SoC is an integrated circuit that combines all the necessary components of a system on a single chip.

- **Components:**

This includes the CPU, memory, graphics processing unit (GPU), connectivity interfaces, and other peripherals.

- **Benefits:**

SoCs offer advantages like smaller size, lower power consumption, and cost-effectiveness compared to systems with separate chips.

- **Use in Embedded Systems:**

SoCs are widely used as the processing core in many embedded devices, providing the necessary computing power and functionality.

Relationship between Embedded Device and SoC:

- An SoC can be considered the heart of many embedded systems.
- The SoC provides the necessary processing, memory, and other functionalities needed for the embedded system to perform its specific task.
- By integrating the SoC, embedded systems can achieve smaller sizes, lower power consumption, and improved reliability.

Difference between RISC and CISC Processor | Set 2

The **microprocessor** is a processing unit on the single chip. It is the integrated circuit that performs the core functions of the computer CPU. It is the multipurpose programmable silicon chip constructed using a Metal Oxide Semiconductor (MOS) technology which is clock driven and register based. It accepts a binary data as a input and provides output after processing it as per a specification of instructions stored in a memory. These microprocessors are capable of the processing the 128 bits at the time at the speed of a one billion instructions per second.

Characteristics of a Microprocessor

1. **Instruction Set** : The set of complete instructions that the microprocessor executes is termed the instruction set.

2. **Word Length** : The number of bits processed in a single instruction is called word length or word size. The Greater the word size is the larger the processing power of the CPU.

3. **System Clock Speed** : A Clock speed determines how fast the single instruction can be executed in the processor. The microprocessor is controlled by the System Clock. A Clock speeds are generally measured in the millions of a cycles per second (MHz) and thousand million cycles per second GHz. A Clock speed is considered to be the very important aspect of predicting a performance of the processor.

What is Reduced Instruction Set Computer(RISC)?

It stands for Reduced Instruction Set Computer. It is a type of microprocessor architecture that uses a small set of instructions of uniform length. These are simple instructions that are generally executed in one clock cycle. RISC chips are relatively simple to design and inexpensive. The setback of this design is that the computer has to repeatedly perform simple operations to execute a larger program having a large number of processing operations.
Examples: SPARC, POWER PC, etc.

Advantages of Reduced Instruction Set Computer (RISC)

- Faster execution speed RISC processors use the simpler instructions that can be executed more quickly leading to improved overall performance in the many tasks.
- Lower power consumption The simpler design of a RISC processors often results in the lower power usage making them the ideal for mobile devices and energy efficient computing.
- Easier to design and manufacture RISC processors have a simpler architecture which can make them the easier and potentially cheaper to design and produce.

Disadvantages of Reduced Instruction Set Computer (RISC)

- Larger code size RISC processors often require more lines of code to perform complex tasks, which can lead to larger program sizes and increased memory usage.
- More work for compilers The simpler instruction set means compilers for RISC processors need to do more work to translate high-level programming languages into machine code.
- Limited built-in functionality RISC processors have fewer complex instructions built into hardware, which can make certain specialized tasks less efficient without additional software support.

What is Complex Instruction Set Computer (CISC)?

It stands for Complex Instruction Set Computer. These processors offer the users, hundreds of instructions of variable sizes. CISC architecture includes a complete set of special-purpose circuits that carry out these instructions at a very high speed. These instructions interact with memory by using complex addressing modes. CISC processors reduce the program size and hence lesser number of memory cycles are required to execute the programs. This increases the overall speed of execution.
Examples: Intel architecture, AMD

Advantages of Complex Instruction Set Computer (CISC)

- Smaller code size CISC processors can perform complex operations with single instructions, often resulting in more compact code and reduced memory usage.
- Rich instruction set The diverse set of complex instructions can make programming easier and more intuitive for certain tasks, especially in assembly language.
- Backwards compatibility CISC architectures, like x86, often maintain compatibility with older software, making system upgrades easier for users and businesses.

Disadvantages of Complex Instruction Set Computer (CISC)

- Slower execution speed Complex instructions typically take longer to execute, potentially resulting in slower overall performance compared to RISC processors.
- Higher power consumption The more complex hardware required for the CISC processors often leads to a increased power usage making them less suitable for the mobile devices.
- More complex hardware design a CISC processors require more complex circuitry to handle their varied the instructions which can make them a more challenging and expensive to design and manufacture.

What is EPIC?

It stands for Explicitly Parallel Instruction Computing. The best features of RISC and CISC processors are combined in the architecture. It implements parallel processing of instructions rather than using fixed-length instructions. The working of EPIC processors is supported by using a set of complex instructions that contain both basic instructions as well as the information of execution of parallel instructions. It substantially increases the efficiency of these processors.

Difference between RISC and CISC processor

CISC	RISC
A large number of a instructions are present in the architecture.	Very few instructions are present. The number of instructions is generally less than 100.
Some instructions with long execution times. These include instructions that copy an entire block from one part of memory to another and others that copy multiple registers to and from memory.	No instruction with a long execution time due to a very simple instruction set. Some early RISC machines did not even have an integer multiply instruction, requiring compilers to implement multiplication as a sequence of additions.
Variable-length encodings of the instructions. **Example:** IA32 instruction size can range from 1 to 15 bytes.	**Fixed-length encodings** of the instructions are used. **Example:** In IA32, generally all instructions are encoded as 4 bytes.
Multiple formats are supported for specifying operands. A memory operand specifier can have many different combinations of displacement, base, and index register.	Simple addressing formats are supported. Only base and displacement addressing is allowed.
CISC supports array.	RISC does not support an array.

CISC	RISC
Arithmetic and logical operations can be applied to both memory and register operands.	Arithmetic and logical operations only use register operands. Memory referencing is only allowed by loading and storing instructions, i.e. reading from memory into a register and writing from a register to memory respectively.
Implementation programs are hidden from machine-level programs. The ISA provides a clean abstraction between programs and how they get executed.	Implementation programs exposed to machine-level programs. Few RISC machines do not allow specific instruction sequences.
Condition codes are used.	No condition codes are used.
The stack is being used for procedure arguments and returns addresses.	Registers are being used for procedure arguments and return addresses. Memory references can be avoided by some procedures.
Successful pipeline with one instruction per cycle	Unsuccessful pipeline
Heavy use of RAM	More efficient use of RAM

Refer for Set-1: RISC and CISC

Conclusion

The Reduced Instruction Set Computing and a Complex Instruction Set Computing are the method of a processor design. The RISC processors use the fewer and simpler instructions that execute quickly while the CISC processors have the complex instructions that can perform the multiple operations. The RISC focuses on a efficiency and speed using a simpler hardware and relying on the software. CISC aims for a versatility with more built in functionality in the hardware. RISC is a generally faster and more energy efficient while a CISC can be more compact in a terms of code size

Difference Between RISC and CISC

DifferencesComputersComputer Architecture

RISC and CISC are two different types of computer architectures that are used to design the microprocessors that are found in computers. The fundamental difference between RISC and CISC is that **RISC (Reduced Instruction Set Computer)** includes simple instructions and takes one cycle, while the **CISC (Complex Instruction Set Computer)** includes complex instructions and takes multiple cycles.

Read this tutorial to find out more about RISC and CISC and how these two architectures are different from each other.

What is RISC?

In the **RISC architecture**, the instruction set of the computer system is simplified to reduce the execution time. RISC architecture has a small set of instructions that generally includes register-to-register operations.

The RISC architecture uses comparatively a simple instruction format that is easy to decode. The instruction length can be fixed and aligned to word boundaries. RISC processors can execute only one instruction per clock cycle.

The following are some important **characteristics** of a RISC Processor ?

- A RISC processor has a few instructions.
- RISC processor has a few addressing modes.
- In the RISC processor, all operations are performed within the registers of the CPU.
- RISC processor can be of fixed-length.
- RISC can be hardwired rather than micro-programmed control.
- RISC is used for single-cycle instruction execution.
- RISC processor has easily decodable instruction format.

RISC architectures are characterized by a small, simple instruction set and a highly efficient execution pipeline. This allows RISC processors to execute instructions quickly, but it also means that they can only perform a limited number of tasks.

What is CISC?

The **CISC architecture** comprises a complex instruction set. A CISC processor has a variable-length instruction format. In this processor architecture, the instructions that require register operands can take only two bytes.

In a CISC processor architecture, the instructions which require two memory addresses can take five bytes to comprise the complete instruction code. Therefore, in a CISC processor, the execution of instructions may take a varying number of clock cycles. The CISC processor also provides direct manipulation of operands that are stored in the memory.

The primary objective of the CISC processor architecture is to support a single machine instruction for each statement that is written in a high-level programming language.

The following are the important **characteristics** of a CISC processor architecture ?

- CISC can have variable-length instruction formats.
- It supports a set of a large number of instructions, typically from 100 to 250 instructions.
- It has a large variety of addressing modes, typically from 5 to 20 different modes.
- CISC has some instructions which perform specialized tasks and are used infrequently.

CISC architectures have a large, complex instruction set and a less efficient execution pipeline. This allows CISC processors to perform a wider range of tasks, but they are not as fast as RISC processors when executing instructions.

Difference between RISC and CISC

The following table highlights all the important differences between RISC and CISC architectures ?

S.No.	RISC	CISC
1.	It stands for Reduced Instruction Set Computer.	It stands for Complex Instruction Set Computer.
2.	It is a microprocessor architecture that uses small instruction set of uniform length.	This offers hundreds of instructions of different sizes to the users.
3.	These simple instructions are executed in one clock cycle.	This architecture has a set of special purpose circuits which help execute the instructions at a high speed.

4.	These chips are relatively simple to design.	These chips are complex to design.
5.	They are inexpensive.	They are relatively expensive.
6.	Examples of RISC chips include SPARC, POWER PC.	Examples of CISC include Intel architecture, AMD.
7.	It has less number of instructions.	It has more number of instructions.
8.	It has fixed-length encodings for instructions.	It has variable-length encodings of instructions.
9.	Simple addressing formats are supported.	The instructions interact with memory using complex addressing modes.
10.	It doesn't support arrays.	It has a large number of instructions. It supports arrays.
11.	It doesn't use condition codes.	Condition codes are used.
12.	Registers are used for procedure arguments and return addresses.	The stack is used for procedure arguments and return addresses.

Conclusion

The most significant difference between RISC and CISC architectures is the size and complexity of the instruction set. RISC is a microprocessor architecture that uses a small instruction set of uniform length that allows fast execution, while the CISC architecture is one that offers hundreds of instructions of different sizes that allows the users to perform a wider range of tasks.

Difference between arm7, arm9, arm11 and arm cortex

The ARM architecture refers to a family of reduced instruction set computing (RISC) processors that are widely used in embedded systems and mobile devices. The ARM processors are known for their power efficiency and performance capabilities. Over the years, ARM has introduced several microarchitecture versions with improvements and new features. The major ARM processor families include ARM7, ARM9, ARM11 and Cortex.

Contents

ARM7ARM9ARM11ARM CortexKey DifferencesARM7 Core ArchitectureARM9 Core ArchitectureARM11 Core ArchitectureARM Cortex-A SeriesEvolution of ARM Instruction Sets

ARM7

The ARM7 processors were one of the first ARM-based CPUs introduced in the early 1990s. The ARM7 core architecture leveraged a 3-stage pipeline and operated at frequencies between 30-100 MHz. Some key features of ARM7 processors include:

- 3-stage pipeline to enable faster instruction execution
- Von Neumann architecture with integrated cache
- Thumb 16-bit instruction set to improve code density
- EmbeddedICE module for debug and trace capabilities
- Support for up to 4GB physical address space
- Optional MMU for memory protection and virtual memory support

The ARM7TDMI core was one of the most popular ARM7 variants that incorporated a thumb instruction set and a debug module. Key applications of ARM7 processors include early mobile phones, PDA devices, disk drives, routers and printers.

ARM9

Introduced in 2001, the ARM9 family was designed to address the growing performance requirements of mobile and embedded applications. The ARM9 cores were based on an evolved 5-stage pipeline executing the ARMv4T instruction set. Here are some major improvements of ARM9 over previous ARM7 cores:

- Higher clock speeds up to 200 MHz
- 5-stage pipeline for improved performance
- Updated thumb instruction set Thumb-2 with conditional execution
- Enhanced DSP instructions and saturating arithmetic

- Coprocessor interface to support multimedia extensions
- Tightly-coupled memory support
- Multi-master AXI bus interface

The ARM9 family includes variants like ARM920T, ARM922T, ARM925T, ARM926EJ-S, ARM946E-S etc. ARM9 processors were used in early smartphones, feature phones, WiFi routers, portable media players and handheld gaming devices.

ARM11

The ARM11 microarchitecture was announced in 2002 built on the ARMv6 instruction set with Jazelle DBX Java acceleration. The ARM11 cores provided substantially higher performance through microarchitectural optimizations like:

- 8-stage pipeline to enable higher clock speeds
- Improved branch prediction and return stack
- Faster integer and floating-point arithmetic
- Pipelined memory architecture
- Wider cache interface and higher L1 cache bandwidth
- SIMD media instructions for multimedia
- Adaptive power control for power efficiency

Some common ARM11 family processors are ARM1136JF-S, ARM1156T2F-S, ARM1176JZF-S, ARM11 MPCore etc. ARM11 processors were used in smartphones like iPhone 3GS/4, Nokia N900 and other mobile devices.

ARM Cortex

Introduced from 2004 onwards, ARM Cortex processors represent the most advanced and contemporary ARM processor family optimized for high performance and energy efficiency. The Cortex cores are branded based on their performance and intended applications as **Cortex-A (Application), Cortex-R (Real-time) and Cortex-M (Microcontroller).** Some major distinguishing features of Cortex processors include:

- Advanced multi-stage pipelines and superscalar execution
- SIMD and VFPv3/v4 floating-point support
- NEON media processing engine
- Thumb-2 instruction set with CLZ, BLX operands
- Hardware divide and floating-point divide

- TrustZone security extensions
- Accelerator Coherency Port (ACP) interface
- Faster inter-core communication with Multiprocessor ILMs
- 64-bit memory addressing and IO coherence

The Cortex-A series targets high-performance application processing and includes variants like Cortex-A5, A7, A8, A9, A12, A15, A17, A32, A35, A53, A55, A57, A65, A72, A73 up to the latest Cortex-A77, A78 and A710. The Cortex-R series is optimized for real-time applications while the Cortex-M series targets low-power microcontroller applications.

Key Differences

Here is a summary of some of the key differences between the ARM processor families:

Features	ARM7	ARM9	ARM11	Cortex
Release timeframe	Early 1990s	2001	2002	2004 onwards
Instruction set	ARMv3/v4	ARMv4T	ARMv6	ARMv7 and above
Pipeline stages	3	5	8	8 and above
Process technology	0.35u to 0.18u	0.18u to 90nm	130nm to 65nm	45nm to 5nm
Clock Speed	**30-100 MHz**	**Up to 200 MHz**	**Up to 600 MHz**	**Up to 3 GHz+**
Features	Basic 3-stage pipeline, Thumb ISA for code density, MMU optional	Improved 5-stage pipeline, multimedia and DSP extensions	Deeper 8-stage pipeline, Jazelle DBX, media instructions	Advanced multi-stage pipelines, NEON, TrustZone, multi-core

Features	ARM7	ARM9	ARM11	Cortex
Applications	PDA, routers, printers	Feature phones, smartphones, gaming devices	Smartphones, mobile devices	Smartphones, tablets, servers, IoT devices

In summary, the ARM architecture has continued to evolve over the years with microarchitecture improvements to deliver higher performance, better power efficiency and advanced features from ARM7 to ARM11 to the latest Cortex processors.

ARM7 Core Architecture

The ARM7 core architecture is based on the original ARMv3 instruction set with a 3-stage integer pipeline. The major components of the ARM7 core include:

- **Integer pipeline** – 3-stage pipeline with Fetch, Decode and Execute stages
- **Register bank** – Thirty-seven 32-bit general purpose registers including PC and SP
- **Barrel shifter** – For shift and rotate operations
- **ALU** – Arithmetic Logic Unit for arithmetic and logic operations
- **Memory interface** – Supports up to 4GB physical address space
- **Write buffer** – To manage write bandwidth between processor and memory
- **EmbeddedICE** – Debug and trace support
- **Thumb decoder** – To decode 16-bit Thumb instructions

The integer pipeline comprises of three main stages – Fetch, Decode and Execute. Some of the key operations in the pipeline are:

- **Fetch** – Fetches instructions from I-cache or memory
- **Decode** – Decodes instructions into control signals
- **Execute** – Performs operations like ALU ops, address calculation, shifts, multiply
- **Memory** – Load/Store operations done in Execute stage
- **Writeback** – Writeback of operation results if required

The ARM7TDMI core also added a 16-bit Thumb instruction set to improve code density. The two main configurations supported are:

- **ARM state** – 32-bit ARM instruction set
- **Thumb state** – 16-bit compressed Thumb instruction set

The EmbeddedICE module provides debug capability and trace support for application development.

ARM9 Core Architecture

The ARM9 family was an evolutionary upgrade over ARM7 with key improvements:

- Higher performance 5-stage integer pipeline
- Introduction of Jazelle DBX for Java acceleration
- Enhanced Thumb-2 instruction set support
- Media processing extensions like saturating arithmetic

The ARM9 core has a 5-stage integer pipeline comprising:

- **Fetch** – Fetch instructions from I-cache or memory
- **Decode** – Decode instructions and read register operands
- **Execute** – Execute ALU operations and calculate memory addresses
- **Memory** – Perform load/store from data cache or memory
- **Writeback** – Writeback results to register bank

The pipeline enables faster instruction execution with reduced stalls. Key components of the ARM9 core include:

- **Register bank** – Thirty-seven 32-bit registers including 3 status registers
- **Barrel shifter** – Combined shifter and logic unit for shifts/rotates
- **ALU** – 32-bit Arithmetic Logic Unit
- **MAC** – Optional 32-bit multiplier accumulator unit
- **Jazelle DBX** – Hardware accelerator for Java bytecodes
- **Write buffer** – To reduce stalls in case of writes
- **System Control** – Handling interrupts, exceptions and coprocessors

The ARM9 implemented the Thumb-2 extensions to the Thumb ISA with conditional execution and branches. This enabled higher code density without compromising on performance.

ARM11 Core Architecture

The ARM11 core architecture was an evolution of the 8-stage pipeline implementing the ARMv6 instruction set. Some of the major improvements included:

- 8-stage pipeline for higher clock speeds
- Improved branch prediction and return stack
- 2-cycle multiplier for better arithmetic performance
- Stores buffer to reduce stalls on writes
- Operand forwarding unit to avoid pipeline bubbles
- Coprocessor interface for multimedia acceleration

The integer pipeline in ARM11 comprises the following key stages:

- **Fetch** – Fetch instructions from I-cache
- **Decode** – Decode instructions into control signals
- **Register Access** – Read register operands for execution
- **Execute** – Perform ALU ops, shift/rotate, multiply, branches
- **Data Cache** – Load/Store from data cache and memory
- **Second Data Cache** – Additional time for cache/memory access
- **Writeback** – Writeback results of operations
- **Register Write** – Write results into register bank

Some key components of the ARM11 core include:

- **Register bank** – Sixteen 32-bit registers and twenty-six 64-bit registers
- **Barrel shifter** – Combined shifter and logic unit
- **ALU** – 32-bit Arithmetic Logic Unit
- **MAC** – 32-bit multiplier Accumulator
- **Jazelle** – Acceleration for Java bytecodes
- **Write buffer** – 8-entry write buffer to reduce stalls
- **System Control** – Handling of interrupts, exceptions, coprocessors

The ARM11 core delivered substantially higher performance coupled with power efficiency through microarchitecture improvements like deeper pipeline, caching, forwarding and branch prediction.

ARM Cortex-A Series

The ARM Cortex-A series targets high-performance application processing requirements. Let us look at the microarchitecture of Cortex-A8 as an example.

The Cortex-A8 is an 8-stage superscalar pipeline with features like:

- Out-of-order execution for improved performance
- Program flow prediction to reduce stalls
- NEON media engine for SIMD processing
- Thumb-2 instruction set with efficient branch encoding
- Optional L2 cache interface for lower latency access

The integer pipeline consists of the following stages:

- **Fetch** – Fetch up to 2 instructions from L1 I-cache
- **Decode** – Decode instructions into micro-ops
- **Issue** – Issue up to 5 micro-ops for execution
- **Execute** – Execute ALU ops,branches, multiplies
- **Memory** – Perform load/store operations to L1 D-cache
- **Complete** – Signals that execution unit has completed
- **Retire** – Retires the instruction results
- **Writeback** – Writes back results to architectural state

Some of the key components of the Cortex-A8 core include:

- **Register files** – Thirty-one 64-bit integer registers, Thirty-two 64-bit NEON registers
- **Execution units** – 2 Integer ALUs, Load/Store unit, SIMD NEON unit
- **Branch predictor** – Advanced branch predictor for low misprediction
- **Instruction caches** – Separate L1 caches for Instruction and Data
- **Cache coherence** – Snooping support for cache coherency
- **Memory management** – MMU with 4KB pages and up to 40-bit physical addresses

The Cortex-A8 was one of the first Cortex processors to feature symmetric multiprocessing capabilities enabling multi-core designs.

Evolution of ARM Instruction Sets

The ARM instruction sets have continued to evolve with additions

MESI Protocol for Cache Coherence:

The **MESI protocol** (Modified, Exclusive, Shared, Invalid) is a **cache coherence protocol** used in multiprocessor systems to maintain consistency between caches that share the same memory.

Here's a breakdown of the MESI states and how they work:

MESI States:

1. **Modified (M)**:
    - The cache line is **modified** (dirty) — different from main memory.
    - Only this cache has the data, and it **must write it back** to memory before another cache can access it.
2. **Exclusive (E)**:
    - The cache line is the same as in main memory.
    - No other cache has this data, but it's **not modified** — just exclusively owned.
3. **Shared (S)**:
    - The data is **shared** among multiple caches.
    - It matches main memory and can be read by any processor, but **not written**.
4. **Invalid (I)**:
    - The cache line is **invalid** — it's no longer a valid copy and must be fetched from another cache or memory.

MESI Protocol Operation (Simplified):

- **Read Miss**:
    - If no other cache has the data: move to **Exclusive** (E).
    - If others have it in **S** or **E**: move to **Shared** (S).
- **Write Miss**:
    - Broadcasts an **invalidate** signal to other caches.
    - Transitions to **Modified** (M) after gaining exclusive write access.
- **Read Hit / Write Hit**:
    - Actions depend on the current state — e.g., writing in **Exclusive** upgrades to **Modified**.

Benefits of MESI:

- Reduces unnecessary memory writes by keeping track of dirty (modified) data.
- Improves performance by reducing memory traffic in multiprocessor systems.

Asynchronous Down Counter

In asynchronous/ripple counter output of the first flip-flop is provided as the clock to the second flip-flop **i.e** flip-flop(FF) are not clocked simultaneously. Circuit is simpler, but speed is slow.

What is Asynchronous Counter?

An offbeat counter, otherwise called a wave counter, is a computerized counter where going back and forth is associated with a series. The result of one flip-flop fills in as the clock input for the following flip-flop in the grouping. This implies that adjustments of state echo through the flip-slumps consistently, creating a setback for refreshing the ensuing flip-flops. Therefore, nonconcurrent counters are easier, but slower, than simultaneous counters, where all flip-flops are driven by a typical clock signal. In many cases, they are utilized in straightforward counting applications, yet they may experience the ill effects of timing issues because of their innate postponements.

Counter

Counters are used to count the clock pulses. The clock pulses occur at regular intervals. They are used to measure the time and frequency. We can also say that counters are sequential circuits which change their pre-defined states with the help of clock pulses. They are constructed with flip flops and logic gates.

They are of two types –

1. Asynchronous counter
2. Synchronous counter

In the asynchronous counter, an external clock pulse is provided for only the first flip flop, thereafter the output of the 1st FF acts as a clock pulse for the second FF and so on. In the case of synchronous FFs, all the flip flops are triggered simultaneously by an external clock pulse.
Please refer **this**, to understand how an asynchronous counter works.

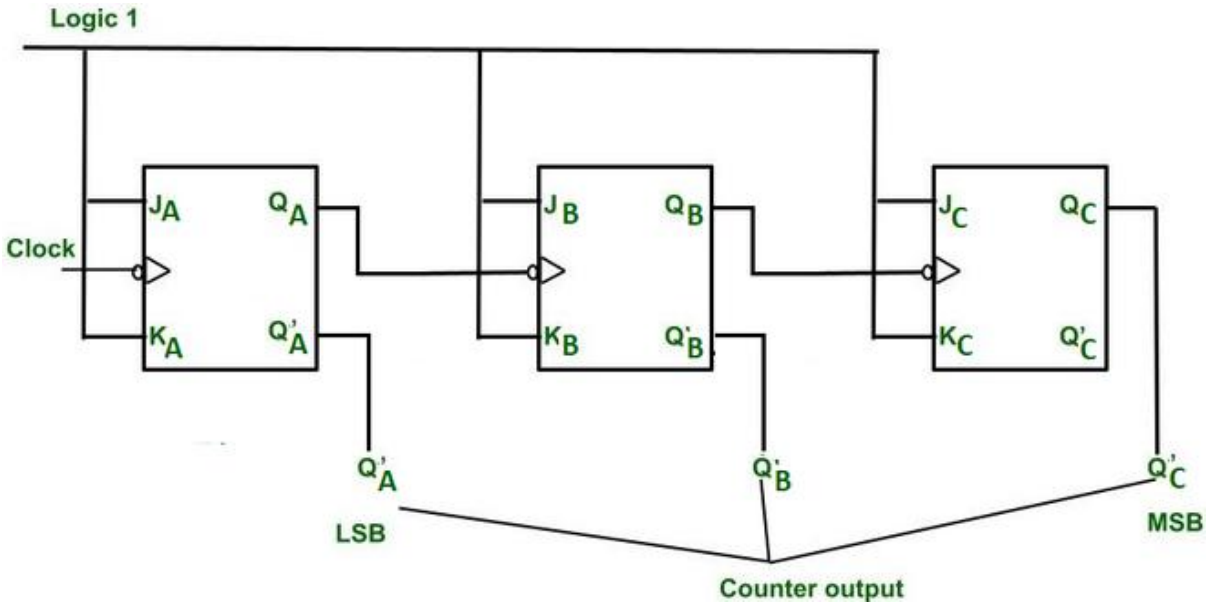
3 Bit Asynchronous Down Counter

- For the 3 bit counter, we require 3 flip flops and we can generate 23 = 8 state and count(111 110 … 000).

- We can generate down counting states in an asynchronous down counter by two ways.

Method 1

In this implementation, the clock pulse(of 50% duty cycle) is given to only the first FF. Thereafter, the output of the first FF is feed as a clock to second FF and the output of the second FF is feed as the clock for the third FF. But the complemented output is taken from each FF(i.e. same as Up counter but output states are complemented). Here QA is LSB and QC is MSB.

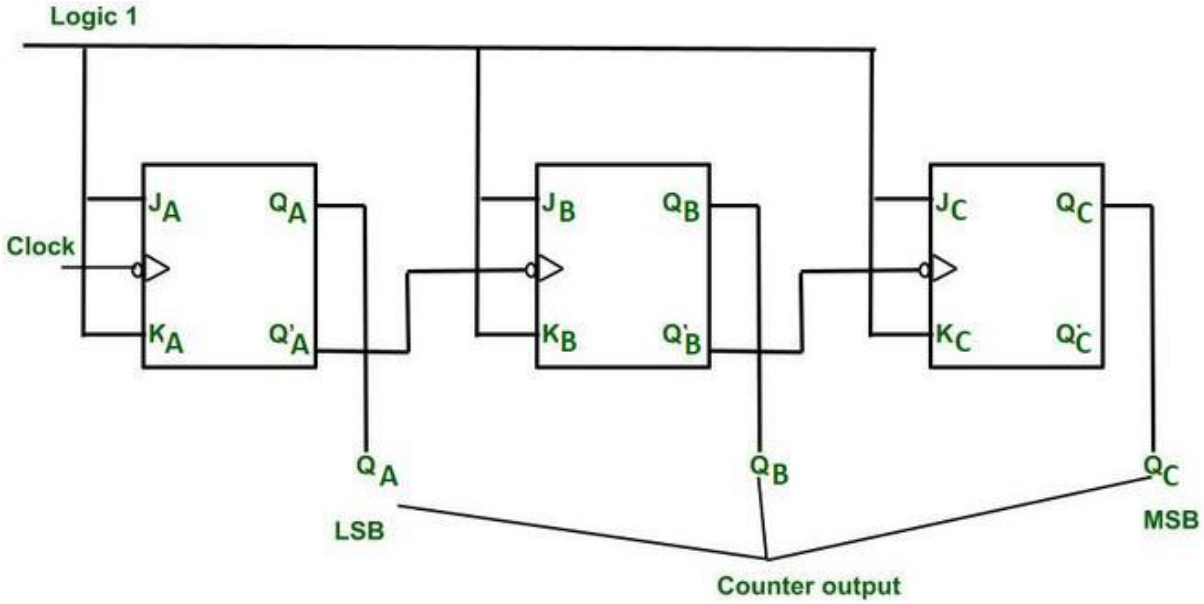


State table –

Clock	Q_C	Q_B	Q_A	Q'_C	Q'_B	Q'_A
Initially	0	0	0	1	1	1
1st	0	0	1	1	1	0
2nd	0	1	0	1	0	1
3rd	0	1	1	1	0	0
4th	1	0	0	0	1	1
5th	1	0	1	0	1	0
6th	1	1	0	0	0	1
7th	1	1	1	0	0	0

Method 2

In this implementation, the clock pulse is given to only the first FF. Thereafter, the complemented output of the first FF(Q'A) is feed as a clock to the second FF and complemented output(i.e. Q'B) of the second FF is feed as a clock for the third FF. But the output(Q) is taken from each FF.



By both implementations, we can acquire the same counting states.

Timing diagram –
The working of counter can be easily understood by the timing diagrams.

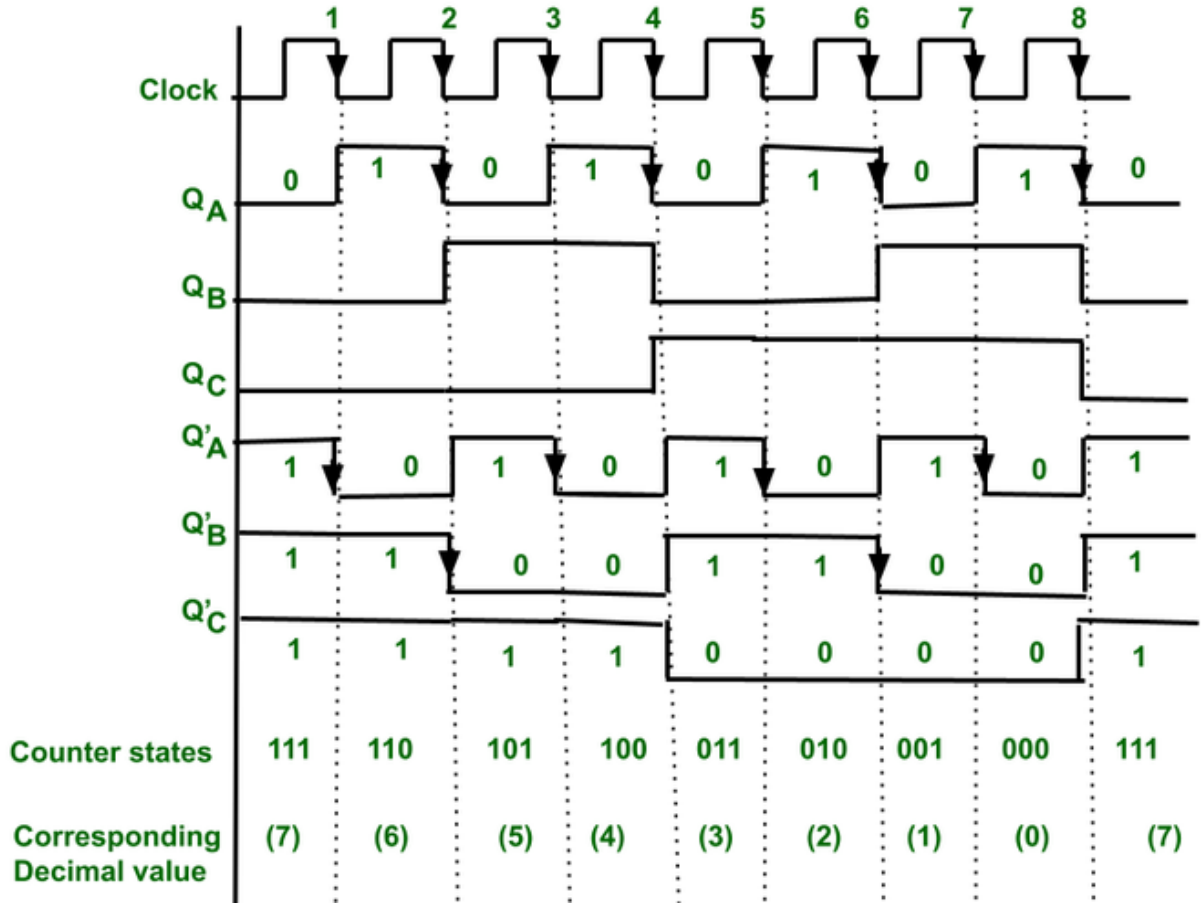
Explanation –

- Here the -ve edge clock pulse is used(i.e. the counter state transition can occur only at falling edge of the clock pulse) , therefore toggling will take place.
- All the counter states are generated by the frequency division.
- Initially QA =0 QB =0 and QC= 0.

First Circuit

- The -ve edge clock pulse is provided to 1st counter. Therefore, the output state of the first counter(i.e. QA) will be toggled at every falling edge of the clock pulse.
- As the QA is feed as a clock to the second FF, therefore the output state (i.e. QB) will be toggled at every falling edge of QA.
- In the same manner, the QB acts as clock for the third FF, therefore the output state (QC) of the third FF will be toggled for every falling edge of QB.
- As we know, this is the working of UP counter, but here the output is taken as in complemented form (i.e. Q'C Q'B Q'A), therefore we get the complemented outputs(i.e. down counting 111 to 000)

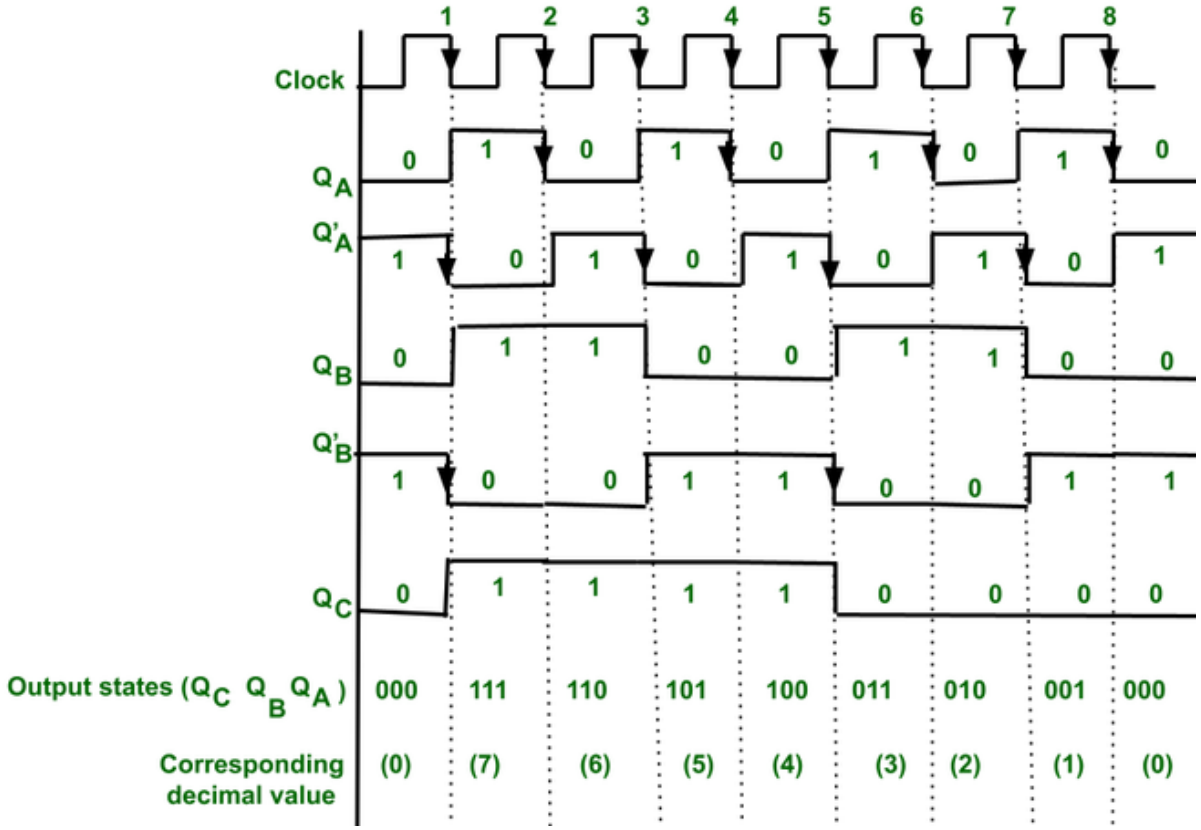
- After the 8th falling edge of the external clock pulse, the counter is reset to 000.



Timing diagram for first circuit

Second circuit –

- The -ve edge clock pulse is provided to 1st counter. Therefore, the output state of the first counter(i.e. QA) will be toggled at every falling edge of the clock pulse.
- As the complemented output(Q'A) is feed as a clock to second FF, therefore the output state (i.e. QB) will be toggled at every falling edge of Q'A.
- In the same manner, the Q'B acts as a clock for the third FF, therefore the output state (QC) of the third FF will be toggled for every falling edge of Q'B.
- In this case, the outputs from the three FF are taken as (QC QB QA).
- After the 8th falling edge of the external clock pulse, the counter is reset to 000.
- Here, the down counting states can be acquired after first -ve edge clock pulse(i.e. after first -ve pulse the counting output becomes 111).



Timing diagram for second circuit

Advantages of Asynchronous Counter

- Due to accumulation of propagation delay, they are used in low speed circuits.
- They are simple to design.
- They are used in mod n counter, and in divide by n counter that divides the input by n(i.e. n is integer).

Disadvantages of Asynchronous Counter

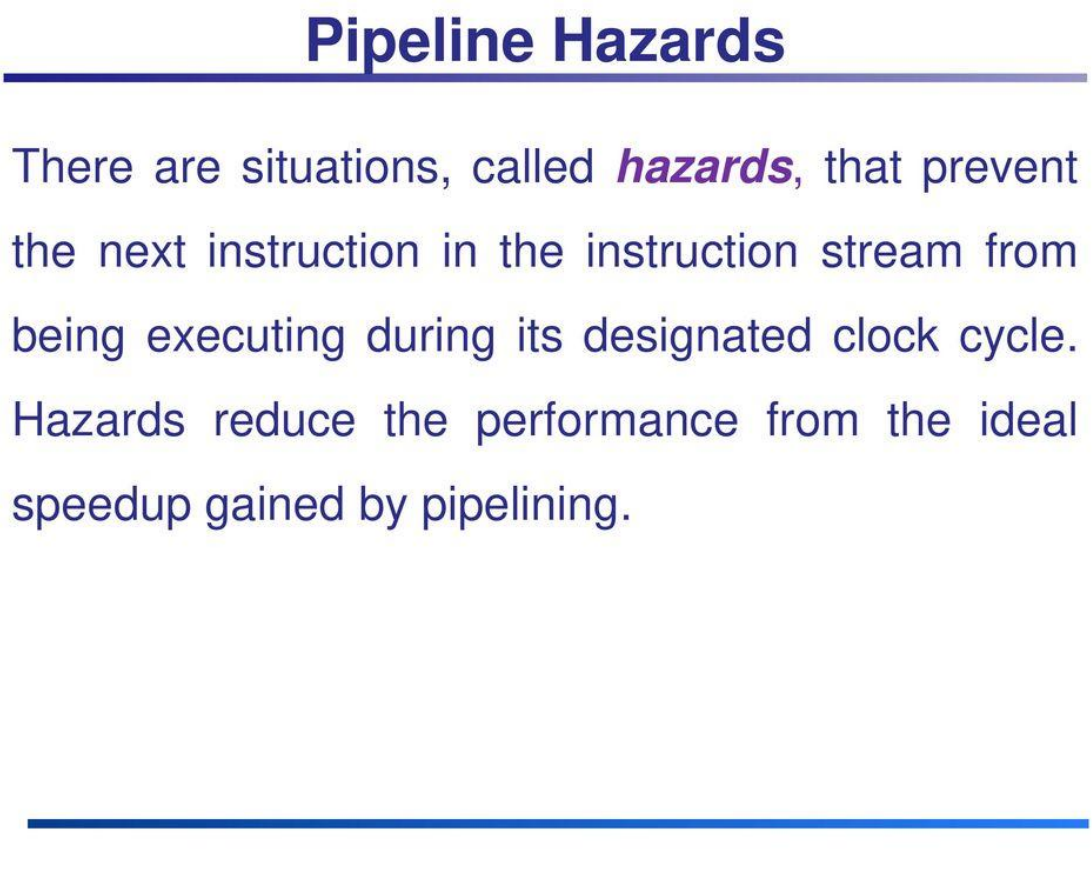
- As the number of flip-flops increases, the propagation delay also increases.
- For high clock frequencies, counting errors may occur because of propagation delay. **For example**:-Due to propagation delay(by counter), the toggling action takes place after some delay, so by this wrong counting states may be generated.
- In the case of truncated counter (i.e. in mod counters), extra feedback logic(i.e. logic gate) is required.

Conclusion

A nonconcurrent down counter, or wave down counter, is a sort of computerized counter that includes descending in twofold successions. Not at all like coordinated counters, its flip-flops are not all determined by a solitary clock signal; all things considered, each flip-failure's

result sets off the following flip-flop in the grouping. This flowing impact presents deferrals and potential timing issues, making the counter more slow compared with simultaneous plans. Regardless of this, nonconcurrent down counters are esteemed for their straightforwardness and simplicity of execution in essential counting applications. Nonetheless, for additional intricate or rapid applications, simultaneous counters may be preferred to stay away from swell postponements and guarantee precise timing.

Pipeline Hazards:



Pipeline Hazards

- There are three classes of hazards:
  - **Structural Hazards**: They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
  - **Data Hazards**: They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
  - **Control Hazards**: They arise from the pipelining of branches and other instructions that change the PC

Structural Hazards

1. When a machine is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.
2. If some combination of instructions cannot be accommodated because of a *resource conflict*, the machine is said to have a **structural hazard**.

Example

- A machine has shared a **single-memory** pipeline for data and instructions. As a result, when an instruction contains a data-memory reference (load), it will conflict with the instruction reference for a later instruction (instr 3):

Clock cycle number								
instr	1	2	3	4	5	6	7	8
load	IF	ID	EX	**MEM**	WB			
Instr 1		IF	ID	EX	MEM	WB		
Instr 2			IF	ID	EX	MEM	WB	
Instr 3				**IF**	ID	EX	MEM	WB

Solution (1/2)

- To resolve this, we *stall* the pipeline for one clock cycle when a data-memory access occurs. The effect of the stall is actually to occupy the resources for that instruction slot. The following table shows how the stalls are actually implemented.

Clock cycle number									
instr	1	2	3	4	5	6	7	8	9
load	IF	ID	EX	**MEM**	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Instr 3				**stall**	**IF**	ID	EX	MEM	WB

Solution (2/2)

- Another solution is to use separate instruction and data memories.
- ARM used **Harvard** architecture, so we do not have this hazard

Data Hazards

- **Data hazards** occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on the unpipelined machine.

		Clock cycle number								
		1	2	3	4	5	6	7	8	9
ADD	R1,R2,R3	IF	ID	EX	MEM	**WB**				
SUB	R4,R5,R1		IF	ID_{sub}	EX	MEM	WB			
AND	R6,R1,R7			IF	ID_{and}	EX	MEM	WB		
OR	R8,R1,R9				IF	ID_{or}	EX	MEM	WB	
XOR	R10,R1,R11					IF	ID_{xor}	EX	MEM	WB

Forwarding

- The problem with data hazards, introduced by this sequence of instructions can be solved with a simple hardware technique called ***forwarding***.

Clock cycle number								
		1	2	3	4	5	6	7
ADD	R1,R2,R3	IF	ID	EX	MEM	**WB**		
SUB	R4,R5,R1		IF	ID_{sub}	EX	MEM	WB	
AND	R6,R1,R7			IF	ID_{and}	EX	MEM	WB

Forwarding

- *Forwarding* involves feeding output data into a previous stage of the pipeline.
- Forwarding is implemented by feeding back the output of an instruction into the previous stage(s) of the pipeline as soon as the output of that instruction is available.

Clock cycle number								
		1	2	3	4	5	6	7
ADD	R1,R2,R3	IF	ID	EX	MEM	**WB**		
SUB	R4,R5,R1		IF	ID_{sub}	EX	MEM	WB	
AND	R6,R1,R7			IF	ID_{and}	EX	MEM	WB

Forward Data

Clock cycle number								
		1	2	3	4	5	6	7
ADD	R1,R2,R3	IF	ID	EX_{add}	MEM_{add}	WB		
SUB	R4,R5,R1		IF	ID	EX_{sub}	MEM	WB	
AND	R6,R1,R7			IF	ID	EX_{and}	MEM	WB

- The first forwarding is for value of **R1** from **EX_{add}** to **EX_{sub}**.
  The second forwarding is also for value of **R1** from **MEM_{add}** to **EX_{and}**.
  This code now can be executed without stalls.
- Forwarding can be generalized to include passing the result directly to the functional unit that requires it
- A result is forwarded from the output of one unit to the input of another, rather than just from the result of a unit to the input of the same unit.

Without Forward

Clock cycle number										
		1	2	3	4	5	6	7	8	9
ADD	R1,R2,R3	IF	ID	EX	MEM	**WB**				
SUB	R4,R5,R1		IF	*stall*	*stall*	ID_{sub}	EX	MEM	WB	
AND	R6,R1,R7			*stall*	*stall*	IF	ID_{and}	EX	MEM	WB

Data Forwarding

- Data dependency arises when an instruction needs to use the result of one of its predecessors before the result has returned to the register file => pipeline hazards
- Forwarding paths allow results to be passed between stages as soon as they are available
- 5-stage pipeline requires each of the three source operands to be forwarded from any of the intermediate result registers
- Still one load stall

```
LDR rN, […]
ADD r2,r1,rN   ;use rN immediately
```

  - One stall
  - Compiler rescheduling

Stalls are required

		1	2	3	4	5	6	7	8
LDR	R1,@(R2)	IF	ID	EX	MEM	WB			
SUB	R4,R1,R5		IF	ID	EX_{sub}	MEM	WB		
AND	R6,R1,R7			IF	ID	EX_{and}	MEM	WB	
OR	R8,R1,R9				IF	ID	EXE	MEM	WB

- The load instruction has a delay or latency that cannot be eliminated by forwarding alone.

The Pipeline with one Stall

		1	2	3	4	5	6	7	8	9
LDR	R1,@(R2)	IF	ID	EX	MEM	WB				
SUB	R4,R1,R5		IF	ID	*stall*	EX_{sub}	MEM	WB		
AND	R6,R1,R7			IF	*stall*	ID	EX	MEM	WB	
OR	R8,R1,R9				*stall*	IF	ID	EX	MEM	WB

- The only necessary forwarding is done for R1 from **MEM** to **EX_{sub}**.

Control hazards

- ***Control hazards*** can cause a greater performance loss for ARM pipeline than data hazards.
- When a branch is executed, it may or may not change the PC (program counter) to something other than its current value plus 4.
- The simplest method of dealing with branches is ***to stall*** the pipeline as soon as the branch is detected until we reach the **EX** stage

Branch	IF	ID	EXE	MEM	WB				
Branch successor		IF (stall)	*Stall*	IF	ID	EXE	MEM	WB	
Branch successor+1					IF	ID	EXE	MEM	WB

Introduction

A major driving force in the microprocessor industry is the never ending desire to miniaturize things. Smaller transistors require less voltage to operate and thus consume less power and produce less heat. Smaller interconnect distances also allow for faster clock speeds. Perhaps most important of all, smaller die areas lead to cheaper processors since more chips can fit in a single wafer. The first microprocessor made by Intel was the 4004, which had 2300 transistors. Today's chips, on the other hand, incorporate 5 to 20 million transistors. So what do they do with all those transistors?

A major hog of real estate is, of course, caches. Caches, and any IC (integrated circuit) based memory device, must have many wires running to and from the read and write ports. For on-chip caches, the load/store unit in the CPU must be able to access every location in the cache from both the read and write ports. The situation is even worse when there are more than one load/store units. That's a lot of wires! In the Pentium Pro, for example, a single package includes both the CPU chip and a L2 cache chip; the CPU chip has about 5 million transistors, while the cache chip has about 15 million transistors.

However, even accounting for caches, there is still a large increase in the number of transistors in today's chips compared to the 4004. Obviously, microprocessors are becoming increasingly more complex. We can understand this increasing complexity since chip designers want to create fast processors which are at the same time affordable. As process technology improved and more transistors could be fitted in the same die area, it became cost effective to add newer or improved features to the processor in an attempt to increase its effective speed. One of these improvements is dynamic scheduling.

Dynamic scheduling, as its name implies, is a method in which the hardware determines which instructions to execute, as opposed to a statically scheduled machine, in which the compiler determines the order of execution. In essence, the processor is executing instructions out of order. Dynamic scheduling is akin to a data flow machine, in which instructions don't execute based on the order in which they appear, but rather on the availability of the source operands. Of course, a real processor also has to take into account the limited amount of resources available. Thus instructions execute based on the availability of the source operands as well as the availability of the requested functional units.

Dynamically scheduled machines can take advantage of parallelism which would not be visible at compile time. They are also more versatile as code does not necessarily have to be recompiled to run efficiently since the hardware takes care of much of the scheduling. In a statically scheduled machine, code would have to be recompiled to take advantage of the machine's particular hardware. (All of this is assuming the machines use the same instruction set architecture. Of course, the code would have to be recompiled no matter what if the machines used different ISAs.)

Scoreboard

Introduced in the CDC 6600, the scoreboard is one method of implementing dynamic scheduling. Like any good dynamically scheduled machine, the scoreboard monitors each instruction waiting to be dispatched. Once it determines that all the source operands and the required functional units are available, it dispatches the instruction so that it can be executed. However, the scoreboard is limited in that it does not handle WAR and WAW hazards very well.

The scoreboard is a central location where information about the currently active instructions is kept. The scoreboard determines when and where an instruction begins and ends execution. In a scoreboard machine, instructions go through four main stages:

1. Issue - In the issue stage, the scoreboard checks for an available functional unit as well as any potential WAW hazards. If it detects a WAW hazard or there are no functional units available for the instruction, the instruction stalls.
2. Read operands - In this stage, the scoreboard checks for the availability of the source operands. If the source operands are available, the scoreboard instructs the functional unit to read the operands from the register file and begin executing, i.e. it resolves RAW hazards. An operand is considered available if no currently issued instruction is going to write to it, or if it is currently being written to the register file. This last idea is important, since it is the reason why WAW hazards cause stalls whereas (as we'll see) WAR hazards are allowed to propagate further down the pipeline. If there were more than one active instruction writing to the same register, there would be ambiguity about when an operand should be read from the register file (i.e. the scoreboard wouldn't know whether the operand should be read after the first or second write).
3. Execution - The execution stage; nothing fancy here, except the functional unit notifies the scoreboard when it has finished execution.
4. Write result - Once the scoreboard receives the notification that a functional unit has finished execution, it checks for any potential WAR hazards, i.e if an earlier instruction which is still in the read operands stage has the destination register as one of its source operands. If a WAR hazard is found, the scoreboard instructs the functional unit to stall until the hazard clears. In the meantime, the functional unit is unavailable to other instructions.

As an example of how scoreboarding works, consider the following code:

```
Loop:  LD     F2,0(R1)
       ADDD   F6,F2,F4
```

```
        MULTD F8,F6,F0
        SUBI  R1,R1,#8
        BNEZ  R1,Loop
```

However, we'll assume that the compiler was nice enough to unroll the loop for us without rescheduling the code (we'll see later how dynamic branch prediction and a reorder buffer allows the loop unrolling to be done by the hardware). Thus the code we'll examine looks like:

```
Loop:   LD    F2,0(R1)
        ADDD  F6,F2,F4
        MULTD F8,F6,F0
        LD    F10,-8(R1)
        ADDD  F12,F10,F4
        MULTD F14,F12,F0
        SUBI  R1,R1,#16
        BNEZ  R1,Loop
```

Tomasulo's Algorithm

Tomasulo's algorithm is another method of implementing dynamic scheduling. This scheme was invented by Robert Tomasulo, and was first used in the IBM 360/91. Tomasulo's algorithm differs from scoreboarding in that it uses register renaming to eliminate output and anti-dependences, i.e. WAW and WAR hazards. Output and anti-dependences are just name dependences, there is no actual data dependence. For example, the code below

```
        MULTD F4,F2,F2
        ADDD  F2,F0,F6
```

contains an anti-dependence since the first instruction reads from F2 and the second instruction writes to F2 (a WAR hazard). However, there is no data dependence, as is shown by the code below (assumin F8 is unused):

```
        MULTD F4,F2,F2
        ADDD  F8,F0,F6
```

The anti-dependence is removed without changing the semantics of the code simply by changing the F2 to an F8. However, we don't have to use F8, we can use any available

register. In fact, suppose there were some physical registers in the chip that the instruction set was not aware of, i.e. there were some extra registers. We can use those extra registers and leave the rest for the compiler to play with. This is what register renaming is; it allows the hardware to detect a name dependence and eliminate it by storing the result of an instruction somewhere else. Thus, with register renaming,

```
MULTD F4,F2,F2
ADDD  F2,F0,F6
```

the add can execute and finish before the multiply starts even though looking at the code suggests that would result in an erroneous answer.

Tomasulo's algorithm implements register renaming through the use of what are called reservation stations. Reservation stations are buffers which fetch and store instruction operands as soon as they're available. Source operands point to either the register file or to other reservation stations. Each reservation station corresponds to one instruction. Once all source operands are available, the instruction is sent for execution, provided a functional unit is also available. Once execution is complete, the result is buffered at the reservation station. Thus, unlike in scoreboarding where the functional unit would stall during a WAR hazard, the functional unit is free to execute another instruction. The reservation station then sends the result to the register file and any other reservation station which is waiting on that result. WAW hazards are handled since only the last instruction (in program order) actually writes to the registers. The other results are buffered in other reservation stations and are eventually sent to any instructions waiting for those results. WAR hazards are handled since reservation stations can get source operands from either the register file or other reservation stations (in other words, from another instruction). In Tomasulo's algorithm, the control logic is distributed among the reservation stations, whereas in scoreboarding, the scoreboard keeps track of everything.

Last Minute Notes Computer Organization

Table of Content

- Basic Terminology
- Instruction Set and Addressing Modes
- Instruction Design and Format
- Control Unit
- Memory Organization
- I/O Interface

- Pipelining
- IEEE Standard 754 Floating Point Numbers

Basic Terminology

- **Control Unit –** A control unit (CU) handles all processor control signals. It directs all input and output flow, fetches the code for instructions and controlling how data moves around the system.
- **Arithmetic and Logic Unit (ALU) –** The arithmetic logic unit is that part of the CPU that handles all the calculations the CPU may need, e.g. Addition, Subtraction, Comparisons. It performs Logical Operations, Bit Shifting Operations, and Arithmetic Operation.

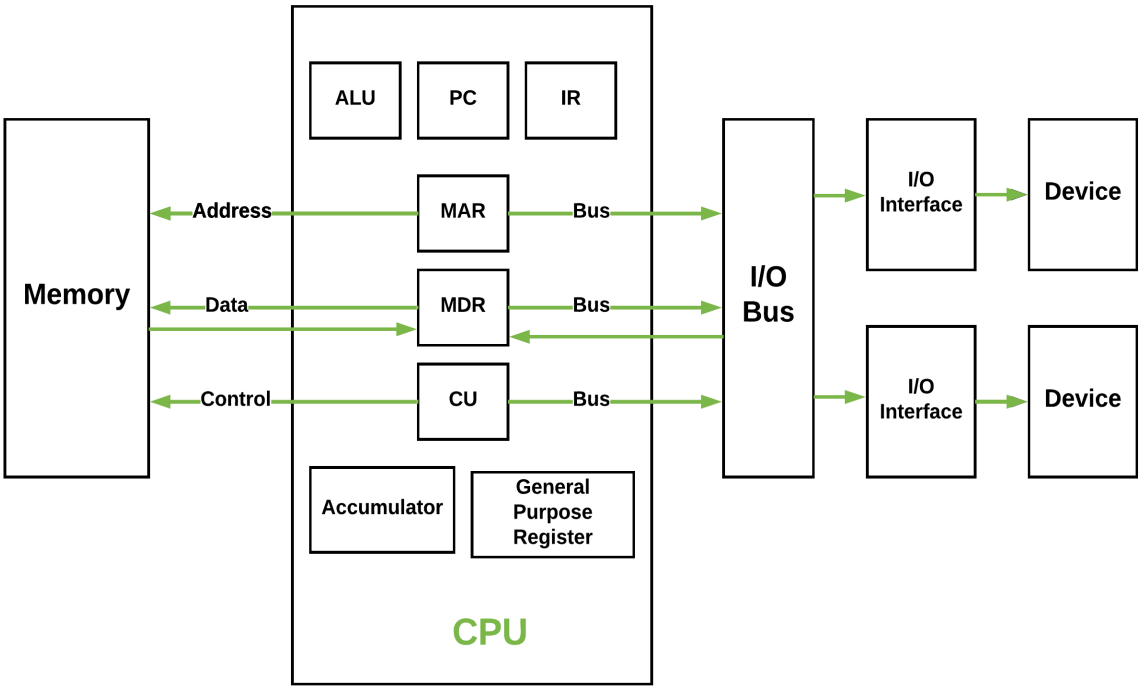


Figure – Basic CPU structure, illustrating ALU

- **Main Memory Unit (Registers) –**

1. **Accumulator:** Stores the results of calculations made by ALU.
2. **Program Counter (PC):** Keeps track of the memory location of the next instructions to be dealt with. The PC then passes this next address to Memory Address Register (MAR).
3. **Memory Address Register (MAR):** It stores the memory locations of instructions that need to be fetched from memory or stored into memory.
4. **Memory Data Register (MDR):** It stores instructions fetched from memory or any data that is to be transferred to, and stored in, memory.

5. **Current Instruction Register (CIR):** It stores the most recently fetched instructions while it is waiting to be coded and executed.
6. **Instruction Buffer Register (IBR):** The instruction that is not to be executed immediately is placed in the instruction buffer register IBR.

- **Input/Output Devices –** Program or data is read into main memory from the *input device* or secondary storage under the control of CPU input instruction. *Output devices* are used to output the information from a computer.
- **Buses –** Data is transmitted from one part of a computer to another, connecting all major internal components to the CPU and memory, by the means of Buses. Types:
    1. **Data Bus:** It carries data among the memory unit, the I/O devices, and the processor.
    2. **Address Bus:** It carries the address of data (not the actual data) between memory and processor.
    3. **Control Bus:** It carries control commands from the CPU (and status signals from other devices) in order to control and coordinate all the activities within the computer.

Types of Computer Architecture

1. Von Neumann Architecture

- It uses one memory to store both the program instructions and the data.
- The CPU fetches instructions and data from the same place, one after another.
- This design is simpler and used in most traditional computers.

2. Harvard Architecture

- It uses two separate memories: one for program instructions and another for data.
- The CPU can fetch instructions and data at the same time, making it faster.
- This design is used in modern systems like embedded processors.

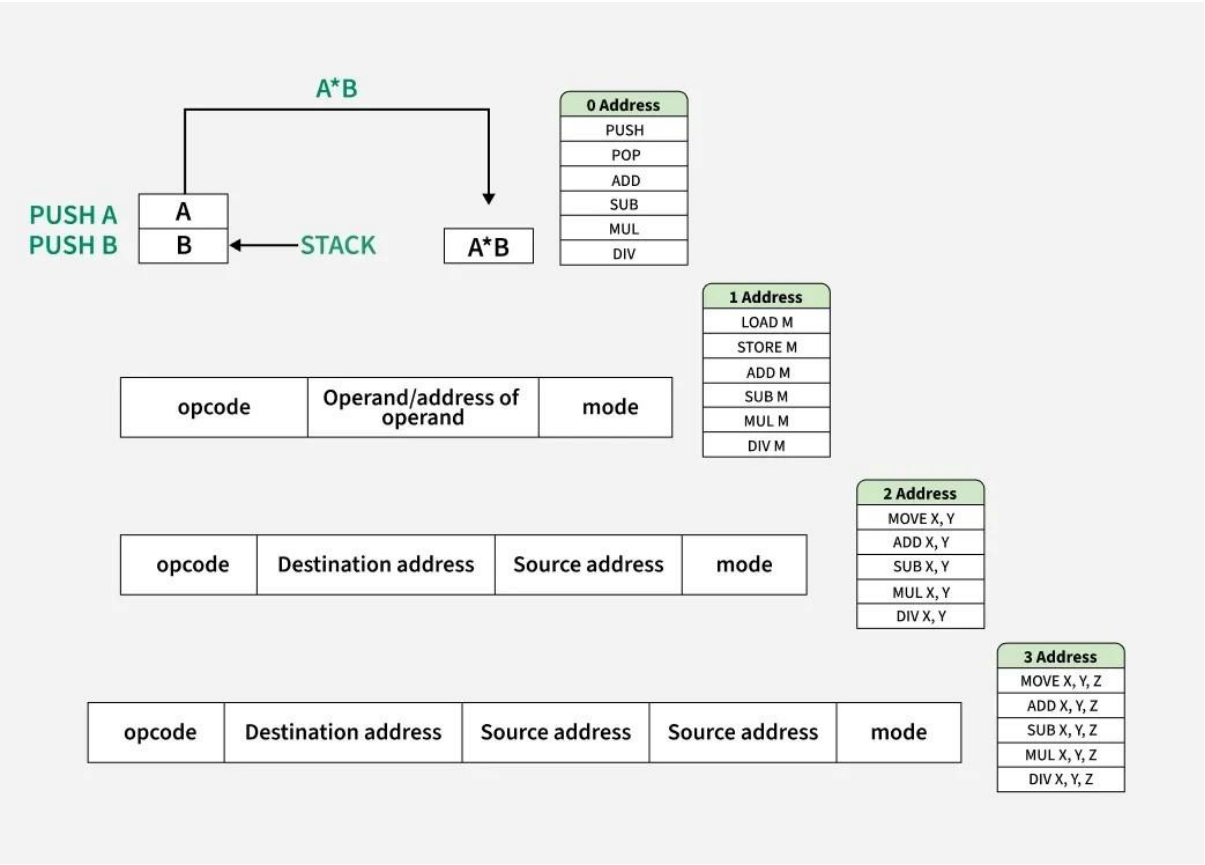
Instruction Set and Addressing Modes

Instruction Formats (Zero, One, Two and Three Address Instruction)

A instruction is of various length depending upon the number of addresses it contain. Generally CPU organization are of three types on the basis of number of address fields:

1. Single Accumulator organization
2. General register organization

3. Stack organization



Read more about Instruction Format, Here.

Basic Machine Instructions in COA

Machine instructions are the basic commands given to the processor to perform tasks. They operate directly on the hardware.

Types of Machine Instructions

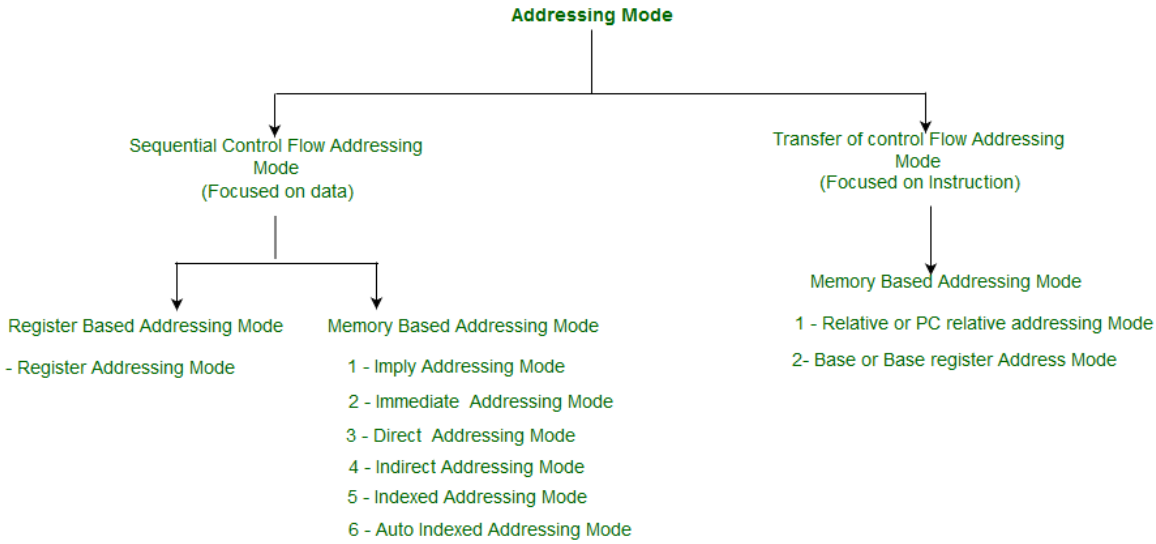
- **Data Transfer Instructions**
    - Move data between memory, registers, or I/O devices.
    - Example: LOAD, STORE, MOVE.
- **Arithmetic Instructions**
    - Perform arithmetic operations like addition, subtraction, multiplication, and division.
    - Example: ADD, SUB, MUL, DIV.

- **Logical Instructions**
    - Perform logical operations such as AND, OR, NOT, XOR.
    - Example: AND, OR, NOT, XOR.
- **Control Transfer Instructions**
    - Change the sequence of execution (jump, branch, or call).
    - Example: JUMP, CALL, RET.
- **Input/Output Instructions**
    - Allow communication between the processor and external devices.
    - Example: IN, OUT.
- **Shift and Rotate Instructions**
    - Shift or rotate bits in a register.
    - Example: SHL (Shift Left), SHR (Shift Right), ROL (Rotate Left), ROR (Rotate Right).

Components of an Instruction

- **Opcode**: Specifies the operation to perform (e.g., ADD, SUB).
- **Operands**: Data to be operated on (e.g., registers, memory locations).

Addressing Modes



The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually executed. An assembly language program instruction consists of two parts :

Opcode	Operand

Addressing Mode	Description	Example
Immediate Addressing	The operand is directly given in the instruction.	ADD R1, 5 (Add 5 to R1)
Register Addressing	The operand is stored in a register.	ADD R1, R2 (Add R2 to R1)
Direct Addressing	The operand is in memory, and the memory address is specified directly in the instruction.	LOAD R1, 1000 (Load data from memory address 1000 into R1)
Indirect Addressing	The address of the operand is stored in a register or memory location, not directly in the instruction.	LOAD R1, (R2) (Load data from the memory address stored in R2 into R1)
Register Indirect	Similar to indirect addressing, but specifically uses registers to hold the address of the operand.	LOAD R1, (R3) (Use R3 as pointer)
Indexed Addressing	The operand's address is calculated by adding an index (offset) to a base address stored in a register.	LOAD R1, 1000(R2) (Load data from memory address 1000 + R2 into R1)
Base Addressing	The base address is stored in a register, and the operand's offset is specified in the instruction.	LOAD R1, 200(RB) (RB = Base Register)

Addressing Mode	Description	Example
Relative Addressing	The operand's address is determined by adding an offset to the current program counter (PC).	JUMP 200 (Jump to PC + 200)
Implicit Addressing	The operand is implied by the instruction itself (no explicit address or operand).	CLR (Clear accumulator)

Effective address or Offset: An offset is determined by adding any combination of three address elements: displacement, base and index.

Read more about Addressing Modes, Here.

RISC vs CISC

RISC	CISC
Focus on software	Focus on hardware
Uses only Hardwired control unit	Uses both hardwired and microprogrammed control unit
Transistors are used for more registers	Transistors are used for storing complex Instructions
Fixed sized instructions	Variable sized instructions
Can perform only Register to Register Arithmetic operations	Can perform REG to REG or REG to MEM or MEM to MEM
Requires more number of registers	Requires less number of registers

RISC	CISC
Code size is large	Code size is small
An instruction executed in a single clock cycle	Instruction takes more than one clock cycle
An instruction fit in one word.	Instructions are larger than the size of one word
Simple and limited addressing modes.	Complex and more addressing modes.
RISC is Reduced Instruction Cycle.	CISC is Complex Instruction Cycle.
The number of instructions are less as compared to CISC.	The number of instructions are more as compared to RISC.
It consumes the low power.	It consumes more/high power.
RISC is highly pipelined.	CISC is less pipelined.
RISC required more RAM .	CISC required less RAM.
Here, Addressing modes are less.	Here, Addressing modes are more.

Read more about RISC vs CISC, Here.

Instruction Design and Format

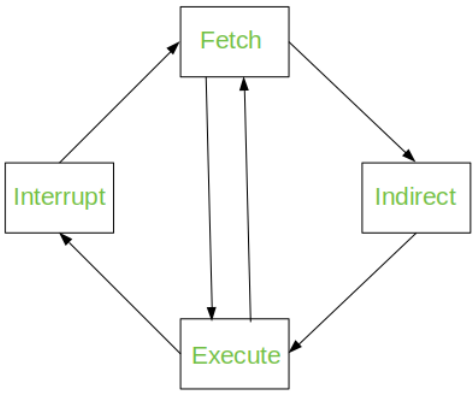
CPU Registers

The instruction cycle involves multiple registers in the CPU to fetch, decode, execute and store results.

- **Program Counter (PC)**

- Holds the address of the next instruction to be executed.
- Updates after each instruction fetch.

- **Instruction Register (IR)**
  - Stores the currently fetched instruction.
  - Used by the control unit for decoding.
- **Memory Address Register (MAR)**
  - Holds the memory address of the data or instruction to be fetched or stored.
- **Memory Data Register (MDR)** (or **Memory Buffer Register, MBR**)
  - Temporarily holds the data being transferred to/from memory.
- **Accumulator (AC)**
  - Stores intermediate arithmetic and logic results during execution.
- **General Purpose Registers (GPR)**
  - Temporary storage for operands, results, or data during execution.
- **Temporary Register (TR)**
  - Stores intermediate data during complex operations or instruction execution.
- **Status Register / Flag Register**
  - Stores condition flags (e.g., zero, carry, overflow) to indicate the result of operations.
- **Stack Pointer (SP)**
  - Points to the top of the stack in memory, used during function calls or interrupts.



The Instruction Cycle

Flag Registers

Status Flags

- **Zero Flag (Z):** When an arithmetic operation results in zero, the flip-flop called the Zero flag – which is set to one.
- **Carry flag (CY):** After an addition of two numbers, if the sum in the accumulator is larger than eight bits, then the flip-flop uses to indicate a carry called the Carry flag, which is set to one.
- **Parity (P):** If the result has an even number of 1s, the flag is set to 1; for an odd number of 1s the flag is reset.
- **Auxiliary Carry (AC):** In an arithmetic operation, when a carry is generated from lower nibble and passed on to higher nibble then this register is set to 1.
- **Sign flag(S):** It is a single bit in a system status (flag) register used to indicate whether the result of the last mathematical operation resulted in a value in which the most significant bit was set.

Instruction Cycle

1. Fetch: The CPU retrieves the next instruction from memory using the Program Counter (PC).

2. Indirect: If the instruction uses an **indirect addressing mode**, the effective memory address of the operand is resolved. Example: For LOAD R1, (100), the CPU fetches the address stored at memory location 100.

3. Execute: The CPU performs the operation specified by the instruction (e.g., arithmetic, logical, control).

4. Interrupt: If an interrupt request occurs (e.g., hardware interrupt or software exception), the CPU temporarily halts the current execution to service the interrupt. After servicing, the CPU resumes the instruction cycle.

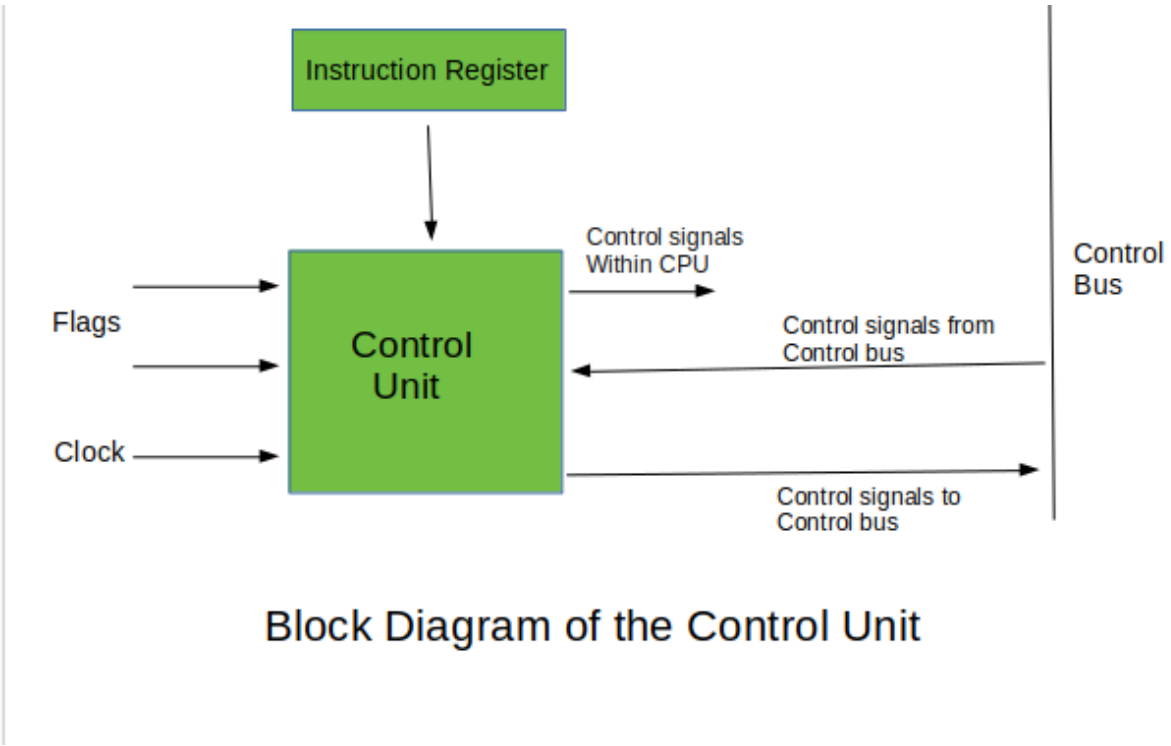
Standard Instruction Cycle

This includes the basic steps for executing instructions:

- **Fetch**: Retrieve the instruction from memory.
- **Decode**: Identify the operation and operands.
- **Execute**: Perform the operation.
- **Store (Write Back)**: Save the result (if any).

Read more about Instruction Cycle, Here.

Control Unit



Block Diagram of the Control Unit

The Control Unit (CU) is a core component of the CPU that directs its operation by generating control signals. It manages the execution of instructions by coordinating with the ALU, registers, and memory.

Types of Control Units

Hardwired Control Unit –

- Fixed logic circuits that correspond directly to the Boolean expressions are used to generate the control signals.
- Hardwired control is faster than micro-programmed control.
- A controller that uses this approach can operate at high speed.
- RISC architecture is based on hardwired control unit.

Micro-programmed Control Unit –

- The control signals associated with operations are stored in special memory units inaccessible by the programmer as Control Words.
- Control signals are generated by a program are similar to machine language programs.
- Micro-programmed control unit is slower in speed because of the time it takes to fetch microinstructions from the control memory.

- There are two type Micro-programmed control Unit:
    - **Horizontal Micro-programmed control Unit-** The control signals are represented in the decoded binary format that is 1 bit/CS.
    - **Vertical Micro-programmed control Unit –** The control signals re represented in the encoded binary format. For N control signals- Logn(N) bits are required.

Read more about Hardwired CU vs Micro-programmed CU, Here.

Microprogram: Program stored in memory that generates all control signals required to execute the instruction set correctly, it consists micro-instructions.

Micro-instruction: Contains a sequencing word and a control word. The control word is all control information required for one clock cycle.

Micro-operations: Micro-operations are the atomic operations which executes a particular micro-instruction. Example of micro-operation during the fetch cycle:

t1: MAR ←(PC)
t2: MBR ←Memory
PC ←(PC) + I
t3: IR ←(MBR)

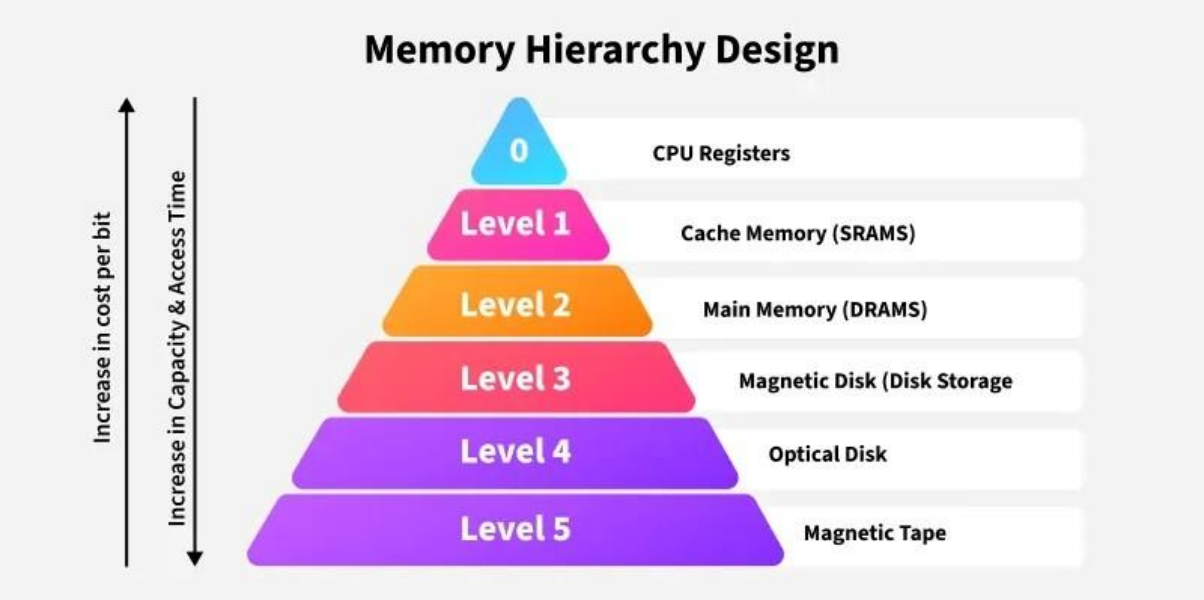
Memory Organization

- Memories are made up of registers.
- Each register in the memory is one storage location.
- The storage location is also called a memory location.
- Memory locations are identified using Address.
- The total number of bit a memory can store is its capacity.

Byte Addressable Memory	**Word Addressable Memory**
When the *data space in the cell = 8 bits* then the corresponding *address space* is called as Byte Address.	When the *data space in the cell = word length of CPU* then the corresponding *address space* is called as Word Address.

Byte Addressable Memory	Word Addressable Memory
Based on this data storage i.e. *Bytewise storage*, the memory chip configuration is named as **Byte Addressable Memory**.	Based on this data storage i.e. *Wordwise storage*, the memory chip configuration is named as **Word Addressable Memory**.

Memory Hierarchy



Simultaneous access memory organization: If H1 and H2 are the Hit Ratios and T1 and T2 are the access time of L1 and L2 memory levels respectively then the

Average Memory Access Time can be calculated as:

$T=(H1*T1)+((1-H1)*H2*T2T=(H1*T1)+((1-H1)*H2*T2$

Hierarchical Access Memory Organization: If H1 and H2 are the Hit Ratios and T1 and T2 are the access time of L1 and L2 memory levels respectively then

Average Memory Access Time can be calculated as:

$T=(H1*T1)+((1-H1)*H2*(T1+T2)T=(H1*T1)+((1-H1)*H2*(T1+T2)$

Read more about Simultaneous and Hierarchical Access Memory Organization, Here.

Cache Memory

Cache Memory is a special very high-speed memory. It is used to speed up and synchronizing with high-speed CPU. Levels of memory: Level 1 or Register, Level 2 or Cache memory, Level 3 or Main Memory, Level 4 or Secondary Memory.

Hit ratio = hit / (hit + miss) = no. of hits/total accesses

Locality of reference – Since size of cache memory is less as compared to main memory. So to check which part of main memory should be given priority and loaded in the cache is decided based on the locality of reference.

Types of Locality of reference

- **Spatial Locality of reference:** Spatial locality means instruction or data near to the current memory location that is being fetched, may be needed soon in the near future.
- **Temporal Locality of reference:** Temporal locality means current data or instruction that is being fetched may be needed soon. So we should store that data or instruction in the cache memory to avoid searching again in main memory for the same data.
- **Cache Mapping:** There are three different types of mapping used for the purpose of cache memory which is as follows: Direct mapping, Associative mapping and Set-Associative mapping.

Direct Mapping – Maps each block of main memory into only one possible cache line. If a line is previously taken up by a memory block and a new block needs to be loaded, the old block is trashed. An address space is split into two parts index field and a tag field. The cache is used to store the tag field whereas the rest is stored in the main memory.

Cache Line Number = Main Memory block Number % Number of Blocks in Cache

Associative Mapping – A block of main memory can map to any line of the cache that is freely available at that moment. The word offset bits are used to identify which word in the block is needed, all of the remaining bits become Tag.

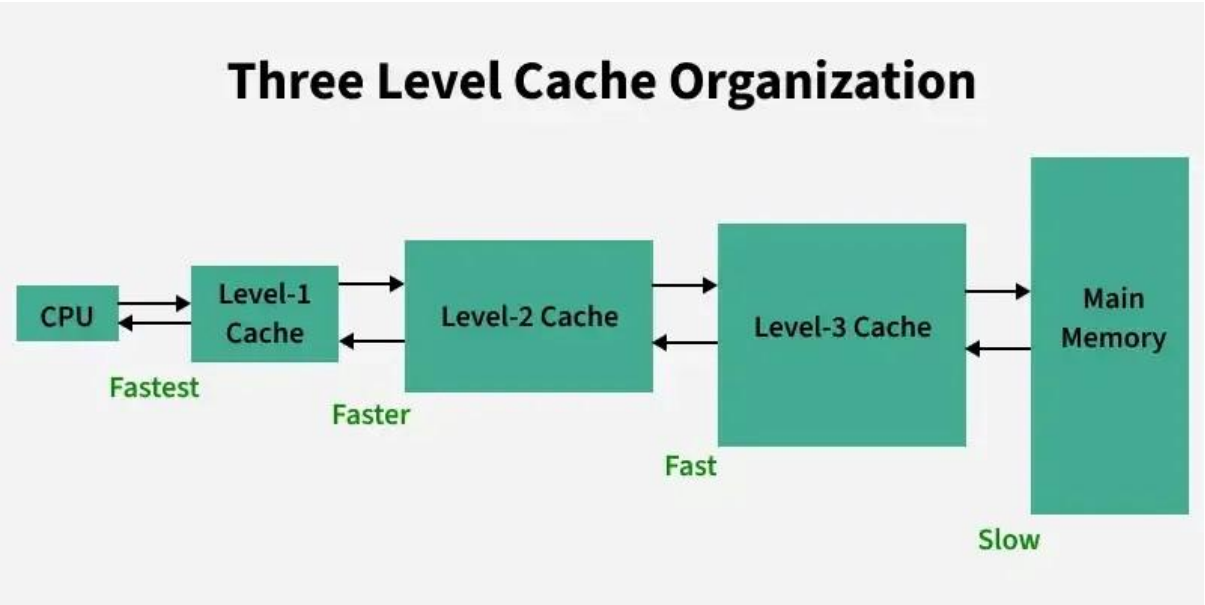
Set-Associative Mapping – Cache lines are grouped into sets where each set contains k number of lines and a particular block of main memory can map to only one particular set of the cache. However, within that set, the memory block can map to any freely available cache line.

Cache Set Number = Main Memory block number % Number of sets in cache

Note: Translation Lookaside Buffer (i.e. TLB) is required only if Virtual Memory is used by a processor. In short, TLB speeds up the translation of virtual address to a physical address by storing page-table in faster memory. In fact, TLB also sits between the CPU and Main memory.

Read more about Cache Mapping Techniques, Here.

Multilevel Cache



Multilevel Cache

Multilevel Caching is used in modern processors to improve memory access speed by introducing multiple levels of cache memory.

Types of Cache Levels

- **L1 Cache (Level 1)**:
    - Smallest, fastest, and closest to the CPU.
    - Usually divided into Instruction Cache and Data Cache.
- **L2 Cache (Level 2)**:
    - Larger and slower than L1 but still faster than main memory.
    - Acts as a bridge between L1 and L3/main memory.
- **L3 Cache (Level 3)**:
    - Shared across multiple cores.
    - Larger and slower than L2 but faster than main memory.

Performance Metrics

1. **Hit Ratio**:Percentage of memory accesses satisfied by the cache. Hit RatioHit Ratio = Cache HitsTotal AccessesTotal AccessesCache Hits
2. **Miss Ratio**: Percentage of memory accesses that result in a miss. Miss Ratio=1–Hit RatioMiss Ratio=1–Hit Ratio**Effective Memory Access Time (EMAT)**
   For **2-level cache**:

EMAT=H1×T1+(1–H1)×[H2×T2+(1–H2)×TM]EMAT=*H*1×*T*1+(1–*H*1)×[*H*2×*T*2+(1–*H*2)×*TM*]

- H1,H2H_1, H_2: Hit ratios for L1 and L2 caches.
- T1,T2T_1, T_2: Access times for L1 and L2 caches.
- TMT_M: Access time for main memory.

Cache Replacement Policies Table

Algorithm	Key Idea
LRU	Replace least recently used block
FIFO	Replace oldest block
Random	Replace random block
LFU	Replace least-used block
Optimal	Replace block not used longest

Cache Updation Policy

Write Through: In this technique, all write operations are made to main memory as well as to the cache, ensuring that main memory is always valid.

For hierarchical access:
Tread=H×Tcache+(1−H)×(Tcache+Tmemory_block)=Tcache+(1−H)×Tmemory_blockTread=H×Tcache+(1−H)×(Tcache+Tmemory_block)=Tcache+(1−H)×Tmemory_block

For simultaneous access : [Tex]T_{read} = H \times T_{cache} + (1-H) \times (T_{memory_block}) \newlineT_{write} = T_{memory_word}[/Tex]

Write Back: In write-back updates are made only in the cache. When an update occurs, a dirty bit, or use bit, associated with the line is set. Then, when a block is replaced, it is written back to main memory if and only if the dirty bit is set.

For hierarchical access:
Tread=Twrite=H×Tcache+(1−H)×(Tcache+Tmemory_block+Twrite_back)=Tcache+(1−H)×(Tmemory_block+Twrite_back), where Twrite_back=x×Tmemory_block, where x is the fraction of

dirty blocksTread=Twrite=H×Tcache+(1−H)×(Tcache+Tmemory_block+Twrite_back)=Tcache +(1−H)×(Tmemory_block+Twrite_back), where Twrite_back=x×Tmemory_block , where x is the fraction of dirty blocks

For simultaneous access : Tread=Twrite=H×Tcache+(1−H)×(Tmemory_block+Twrite_back), where Twrite_back=x×Tmemory_block, where x is the fraction of dirty blocksTread=Twrite=H×Tcache +(1−H)×(Tmemory_block+Twrite_back), where Twrite_back=x×Tmemory_block , where x is the fraction of dirty blocks

Read more about Cache Memory, Here**.**

Cache Miss

Type of Miss	**Reason**
Compulsory Miss	First-time access to data
Conflict Miss	Multiple blocks mapped to same cache line
Capacity Miss	Cache cannot hold all required data

Read about Types of Cache Miss, Here.

I/O Interface

- An I/O (Input/Output) Interface connects the CPU and memory with external devices like keyboards, monitors, printers, etc.
- It acts as a bridge between the CPU and I/O devices to ensure smooth data transfer.

Data transfer between the main memory and I/o device may be handled in a variety of modes like :

Programmed I/O: In Programmed I/O, the CPU controls data transfer between the I/O device and memory without allowing direct access for the device. The I/O device sends one byte at a time, placing the data on the I/O bus and enabling the data valid line. The interface stores the byte in its data register, activates the data accepted line, and sets a flag bit to notify the CPU. The I/O device waits for the data accepted line to reset before sending the next byte. This process is managed step-by-step by the CPU, making it slower but synchronized.

Interrupt driven I/O: In interrupt driven I/O, the processor issues an I/O command, continues to execute other instructions, and is interrupted by the I/O module when the I/O module completes its work.

Read more about Interrupt, Here.

Interrupt Handling Techniques

- **Daisy Chaining in Interrupts**

Daisy chaining is a method of handling multiple interrupts in a system by connecting the devices in a serial or chain-like manner. When an interrupt request is generated, the priority is determined by the position of the device in the chain. The device closer to the CPU has higher priority. The interrupt signal travels through the chain, and each device checks if it is the source of the interrupt. If not, it passes the signal to the next device in the chain. This approach is simple to implement but suffers from longer delays for devices farther down the chain and is unsuitable for systems requiring precise or equal priority handling.

- **Parallel Priority Interrupt**

Parallel priority interrupts use a priority encoder to handle multiple interrupt requests simultaneously. All devices send their interrupt requests in parallel to the encoder, which determines the highest-priority interrupt and sends it to the CPU. This method is faster and more efficient than daisy chaining because it does not rely on signal propagation through a chain. Each device is assigned a priority, and the encoder ensures that the device with the highest priority gets serviced first. Parallel priority interrupts are commonly used in systems where speed and fair priority handling are essential.

Direct Memory Access(DMA): In Direct Memory Access (DMA), the I/O module and main memory exchange data directly without processor involvement.

Modes of DMA Transfer

1. Burst Mode (Block Transfer Mode)

In burst mode, the DMA controller takes full control of the system bus and transfers an entire block of data in one go before releasing the bus back to the CPU. This method is fast but can cause the CPU to be idle during the transfer, as it doesn't get access to the bus until the transfer is complete.

2. Cycle Stealing Mode

In cycle stealing mode, the DMA controller takes control of the bus for one data transfer (one word or one byte) at a time and then releases it back to the CPU. This allows the CPU and DMA to share the bus alternately, improving overall system efficiency while slightly slowing the DMA transfer.

Read more about Modes of DMA Transfer, Here.

Mode	Key Feature	CPU Involvement	Use Case
Programmed I/O	CPU waits for device (polling)	High	Slow devices
Interrupt I/O	Device signals CPU via interrupt	Medium	Keyboards, printers
DMA	DMA controller handles transfer	Low (only initiation)	High-speed or bulk data devices

DMA Controller

The DMA (Direct Memory Access) Controller is a hardware component that manages data transfer between memory and I/O devices without constant CPU involvement. It communicates with the CPU, memory, and I/O devices through control and data lines.

The CPU interacts with the DMA controller by selecting its registers via the address bus while enabling the DS (Data Select) and RS (Register Select) inputs. When the CPU grants the bus to the DMA (indicated by BG = 1, Bus Grant), the DMA takes control of the buses. The DMA then directly communicates with memory by placing the memory address on the address bus and activating the RD (Read) or WR (Write) control signals to perform data transfer.

The DMA controller communicates with external I/O devices using request and acknowledge lines:

- The I/O device sends a request signal when it needs to transfer data.
- The DMA acknowledges this request, initiates the data transfer, and ensures synchronization.

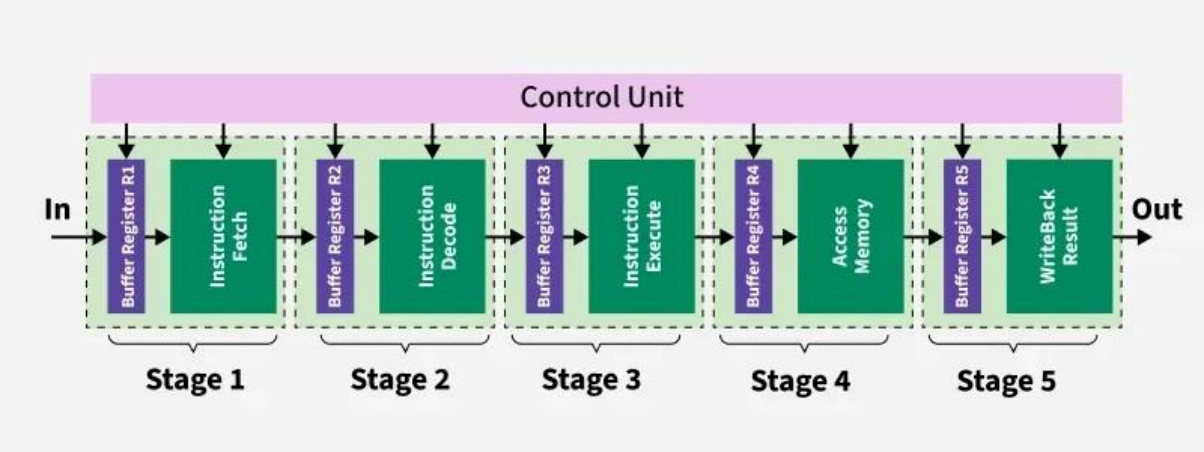
This process enables efficient and high-speed data transfer while freeing the CPU to perform other tasks.

Read more about I/O Interface, Here.

Pipelining

- Pipelining is a process of arrangement of hardware elements of the CPU such that its overall performance is increased.

- Simultaneous execution of more than one instruction takes place in a pipelined processor.
- RISC processor has 5 stage instruction pipeline to execute all the instructions in the RISC instruction set. Following are the 5 stages of RISC pipeline with their respective operations:
    - **Stage 1 (Instruction Fetch)** In this stage the CPU reads instructions from the address in the memory whose value is present in the program counter.
    - **Stage 2 (Instruction Decode)** In this stage, instruction is decoded and the register file is accessed to get the values from the registers used in the instruction.
    - **Stage 3 (Instruction Execute)** In this stage, ALU operations are performed.
    - **Stage 4 (Memory Access)** In this stage, memory operands are read and written from/to the memory that is present in the instruction.
    - **Stage 5 (Write Back)** In this stage, computed/fetched value is written back to the register present in the instructions.



5 stages of pipeline

Performance of a pipelined processor

Consider a 'k' segment/stages pipeline with clock cycle time as 'Tp'. Let there be 'n' tasks to be completed in the pipelined processor. So, time taken to execute 'n' instructions in a pipelined processor:

ETpipeline = k + n – 1 cycles
= (k + n – 1) Tp

In the same case, for a non-pipelined processor, execution time of 'n' instructions will be:

*ETnon-pipeline = n * k * Tp*

So, speedup (S) of the pipelined processor over non-pipelined processor, when 'n' tasks are executed on the same processor is:

S = Performance of pipelined processor / Performance of Non-pipelined processor

As the performance of a processor is inversely proportional to the execution time, we have:

S = ETnon-pipeline / ETpipeline
*=> S = [n * k * Tp] / [(k + n – 1) * Tp]*
*S = [n * k] / [k + n – 1]*

When the number of tasks 'n' are significantly larger than k, that is, n >> k

*S = n * k / n*
S = k

where 'k' are the number of stages in the pipeline. Also,

Efficiency = Given speed up / Max speed up = S / Smax

We know that, Smax = k So,

***Efficiency** = S / k*

Throughput = Number of instructions / Total time to complete the instructions So,

***Throughput** = n / (k + n – 1) * Tp*

Note: The cycles per instruction (CPI) value of an ideal pipelined processor is 1

Performance of pipeline with stalls

Speed Up (S) = CPInon-pipeline/ (1 + Number of stalls per instruction)

Read more about Pipelining, Here.

Dependencies and Data Hazard

There are mainly three types of dependencies possible in a pipelined processor. These are :

Structural dependency:

- This dependency arises due to the resource conflict in the pipeline. A resource conflict is a situation when more than one instruction tries to access the same resource in the same cycle. A resource can be a register, memory, or ALU.
- To minimize structural dependency stalls in the pipeline, we use a hardware mechanism called Renaming.

Control Dependency:

- This type of dependency occurs during the transfer of control instructions such as BRANCH, CALL, JMP, etc. On many instruction architectures, the processor will not know the target address of these instructions when it needs to insert the new instruction into the pipeline. Due to this, unwanted instructions are fed to the pipeline.
- Branch Prediction is the method through which stalls due to control dependency can be eliminated. In this at 1st stage prediction is done about which branch will be taken.

Data Dependency :

- Data dependency occurs when one instruction depends on the result of another instruction. It can cause data hazards in pipelined processors.

Types of Hazards in Pipelined Processors

Hazards are situations that cause the pipeline to stall or delay instruction execution. There are three main types of hazards:

1. **Structural Hazards**

- Occur when hardware resources are insufficient to handle the current instruction stream.
- Example: If only one memory unit exists, and both instruction fetch and data access need it simultaneously.

Solution:

- Add more resources (e.g., separate instruction and data memory – Harvard architecture).
- Use scheduling to avoid conflicts.

2. **Data Hazards**

- Arise when an instruction depends on data from a previous instruction that has not yet completed.

Types of Data Hazards:

- **RAW (Read After Write)**: True dependency.
- **WAR (Write After Read)**: Anti-dependency.
- **WAW (Write After Write)**: Output dependency.

Solution:

- Data forwarding/bypassing.

- Insert pipeline stalls (NOPs).
- Instruction scheduling.

3. **Control Hazards**

- Occur due to branch or jump instructions, where the next instruction to execute is uncertain until the branch is resolved.

Solution:

- Branch prediction techniques.
- Delayed branching (use NOPs).
- Dynamic scheduling.

Read more about Dependencies and Hazards, Here.

IEEE Standard 754 Floating Point Numbers

There are several ways to represent floating point number but IEEE 754 is the most efficient in most cases. IEEE 754 has 3 basic components:

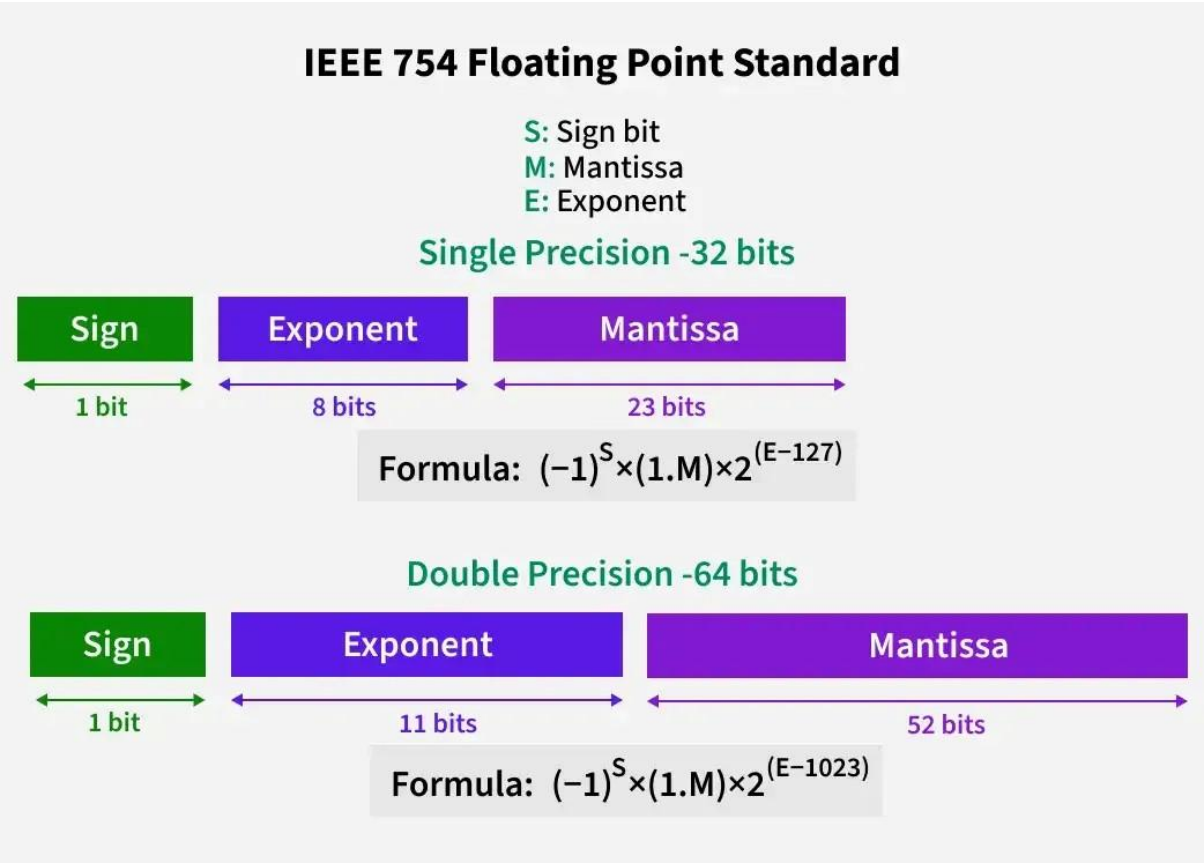
The Sign of Mantissa – This is as simple as the name. 0 represents a positive number while 1 represents a negative number.

The Biased exponent – The exponent field needs to represent both positive and negative exponents. A bias is added to the actual exponent in order to get the stored exponent.

The Normalised Mantisa – The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits. Here we have only 2 digits, i.e. O and 1. So a normalised mantissa is one with only one 1 to the left of the decimal.

The IEEE 754 Standard is used to represent floating-point numbers in binary. It has two formats:

1. **Single Precision (32-bit)**
2. **Double Precision (64-bit)**



IEEE 754 Floating Point Standard

E=0,M=0: Zero

Read more about IEEE Floating Point Notation, Here.

$ and & in computer architecture:

In computer architecture, both the dollar sign ($) and ampersand (&) have specific meanings, primarily within assembly language and programming. The dollar sign is often used to denote immediate values or constants, while the ampersand typically represents a bitwise AND operation or, in some cases, the memory address of a variable.

Dollar Sign ($):

- **Immediate values:**

In x86 assembly languages, the dollar sign ($) is used to indicate that the following number or expression represents an immediate value, meaning it's a constant that's directly included in the instruction.

- **AT&T syntax:**

In AT&T syntax, the dollar sign ($) is used to specify immediate values, while the percentage sign (%) is used to denote registers.

Ampersand (&):

- **Bitwise AND:**

The ampersand (&) is a common operator in many programming languages, including C, C++, and C#, used to perform a bitwise AND operation on two operands.

- **Address-of operator:**

In C and C++, the ampersand (&) can also be used as an address-of operator, which returns the memory address of a variable.

- **Logical AND:**

In some contexts, especially in C and C++, the double ampersand (&&) is used to perform a logical AND operation on two Boolean operands.

- **Other usages:**

The ampersand can also be used in other contexts, such as in string concatenation or as a special character in certain programming languages.

What is Register in Digital Electronics ?

A register is a small and temporary storage unit inside a computer's (CPU). It plays a vital role in holding the data required by the CPU for immediate processing and is made up of flip-flops. It usually holds a limited amount of data ranging from 8 to 64 bits, depending on the processor architecture.

Registers act as intermediate storage for data during arithmetic logic and other processing operations. In this article, we explore the definition, examples, applications, advantages, and disadvantages of registers in computer systems.

Table of Content

- What is Register?
- Memory Hierarchy
- Types of Registers
- Applications
- Advantages and Disadvantages

What is Register?

A register is a tiny, fast storage memory within the central processing unit (CPU) or the arithmetic logic unit (ALU) of a computer. Registers are utilized for a variety of functions in handling and controlling instructions and data and play an important role in the operation of a computer's CPU.

There are different types of registers listed below :

- Accumulator Register
- Program Counter (PC) Register
- General-Purpose Registers
- Instruction Register (IR)
- Memory Address Register (MAR)
- Memory Data Register (MDR)
- Stack Pointer (SP)
- Floating-Point Registers

Fetch

When we talk about "fetch" in computers, it is like a computer going on fetching next instruction which has to be followed. The first stage is where computer picks his next operation from its memory – this allows it to know what task has to come at the next moment and provides smooth execution.

Decode

When we talk about "decode" in computers, computer gets instruction out of memory and decodes it by figuring out what would be the meaning of the instruction that was fetched. After getting the instruction the computer has to decode it so as to know what will be its next course of action as per that instruction. This is an important step since it shows how best a computer can perform a certain task.

Execute

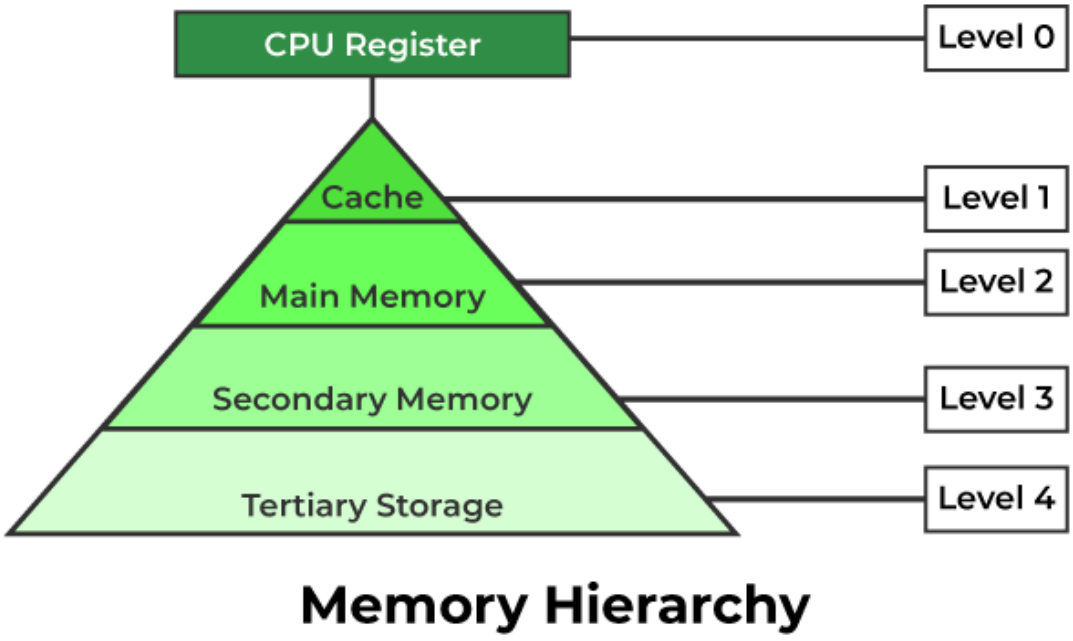
The term "execution" as used in computers refers to an action or task the computer actually undertakes as specified in the instruction that has been fetched and decoded. Once it has understood what is required of it, the computer carries out a certain operation. This could involve performing calculations, transferring data from one location to another, or accomplishing any other task defined by the instructions provided. This is a finishing process marking the end of the instruction cycle after which time it proceeds with the next command so as to maintain its rhythm.

Register Definition

In simple words, we can say that Register is very small, tiny storage unit which is temporary in nature but fast storage memory of a computer.

Memory Hierarchy and the Role of Registers

Computer systems have a memory hierarchy that includes multiple levels of memory with varying access speeds and capacities. At the top of this hierarchy are the CPU registers, which play a vital role in enhancing CPU performance. Registers are small**,** high-speed storage units located within the CPU itself, providing fast access to frequently used data.



Memory Hierarchy

The memory hierarchy typically includes the following levels:

1. **CPU Registers: The** temporary storage units within the CPU are directly accessible by the processor. They hold data required for immediate processing and act as intermediate storage during operations.
2. **Cache Memory: The** fast memory located between the CPU and main memory. Cache memory exploits the principle of temporal locality, caching frequently accessed data to reduce the need to access slower main memory frequently.
3. **logic, (RAM): S**lower than cache memory but larger in capacity the main memory holds data and instructions that are currently being used by the CPU.
4. **Secondary Storage**: The slowest but highest-capacity memory used for the long-term storage of data and programs.

Types of Registers

There are different types of registers listed below as per the different parameters :

- Accumulator Register
- Program Counter (PC) Register
- General-Purpose Registers
- Instruction Register (IR)
- Memory Address Register (MAR)
- Memory Data Register (MDR)
- Stack Pointer (SP)
- Floating-Point Registers
- Index Register (IR)
- Memory Buffer Register (MBR)
- Data Register (DR)

Accumulator Register

The accumulator acts as a central point for arithmetic and logical operations within the CPU. It fetches data from memory and stores intermediate results during calculations. Arithmetic operations such as addition, subtraction, multiplication, and division often take place in the accumulator. The final result may be stored in the accumulator or transferred to other registers or memory locations.

Program Counter (PC) Register

The program counter is a special register that keeps track of the memory address of the next instruction to be fetched and executed. As the CPU executes each instruction in sequence the program counter is updated to indicate the next instruction's address in memory. This process continues until the program's execution is complete.

General-Purpose Registers

General-purpose registers are versatile because they can hold data and memory addresses. They are used for various calculations and data manipulation tasks during program execution. General-purpose registers are essential for performing arithmetic and logical operations on data stored in the CPU.

Instruction Register (IR)

The instruction register holds the currently fetched instruction from memory. It allows the CPU to decode and execute the instruction based on its opcode and operands.

Memory Address Register (MAR)

The memory address register stores the memory address of data or instructions to be accessed or written in memory. It plays a crucial role in memory operations by indicating the location of the data or instruction the CPU needs to access.

Memory Data Register (MDR)

The Memory Data Register holds the actual data fetched from or written to memory. When the CPU retrieves data from memory, it is temporarily stored in the MDR before being processed further.

Stack Pointer (SP)

The stack pointer is used in stack-based memory operations. It keeps track of the top of the stack. which is a region of memory used for temporary storage of data and return addresses during function calls.

Floating-Point Registers

The Floating-point registers are specialized for handling floating-point numbers and performing floating-point arithmetic operations. These registers can store and manipulate floating-point numbers with higher precision.

Index Register (IR)

The Index Register is a kind of useful instrument in the computer that tracks specific locations of data. It is like a small helper that makes it easy for the computer to get to where memory is stored quickly and process it.

Memory Buffer Register (MBR)

The Memory Buffer Register can be viewed as a temporary store in the computer where data takes time off when it is being transferred from one place to another by the CPU. This register can be seen as some kind of traffic police man, ensuring smooth movements of information between CPU and memory without any bumps.

Data Register (DR)

The Data Register is an example of a fast storage place in the computer where information rests while all these processes take place on the same machine. It's a location where transient amounts of data are kept within the system for temporary usage like pit stops within then performing calculations or tasks or others.

Applications of Registers

- **Arithmetic and Logic Operations:** The Registers are extensively used during arithmetic and logic operations in the CPU. They temporarily store operands, intermediate results, and flags facilitating quick and efficient calculations.
- **Instruction Execution:** The Registers play a crucial role in the execution of machine instructions. They hold instructions and data needed for immediate processing ensuring smooth program execution.
- **CPU Pipelining:** The pipelining is employed to increase performance by overlapping the execution of multiple instructions. Registers help in holding intermediate results and data between different stages of the pipeline.
- **Context Switching:** the CPU switches between different processes or threads. it needs to save and restore the context of each process. Registers are used to store the CPU's state including the program counter stack pointer and general-purpose registers during context switching.
- **Memory Addressing:** The memory address register (MAR) and memory data register (MDR) are involved in memory operations. MAR holds the memory address being accessed. while MDR temporarily stores data fetched from or written to memory.
- **Input/Output Operations:** The Registers are used to buffer data during input/output operations. Input data is stored in input registers and output data is temporarily held in output registers before being processed further.
- **Floating-Point Operations:** To Specialized floating-point registers are used for handling floating-point arithmetic operations. These registers can store and manipulate floating-point numbers with higher precision.
- **Control Unit Operations: The** Registers play a critical role in the control unit of the CPU. They hold control signals and flags that determine the control flow and sequencing of the instructions during program execution.

Advantages and Disadvantages of Registers

Advantages and Disadvantages of Registers are given below :

Advantages of Registers

- **Speed:** The Registers offer fast access times due to their proximity to the CPU, enhancing overall system performance.
- **Data Processing Efficiency:** They enable quick data manipulation, reducing the need to access slower main memory frequently.

Disadvantages of Registers

- **Limited Capacity:** The Registers have a small size, restricting the amount of data they can hold at a time.
- **Cost:** The Registers are made from flip-flops and require more hardware, contributing to the overall cost of the processor.

Conclusion

In conclusion, a register is a small and high-speed storage unit within a computer's central processing unit (CPU). It serves as temporary storage for the data that the CPU requires for**,** immediate processing during arithmetic, logic, and other operations. The Registers play a critical role in enhancing CPU performance by providing fast access to frequently used data and facilitating efficient data manipulation.

Instruction Codes and Operands in Computer Architecture

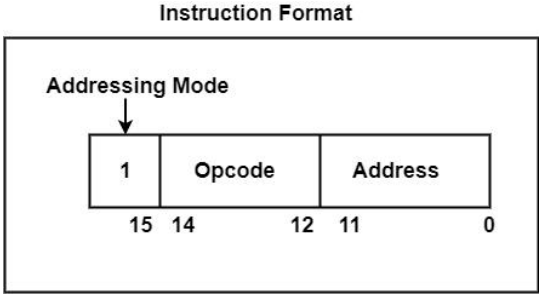
Computer ArchitectureComputer ScienceNetwork

A **computer instruction** is a binary code that determines the micro-operations in a sequence for a computer. They are saved in the memory along with the information. Each computer has its specific group of instructions.

They can be categorized into two elements as Operation codes (Opcodes) and Address. Opcodes specify the operation for specific instructions. An address determines the registers or the areas that can be used for that operation. Operands are definite elements of computer instruction that show what information is to be operated on.

It consists of 12 bits of memory that are required to define the address as the memory includes 4096 words. The 15th bit of the instruction determines the addressing mode (where direct addressing corresponds to 0, indirect addressing corresponds to 1). Therefore, the instruction format includes 12 bits of address and 1 bit for the addressing mode, 3 bits are left for Opcodes.

The following block diagram shows the instruction format for a basic computer.



There are three parts of the Instruction Format which are as follows −

Addressing Modes

Instructions that define the address of a definite memory location are known as memory reference instructions. The method in which a target address or effective address is recognized within the instruction is known as addressing mode.

The address field for instruction can be represented in two different ways are as follows −

- **Direct Addressing** − It uses the address of the operand.
- **Indirect Addressing** − It facilitates the address as a pointer to the operand.

The address of the operand or the target address is called the effective address.

Effective Address (EA) − It defines the address that can be executed as a target address for a branch type instruction or the address that can be used directly to create an operand for a computation type instruction, without creating any changes.

Opcodes

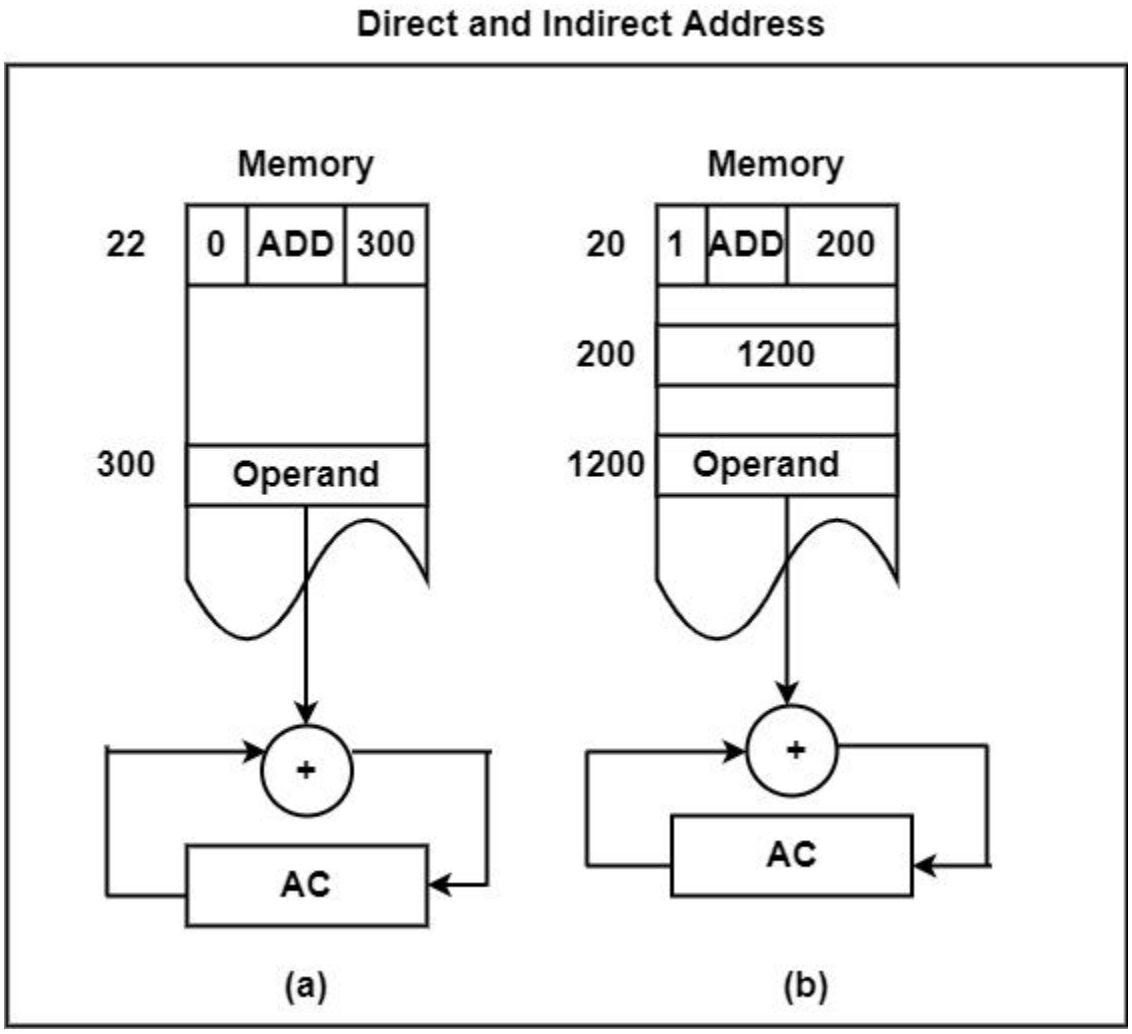
An opcode is a collection of bits that represents the basic operations including add, subtract, multiply, complement, and shift. The total number of operations provided through the computer determines the number of bits needed for the opcode. The minimum bits accessible to the opcode should be n for 2n operations. These operations are implemented on information that is saved in processor registers or memory.

Address

The address is represented as the location where a specific instruction is constructed in the memory. The address bits of an instruction code is used as an operand and not as an address. In such methods, the instruction has an immediate operand. If the second part has an address, the instruction is referred to have a direct address.

There is another possibility in the second part including the address of the operand. This is referred to as an indirect address. In the instruction code, one bit can signify if the direct or indirect address is executed.

The figure shows a diagram showing direct and indirect addresses.



Computer Organization – Basic Computer Instructions

Computer organization refers to the way in which the components of a computer system are organized and interconnected to perform specific tasks. One of the most fundamental aspects of computer organization is the set of basic computer instructions that the system can execute.

Basic Computer Instructions

Basic computer instructions are commands given to a computer to perform specific tasks. These instructions are typically divided into three categories:

1. **Data Transfer Instructions**: Move data between memory and registers (e.g., Load, Store).

1. **Arithmetic and Logic Instructions**: Perform math or logic operations (e.g., Add, Subtract, AND, OR).

1. **Control Instructions**: Guide the flow of the program (e.g., Jump, Branch, Call).

Data Transfer Instructions

These instructions deal with moving data from one location to another within the computer. Think of it as copying or transferring information between various components like memory, registers, or storage.

Common Data Transfer Instructions:

- **Load**: Copies data from memory to a register (temporary storage inside the CPU).
- **Store**: Transfers data from a register to memory.
- **Move**: Transfers data from one register to another.

Arithmetic and Logic Instructions

These instructions are used to perform mathematical and logical operations. They enable computers to handle calculations and make decisions based on certain conditions.

Arithmetic Instructions:

- **Add**: Adds two numbers.
- **Subtract**: Subtracts one number from another.
- **Multiply**: Multiplies two numbers.
- **Divide**: Divides one number by another.

Logic Instructions:

- **AND**: Compares two bits and returns 1 if both are 1; otherwise, returns 0.
- **OR**: Compares two bits and returns 1 if at least one is 1.
- **NOT**: Inverts a bit (1 becomes 0, and 0 becomes 1).
- **XOR (Exclusive OR)**: Returns 1 if the bits are different, 0 if they are the same.

Control Instructions

Control instructions determine the flow of execution in a program. They guide the computer on which instruction to execute next, allowing flexibility in decision-making and repeated actions.

Common Control Instructions:

- **Jump (JMP)**: Directs the program to execute a specific instruction elsewhere in the code.
- **Conditional Branch**: Jumps to another instruction only if a specific condition is true. **Example:** Branch if zero (BZ) or branch if not zero (BNZ).

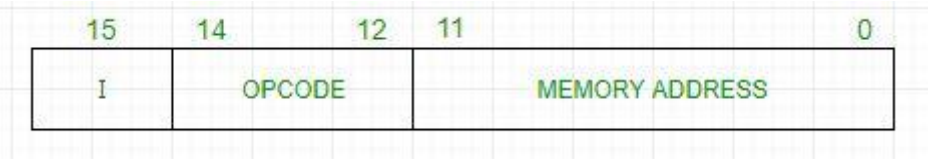
- **Call**: Transfers control to a subroutine (a set of instructions performing a specific task) and returns afterward.
- **Return**: Brings back control to the main program after a subroutine is executed.

Instruction Set of a Basic Computer

The basic computer has 16-bit instruction register (IR) which can denote either memory reference or register reference or input-output instruction.

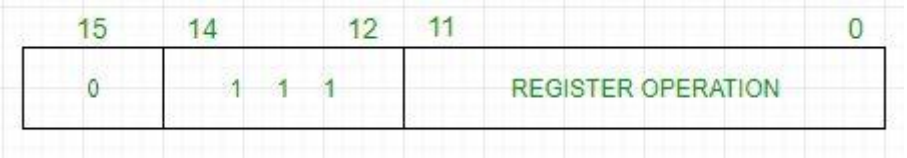
Memory Reference Instructions

These instructions refer to memory address as an operand. The other operand is always accumulator. Specifies 12-bit address, 3-bit opcode (other than 111) and 1-bit addressing mode for direct and indirect addressing.



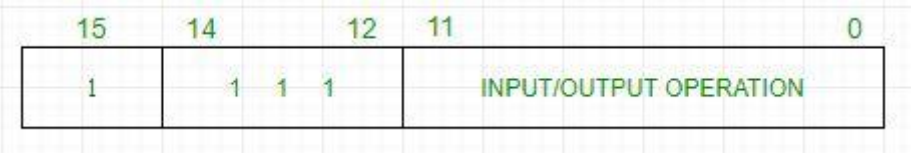
Register Reference Instructions

These instructions perform operations on registers rather than memory addresses. The IR(14 – 12) is 111 (differentiates it from memory reference) and IR(15) is 0 (differentiates it from input/output instructions). The rest 12 bits specify register operation.



Input/Output Instructions

These instructions are for communication between computer and outside environment. The IR(14 – 12) is 111 (differentiates it from memory reference) and IR(15) is 1 (differentiates it from register reference instructions). The rest 12 bits specify I/O operation.



Essential Instructions in a Basic Computer

Program Counter (PC) is a key part of a computer, and its instructions are the basic tasks a computer performs. These instructions are handled by the computer's **CPU (Central Processing Unit)** and form the foundation for more complex operations. Some examples of basic PC instructions include:

Symbol	Hexadecimal Code	Description
AND	0xxx, 8xxx	AND memory word to AC
ADD	1xxx, 9xxx	Add memory word to AC
LDA	2xxx, Axxx	Load memory word to AC
STA	3xxx, Bxxx	Store AC content in memory
BUN	4xxx, Cxxx	Branch Unconditionally
BSA	5xxx, Dxxx	Branch and Save Return Address
ISZ	6xxx, Exxx	Increment and skip if 0
CLA	7800	Clear AC
CLE	7400	Clear E(overflow bit)
CMA	7200	Complement AC
CME	7100	Complement E
CIR	7080	Circulate right AC and E
CIL	7040	Circulate left AC and E
INC	7020	Increment AC

Symbol	Hexadecimal Code	Description
SPA	7010	Skip next instruction if AC > 0
SNA	7008	Skip next instruction if AC < 0
SZA	7004	Skip next instruction if AC = 0
SZE	7002	Skip next instruction if E = 0
HLT	7001	Halt computer
INP	F800	Input character to AC
OUT	F400	Output character from AC
SKI	F200	Skip on input flag
SKO	F100	Skip on output flag
ION	F080	Interrupt On
IOF	F040	Interrupt Off

Note: In the given table,

1. **AC (Accumulator)**: A register that temporarily stores data during arithmetic or logic operations.
2. **E (Carry/Overflow Bit)**: A single-bit register used for carry operations or overflow detection in arithmetic calculations.

Example of Instruction Execution

- **ADD Instruction**:
  Suppose the memory address 0001 contains the value 5, and the AC currently holds 10. The ADD instruction fetches the value from memory, adds it to the AC, and stores the result in the AC.

AC ← AC + M[0001]

Result: AC = 10 + 5 = 15

Uses of Basic Computer Instructions

- **Data manipulation**: Basic computer instructions are used to handle data in the computer system, including moving data between memory and the CPU.
- **Control flow**: Basic instructions guide the flow of a program by branching to different parts based on conditions.
- **Input/output operations**: Instructions are used to transfer data between the computer and external devices like keyboards, printers, or monitors.
- **Program execution**: They load programs into memory and control their execution by moving data in and out of the program.
- **System maintenance**: These instructions handle tasks like memory allocation, interrupt management, error detection and correction.

Issues of Basic Computer Instructions

- **Complexity**: Instructions can be hard to understand, especially for beginners, making programming challenging.
- **Limited functionality**: Basic instructions can handle only simple tasks. Programmers require to write extra code for complex operations.
- **Compatibility**: Instructions may vary across different computer systems This can require programmers to write separate code for each system, which can be time-consuming and inefficient.
- **Security**: Basic computer instructions can be vulnerable to security threats, such as buffer overflows and code injection attacks.
- **Maintenance**: Basic computer instructions can be difficult to maintain, particularly as systems become more complex and code becomes more extensive. This can require significant resources, time and effort.

The x86-64 architecture, a 64-bit extension of the x86 architecture, has approximately 981 unique instruction mnemonics and a total of 3,684 instruction variants. This includes a wide range of instructions, from simple arithmetic operations to complex SIMD (Single Instruction, Multiple Data) instructions.

Elaboration:

- **Instruction Mnemonics:**

These are the symbolic names used in assembly language to represent specific instructions (e.g., ADD, MOV, JMP).

- **Instruction Variants:**

Each instruction mnemonic can have different forms or variants depending on the operands (the data the instruction operates on) and addressing modes.

- **x86-64:**

This is the 64-bit version of the x86 architecture, which includes extensions to the older 32-bit x86 instructions.

- **Instruction Set:**

The x86 instruction set is a comprehensive collection of instructions that allow a processor to perform a wide variety of tasks.

- **Complexity:**

The x86 architecture is known for its complex instruction set and addressing modes, which can make programming in assembly language challenging but also offer significant performance benefits.

The RISC-V base user-level 32-bit integer instruction set, known as RV32I, includes 47 instructions. These instructions can be grouped into six types: R-type (register-register), I-type (short immediates and loads), S-type (stores), B-type (branches), J-type (jumps), and L-type (loads).

Here's a more detailed breakdown:

- **RV32I:** The base instruction set with 47 core instructions.
- **RV64I:** The 64-bit version also has 47 instructions.
- **RV32E:** A reduced 32-bit version with 16 registers, including instructions like add, sub, and, or, xor, and slt.

Dynamic Scheduling (Tomasolu and Scoreboard technique)?

In Tomasulo's algorithm, the control logic is distributed among the reservation stations, whereas in scoreboarding, the scoreboard keeps track of everything.

Dynamic Scheduling with Scoreboards

Objectives & Prior Learning

- Explore advantages of dynamic scheduling
- Evaluate the use of scoreboards for dynamic scheduling
- Analyze scoreboard architecture and stages of control

Patterson, Chapter 3 and Appendix C

- Modified MIPS pipeline
- Four branch hazards alternatives
- Stall until direction is known
- Static prediction
- Delayed branch
- Dynamic branch prediction

Dynamic Scheduling

Advantages

- Handles cases when dependencies unknown at compile time
- Simplifies the compiler
- Allows code that compiled for one pipeline to run efficiently on a different pipeline
- Hardware speculation, a technique with significant performance advantages, builds on dynamic scheduling
- Key idea: Allow instructions behind stall to proceed
- Enables out-of-order execution & out-of-order completion

Examples

Allow Instructions Behind Stall to Proceed

fdiv.d f0,f2,f4

fadd.d f10,f0,f8

fsub.d f12,f8,f14

The fadd.d needs to wait for the slow fdiv.d to produce f0 but the fsub.d can proceed.

HW Speculation

ADD R1,R2,R3

BEQZ R5 Next

...

Next: SUB R7,R6,R4

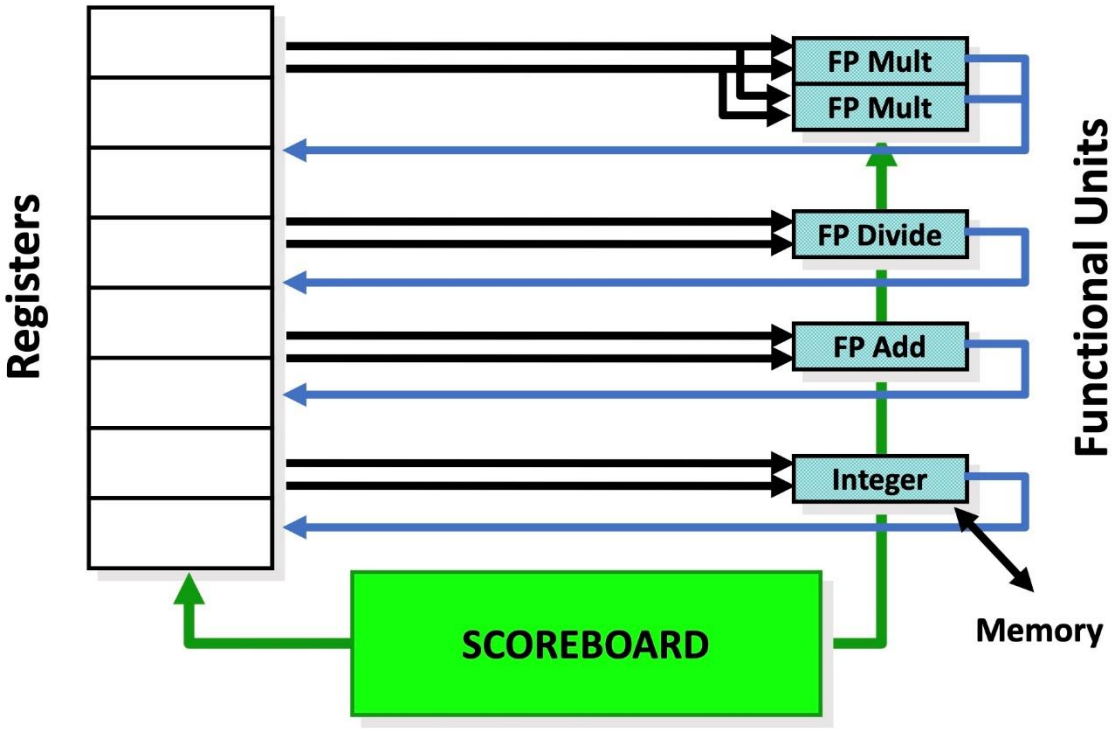
Predict the outcome of a branch and start executing instructions accordingly.

Scoreboard: a Bookkeeping Technique

- Scoreboards date to CDC 6600 in 1963
- Instructions execute whenever not dependent on previous instructions and no structural hazards
- CDC 6600: In order issue, out-of-order execution, out-of-order commit (or completion)
    - No forwarding (in original implementation)
    - But can be added
- Out-of-order execution divides ID stage:

1. Issue — decode instructions, check for structural hazards
2. Read operands — wait until no data hazards, then read operands

Scoreboard Architecture (CDC 6600)



Scoreboard Implications

- Out-of-order completion => WAR, WAW hazards
- Solution for WAR:
    - Stall writeback until registers have been read
- Solution for WAW:
    - Detect hazard and stall issue of new instruction until other instruction completes
- Need to have multiple instructions in execution phase => multiple execution units or pipelined execution units
- **Scoreboard** keeps track of dependencies between instructions that have already issued
- Check slide 11 for an example

Four Steps of Scoreboard Control

- Scoreboard replaces ID, EX, WB with 4 steps
- **Issue** — decode instructions & check for structural hazards (ID1)
    - Instructions issued in program order (for hazard checking)
    - Don’t issue if structural hazard

    - Don’t issue if instruction is output dependent on any previously issued but uncompleted instruction (no WAW hazards)
- **Read operands** — wait until no data hazards, then read operands (ID2)
    - All real dependencies (RAW hazards) resolved in this step, since we wait for instructions to write back data.
    - **No forwarding of data in this design**
- **Execution** — operate on operands (EX)
    - The functional unit begins execution upon receiving operands. When the result is ready, it **notifies the scoreboard** that it has completed execution.
- **Write result** — finish execution (WB)
    - Stall until no WAR hazards with previous instructions:
    - DIVD F0,F2,F4
    - ADDD F10,F0,F8
    - SUBD F8,F6,F14
    - CDC 6600 scoreboard would stall SUBD until ADDD reads operands.

Three Parts of the Scoreboard

- **Instruction status**: Which of 4 steps the instruction is in
- **Functional unit status**: Indicates the state of the functional unit (FU). 9 fields for each functional unit
    - Busy: Indicates whether the unit is busy or not
    - Op: Operation to perform in the unit (e.g., ADD or SUB)
    - Fi: Destination register
    - Fj,Fk: Source-register numbers
    - Rj,Rk: Flags indicating when Fj, Fk are ready
    - Qj,Qk: Functional units producing source registers Fj, Fk
- **Register result status**: Indicates which functional unit will write each register, if such FU exists. Blank when no pending instructions will write that register

How is it done?

Central control keeps track of

1. Current step of each instruction (out of 4 steps: Issue, Get operands, Execute, Write);
2. Status of functional unit: to keep track of RAW & structural hazards (if busy):
    - Rj, Rk - status of source registers (Yes means ready); a No in Rj means must wait for a FU to produce a new value for the source operand
    - Qj will include the ID of the FU that will provide the operand
3. Status of each register Rn: if equal zero, Rn has a "stable" value that is not about to change, otherwise status of Rn includes the ID of the FU that will write into Rn. The register result status allows checking for WAW & RAW. Note that an FU should not have an ID=0.

Scoreboard Example

Op dest,S1,S2 = Op Fi,Fj,Fk

Instruction status	Wait until	Bookkeeping
Issue	Not busy (FU) and not Rg.Result(dest)	Busy(FU)← yes; Op(FU)← op; Fi(FU)← `Dest'; Fj(FU)← `S1'; Fk(FU)←`S2'; Qj← Rg.Result('S1'); Qk←Rg.Result(`S2'); Rj← not Qj; Rk←not Qk; Rg.Result('dest')← FU;
Read operands	Rj and Rk	Rj← No; Rk← No
Execution complete	Functional unit done	
Write result	∀f((Fj(f)≠Fi(FU) or Rj(f)=No) & (Fk(f)≠Fi(FU) or Rk(f)=No))	∀f(if Qj(f)=FU then Rj(f)← Yes); ∀f(if Qk(f)=FU then Rk(f)← Yes); Rg.Result(Fi(FU))←0; Busy(FU)←No

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2				
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Op Dest,S1,S2 = Op Fi,Fj,Fk

Functional unit status:

Time	Name	Busy	Op	Dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
FU									

Scoreboard Example: Cycle 1

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1			
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

FU latencies:

Add: 2 cycles

Mult: 10

Div: 40

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
1									
FU				Integer					

Scoreboard Example: Cycle 2

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2		
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
2	FU	Integer								

Issue 2nd LD?

❓ Issue 2nd LD?

👷 No! Since load needs Integer FU, but it is being used by previous load.

Scoreboard Example: Cycle 3

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F6		R2				No
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU	Integer								

❓ Can we instead issue the MULTD? Why or why not?

👷 No! But why?

🧐 Since MULTD needs data from previous load F2. Data hazard.

Scoreboard Example: Cycle 4

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU									

Scoreboard Example: Cycle 5

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5			
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F2		R3				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
5	FU		Integer							

Scoreboard Example: Cycle 6

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6		
MULTD	F0	F2	F4	6			
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F2		R3				Yes
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
6	FU	Mult1	Integer							

Scoreboard Example: Cycle 7

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	
MULTD	F0	F2	F4	6			
SUBD	F8	F6	F2	7			
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F2		R3				No
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	Mult1	Integer			Add				

Two instructions in Issue; No forwarding

Scoreboard Example: Cycle 8a (part 1)

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	
MULTD	F0	F2	F4	6			
SUBD	F8	F6	F2	7			
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F2		R3				No
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
8	FU	Mult1	Integer			Add	Divide			

Scoreboard Example: Cycle 8b (part 2)

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6			
SUBD	F8	F6	F2	7			
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
8	FU	Mult1				Add	Divide			

Scoreboard Example: Cycle 9

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9		
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status:

Note remaining cycles →

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
10	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
2	Add	Yes	Sub	F8	F6	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
9	FU	Mult1				Add	Divide			

❓ Can we instead read operands for MULT & SUB?

🧐 A confusing question since in cycle 9, we are reading operands for MULT & SUB.

🤦 Seems like professor is also confused about it.

❓ Issue ADDD?

🧐 No, since the Add FU is being occupied.

🤦 Right.

∘ ∘ ∘

Scoreboard Example: Cycle 12

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
7	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	No								
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
12	FU	Mult1					Divide			

❓ Where is cycle 10? Why don't execute MULTD? And why its Rj and Rk become No?

🤔 I forget one important thing, some operations have high FU latencies. Load just needs 1 cycle, but Add needs 2 cycles. 10 cycles for Mult and 40 cycles for Div.

❓ Can we instead read operands for DIVD?

🤔 No. It needs F0 which is the result of MULTD.

❓ Issue ADDD?

🤔 Yes. Since SUBD has completed.

。 。 。

Scoreboard Example: Cycle 17

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

WAR Hazard!

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
2	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
17	FU	Mult1			Add		Divide			

❓ Why not write the result of ADDD?

🧐 There is a WAR Hazard.

Scoreboard Example: Cycle 21

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21		
ADDD	F6	F8	F2	13	14	16	

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6			Yes	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
21	FU				Add		Divide			

WAR Hazard is now gone...

Scoreboard Example: Cycle 22

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21		
ADDD	F6	F8	F2	13	14	16	22

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
39	Divide	Yes	Div	F10	F0	F6			No	No

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
22	FU						Divide			

Review: Scoreboard Example: Cycle 62

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	62
ADDD	F6	F8	F2	13	14	16	22

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
62	FU									

In-order issue; out-of-order execute & commit

CDC 6600 Scoreboard - Summary

Key idea of Scoreboard: Allow instructions behind stall to proceed (Decode ⇒ Issue instruction & read operands)

- Enables out-of-order execution ⇒ out-of-order completion
- ID stage checked both for structural & data dependencies

- Original version didn't handle forwarding
- Limited to instructions in basic block (small window)
- Prevent WAW hazards, but wait for WAR hazards

Dynamic Scheduling – Tomasulo's Algorithm

[TOC]

Objectives & Prior Learning

- Analyze the use of dynamic scheduling
- Demonstrate understanding of Tomasulo's Algorithm
- Analyze advantages and drawbacks related to Tomasulo's Algorithm

Patterson, Chapter 3 and Appendix C

- Advantages of dynamic scheduling
- Scoreboard example

Dynamic Scheduling: Tomasulo's Algorithm

Background

- For IBM 360/91
- Goal: High Performance without special compilers
- Small number of floating point registers (4 in 360) prevented interesting compiler scheduling of operations
    - This led Tomasulo to try to figure out how to get more effective registers — **renaming in hardware!**
- Why Study 1966 Computer?
- The descendants of this have flourished! Alpha 21264, HP 8000, MIPS 10000, Pentium, PowerPC, ...

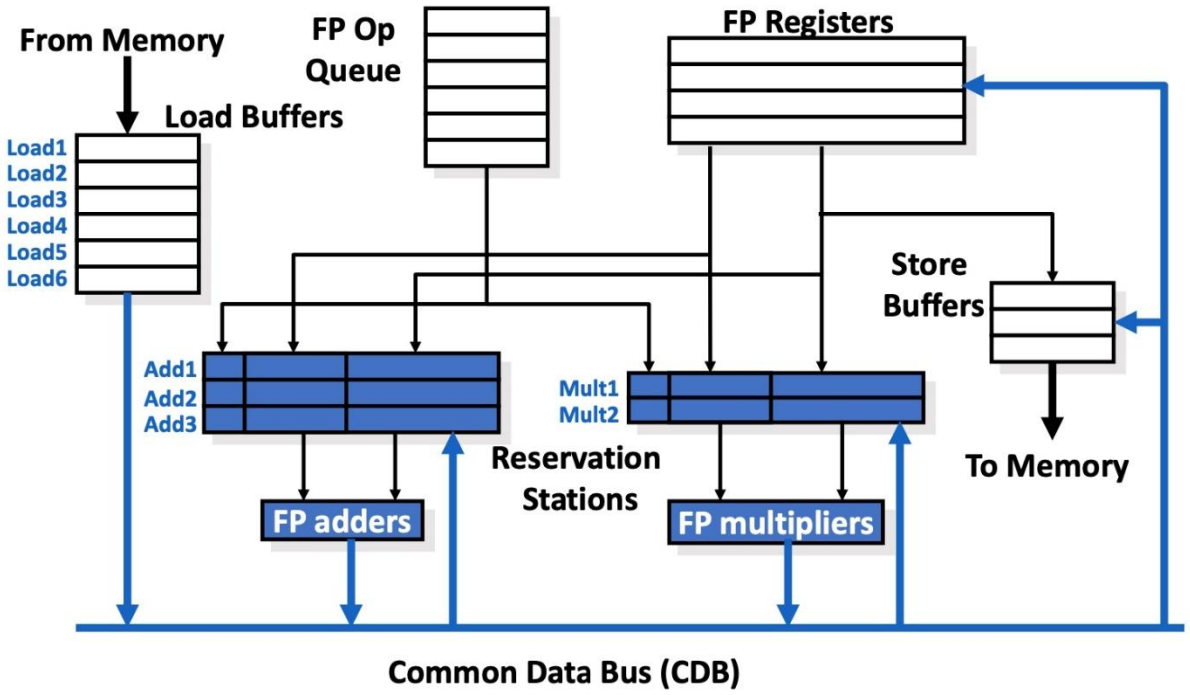
Tomasulo Algorithm

- Control & buffers **distributed** with Functional Units (FU)
- FU buffers called "**reservation stations**"; have pending operands
- Registers in instructions replaced by values or pointers to reservation stations (RS);
    - Form of **register renaming**;

    - Avoids WAR, WAW hazards
    - More reservation stations than registers, so can do optimizations compilers can't
- Results to FU from RS, **not through registers**, over **Common Data Bus** that broadcasts results to all FUs
- Load and Stores treated as FUs with RSs as well

- The main difference between Tomasulo's algorithm and scoreboarding is that the control is distributed vs. the centralized scoreboard control.
- Each FU has its own buffers (for holding a queue of instructions waiting to be executed and their operands) called "reservation stations (RS)," and has its own local control so the overall control is distributed.
- The values of source operands for instructions are stored in the RS rather than kept in the RF and if operand is not yet available, a pointer to the RS that will produce it, is generated.
- A form of "register renaming" as RS buffers act as RF registers.
- It avoids WAW & WAR as they are due to reuse of registers.
- More RS than registers in the RF – can do optimizations that compilers can not as they can only access RF registers.
- Operands from one instruction to another not through RF registers but through a Common Data Bus (CDB).
- The result of an FU is broadcasted over the CDB and all RS monitor the CDB and grab the operand if they need it.
- CDB can be viewed as an extended version of forwarding: in forwarding the source and destination are known, CDB – bill board – destinations are not known to the sender.
- Load and Store behave similarly with load buffers and store buffers acting like RS.

Tomasulo's Organization



❓ Does the CDB transmit only data?

a. Yes b. No

🧐 b. No. Some operation results in an address.

Reservation Station Components

- Op: Operation to perform in the unit (e.g., ADD)
- Vj, Vk: Value of Source operands
    - Store buffers has Vj field, result to be stored
- Qj, Qk: Reservation stations producing source registers (value to be written)
    - Note: Qj,Qk=0 => ready
- Busy: Indicates reservation station or FU is busy
- Register result status (Rg.Result) — Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

❓ What is the benefit of this?

🧐 benefit of what?

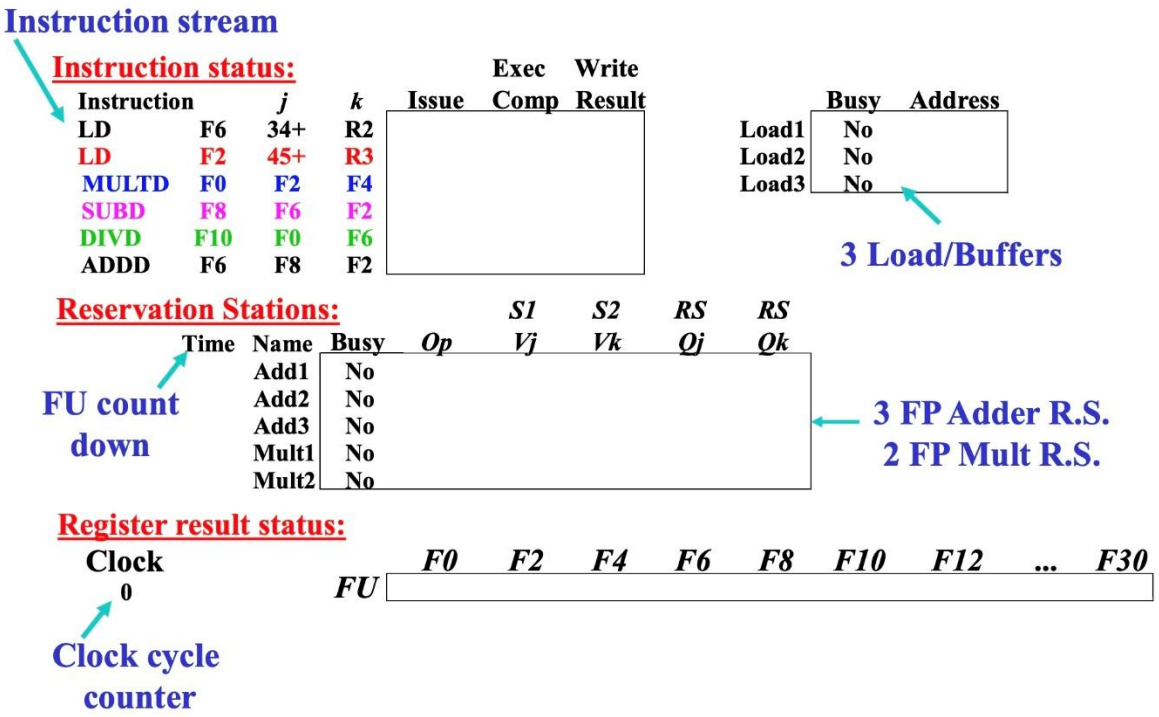
Three Phases of Tomasulo's Algorithm

1. Issue — get instruction from FP Op Queue

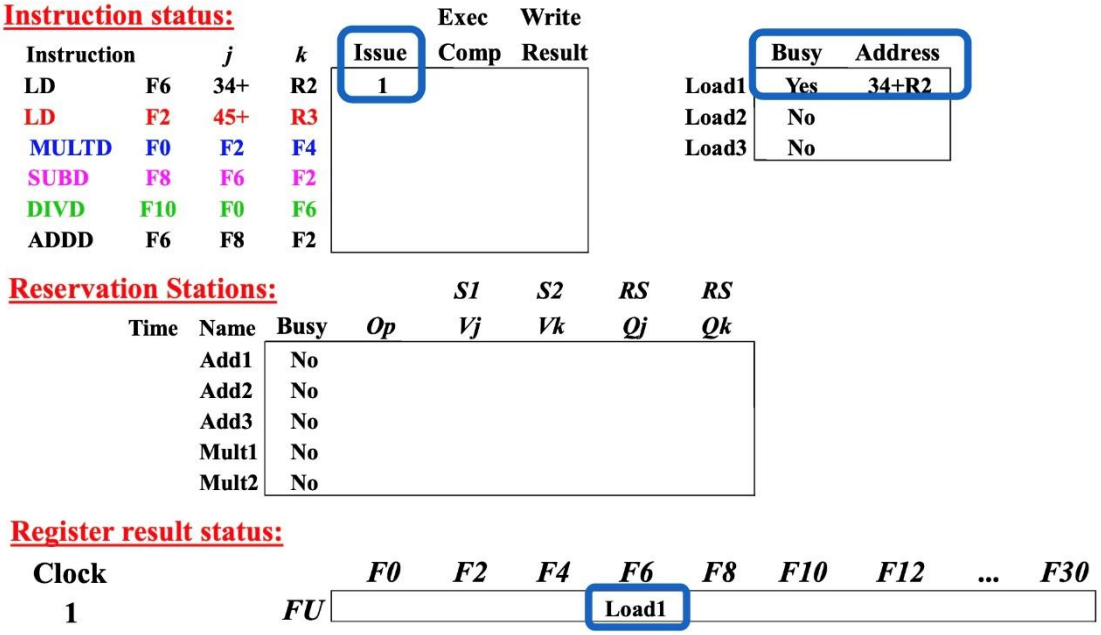
    - If reservation station free (no structural hazard), control issues instruction & gets operands (renames registers)
2. Execute—operate on operands(EX)
    - When both operands ready then execute; if not ready, watch Common Data Bus for result
3. Write result—finish execution (WB)
    - Write on Common Data Bus to all awaiting units; mark reservation station available
    - Normal data bus: data + destination ("go to" bus)
    - Common data bus: data + source ("come from" bus)
        - 64 bits of data + 4 bits of Functional Unit (RS) source address
        - Does the broadcast
        - Other RS get data if "source" matches the ID of RS that is expected to produce the required operand
    - Example - speed of operations: 2 clocks for Fl.pt.ADD,SUB and LOAD; 10 for MULT; 40 clocks for DIV

Tomasulo Example

- Same program that we used before
- Register Result Status – identical to scoreboard
- RS status – similar to FU status in scoreboard but main difference Vj and Vk each have 64 bits
- Instruction status table is not implemented in HW and appears here only for purpose of us understanding what is happening in each cycle.
- Approximate number of bits in an RS: 1+3+64+64+4+4 = 140 bits!
- Assuming here 3 RS in front of FP adder, 2 in front of FP multiplier and 3 load buffers (no Store instructions in our program)



Tomasulo Example Cycle 1



Tomasulo Example Cycle 2

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3	2		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
2	FU	Load2		Load1					

Note: Can have multiple loads outstanding

Tomasulo Example Cycle 3

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU	Mult1	Load2		Load1				

Note: MULT issued; registers names are removed ("renamed") in Reservation Stations

Tomasulo Example Cycle 4

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	Yes	SUBD	M(LD.1)			Load2
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	Mult1	Load2		0000	Add1				

Tomasulo Example Cycle 5

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
2	Add1	Yes	SUBD	M(LD.1)	M(LD.2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(LD.2)	R(F4)		
	Mult2	Yes	DIVD		R(F6)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
5	FU	Mult1	0000			Add1	Mult2			

Timer starts down for Add1, Mult1

Tomasulo Example Cycle 6

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
1	Add1	Yes	SUBD	M(LD.1)	M(LD.2)		
	Add2	Yes	ADDD		R(F2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(LD.2)	R(F4)		
	Mult2	Yes	DIVD		R(F6)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
6	FU	Mult1			Add2	Add1	Mult2			

❓ Issue ADDD here despite name dependency on F6?

🧐 I think it's feasible since the register content has already been fetched.

Tomasulo Example Cycle 7

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
0	Add1	Yes	SUBD	M(LD.1)	M(LD.2)		
	Add2	Yes	ADDD		M(LD.2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(LD.2)	R(F4)		
	Mult2	Yes	DIVD		R(F6)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	Mult1			Add2	Add1	Mult2			

Tomasulo Example Cycle 8

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
2	Add2	Yes	ADDD	(M1-M2	M(LD.2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(LD.2)	R(F4)		
	Mult2	Yes	DIVD		R(F6)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
8	FU	Mult1			Add2	0000	Mult2			

Tomasulo Example Cycle 11

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	MULTD	M(LD.2)	R(F4)		
	Mult2	Yes	DIVD		R(F6)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
11	FU	Mult1			0000		Mult2			

Tomasulo Example Cycle 15

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(LD.2)	R(F4)		
	Mult2	Yes	DIVD		R(F6)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
15	FU	Mult1					Mult2			

Tomasulo Example Cycle 16

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

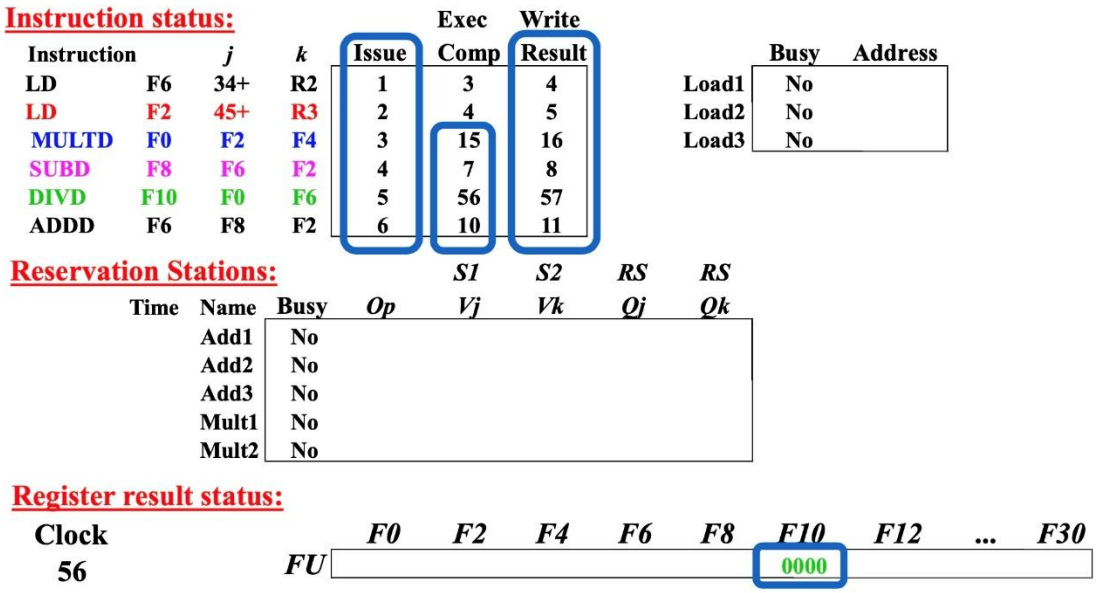
Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	M2*F4	R(F6)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
16	FU	0000					Mult2			

Just waiting for Mult2 (DIVD) to complete

Tomasulo Example Cycle 57



Once again: In-order issue, out-of-order execution and out-of-order completion.

Drawbacks

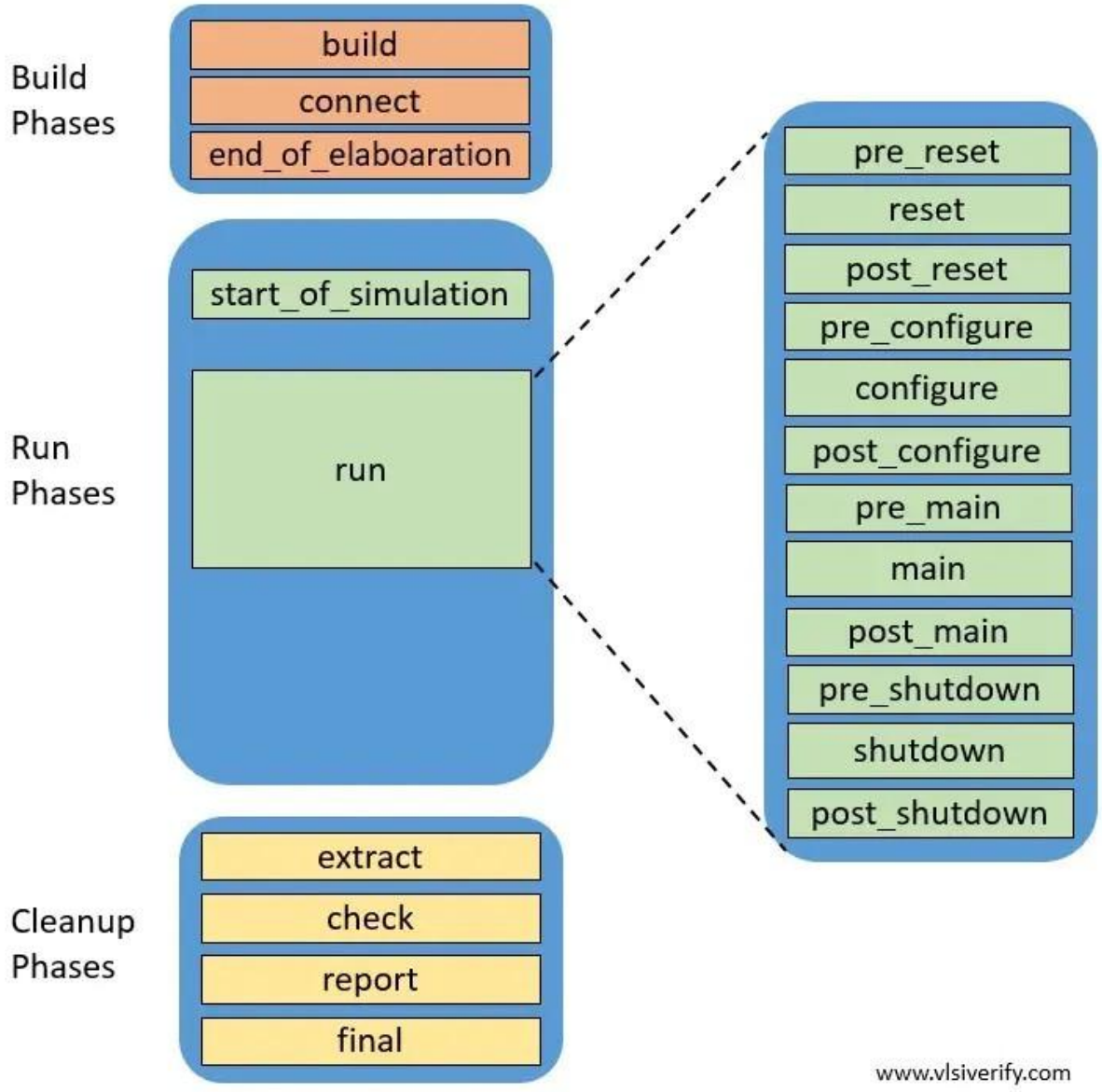
- High Complexity (large circuit, high power consumption)
- Many stores (CDB) at high speed resulting limit on performance by Common Data Bus
    - Each CDB must go to multiple functional units =》 high wiring density, high capacitance
    - Number of functional units that can complete per cycle limited to one!
        - Multiple CDBs =》 more FU logic for parallel stores
        - CDB is the only centralized resource – may become a bottleneck;
        - CDB must be connected to all FUs – large number of wires with high capacitance – slowing down the operation
- Non-precise interrupts!
    - We will address this later

❓ Explain the fundamentals of Tomasulo's Algorithm, including:

- Purpose and goal of dynamic scheduling
- Benefits
- Drawbacks

- Difference between Tomasulo's Algorithm and scoreboarding
- Relationship to registers (effects, etc.)
- Phases

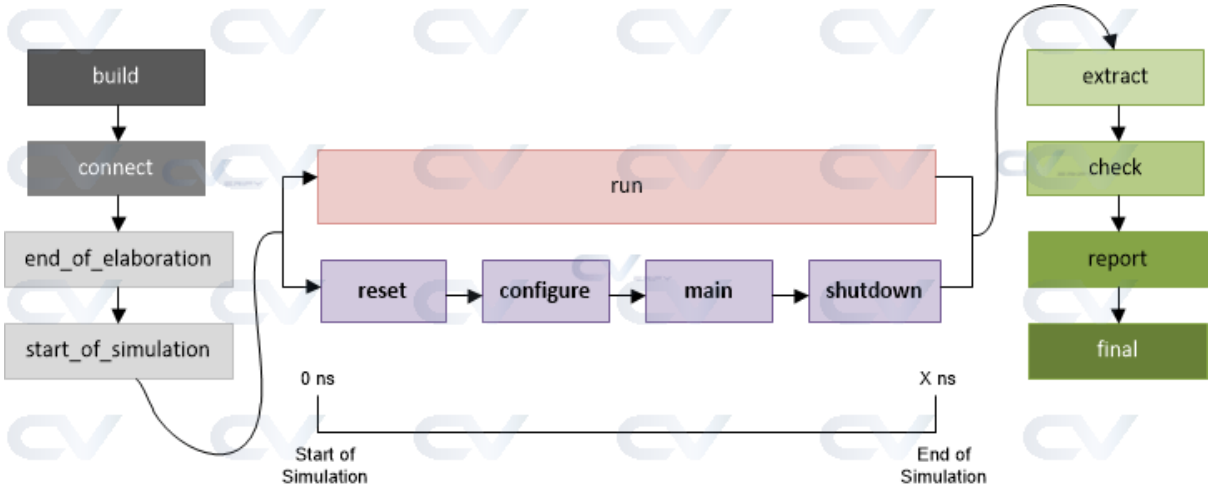
UVM Phases:



All testbench components are derived from uvm_component and are aware of the *phase* concept. Each component goes through a pre-defined set of phases, and it cannot proceed to the next phase until all components finish their execution in the current phase. So UVM phases act as a **synchronizing mechanism** in the life cycle of a simulation.

Because phases are defined as callbacks, classes derived from uvm_component can perform useful work in the callback phase method. Methods that do not consume simulation time are functions and methods that consume simulation time are tasks. All phases can be grouped into three categories:

1. Build time phases
2. Run time phases
3. Clean-Up phases



Note that run_phase is launched in parallel with other run-time phases.

Main UVM Phases

Phase Category	UVM Phase Name	Method Type	Description
Build	build_phase	function	Used to build testbench components and create their instances
	connect_phase	function	Used to connect between different testbench components via TLM ports
	end_of_elaboration_phase	function	Used to display UVM topology and other functions required to be done after connection
	start_of_simulation_phase	function	Used to set initial run-time configuration or display topology
Run	run_phase	task	Actual simulation that consumes time happens in this UVM phase and runs parallel to other UVM run-time phases.

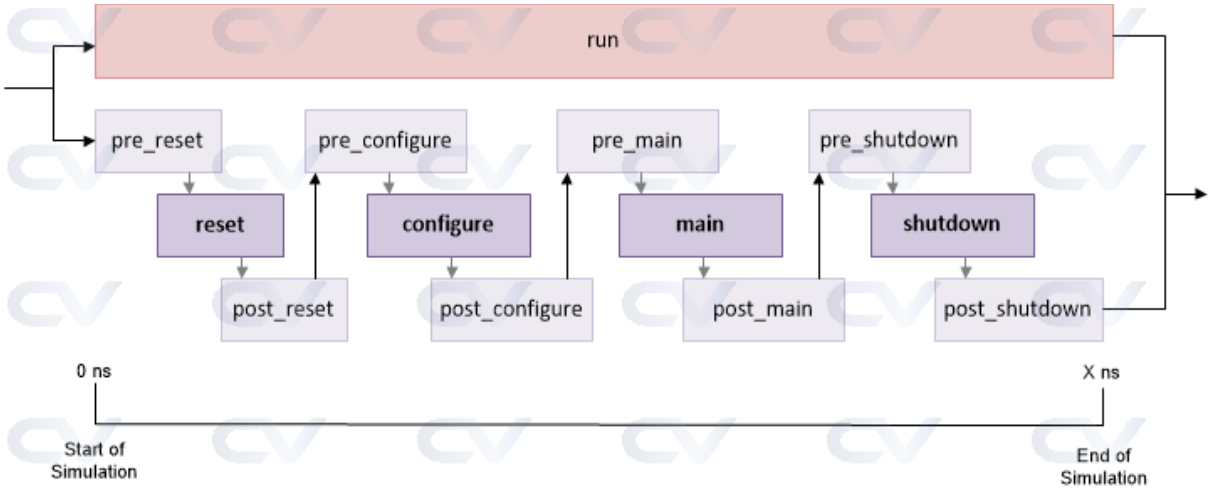
Clean	extract_phase	function	Used to extract and compute expected data from scoreboard, an
	check_phase	function	Used to perform scoreboard tasks that check for errors between expected and actual values from design
	report_phase	function	Used to display result from checkers, or summary of other test objectives
	final_phase	function	Typically used to do last minute operations before exiting the simulation

Logically, the first thing to be done is to create testbench component objects so that they can be connected together. This is the reason for the build_phase. It is better to not start connecting them while other testbench components are still building their sub-components. So we have connect_phase which will connect all the components that were built in the previous phase. Although the next two phases are rarely used or are typically used to display UVM hierhachy information. Test stimulus is driven to the design during the run_phase which is launched in parallel with other run-time phases that are described shown below.

Why is build done top to bottom ?

Verification environment usually looks like a nested structure as shown in the image below. In order to build the agent, UVM environment is required to be defined and similarly in order to build the environment, the test class has to be defined. Hence, we can only go from top to bottom.

To add more flexibility, there are pre and post callbacks to all the runtime phases.



Why doesn't Verilog testbench need phases ?

A Verilog testbench would have all its components made of static containers or *modules*. So each module will have a set of ports or signals that it can utilize to communicate with other testbench components. Since a module is static, all modules will be created at beginning of the simulation and we won't have to worry about any component getting called without it being created or initialized. It would be useful to have some *run-time* phases for resetting DUT, driving the actual stimulus and maybe another phase to enable and run checkers.

```
module driver (...);
        task reset_phase (...);
        task run_phase (...);
        task check_phase (...);

        initial begin
                reset_phase ();
                run_phase ();
                check_phase ();
        end
endmodule

module tb_top;
        dut      d0   (...);
        driver    drvr (...);
        checker1  chk0 (...);
        checker2  chk1 (...);
endmodule
```

Verilog testbenches are limited in functionality and are not scalable.

Why SystemVerilog testbench require phases ?

SystemVerilog changes the whole game with the introduction of OOP (Object Oriented Programming) features. This enables the creation of well structured entities that can be reused and deployed when required. These entities are class objects, and it is possible to create a brand new object in the middle of the entire simulation, say at time Xns. What this means is that testbench components can be created at different times, and hence you could

end up calling a component while it hasn't been initialized yet leading to wrong testbench outputs. Also, different companies maintain their own hierarchy and arrangement of testbench environments and takes a lot of effort to plug in different verification IPs from other companies. So, *synchronization* between testbench components is a **key** requirement.

```
typedef class driver;

typedef class monitor;


class env;
        driver  d0;
        monitor mon0;


        function init_drvr ();
                d0 = new ();
                // initialize
        endfunction


        function init_mon ();
                mon0 = new ();
                // initialize
        endfunction
endclass
```

What should be done in each UVM Phase ?

The table below summarizes the intent, expectation and result of each phase as documented in the UVM reference manual.

Phase	Upon Entry	Typical Uses	Exit Criteria

build	Top-Level components are instantiated under uvm_root	• Instantiate sub-components • Instantiate register model • Get configuration values for components being built • Set configuration values for sub-components	• All uvm_components have been instantiated.
connect	All components have been instantiated	• Connect TLM ports and exports, initiator sockets and target sockets • Connect register model to adapter components • Setup explicit phase domains	• Connections between various components have been established • All independent phase domains are set
end_of_elaboration	Verification environment has been assembled	• Display environment topology • Open files • Define additional configuration settings	None
start_of_simulation	• External tools/simulation engines have started and synced • Verification environment	• Display environment topology • Set debugger breakpoint	None

	is configured and ready to be started	• Set initial run-time configuration values	
run	• Indicates that power has been applied • No active edges of clock should have happened before (x->1)	Components run over various run-time phases	• DUT no longer needs to be simulated • post_shutdown_phase is ready to end
extract	• DUT no longer needs to be simulated • Simulation time will no longer advance	• Extract data/final state information from scoreboard/testbench components • Probe the DUT (hierarchical reference/backdoor access) • Calculate statistics and summaries • Display final state information • Close files	All data has been collected and summarized
check	All data has been collected	Check that no unaccounted for data remain	Test is known to have passed or failed
report	Test is known to have passed or failed	• Report test results	End of test

		• Write results to file	
final	All test-related activity has completed	• Close files • Terminate co-simulation engines	Ready to exit simulator

ARM Interview Questions

Verilog code for a Dual Port RAM with Testbench

There are many types RAM. These differ in terms of the number of ports, synchronous or asynchronous modes of operation etc.

Here in this post, I have written the Verilog code for a simple Dual port RAM, with two ports 0 and 1. The writing is allowed to only one port, on the positive edge the clock. The reading is done from both the ports asynchronously, that means we don't have to wait for the clock signal to read from the memory.

I have fixed the RAM size as 16*8 bits, meaning 16 elements of 8 bits each. These specifications can be changed easily by altering few lines in the code.

Verilog code:

```
module dual_port_ram
  (  input clk, //clock
     input wr_en,    //write enable for port 0
     input [7:0] data_in,    //Input data to port 0.
     input [3:0] addr_in_0,  //address for port 0
     input [3:0] addr_in_1,  //address for port 1
     input port_en_0,    //enable port 0.
     input port_en_1,    //enable port 1.
     output [7:0] data_out_0,    //output data from port 0.
     output [7:0] data_out_1 //output data from port 1.
  );

//memory declaration.
reg [7:0] ram[0:15];

//writing to the RAM
always@(posedge clk)
begin
  if(port_en_0 == 1 && wr_en == 1)    //check enable signal and if write enable is ON
     ram[addr_in_0] <= data_in;
end
```

```
//always reading from the ram, irrespective of clock.
assign data_out_0 = port_en_0 ? ram[addr_in_0] : 'dZ;
assign data_out_1 = port_en_1 ? ram[addr_in_1] : 'dZ;

endmodule
```

Testbench code:

```
module tb;

   // Inputs
   reg clk;
   reg wr_en;
   reg [7:0] data_in;
   reg [3:0] addr_in_0;
   reg [3:0] addr_in_1;
   reg port_en_0;
   reg port_en_1;

   // Outputs
   wire [7:0] data_out_0;
   wire [7:0] data_out_1;

   integer i;

   // Instantiate the Unit Under Test (UUT)
   dual_port_ram uut (
      .clk(clk),
      .wr_en(wr_en),
      .data_in(data_in),
      .addr_in_0(addr_in_0),
      .addr_in_1(addr_in_1),
      .port_en_0(port_en_0),
      .port_en_1(port_en_1),
      .data_out_0(data_out_0),
      .data_out_1(data_out_1)
   );

   always
      #5 clk = ~clk;
```

```
  initial begin
    // Initialize Inputs
    clk = 1;
    addr_in_1 = 0;
    port_en_0 = 0;
    port_en_1 = 0;
    wr_en = 0;
    data_in = 0;
    addr_in_0 = 0;
    #20;
    //Write all the locations of RAM
    port_en_0 = 1;
    wr_en = 1;
   for(i=1; i <= 16; i = i + 1) begin
       data_in = i;
       addr_in_0 = i-1;
       #10;
    end
    wr_en = 0;
    port_en_0 = 0;
    //Read from port 1, all the locations of RAM.
    port_en_1 = 1;
    for(i=1; i <= 16; i = i + 1) begin
       addr_in_1 = i-1;
       #10;
    end
    port_en_1 = 0;
  end

endmodule
```

The code synthesised and simulated using Xilinx ISE 14.6 tool.

Simulation waveform:

Synthesis:

The synthesis was done for a Virtex 6 device. Note that, for this particular device the inbuilt BRAM's were not used to implement this RAM because of asynchronous read operations. Making it synchronous might use the available BRAM's.

WRITING SRAM VERILOG

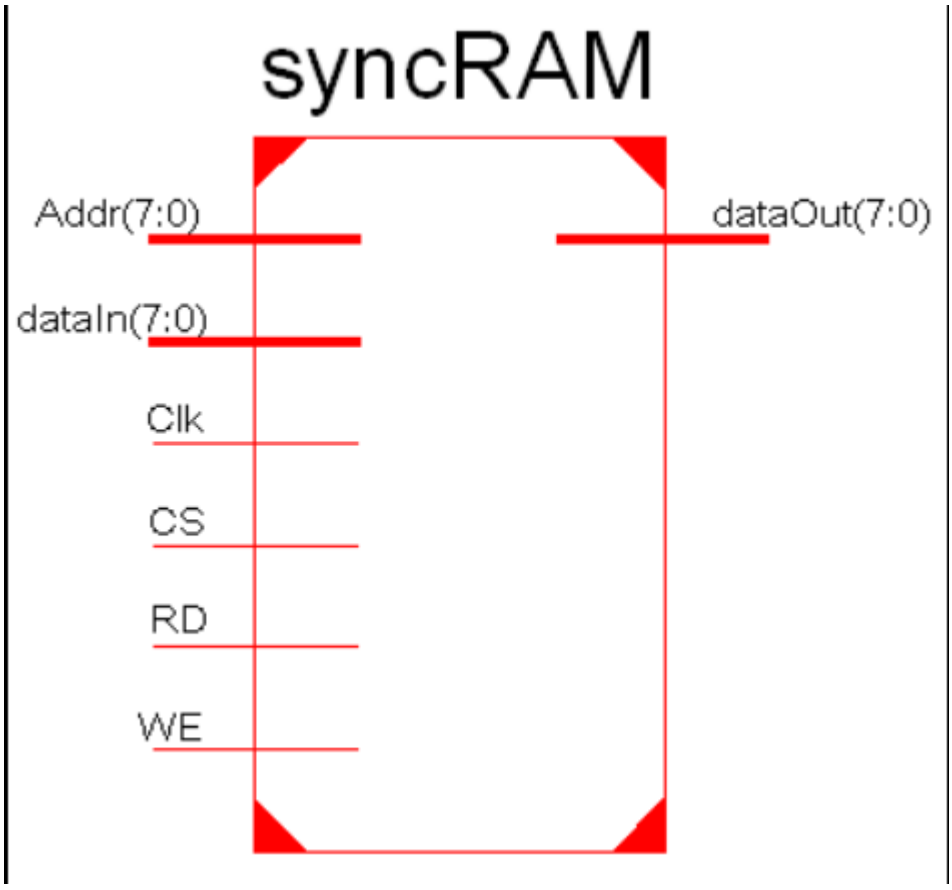
Studying About SRAM (Static Random Access Memory)

Definition:"**Memory** is a basic element in any system whether the memory is **volatile** or **non-volatile.**

In this example, a **volatile memory unit** is designed in the form of a **Synchronous Static RAM**.

Static Random-Access Memory (SRAM) is a type of **semiconductor memory** that uses **bi-stable latching circuitry** to store each bit.

The term **Static** differentiates it from **Dynamic RAM (DRAM)** which must be periodically refreshed. **SRAM** retains data, but it is still **volatile** as data is lost when the power to the memory unit is cut off."

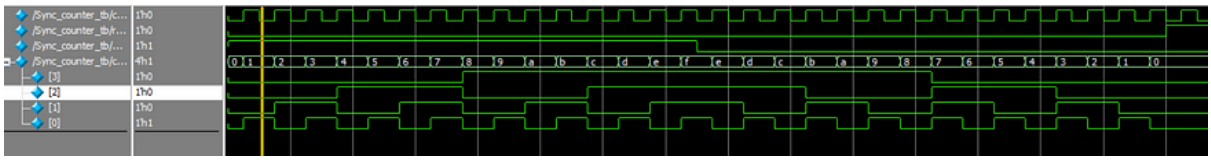


Simple 8-Bit SRAM

Verilog Module of SRAM we get from dataflow

Let’s Take Verilog

Verilog RTL-SRAM



result of EP waveform

let’s take shot on module of syncRAM

Here is the verilog code

```
module syncRAM( dataIn, dataOut, Addr, CS, WE, RD, Clk );



// parameters for the width

parameter ADR   = 8;

parameter DAT   = 8;

parameter DPTH  = 8;


//ports

input   [DAT-1:0]  dataIn;

output reg [DAT-1:0]  dataOut;

input   [ADR-1:0]  Addr;

input CS,

    WE,

    RD,

    Clk;
```

```
//internal variables

reg [DAT-1:0] SRAM [DPTH-1:0];

always @ (posedge Clk)
begin
 if (CS == 1'b1) begin
  if (WE == 1'b1 && RD == 1'b0) begin
   SRAM [Addr] = dataIn;
  end
  else if (RD == 1'b1 && WE == 1'b0) begin
   dataOut = SRAM [Addr];
  end
  else;
 end
 else;
end
endmodule
```

This is the module code of SRAM

```
`timescale 1ns / 1ps

module syncRAM_tb;
```

```
// Inputs
reg [7:0] dataIn;
reg [7:0] Addr;
reg CS;
reg WE;
reg RD;
reg Clk;

// Outputs
wire [7:0] dataOut;

// Instantiate the Unit Under Test (UUT)
syncRAM uut (
 .dataIn(dataIn),
 .dataOut(dataOut),
 .Addr(Addr),
 .CS(CS),
 .WE(WE),
 .RD(RD),
 .Clk(Clk)
 );
```

```
initial begin

 // Initialize Inputs

 dataIn  = 8'h0;

 Addr  = 8'h0;

 CS  = 1'b0;

 WE  = 1'b0;

 RD  = 1'b0;

 Clk  = 1'b0;


 // Wait 100 ns for global reset to finish

 #100;



 // Add stimulus here

 dataIn  = 8'h0;

 Addr  = 8'h0;

 CS  = 1'b1;

 WE  = 1'b1;

 RD  = 1'b0;

 #20;

 dataIn  = 8'h0;
```

```
Addr  = 8'h0;

#20;

dataIn  = 8'h1;

Addr  = 8'h1;

#20;

dataIn  = 8'h10;

Addr  = 8'h2;

#20;

dataIn  = 8'h6;

Addr  = 8'h3;

#20;

dataIn  = 8'h12;

Addr  = 8'h4;

#40;

Addr  = 8'h0;

WE  = 1'b0;

RD  = 1'b1;

#20;

Addr   = 8'h1;

#20;

Addr   = 8'h2;
```

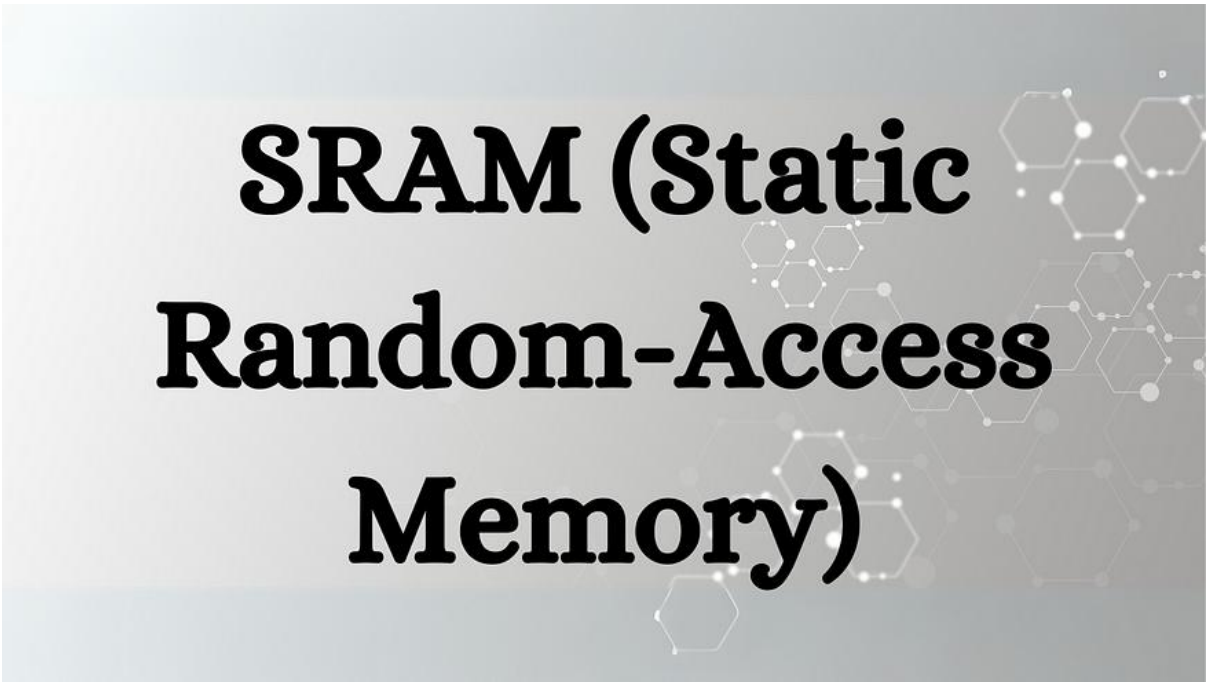
```
 #20;

 Addr   = 8'h3;

 #20;

 Addr   = 8'h4;

end



always #10 Clk = ~Clk;



endmodule
```

This is the testbench of SRAM and the values are written in Different Addr also the dataIn will be from 0..f and then dataOut is in also from 0..f

then we can see that there is another value which is represented as the DataOut and also we can see that last part is repeated as per usual.

SRAM Memory Design and Verification with SystemVerilog Testbench

This project involves verifying the functionality of an SRAM (Static Random-Access Memory) module using a SystemVerilog testbench. The testbench generates random transactions for reading and writing to the SRAM, and a comprehensive environment is set up to monitor and scoreboard the results. The project aims to ensure the correct operation of the SRAM memory in a digital design context.



SRAM (Static Random-Access Memory)

The following Verilog code defines a simple SRAM(Static Random-Access Memory) module that has a 256-word memory.

In SRAM, data is stored in a flip-flop-based configuration, allowing for faster and direct access to data without the need for refreshing cycles

It operates based on a clock signal and supports read and write operations, with a reset functionality to clear the memory.

SRAM DUT using Verilog Code:

```
module ram(

input clk, rst, wr,

input [7:0] din, addr,

output reg [7:0] dout

);

reg [7:0] mem [256];

integer i;

always @(posedge clk) begin
```

```
if(rst) begin

for(i = 0; i <256; i++) begin

mem[i] = 0;

end

end

else if(wr) begin

mem[addr] = din;

end


else begin

dout <= mem[addr];

end

end

endmodule
```

— —

SRAM testbench using System Verilog:

```
class transaction;
rand bit [7:0] din;
rand bit [7:0] addr;
bit wr;
bit [7:0] dout;
endclass

class generator;
mailbox mbx;
transaction t;
```

```
event done;
integer i;

function new(mailbox mbx);
this.mbx = mbx;
endfunction

task run();
t = new();
for(i = 0; i < 100; i++)begin
t.randomize();
mbx.put(t);
$display("[GEN] : Data send to driver");
@(done);
end
endtask
endclass


interface counter_intf();
logic clk,rst, wr;
logic [7:0] din, addr;
logic [7:0] dout;
endinterface

class driver;
mailbox mbx;
event done;
transaction t;
virtual counter_intf vif;

function new(mailbox mbx);
this.mbx = mbx;
endfunction

task run();
t = new();
forever begin
mbx.get(t);
vif.din = t.din;
vif.addr = t.addr;
```

```
$display("[DRV] : Interface Trigger");
->done;
@(posedge vif.clk);
end
endtask
endclass

class monitor;
mailbox mbx;
virtual counter_intf vif;
transaction t;

function new(mailbox mbx);
this.mbx = mbx;
endfunction

task run();
t = new();
forever begin
t.din = vif.din;
t.addr = vif.addr;
t.dout = vif.dout;
t.wr = vif.wr;
mbx.put(t);
$display("[MON] : data send to Scoreboard");
@(posedge vif.clk);
end
endtask
endclass

class scoreboard;
mailbox mbx;
transaction tarr[256];
transaction t;

function new(mailbox mbx);
this.mbx = mbx;
endfunction

task run();
forever begin
```

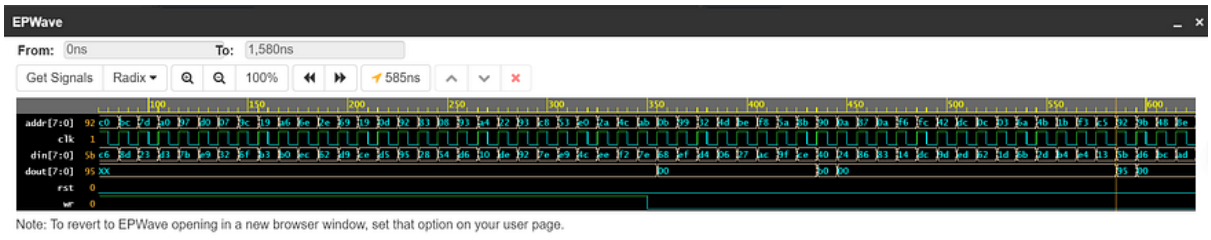
```
mbx.get(t);

if(t.wr == 1'b1) begin
if(tarr[t.addr] == null) begin
tarr[t.addr] = new();
tarr[t.addr] = t;
$display("[SCO] : Data stored");
end
end
else begin
if(tarr[t.addr] == null) begin
if(t.dout == 0)
$display("[SCO] : Data read Test Passed");
else
$display("[SCO] : Data read Test Failed");
end
else begin
if(t.dout == tarr[t.addr].din)
$display("[SCO] : Data read Test Passed");
else
$display("[SCO] : Data read Test Failed");
end
end
end
endtask
endclass

class environment;
generator gen;
driver drv;
monitor mon;
scoreboard sco;

virtual counter_intf vif;

mailbox gdmbx;
mailbox msmbx;

event gddone;

function new(mailbox gdmbx, mailbox msmbx);
```

```
this.gdmbx = gdmbx;
this.msmbx = msmbx;

gen = new(gdmbx);
drv = new(gdmbx);

mon = new(msmbx);
sco = new(msmbx);
endfunction

task run();
gen.done = gddone;
drv.done = gddone;

drv.vif = vif;
mon.vif = vif;

fork
gen.run();
drv.run();
mon.run();
sco.run();
join_any

endtask

endclass

module tb;

environment env;
counter_intf vif();
mailbox gdmbx, msmbx;

ram dut (vif.clk, vif.rst, vif.wr, vif.din, vif.addr, vif.dout);

always #5 vif.clk = ~vif.clk;


initial begin
vif.clk = 0;
```

```
vif.rst = 1;
vif.wr = 1;
#50;
vif.wr = 1;
vif.rst = 0;
#300;
vif.wr = 0;
#200
vif.rst = 0;
#50;

end

initial begin
gdmbx = new();
msmbx = new();

env = new(gdmbx,msmbx);
env.vif = vif;
env.run();
#600;
$finish;
end

initial begin
$dumpvars;
$dumpfile("dump.vcd");

end


endmodule
```

— —

Output:



is wrap same as incr in axi

In the **AXI protocol**, **WRAP** and **INCR** are *not* the same—they define different types of **burst transactions**, and they affect how addresses are generated during a burst.

Key Differences:

Feature	INCR (Incrementing burst)	WRAP (Wrapping burst)
Addressing	Address increases linearly with each beat	Address increases, but wraps around after reaching a boundary
Use case	For linear memory accesses	For cache line accesses that wrap
Example	Reading 4 words from addresses 0x00, 0x04, 0x08, 0x0C	Reading 4 words from 0x0C, 0x10, 0x00, 0x04 (wraps around a 16-byte boundary)
Burst Type	INCR (value = 0b001)	WRAP (value = 0b010)

Summary:

- Use **INCR** when accessing memory sequentially.
- Use **WRAP** when you need to wrap within a specific address boundary, often useful for cache line fills.

Would you like a visual example of how a wrapping burst behaves compared to an incrementing burst?

incr burst same as wrap burst in axi?

No, an **INCR burst** is **not the same** as a **WRAP burst** in the AXI protocol. Here's a clear breakdown:

🔁 INCR Burst (Incrementing Burst)

- **Definition:** Each beat accesses the next sequential address.
- **Addressing:** The address increases by the size of the data (e.g., 4 bytes per word).
- **Use case:** Normal memory reads/writes.
- **Example (4-beat burst, 4 bytes each, start at 0x00):**

Address sequence: 0x00 → 0x04 → 0x08 → 0x0C

🔁 **WRAP Burst (Wrapping Burst)**

- **Definition:** Like INCR, but wraps around after reaching the wrap boundary.
- **Addressing:** Increases like INCR, but when the upper boundary is reached, it wraps back to a lower boundary within a fixed address window.
- **Use case:** Common in cache line fills or circular buffers.
- **Example (4-beat burst, 4 bytes each, wrap boundary = 16 bytes, start at 0x0C):**

Address sequence: 0x0C → 0x10 → 0x00 → 0x04

🧠 **Key Technical Difference:**

- **INCR:** No wrap-around; just linear increment.
- **WRAP:** Forces addresses to wrap within a boundary that's typically burst_length × data_size.

WRAP Address Calculation

AHB AXI WRAP Burst

Table of Contents

- AHB AXI WRAP Burst
  - Equations for WRA Address Calculation
  - WRAP Address calculation examples
    - WRAP Example-1
    - WRAP Example-2
    - WRAP Example-3

A WRAP burst is similar to INCR burst. In WRAP the address will be incremented based the SiZE,
but on reaching the upper address limit address will wrap to lower address.

From the above statement, we could see that there are two considerations during WRAP address calculation,

1. Upper address limit to make WRAP
2. Lower address to WRAP to

There are restrictions on WRAP bursts,

- The start address must be aligned to the size of each transfer
- The length of the burst must be 2, 4, 8, or 16 transfers

Equations for WRA Address Calculation

Below equations are used for WRAP address calculation,

- Lower address to WRAP to

Wrap_Boundary =
(INT(Start_Address/(Number_Bytes×Burst_Length)))×(Number_Bytes×Burst_Length)

- Upper address limit to make WRAP

Address_N = Wrap_Boundary + (Number_Bytes × Burst_Length)

WRAP Address calculation examples

WRAP Example-1

AxADDR = 0x04, AxLEN = 3, AxSIZE = 2, AxBURST = 3

Start_Address = 0x04
Burst_Length = 4
Number_Bytes = 4

Wrap_Boundary =
(INT(Start_Address/(Number_Bytes×Burst_Length)))×(Number_Bytes×Burst_Length)
= (0x04/(4×4)) x (4×4)
= 0

Address_N = Wrap_Boundary + (Number_Bytes × Burst_Length)
= 0 + (4 x 4)
= 16

As Burst_Length is 4, Burst consists of 4 Address,
Address_0
Address_1
Address_2
Address_3

Condition, If the Address_n == 16, Address_n = wrap_boundary = 0

Address_0 = 4
Address_1 = Address_0 + Number_Bytes = 4 + 4 = 8
Address_2 = Address_1 + Number_Bytes = 8 + 4 = 12
Address_3 = Address_2 + Number_Bytes = 12 + 4 = 16

As per condition, Address_3 == 16, so Address_3 = 0

Therefore, ADDRESSES are **4, 8, 12, 0.**

WRAP Example-2

AxADDR = 0x38, AxLEN = 3, AxSIZE = 2, AxBURST = 3

Start_Address = 0x38
Burst_Length = 4
Number_Bytes = 4

Wrap_Boundary =
(INT(Start_Address/(Number_Bytes×Burst_Length)))×(Number_Bytes×Burst_Length)
= (0x38/(4×4)) x (4×4)
= 0x30

Address_N = Wrap_Boundary + (Number_Bytes × Burst_Length)
= 0x30 + (4 x 4)
= 0x40

As Burst_Length is 4, Burst consists of 4 Address,
Address_0
Address_1
Address_2
Address_3

Condition, If the Address_n == 0x34, Address_n = wrap_boundary = 0x30

Address_0 = 0x38
Address_1 = Address_0 + Number_Bytes = 0x38 + 0x4 = 0x3C
Address_2 = Address_1 + Number_Bytes = 0x3C + 0x4 = 0x40

As per condition, Address_2 == 0x40, so Address_2 = 0x30

Address_3 = Address_2 + Number_Bytes = 0x30 + 0x4 = 0x34

Therefore, ADDRESSES are **0x38, 0x3C, 0x30, 0x34.**

WRAP Example-3

Lets consider an example,
AxADDR = 0x34, AxLEN = 7, AxSIZE = 2, AxBURST = 3

Start_Address = 0x34
Burst_Length = 8
Number_Bytes = 4

Wrap_Boundary =
(INT(Start_Address/(Number_Bytes×Burst_Length)))×(Number_Bytes×Burst_Length)
= (0x34/(4×8)) x (4×8)
= 0x20

Address_N = Wrap_Boundary + (Number_Bytes × Burst_Length)
= 0x20 + (4 x 8)
= 0x40

As Burst_Length is 4, Burst consists of 4 Address,
Address_0
Address_1
Address_2
Address_3
…
Address_7

Condition, If the Address_n == 0x40, Address_n = wrap_boundary = 0x20

Address_0 = 0x34
Address_1 = Address_0 + Number_Bytes = 0x34 + 4 = 0x38
Address_2 = Address_1 + Number_Bytes = 0x38 + 4 = 0x3C
Address_3 = Address_2 + Number_Bytes = 0x3C + 4 = 0x40

As per condition, Address_3 == 0x40, so Address_3 = 0x20

Address_4 = Address_3 + Number_Bytes = 0x20 + 4 = 0x24
Address_5 = Address_4 + Number_Bytes = 0x24 + 4 = 0x28
Address_6 = Address_5 + Number_Bytes = 0x28 + 4 = 0x2C
Address_7 = Address_6 + Number_Bytes = 0x2C + 4 = 0x30

Therefore, ADDRESSES are **0x34, 0x38, 0x3C, 0x20, 0x24, 0x28, 0x2C, 0x30**

Constraint for AXI 4kb Boundary

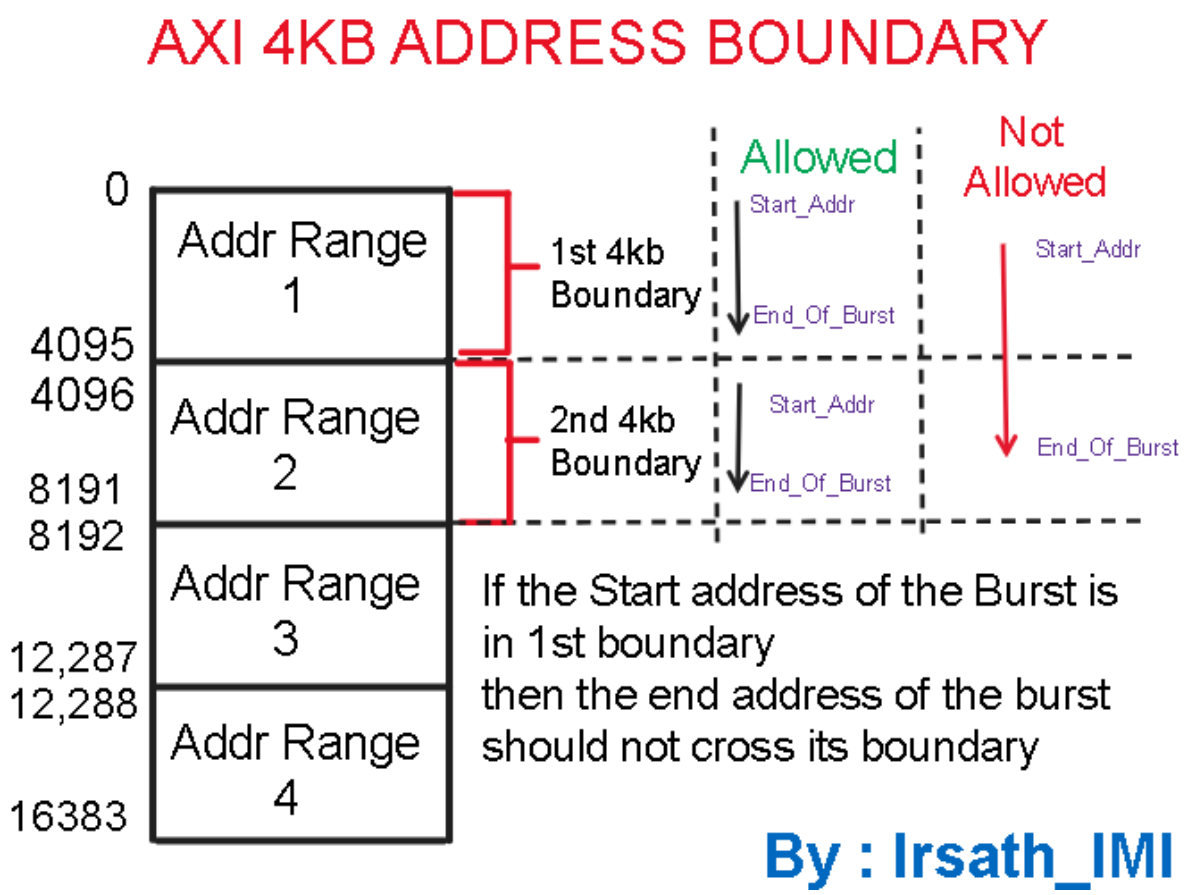
In order to achieve the above constraint, first, we need to understand what the AXI-4Kb Boundary is. AXI is a parallel data transfer protocol that supports burst transfers, meaning it

can send multiple data phases followed by a single address phase. In the address phase, all the information required for the upcoming data transfers is mentioned, including the start address, size of beat (number of bytes transferred in every transfer), and burst size (number of beats or data transfers for the transaction).

By using this information, the slave device needs to store the incoming data into its corresponding address by performing some calculations.

For example, if the awaddr = 100 awsize = 0 awlen = 15, the first data will be stored at the 100th address, the second at the 101st address, and so on. The 16th data will be stored at the address calculated as (awaddr + 2^awsize (awlen - 1)), which is 100 + 2^0 * (16 - 1) = 115th address. Thus, by providing one address, we can send multiple data, and the slave will calculate its corresponding address to store the incoming data. In our case, we stored 16 data starting from address 100 to 115.

Now, let's understand the AXI 4Kb Boundary. In the context of the AXI protocol, a 4Kb boundary refers to a memory address boundary that is aligned to a 4-kb boundary. Memory addresses in the AXI protocol are typically required to be aligned to certain boundaries for efficient memory access.



By seeing the above image we can understand that All the slave addresses are separated by a boundary of 4Kb So, if the start address falls under 0 to 4095, the transaction should not cross this boundary. For example, if the start address is 4000, the transaction should be

stored from 4000 to 4099 (4000 * 1 + 100 - 1). This is an illegal transfer that should not happen, as the start address lies between 0 to 4095. Transactions that start within this range should end inside this range and should not cross this boundary.

With this understanding, let's dive into writing a constraint for this. If we have any address:

1. Find the address offset for the 4Kb Boundary. The address may fall under any boundary like 0 to 4095 or 4096 to 8191 or 8192 to 12287, and so on. All the ranges have 4096 address locations. address offset = awaddr % 4096.
2. Calculate total transfers: Total transfers = 2^awsize * (awlen + 1).
3. The addition of the offset and the total transfer should not cross 4096.

Example: awaddr = 12100 awsize = 4 awlen = 15 Total transfers = 2^4 *(15 + 1) = 16 **** 16 = 256 The transaction will happen from 12100 to (12100 + 256 - 1) = 12355 (**illegal**) as it crosses the boundary 12287.

Applying our three steps:

1. Address offset = 12100 % 4096 = 3908
2. Total transfers = 256
3. Address offset + Total transfer = 3908 + 256 = 4164, which is greater than 4096.

The constraint below will ensure that a transaction does not cross the 4Kb boundary:

```
constraint axi_4kb {
   awaddr % 4096 + (2**awsize * (awlen + 1)) <= 4096;
}
```

Check out the Below Executable Code and Output Below:

```
class axi_tx;
  rand bit [31:0] awaddr;
  rand bit [ 3:0] awsize;
  rand bit [ 7:0] awlen;

  constraint axi_4kb {
    awaddr % 4096 + (2**awsize * (awlen + 1)) <= 4096;
  }
endclass
```

```
module top;
 axi_tx tx=new();
 initial begin
        repeat (5) begin
        assert(tx.randomize());
        $display("awaddr = %0d         awsize = %0d   awlen =
%0d",tx.awaddr,tx.awsize,tx.awlen);
    $display("addr_offset = %0d Total_txn =
%0d",tx.awaddr%4096,(2^tx.awsize*(tx.awlen+1)));
        $display("offset + total_txn <= 4096");
     $display("    %0d   <= 4096 \n",(tx.awaddr%4096+(2**tx.awsize*(tx.awlen+1))));
  end
 end
endmodule
```

```
awaddr = 1154442016   awsize = 1      awlen = 239
addr_offset = 800 Total_txn = 480
offset + total_txn <= 4096
    1280    <= 4096

awaddr = 1550927203   awsize = 0      awlen = 32
addr_offset = 1379 Total_txn = 33
offset + total_txn <= 4096
    1412    <= 4096

awaddr = 3665678381   awsize = 2      awlen = 27
addr_offset = 45 Total_txn = 112
offset + total_txn <= 4096
    157    <= 4096

awaddr = 502678775    awsize = 3      awlen = 138
addr_offset = 1271 Total_txn = 1112
offset + total_txn <= 4096
    2383    <= 4096

awaddr = 1963309261   awsize = 4      awlen = 14
addr_offset = 2253 Total_txn = 240
offset + total_txn <= 4096
    2493    <= 4096
```

In the AXI protocol, the 4KB boundary refers to the alignment of memory addresses, where addresses are multiples of 4096 bytes (4KB). The wrap boundary occurs when a burst transfer, particularly with a wrap type burst, returns to the beginning of the 4KB block after reaching the end of the block. A burst should not cross a 4KB boundary in AXI, and wrap boundary is used in conjunction with burst addressing for transfers within a 4KB block.

Elaboration:

- **4KB Address Boundary:**

Each slave in an AXI system is typically assigned a contiguous block of memory addresses, with the granularity of 4KB. Addresses that are multiples of 4KB are considered 4KB boundaries.

- **Burst Addressing:**

AXI uses bursts to transfer multiple data words in a single transaction. The burst length specifies the number of data words in the burst, and the burst type can be INCR (incrementing address), WRAP (wrapping around to the beginning of the block), or FIXED.

- **Wrap Boundary:**

A wrap burst, when used, will "wrap" or return to the beginning of the 4KB block after reaching the end of the block. This is useful for accessing memory in a circular fashion within the 4KB region.

- **Crossing 4KB Boundary:**

A burst transfer should not cross a 4KB boundary. If a burst crosses a 4KB boundary, it would mean that part of the burst is accessing one slave and the rest is accessing another, which is not supported by the AXI protocol.

- **Wrap Boundary Calculation:**

When using wrap burst, the wrap boundary is calculated based on the starting address, the burst length, and the address size. The address within a burst can be calculated by wrapping the address back to the beginning of the 4KB block if it exceeds the boundary.

In summary:

- **4KB boundary:** AXI protocol requires memory addresses to be aligned to 4KB boundaries.
- **Burst transfers should not cross 4KB boundaries** .
- **Wrap burst**: allows for addressing within a 4KB block in a circular fashion, returning to the start address after reaching the end of the block.

Puzzle:

To measure 7 liters using a 5-liter and an 8-liter bottle, you can fill the 8-liter bottle, then pour water from it into the 5-liter bottle. You'll be left with 3 liters in the 8-liter bottle. Then, fill the 8-liter bottle again and pour water into the 5-liter bottle until the 5-liter bottle is full. This leaves you with 7 liters in the 8-liter bottle.

Here's a step-by-step solution:

1. **Fill the 8-liter bottle completely.**
2. **Pour water from the 8-liter bottle into the 5-liter bottle.** This will leave 3 liters in the 8-liter bottle.
3. **Empty the 5-liter bottle.**
4. **Pour the 3 liters from the 8-liter bottle into the 5-liter bottle.**
5. **Fill the 8-liter bottle completely again.**
6. **Pour water from the 8-liter bottle into the 5-liter bottle (which already has 3 liters).** This will fill the 5-liter bottle completely. At this point, you will have 7 liters remaining in the 8-liter bottle.

PCIe 6.0 offers significantly faster data transfer speeds compared to PCIe 5.0, doubling the data rate to 64 GT/s per lane. This translates to a theoretical maximum bidirectional bandwidth of 256 GB/s for a 16-lane configuration, compared to PCIe 5.0's 128 GB/s. PCIe 6.0 also utilizes PAM4 (Pulse Amplitude Modulation 4-level) encoding, which allows for more data to be transmitted per clock cycle than PCIe 5.0's NRZ (Non-Return-to-Zero) encoding.

Here's a more detailed breakdown:

PCIe 5.0:

Data rate: 32 GT/s per lane, Bandwidth (x16): 64 GB/s, and Encoding: PAM4.

PCIe 6.0:

- Data rate: 64 GT/s per lane
- Bandwidth (x16): 256 GB/s
- Encoding: PAM4 with enhancements for signal integrity and error reduction

Key Differences:

- **Speed:**

PCIe 6.0 doubles the data rate of PCIe 5.0, offering a significant performance boost for data-intensive applications like AI, machine learning, and data centers.

- **Encoding:**

While both use PAM4, PCIe 6.0 includes additional enhancements to improve signal integrity and reduce errors, enabling reliable data transmission at higher speeds.

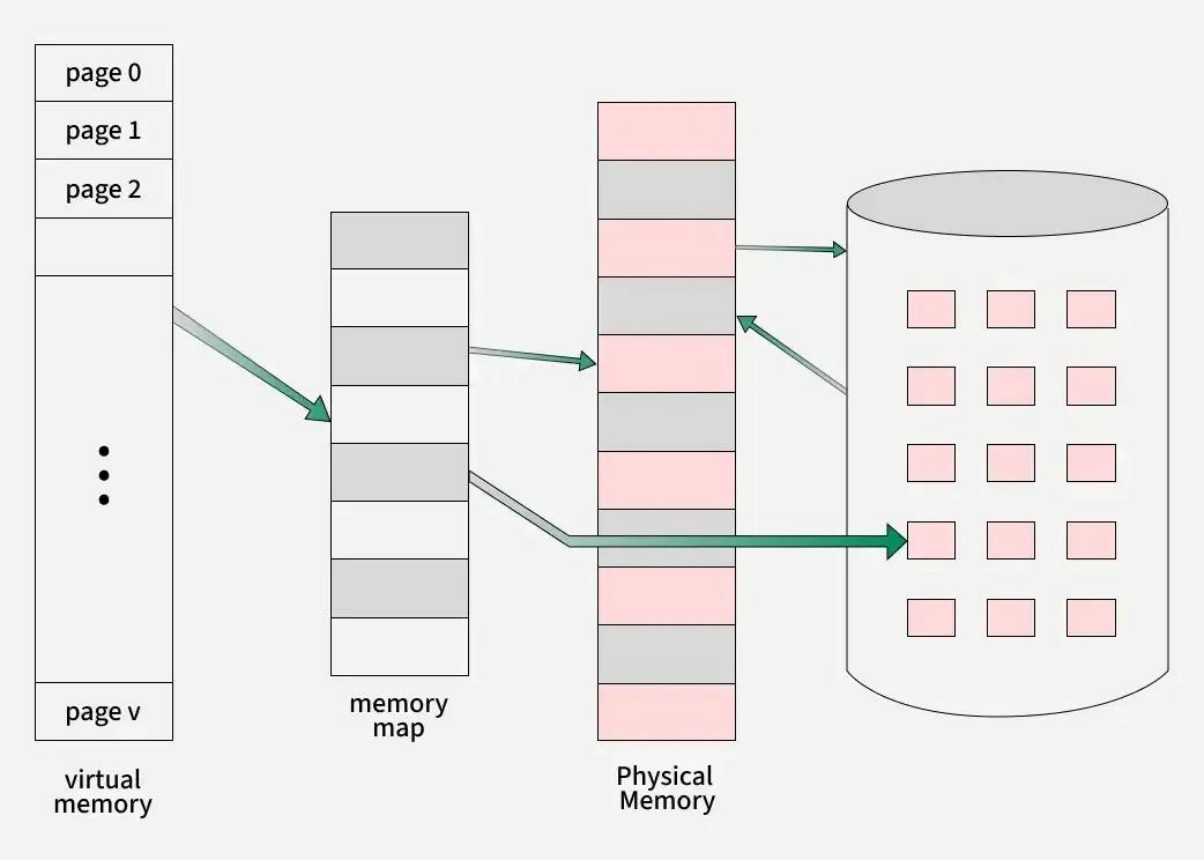
- **Bandwidth:**

The theoretical bandwidth of PCIe 6.0 (256 GB/s x16) is twice that of PCIe 5.0 (128 GB/s x16).

Virtual Memory in Operating System

Virtual memory is a memory management technique used by operating systems to give the appearance of a large, continuous block of memory to applications, even if the physical memory (RAM) is limited. It allows larger applications to run on systems with less RAM.

- The main objective of virtual memory is to support multiprogramming, The main advantage that virtual memory provides is, a running process does not need to be entirely in memory.
- Programs can be larger than the available physical memory. Virtual Memory provides an abstraction of main memory, eliminating concerns about storage limitations.
- A memory hierarchy, consisting of a computer system's memory and a disk, enables a process to operate with only some portions of its address space in RAM to allow more processes to be in memory.



A virtual memory is what its name indicates- it is an illusion of a memory that is larger than the real memory. We refer to the software component of virtual memory as a virtual memory manager. The basis of virtual memory is the noncontiguous memory allocation model. The virtual memory manager removes some components from memory to make room for other components.

The size of virtual storage is limited by the addressing scheme of the computer system and the amount of secondary memory available not by the actual number of main storage locations.

The History of Virtual Memory

Before virtual memory, computers used RAM and secondary memory for data storage. Early computers used magnetic core in place of main memory and magnetic drums in place of secondary memory. In the 1940s and 1950s, computer memory was very expensive and limited in size. As programs became larger and more complex, developers had to worry that their programs might use up all the available memory and cause the computer to run out of space to work.

In those early days, if program was larger than available space then programmers use the concept of overlaying. If some portion of the code that is not currently used move to

overlying if needed it back overwritten to memory this cause extensive programming. This is the reason for developing virtual memory.

In 1956, German physicist **Fritz-Rudolf Güntsch** develop virtual memory. The first real example of a virtual memory system was created at the University of Manchester in England while developing the Atlas computer. This system used a method called paging, which allowed virtual addresses (used by programs) to be mapped to the computer's main memory. The Atlas computer was built in 1959 and started working in 1962.

The first computer with virtual memory was released by Burroughs Corp in 1961. This version of virtual memory do not using paging it oppose paging but support segmentation.

How Virtual Memory Works?

Virtual Memory is a technique that is implemented using both hardware and software. It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory.

- All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process can be swapped in and out of the main memory such that it occupies different places in the main memory at different times during the course of execution.
- A process may be broken into a number of pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time address translation and the use of a page or segment table permits this.

If these characteristics are present then, it is not necessary that all the pages or segments are present in the main memory during execution. This means that the required pages need to be loaded into memory whenever required. Virtual memory is implemented using Demand Paging or Demand Segmentation.

Types of Virtual Memory

In a computer, virtual memory is managed by the Memory Management Unit (MMU), which is often built into the CPU. The CPU generates virtual addresses that the MMU translates into physical addresses.

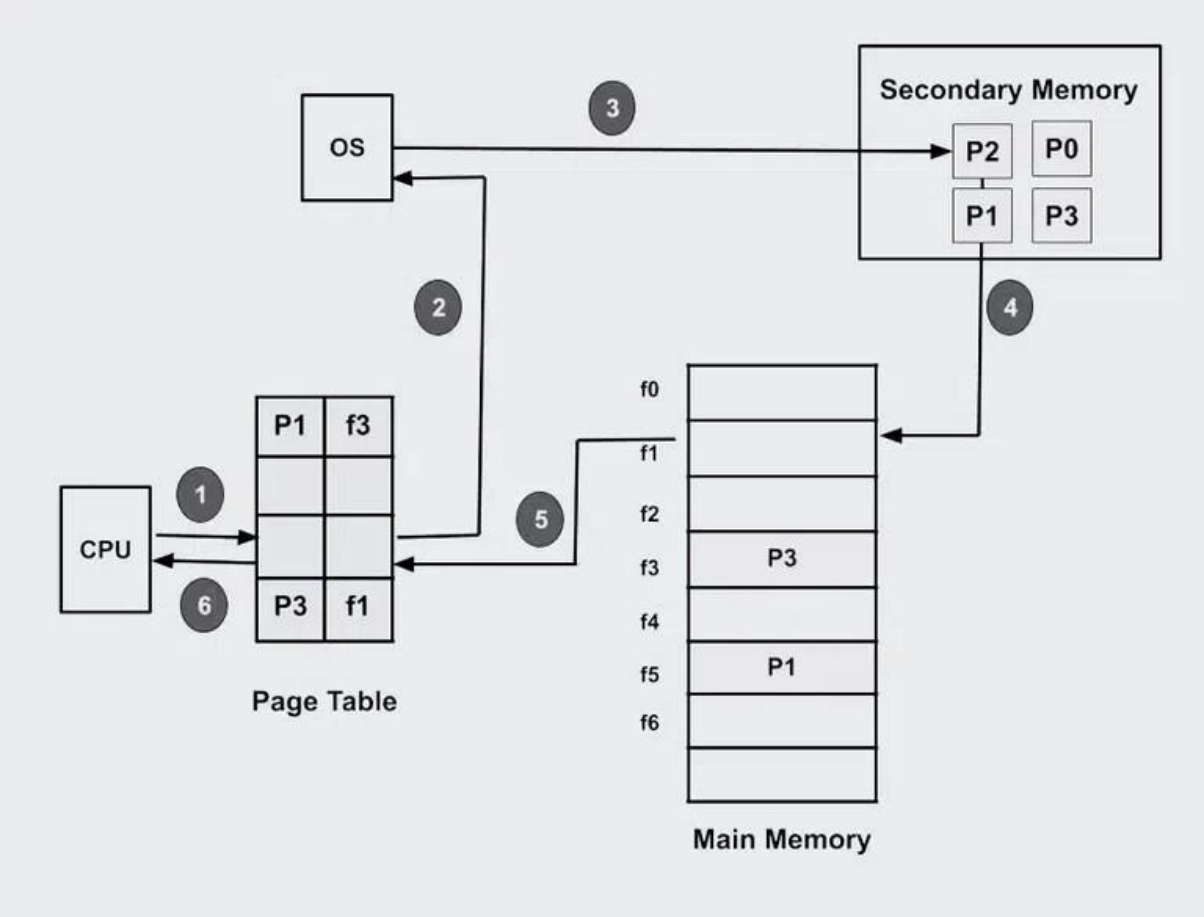
There are two main types of virtual memory:

- Paging
- Segmentation

Paging

Paging divides memory into small fixed-size blocks called pages. When the computer runs out of RAM, pages that aren't currently in use are moved to the hard drive, into an area called a swap file. The swap file acts as an extension of RAM. When a page is needed again, it is swapped back into RAM, a process known as page swapping. This ensures that the operating system (OS) and applications have enough memory to run.

Demand Paging: The process of loading the page into memory on demand (whenever a page fault occurs) is known as demand paging. The process includes the following steps are as follows:



- If the CPU tries to refer to a page that is currently not available in the main memory, it generates an interrupt indicating a memory access fault.
- The OS puts the interrupted process in a blocking state. For the execution to proceed the OS must bring the required page into the memory.
- The OS will search for the required page in the logical address space.
- The required page will be brought from logical address space to physical address space. The page replacement algorithms are used for the decision-making of replacing the page in physical address space.

- The page table will be updated accordingly.
- The signal will be sent to the CPU to continue the program execution and it will place the process back into the ready state.

Hence whenever a page fault occurs these steps are followed by the operating system and the required page is brought into memory.

What is Page Fault Service Time?

The time taken to service the page fault is called page fault service time. The page fault service time includes the time taken to perform all the above six steps.

Let Main memory access time is: m
Page fault service time is: s
Page fault rate is : p
Then, Effective memory access time = (p*s) + (1-p)*m

Page vs Frame

A page is a fixed size block of data in virtual memory and a frame is a fixed size block of physical memory in RAM where these pages are loaded. Think of a page as a piece of a puzzle (virtual memory) and a frame as the spot where it fits on the board (physical memory). When a program runs its pages are mapped to available frames so the program can run even if the program size is larger than physical memory.

Segmentation

Segmentation divides virtual memory into segments of different sizes. Segments that aren't currently needed can be moved to the hard drive. The system uses a segment table to keep track of each segment's status, including whether it's in memory, if it's been modified, and its physical address. Segments are mapped into a process's address space only when needed.

read more about – Segmentation

Combining Paging and Segmentation

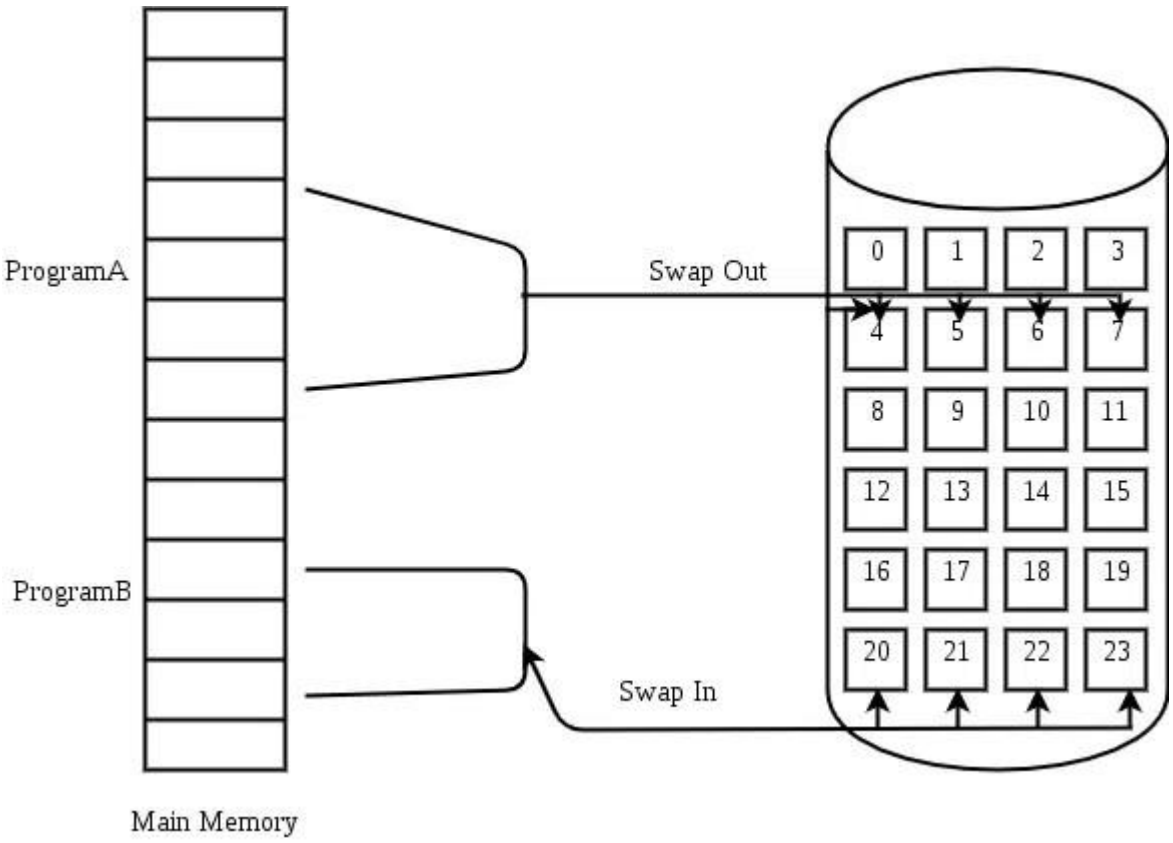
Sometimes, both paging and segmentation are used together. In this case, memory is divided into pages, and segments are made up of multiple pages. The virtual address includes both a segment number and a page number.

Virtual Memory vs Physical Memory

Feature	Virtual Memory	Physical Memory (RAM)
Definition	An abstraction that extends the available memory by using disk storage	The actual hardware (RAM) that stores data and instructions currently being used by the CPU
Location	On the hard drive or SSD	On the computer's motherboard
Speed	Slower (due to disk I/O operations)	Faster (accessed directly by the CPU)
Capacity	Larger, limited by disk space	Smaller, limited by the amount of RAM installed
Cost	Lower (cost of additional disk storage)	Higher (cost of RAM modules)
Data Access	Indirect (via paging and swapping)	Direct (CPU can access data directly)
Volatility	Non-volatile (data persists on disk)	Volatile (data is lost when power is off)

What is Swapping?

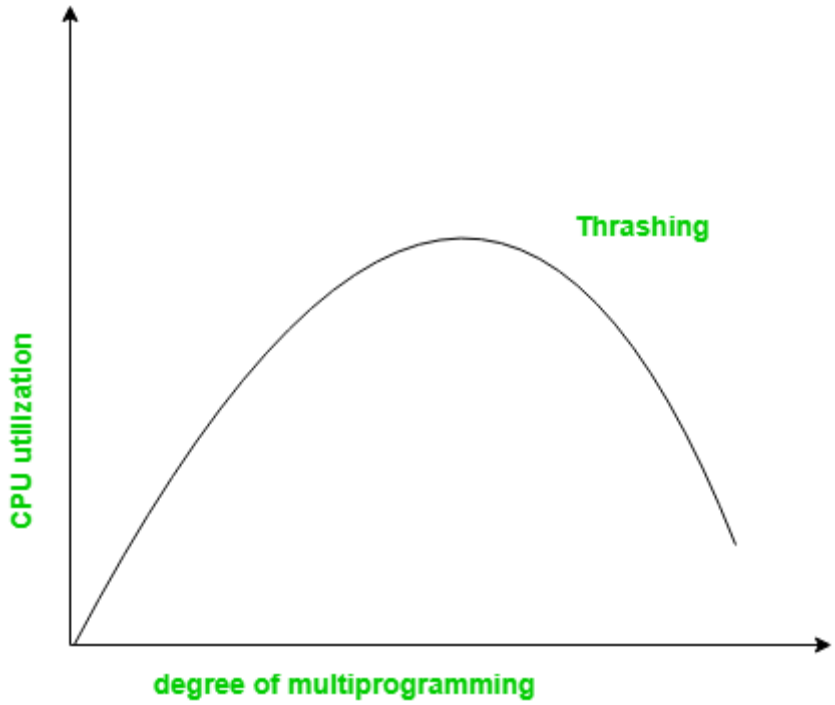
Swapping is a process out means removing all of its pages from memory, or marking them so that they will be removed by the normal page replacement process. Suspending a process ensures that it is not runnable while it is swapped out. At some later time, the system swaps back the process from the secondary storage to the main memory. When a process is busy swapping pages in and out then this situation is called thrashing.



Swappinghierar

What is Thrashing?

At any given time, only a few pages of any process are in the main memory, and therefore more processes can be maintained in memory. Furthermore, time is saved because unused pages are not swapped in and out of memory. However, the OS must be clever about how it manages this scheme. In the steady state practically, all of the main memory will be occupied with process pages, so that the processor and OS have direct access to as many processes as possible. Thus when the OS brings one page in, it must throw another out. If it throws out a page just before it is used, then it will just have to get that page again almost immediately. Too much of this leads to a condition called Thrashing. The system spends most of its time swapping pages rather than executing instructions. So a good page replacement algorithm is required.



In the given diagram, the initial degree of multiprogramming up to some extent of point(lambda), the CPU utilization is very high and the system resources are utilized 100%. But if we further increase the degree of multiprogramming the CPU utilization will drastically fall down and the system will spend more time only on the page replacement and the time taken to complete the execution of the process will increase. This situation in the system is called thrashing.

Causes of Thrashing

Thrashing occurs in a computer system when the CPU spends more time swapping pages in and out of memory than executing actual processes. This happens when there is insufficient physical memory, causing frequent page faults and excessive paging activity. Thrashing reduces system performance and makes processes run very slowly. There are many cause of thrashing as discussed below.

1. High Degree of Multiprogramming

If the number of processes keeps on increasing in the memory then the number of frames allocated to each process will be decreased. So, fewer frames will be available for each process. Due to this, a page fault will occur more frequently and more CPU time will be wasted in just swapping in and out of pages and the utilization will keep on decreasing.

For example:
Let free frames = 400
Case 1: Number of processes = 100
Then, each process will get 4 frames.

Case 2: Number of processes = 400
Each process will get 1 frame.
Case 2 is a condition of thrashing, as the number of processes is increased, frames per process are decreased. Hence CPU time will be consumed just by swapping pages.

2. Lacks of Frames

If a process has fewer frames then fewer pages of that process will be able to reside in memory and hence more frequent swapping in and out will be required. This may lead to thrashing. Hence a sufficient amount of frames must be allocated to each process in order to prevent thrashing.

Recovery of Thrashing

- Do not allow the system to go into thrashing by instructing the long-term scheduler not to bring the processes into memory after the threshold.
- If the system is already thrashing then instruct the mid-term scheduler to suspend some of the processes so that we can recover the system from thrashing.

Performance in Virtual Memory

- Let p be the page fault rate(0 <= p <= 1).
- if p = 0 no page faults
- if p =1, every reference is a fault.

Effective access time (EAT) = (1-p)* Memory Access Time + p * Page fault time.
Page fault time = page fault overhead + swap out + swap in +restart overhead

The performance of a virtual memory management system depends on the total number of page faults, which depend on "paging policies" and "frame allocation"

read more about – Techniques to handle Thrashing

Frame Allocation

A number of frames allocated to each process in either static or dynamic.

- **Static Allocation:** The number of frame allocations to a process is fixed.
- **Dynamic Allocation:** The number of frames allocated to a process changes.

Paging Policies

- **Fetch Policy:** It decides when a page should be loaded into memory.
- **Replacement Policy:** It decides which page in memory should be replaced.
- **Placement Policy:** It decides where in memory should a page be loaded.

What are the Applications of Virtual memory?

Virtual memory has the following important characteristics that increase the capabilities of the computer system. The following are five significant characteristics of Lean.

- **Increased Effective Memory:** One major practical application of virtual memory is, virtual memory enables a computer to have more memory than the physical memory using the disk space. This allows for the running of larger applications and numerous programs at one time while not necessarily needing an equivalent amount of DRAM.
- **Memory Isolation:** Virtual memory allocates a unique address space to each process and that also plays a role in process segmentation. Such separation increases safety and reliability based on the fact that one process cannot interact with and or modify another's memory space through a mistake, or even a deliberate act of vandalism.
- **Efficient Memory Management:** Virtual memory also helps in better utilization of the physical memories through methods that include paging and segmentation. It can transfer some of the memory pages that are not frequently used to disk allowing RAM to be used by active processes when required in a way that assists in efficient use of memory as well as system performance.
- **Simplified Program Development:** For case of programmers, they don't have to consider physical memory available in a system in case of having virtual memory. They can program 'as if' there is one big block of memory and this makes the programming easier and more efficient in delivering more complex applications.

How to Manage Virtual Memory?

Here are 5 key points on how to manage virtual memory:

1. Adjust the Page File Size

- **Automatic Management**: All contemporary operating systems including Windows contain the auto-configuration option for the size of the empirical page file. But depending on the size of the RAM, they are set automatically, although the user can manually adjust the page file size if required.
- **Manual Configuration:** For tuned up users, the setting of the custom size can sometimes boost up the performance of the system. The initial size is usually advised to be set to the minimum value of 1. To set the size of the swap space equal to 5 times the amount of physical RAM and the maximum size 3 times the physical RAM.

2. Place the Page File on a Fast Drive

- **SSD Placement:** If this is feasible, the page file should be stored in the SSD instead of the HDD as a storage device. It has better read and write times, and the virtual memory may prove beneficial in an SSD.

- **Separate Drive:** Regarding systems having multiple drives involved, the page file needs to be placed on a different drive than the OS and that shall in turn improve its performance.

3. Monitor and Optimize Usage

- **Performance Monitoring:** Employ the software tools used in monitoring the performance of the system in tracking the amounts of virtual memory. High page file usage may signify that there is a lack of physical RAM or that virtual memory needs a change of settings or addition in physical RAM.
- **Regular Maintenance:** Make sure there is no toolbar or other application running in the background, take time and uninstall all the tool bars to free virtual memory.

4. Disable Virtual Memory for SSD

- **Sufficient RAM:** If for instance your system has a big physical memory, for example 16GB and above then it would be advised to freeze the page file in order to minimize SSD usage. But it should be done, in my opinion, carefully and only if the additional signals that one decides to feed into his applications should not likely use all the available RAM.

5. Optimize System Settings

- **System Configuration:** Change some general properties of the system concerning virtual memory efficiency. This also involves enabling additional control options in Windows such as adjusting additional system setting option on the operating system, or using other options in different operating systems such as Linux that provides different tools and commands to help in adjusting how virtual memory is utilized.
- **Regular Updates:** Ensure that your drivers are run in their newest version because new releases contain some enhancements and issues regarding memory management.

What are the Benefits of Using Virtual Memory?

- Many processes maintained in the main memory.
- A process larger than the main memory can be executed because of demand paging. The OS itself loads pages of a process in the main memory as required.
- It allows greater multiprogramming levels by using less of the available (primary) memory for each process.
- It has twice the capacity for addresses as main memory.
- It makes it possible to run more applications at once.

- Users are spared from having to add memory modules when RAM space runs out, and applications are liberated from shared memory management.
- When only a portion of a program is required for execution, speed has increased.
- Memory isolation has increased security.
- It makes it possible for several larger applications to run at once.
- Memory allocation is comparatively cheap.
- It doesn't require outside fragmentation.
- It is efficient to manage logical partition workloads using the CPU.
- Automatic data movement is possible.

What are the Limitation of Virtual Memory?

- It can slow down the system performance, as data needs to be constantly transferred between the physical memory and the hard disk.
- It can increase the risk of data loss or corruption, as data can be lost if the hard disk fails or if there is a power outage while data is being transferred to or from the hard disk.
- It can increase the complexity of the memory management system, as the operating system needs to manage both physical and virtual memory.

Conclusion

In conclusion, virtual memory is a crucial feature in operating systems that allows computers to run larger applications and handle more processes than the physical RAM alone can support. By using techniques like paging and segmentation, the system extends the available memory onto the hard drive, ensuring that the operating system and applications can operate smoothly. Although virtual memory can introduce some performance overhead due to the slower speed of hard drives compared to RAM, it provides significant benefits in terms of memory management, efficiency, and multitasking capabilities.

forwarding in pipeline example:

In pipelined processing, forwarding, also known as bypassing, is a technique to mitigate data hazards by delivering the result of an earlier instruction directly to the next instruction that needs it, bypassing the register file. This prevents pipeline stalls, improving execution speed.

Here's a more detailed explanation with an example:

Data Hazards and the Need for Forwarding

- **Pipelining:**

Instructions are broken down into stages (fetch, decode, execute, memory access, write back) and processed in a parallel, sequential manner.

- **Data Hazards:**

A situation where an instruction needs the result of a previous instruction before that result is written back to the register file. This can cause the pipeline to stall, waiting for the result.

- **Example:**

Consider the following instructions:

    - ADD R1, R2, R3 (adds R2 and R3, storing the result in R1)
    - SUB R4, R1, R5 (subtracts R5 from R1)
    - Without forwarding, SUB would have to wait for ADD to write the result to the register file before it could read the value.

- **Forwarding:**

Instead of stalling, the forwarding mechanism delivers the result of ADD (R1's new value) directly to SUB in the execute stage, where it's needed, before it's written back to the register file.

How Forwarding Works

1. **1. Identify Data Dependencies:**

Hardware checks if an instruction's input registers are being written to by a prior instruction in the pipeline.

2. **2. Forwarding Logic:**

If a dependency is found, the forwarding unit takes the result from the appropriate pipeline stage (e.g., EX/MEM, MEM/WB) and delivers it to the input of the dependent instruction's execute stage.

3. **3. Bypassing:**

The dependent instruction uses the forwarded result instead of reading from the register file, avoiding the stall.

Benefits of Forwarding

- **Improved Performance:** Reduces stalls and increases the throughput of the pipeline.
- **Increased Parallelism:** Allows instructions to be executed in a more parallel manner, as they don't have to wait for the register file to be updated.

In essence, forwarding makes the data available to the next instruction as soon as it is calculated, bypassing the need to wait for the result to be written to the register file.

Difference between struct and union in C?

Structures and unions are both user-defined data types in C that allow grouping variables of different types. However, they differ in how they allocate and manage memory.

- **Memory Allocation:**
    - **Structure:** Each member of a structure is assigned its own unique memory location. The size of a structure is equal to or greater than the sum of the sizes of its members, due to potential padding for alignment purposes.
    - **Union:** All members of a union share the same memory location. The size of a union is equal to the size of its largest member.
- **Data Access:**
    - **Structure:** All members of a structure can be accessed and modified independently at any time.
    - **Union:** Only one member of a union can hold a valid value at any given time. Assigning a value to one member overwrites the value of any other member.
- **Keyword:**
    - **Structure:** struct
    - **Union:** union

C

```c
#include <stdio.h>

// Structure definition
struct Student {
    char name[50];
    int roll_number;
    float marks;
};

// Union definition
union Data {
    int i;
    float f;
    char str[20];
};

int main() {
    // Structure usage
    struct Student s1;
    strcpy(s1.name, "John Doe");
    s1.roll_number = 10;
    s1.marks = 85.5;

    printf("Student Name: %s\n", s1.name);
    printf("Roll Number: %d\n", s1.roll_number);
    printf("Marks: %.2f\n", s1.marks);

    // Union usage
    union Data data;
    data.i = 10;
    printf("data.i: %d\n", data.i);
    data.f = 220.5;
    printf("data.f: %.2f\n", data.f);
    strcpy(data.str, "C Programming");
    printf("data.str: %s\n", data.str);
    printf("data.i: %d\n", data.i); // data.i is now overwritten
    printf("data.f: %.2f\n", data.f); // data.f is also overwritten

    printf("Size of Student structure: %lu bytes\n", sizeof(s1));
    printf("Size of Data union: %lu bytes\n", sizeof(data));
```

```
    return 0;
}
```

In C programming, both structures and unions are used to group different types of data under a single name, but they behave in different ways. The main difference lies in how they store data.

The below table lists the primary differences between the C structures and unions:

Parameter	Structure	Union
Definition	A structure is a user-defined data type that groups different data types into a single entity.	A union is a user-defined data type that allows storing different data types at the same memory location.
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union
Size	The size is the sum of the sizes of all members, with padding if necessary.	The size is equal to the size of the largest member, with possible padding.
Memory Allocation	Each member within a structure is allocated unique storage area of location.	Memory allocated is shared by individual members of union.
Data Overlap	No data overlap as members are independent.	Full data overlap as members shares the same memory.
Accessing Members	Individual member can be accessed at a time.	Only one member can be accessed at a time.

Structures

A **structure in C** is a collection of variables, possibly of different types, under a single name. Each member of the structure is allocated its own memory space, and the size of the structure is the sum of the sizes of all its members.

Syntax

struct *name {*
member1 definition;
member2 definition;
…
memberN definition;
};

Example:

```
#include <stdio.h>


struct Student {
    char name[50];
    int age;
    float grade;
};


int main() {


    // Create a structure variable
    struct Student s1 = {"Geek", 20, 85.5};


    // Access structure members
    printf("%s\n", s1.name);
    printf("%d\n", s1.age);
    printf("%.2f\n", s1.grade);
        printf("Size: %d bytes", sizeof(s1));


    return 0;
}
```

Output

Geek

20

85.50

Size: 60 bytes

Explanation: In this example, we create a structure Student to store a student's name, age, and grade. Each of the members (name, age, grade) is stored in its own separate memory location, and we access them individually. The size is also the size of all members combined plus structure padding.

Unions

A **union in C** is similar to a structure, but with a key difference: all members of a union share the same memory location. This means only one member of the union can store a value at any given time. The size of a union is determined by the size of its largest member.

Syntax:

union *name {*
member1 definition;
member2 definition;
…
memberN definition;
};

Example:

{...}

union Data {

int i;

double d;

char c;

};

int main() {

```
    // Create a union variable
    union Data data;

    // Store an integer in the union
    data.i = 100;
    printf("%d
", data.i);

    // Store a double in the union (this will
        // overwrite the integer value)
    data.d = 99.99;
    printf("%.2f
", data.d);

    // Store a character in the union (this will
        // overwrite the double value)
    data.c = 'A';
    printf("%c
", data.c);

        printf("Size: %d", sizeof(data));

 {...}
```

Output

100

99.99

A

Size: 8

Similarities Between Structure and Union

Structures and unions are also similar in some aspects listed below:

- Both are user-defined data types used to store data of different types as a single unit.
- Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.
- Both structures and unions support only assignment = and sizeof operators. The two structures or unions in the assignment must have the same members and member types.
- A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.
- '.' operator or selection operator, which has one of the highest precedences, is used for accessing member variables inside both the user-defined datatypes.

Here's how to get a directory list in Python, Perl, and C:

Python

Python

```
import os

def get_directory_list(path):
  """Returns a list of files and directories in the given path."""
  return os.listdir(path)

# Example usage:
path = "."  # Current directory
directory_list = get_directory_list(path)
print(directory_list)
```

Perl

Code

```
use strict;
use warnings;
use File::Slurp;

sub get_directory_list {
   my ($path) = @_;
   return read_dir($path);
}

# Example usage:
my $path = "."; # Current directory
my @directory_list = get_directory_list($path);
print "@directory_list\n";
```

C

C

```
#include <stdio.h>
#include <dirent.h>
#include <errno.h>
#include <stdlib.h>

char **get_directory_list(const char *path, int *count) {
   DIR *dir;
   struct dirent *entry;
   char **file_list = NULL;
   *count = 0;

   dir = opendir(path);
   if (dir == NULL) {
      perror("opendir");
      return NULL;
   }

   while ((entry = readdir(dir)) != NULL) {
      file_list = realloc(file_list, (*count + 1) * sizeof(char*));
       if (file_list == NULL) {
         perror("realloc");
         closedir(dir);
         return NULL;
      }
```

```
        file_list[*count] = strdup(entry->d_name);
        if (file_list[*count] == NULL) {
            perror("strdup");
            closedir(dir);
            // Free previously allocated strings
            for (int i = 0; i < *count; i++) {
                free(file_list[i]);
            }
            free(file_list);
            return NULL;
        }
        (*count)++;
    }

    closedir(dir);
    return file_list;
}

int main() {
    const char *path = "."; // Current directory
    int count;
    char **directory_list = get_directory_list(path, &count);

    if (directory_list != NULL) {
        for (int i = 0; i < count; i++) {
            printf("%s\n", directory_list[i]);
            free(directory_list[i]); // Free the allocated string
        }
        free(directory_list); // Free the array of strings
    }

    return 0;
}
```

How to resolve RAW WAR WAW hazards?

RAW (Read After Write), WAR (Write After Read), and WAW (Write After Write) hazards can be resolved through various techniques, including register renaming, code reorganization,

and forwarding. These hazards arise from instruction dependencies and the pipelined execution of instructions.

RAW (Read After Write) Hazards:

- **Description:**

A RAW hazard occurs when an instruction needs the result of a previous instruction that hasn't yet finished writing it.

- **Resolution:**
    - **Forwarding:** The result of the previous instruction can be forwarded directly to the next instruction, avoiding the need to wait for the register to be updated.
    - **Stalling:** The pipeline can be stalled if forwarding is not available, ensuring that the next instruction waits until the previous instruction's result is available.
    - **Code Reorganization:** The compiler can reorder instructions to execute the dependent instruction later, after the result is available.
- **Example:**

If instruction A writes to register R1, and instruction B immediately tries to read from R1, there is a RAW hazard.

WAR (Write After Read) and WAW (Write After Write) Hazards:

- **Description:**

WAR and WAW hazards involve a conflict in register access due to the same register being used for reading and writing in different instructions.

- **Resolution:**
    - **Register Renaming:** The compiler or hardware can rename one or both of the registers involved in the hazard to different names, eliminating the conflict.
    - **Stalling:** The pipeline can be stalled to ensure the write is completed before the read.
    - **Code Reorganization:** The compiler can reorder instructions to avoid the conflicting access.
- **Example:**

If instruction A reads from register R1, and instruction B then writes to R1, there is a WAR hazard. If instruction A and instruction B both write to R1, there is a WAW hazard.

Key Concepts:

- **Instruction Dependencies:**

Hazards arise from dependencies between instructions, where one instruction's execution depends on the result of another.

- **Register Renaming:**

A technique where the compiler or hardware changes the register names to avoid conflicts.

- **Forwarding:**

A technique where the result of a previous instruction is sent directly to the next instruction without waiting for it to complete writing to the register.

- **Stalling:**

The pipeline can be paused to ensure the result is available before the dependent instruction attempts to read it.

- **Out-of-Order Execution:**

Modern CPUs can execute instructions out of order, allowing for more efficient pipeline utilization and hazard resolution.

File handling is a crucial aspect of programming, allowing programs to interact with data stored in files. Here's a comparison of file handling in Python, Perl, and C:
Python
Python simplifies file handling with built-in functions. Opening a file.
Python

```
file = open("filename.txt", "mode")
```

where mode can be 'r' (read), 'w' (write), 'a' (append), or 'x' (create). Reading from a file.
Python

```
with open("filename.txt", "r") as file:
    content = file.read() # Read the entire file
    # or
    for line in file: # Read line by line
        print(line)
```

Writing to a file.
Python

```
with open("filename.txt", "w") as file:
    file.write("Hello, world!")
```

Closing a file: Using with open() automatically closes the file. Otherwise, use file.close().
Perl
Perl, known for its text processing capabilities, also offers robust file handling. Opening a file.
Code

```
open my $filehandle, "mode", "filename.txt" or die $!;
```

where mode is similar to Python's. Reading from a file.
Code

```
while (my $line = <$filehandle>) {
    print $line;
}
```

Writing to a file.
Code

```
print $filehandle "Hello, world!\n";
```

Closing a file.
Code

```
close $filehandle;
```

C
C requires more explicit handling of file operations. Opening a file.
C

```
FILE *file_pointer = fopen("filename.txt", "mode");
if (file_pointer == NULL) {
    perror("Error opening file");
    return 1;
}
```

Reading from a file.
C

```
char buffer[255];
while (fgets(buffer, sizeof(buffer), file_pointer) != NULL) {
    printf("%s", buffer);
}
```

Writing to a file.
C

```
    fprintf(file_pointer, "Hello, world!\n");
```

Closing a file.

C

```
    fclose(file_pointer);
```

In pipelined processors, forwarding, also known as bypassing, is a technique used to avoid data hazards, where a later instruction needs a result from an earlier instruction before the result is written back to the register file. Instead of stalling the pipeline, forwarding routes the result directly from the pipeline stage where it's produced to the dependent instruction, preventing delays.

Example:

Imagine a sequence of instructions where the second instruction, an AND operation, needs the result of the SUB instruction that writes into register X2. Without forwarding, the AND instruction would have to wait for the SUB instruction to complete and write the result back to the register file.

With forwarding, the result of the SUB instruction is available at the output of the EX stage (the stage where the ALU performs the operation). The forwarding mechanism can detect that the AND instruction needs the result of SUB and directly forward it from the ALU output to the ALU input for the AND instruction. This way, the AND instruction receives the correct value without waiting for the register file update, avoiding a stall in the pipeline.

In essence: Forwarding allows instructions to use results from earlier instructions before they are written to the register file, optimizing pipeline performance by preventing stalls caused by data dependencies.

Is wrap same as incr in axi

In AXI (Advanced eXtensible Interface), WRAP and INCR are not the same—they define different burst types used in AXI transactions:

1. INCR (Incrementing Burst)

Address increments by the size of each data transfer.

Useful for sequential access (e.g., reading an array).

Example: If the starting address is 0x1000 and burst length is 4 with 4-byte data, addresses would be:
0x1000, 0x1004, 0x1008, 0x100C.

2. WRAP (Wrapping Burst)
Similar to INCR, but the address wraps around to a lower boundary when it reaches a certain point.

Used for cache-line aligned accesses or circular buffers.

Example: With a 4-beat, 4-byte WRAP burst starting at 0x1010, addresses might be:
0x1010, 0x1014, 0x1018, 0x1000 (assuming a 16-byte wrap boundary).

Key Difference:
INCR continues incrementing normally.

WRAP increments but wraps around at a boundary determined by the burst length and data size.

Operating system concepts

Operating system (OS) concepts are foundational principles that govern how an operating system manages hardware and software resources. Here's an overview of the key concepts:

Processes

A process is an instance of a running program.

The OS handles process creation, scheduling, synchronization, and termination.

Threads

Threads are the smallest unit of CPU execution within a process.

Multithreading allows multiple threads within a process to run concurrently.

Memory Management

The OS manages RAM allocation to processes.

Techniques include paging, segmentation, and virtual memory.

File Systems

Organizes and stores data on storage devices.

Provides access control, file naming, directories, and metadata management.

Input/Output (I/O) Management

Manages communication between hardware devices and applications.

Uses drivers and I/O scheduling algorithms.

Scheduling

Determines the order in which processes run on the CPU.

Algorithms include FCFS, SJF, Round Robin, and priority scheduling.

Deadlocks

Occur when processes are stuck waiting for each other indefinitely.

The OS uses prevention, avoidance, or detection and recovery methods.

Synchronization

Ensures correct execution of processes that share resources.

Mechanisms include semaphores, mutexes, and monitors.

Security and Protection

Prevents unauthorized access to system resources.

Involves user authentication, permissions, and isolation.

Virtualization

Allows multiple OSes to run on the same hardware using virtual machines.

Would you like a deep dive into any of these topics or a summary in the form of notes or diagrams?

Mutexes in operating system

Mutexes (Mutual Exclusion Objects) are synchronization primitives used in operating systems to prevent race conditions when multiple threads or processes access shared resources (like variables, files, or memory) concurrently.

Key Concepts of Mutex:
Purpose:

Ensures mutual exclusion, meaning only one thread can access the critical section (shared resource) at a time.

How Mutex Works:

A mutex has two states: locked and unlocked.

When a thread wants to enter a critical section:

It locks the mutex.

If the mutex is already locked, the thread must wait.

When the thread finishes, it unlocks the mutex, allowing others to enter.

Basic Operations:

lock() – Acquire the mutex (wait if already locked).

unlock() – Release the mutex.

try_lock() – Attempt to lock without blocking.

Example (Pseudocode):

```
mutex.lock();      // Enter critical section
// critical section
mutex.unlock();    // Leave critical section
```

Problems Addressed:

Prevents race conditions.

Maintains data consistency in concurrent environments.

Pitfalls:

Deadlocks – Two threads waiting on each other's mutex.

Priority Inversion – A low-priority thread holds a mutex needed by a high-priority thread.

Common Uses:

Thread synchronization in multithreaded programs.

Shared resource protection in OS kernels.

Would you like a visual diagram or a real-code example (like in C or Python)?

Multithreading and multitasking

Multithreading and Multitasking are both concepts in operating systems that deal with running multiple operations, but they differ in scope and implementation:

Multithreading:
Definition: The ability of a single process to manage multiple threads (smaller units of a process) simultaneously.

Level: Happens within a process.

Purpose: Increases efficiency by allowing parallel execution of tasks within the same application.

Example Use Case: A web browser that loads images, renders text, and downloads files using different threads.

Benefits:

Efficient use of CPU.

Faster execution due to shared memory.

Lower overhead compared to multiple processes.

Multitasking:
Definition: The OS runs multiple processes at the same time.

Level: Happens at the OS level with multiple independent programs.

Purpose: Ensures multiple applications can run seemingly simultaneously.

Types:

Preemptive Multitasking: OS decides when to switch tasks (used in most modern OSes).

Cooperative Multitasking: Tasks voluntarily yield control.

Example Use Case: Listening to music while browsing the web and downloading a file.

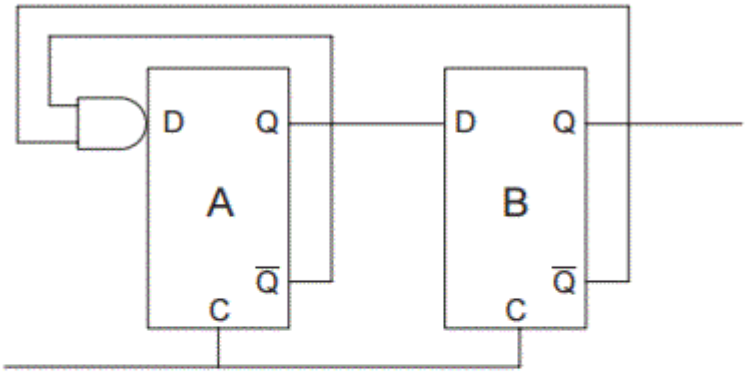
Benefits:

Better system responsiveness.

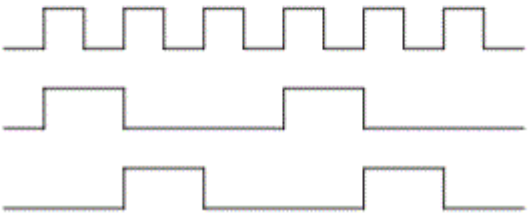
Improved resource utilization.

Key Differences:
Feature Multithreading Multitasking
Scope Within a single process Across multiple processes

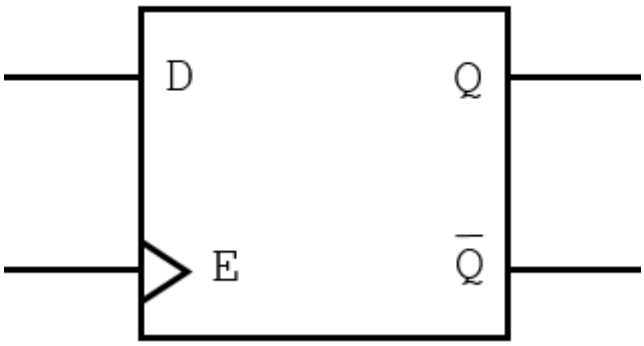
Memory Shared memory space Separate memory for each process
Overhead Low (lightweight) Higher (heavyweight)
Communication Easier (same memory space) Complex (needs inter-process comm.)
Would you like code examples or a diagram comparing the two?

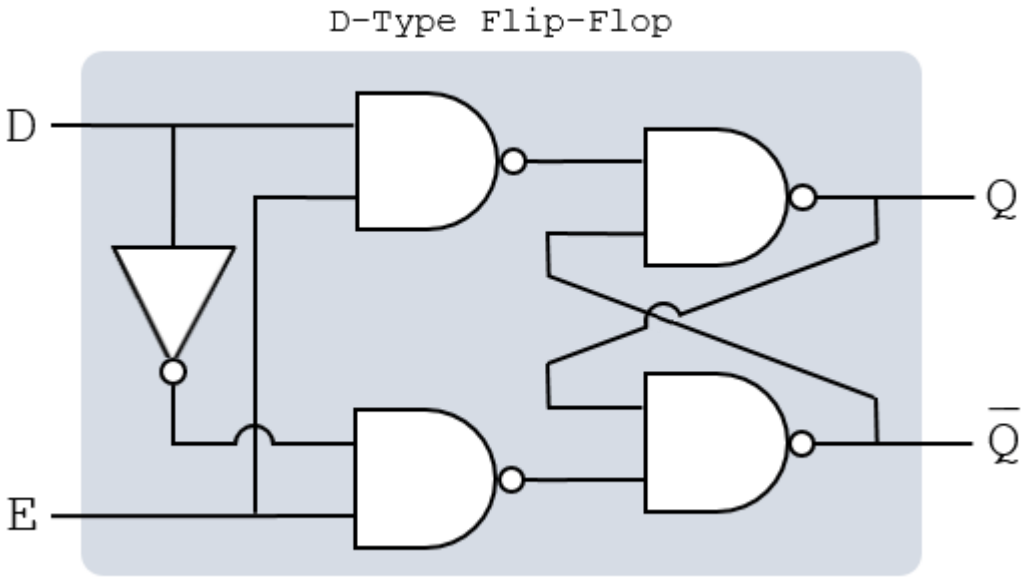


Divide By 3



D-Type Flip-Flop





PCIe Specification	Data Rate per lane	Encoding	x16 bandwidth (GB/s)	Specification ratification year
1.x	2.5	8b/10b	4	2003
2.x	5	8b/10b	8	2007
3.x	8	128b/130b	15.75	2010
4.0	16	128b/130b	31.5	2017
5.0	32	128b/130b	63	2019
6.0	64	PAM4/FLIT	128	2022

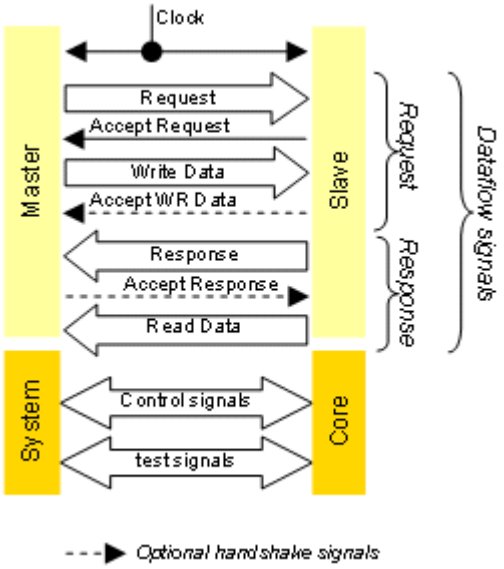


Table 4-3 Burst type encoding

ARBURST[1:0] AWBURST[1:0]	Burst type	Description	Access
b00	FIXED	Fixed-address burst	FIFO-type
b01	INCR	Incrementing-address burst	Normal sequential memory
b10	WRAP	Incrementing-address burst that wraps to a lower address at the wrap boundary	Cache line
b11	Reserved	-	-

Table A3-2 Burst size encoding

AxSIZE[2:0]	Bytes in transfer
0b000	1
0b001	2
0b010	4
0b011	8
0b100	16
0b101	32
0b110	64
0b111	128

Table A3-3 Burst type encoding

AxBURST[1:0]	Burst type
0b00	FIXED
0b01	INCR
0b10	WRAP
0b11	Reserved

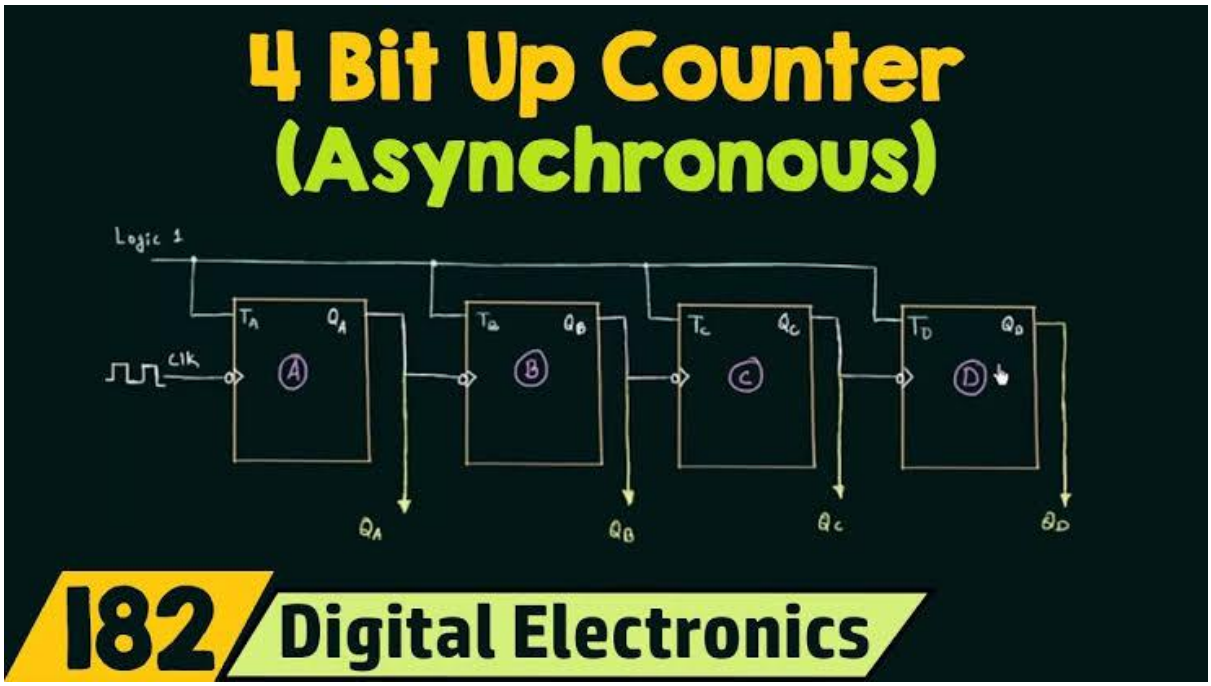


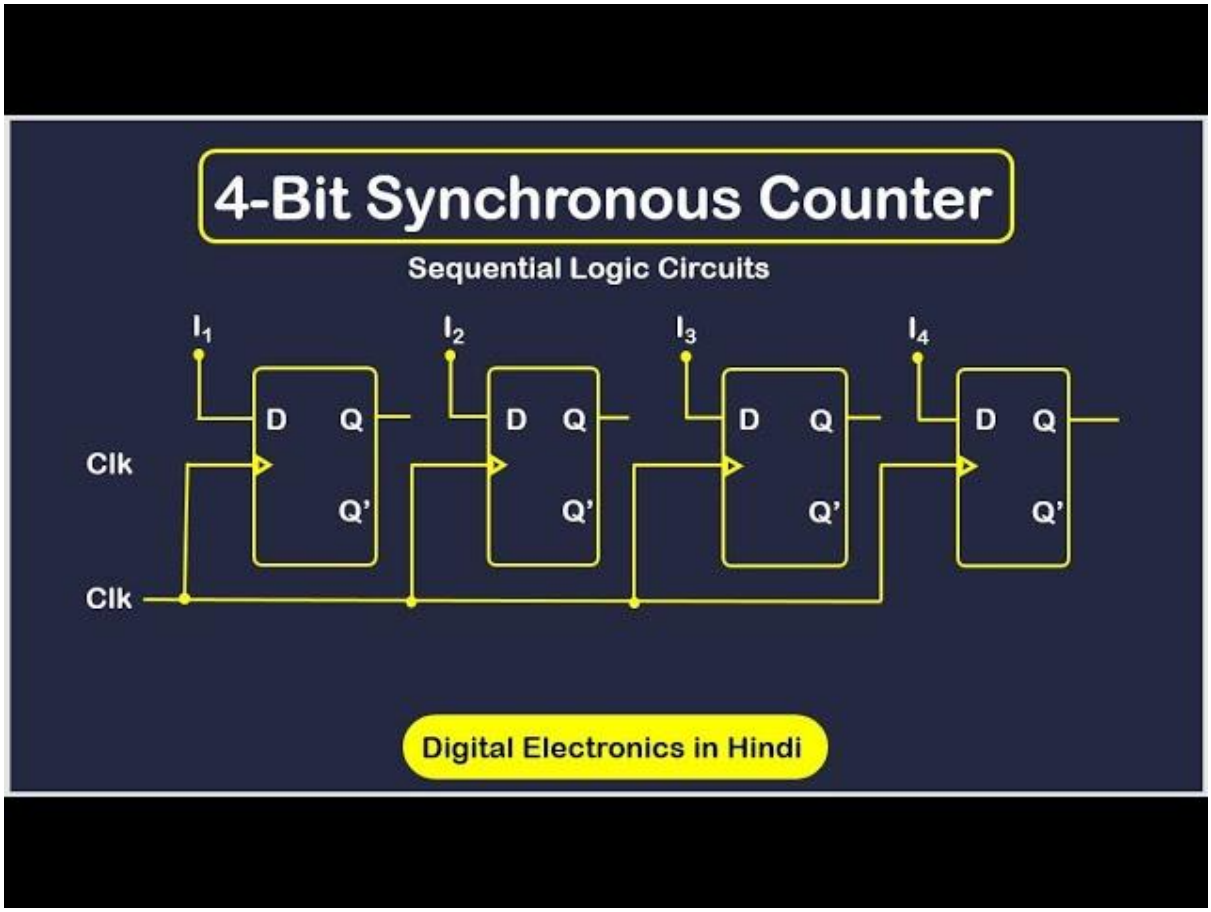
Table 7-1 RRESP[1:0] and BRESP[1:0] encoding

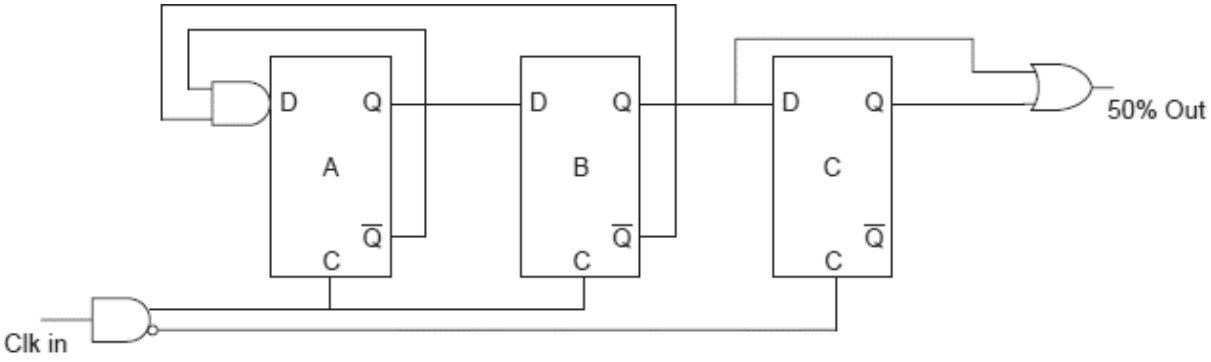
RRESP[1:0] BRESP[1:0]	Response	Meaning
b00	OKAY	Normal access okay indicates if a normal access has been successful. Can also indicate an exclusive access failure.
b01	EXOKAY	Exclusive access okay indicates that either the read or write portion of an exclusive access has been successful.
b10	SLVERR	Slave error is used when the access has reached the slave successfully, but the slave wishes to return an error condition to the originating master.
b11	DECERR	Decode error is generated typically by an interconnect component to indicate that there is no slave at the transaction address.

ARLOCK[1:0] AWLOCK[1:0]	Access type
b00	Normal access
b01	Exclusive access
b10	Locked access
b11	Reserved

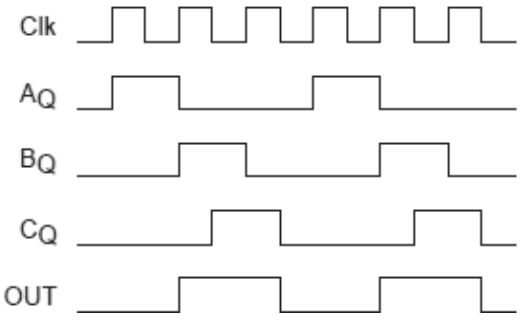
Revision	Max Data Rate	Encoding	Signaling
PCIe 7.0 (2025)	128.0 GT/s	1b/1b (Flit Mode*)	PAM4
PCIe 6.0 (2022)	64.0 GT/s	1b/1b (Flit Mode*)	PAM4
PCIe 5.0 (2019)	32.0 GT/s	128b/130b	NRZ
PCIe 4.0 (2017)	16.0 GT/s	128b/130b	NRZ
PCIe 3.0 (2010)	8.0 GT/s	128b/130b	NRZ
PCIe 2.0 (2007)	5.0 GT/s	8b/10b	NRZ
PCIe 1.0 (2003)	2.5 GT/s	8b/10b	NRZ

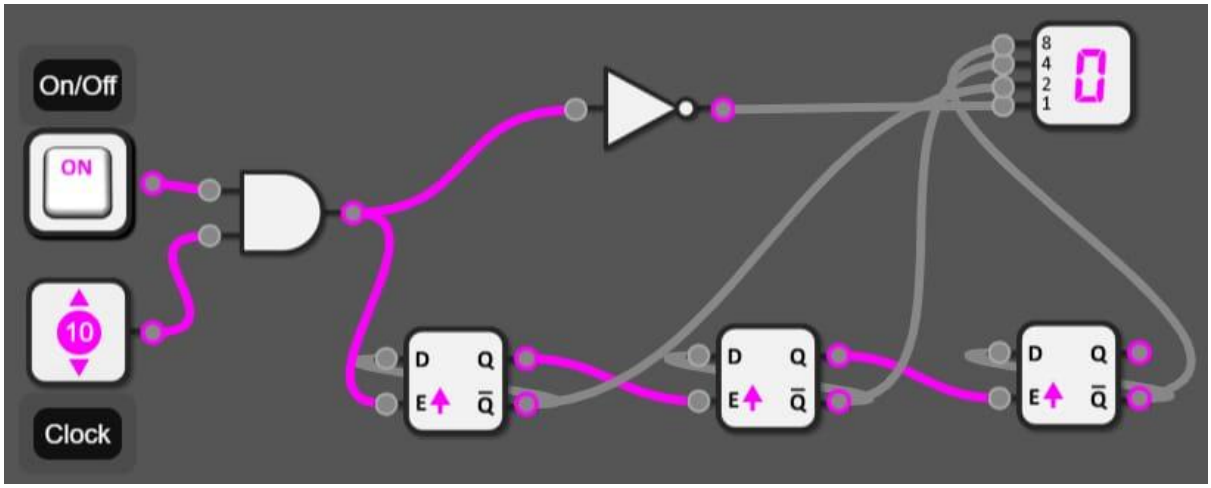
(*Flit Mode also enabled in other Data Rate with their respective encoding)

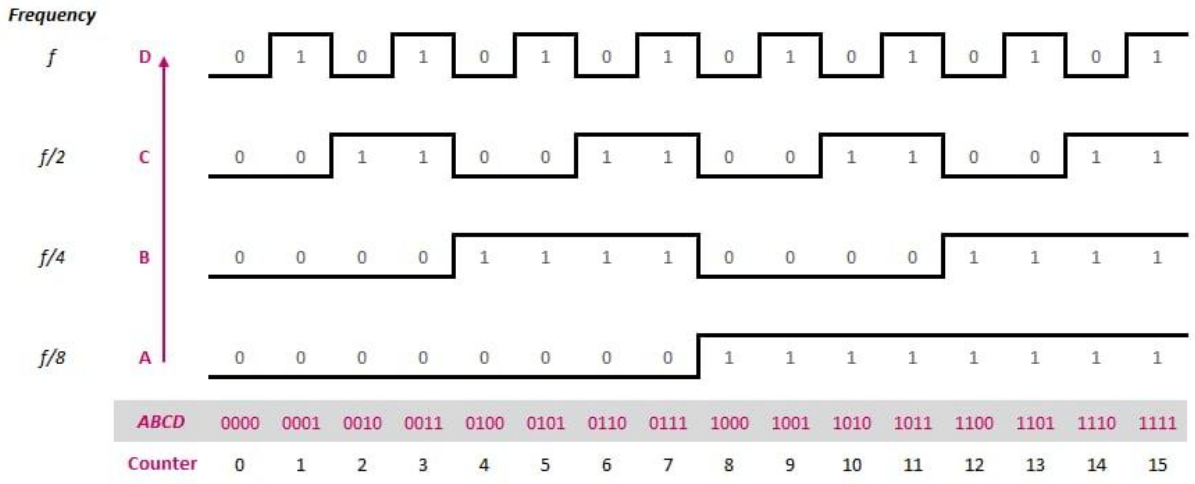


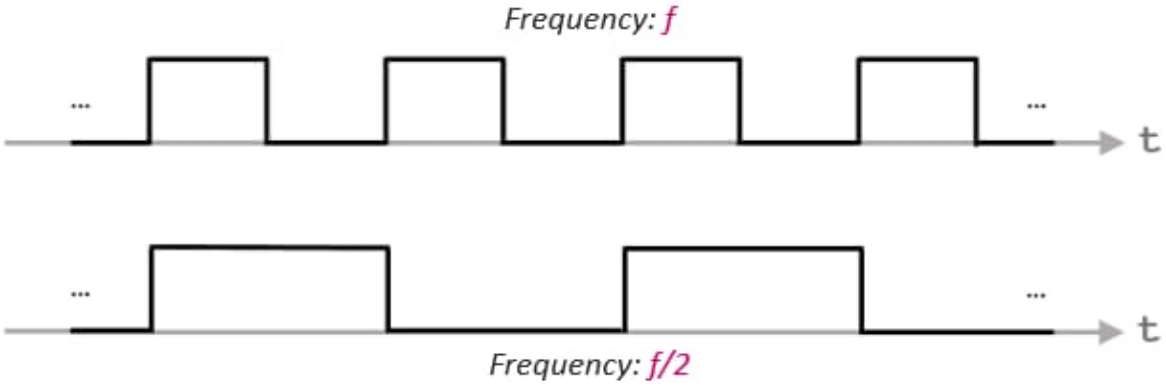


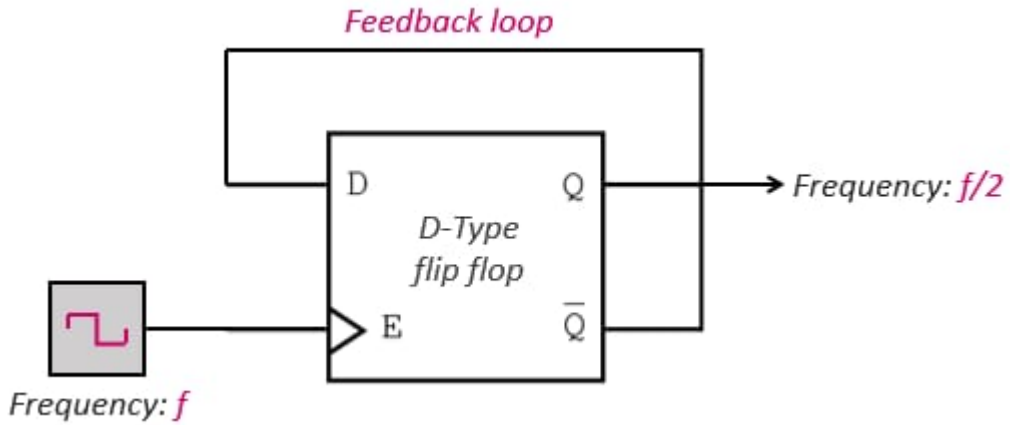
Divide By 3 W/50% out

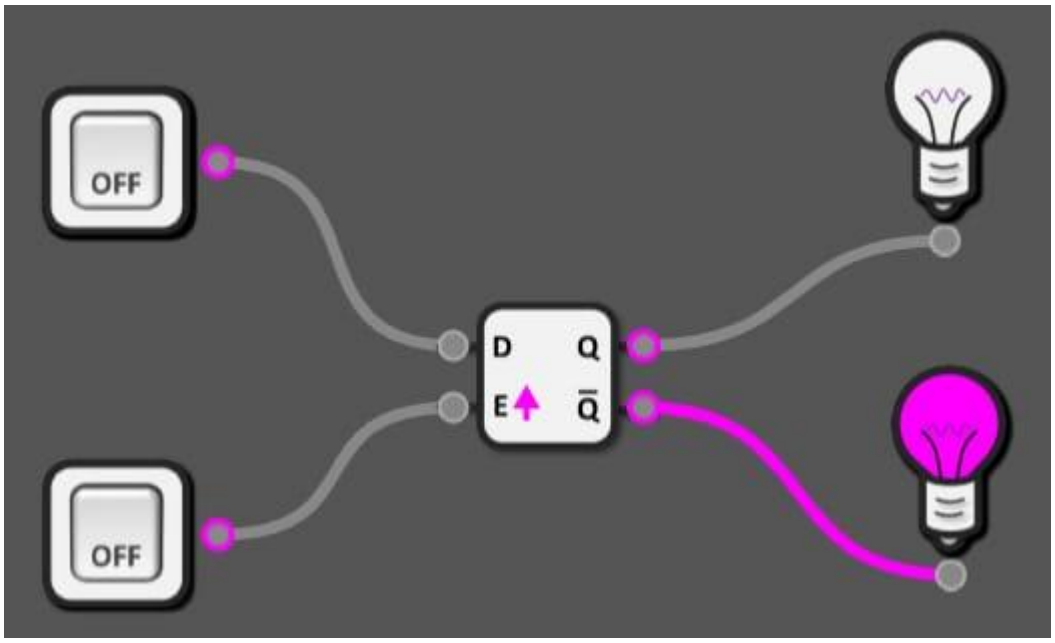


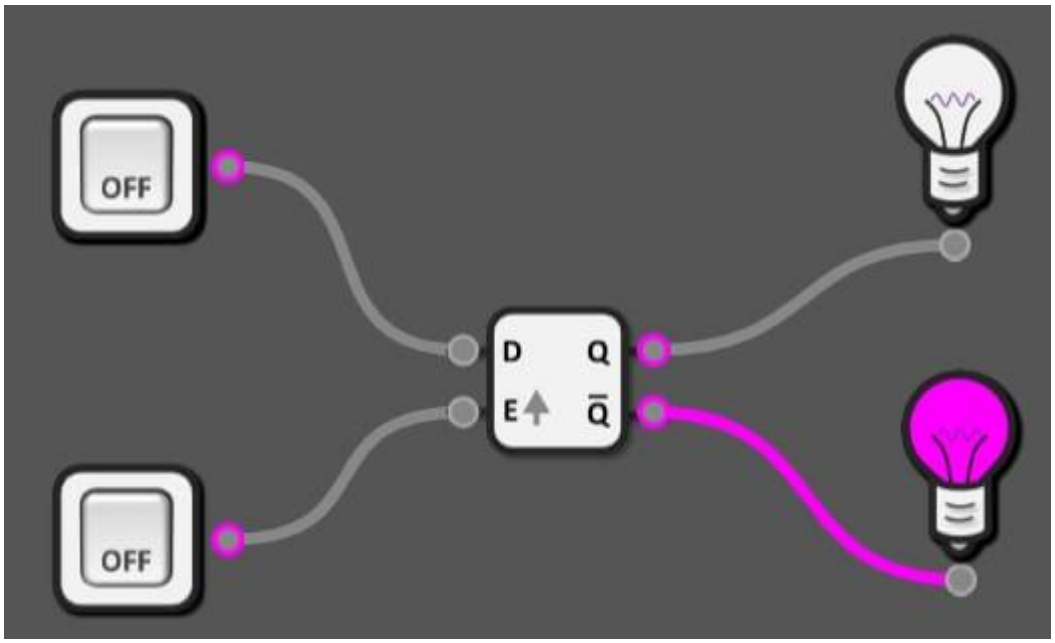

On/Off
ON
10
Clock
D
Q
E
Q̄
8
4
2
1
0


Frequency
f
D
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
f/2
C
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
f/4
B
0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
f/8
A
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
ABCD
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
Counter
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

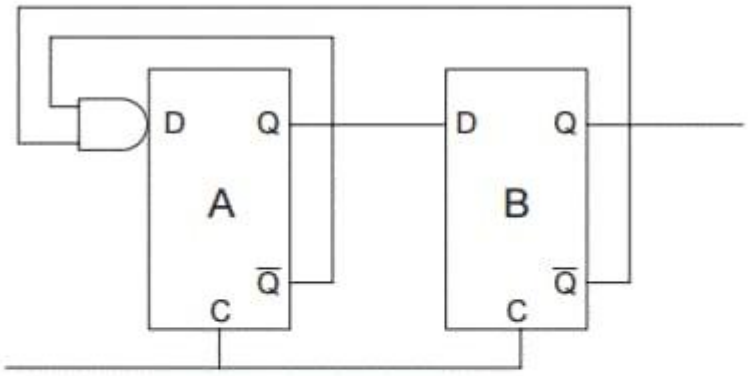

Frequency: f
t
Frequency: f/2
t


Feedback loop
D
Q
D-Type
flip flop
E
Q̄
Frequency: f/2
Frequency: f

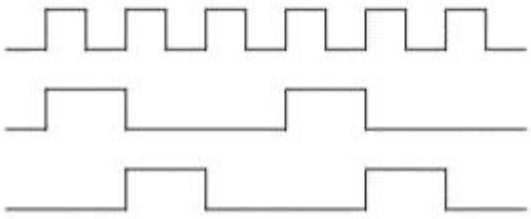

OFF
OFF
D
Q
E
Q̄

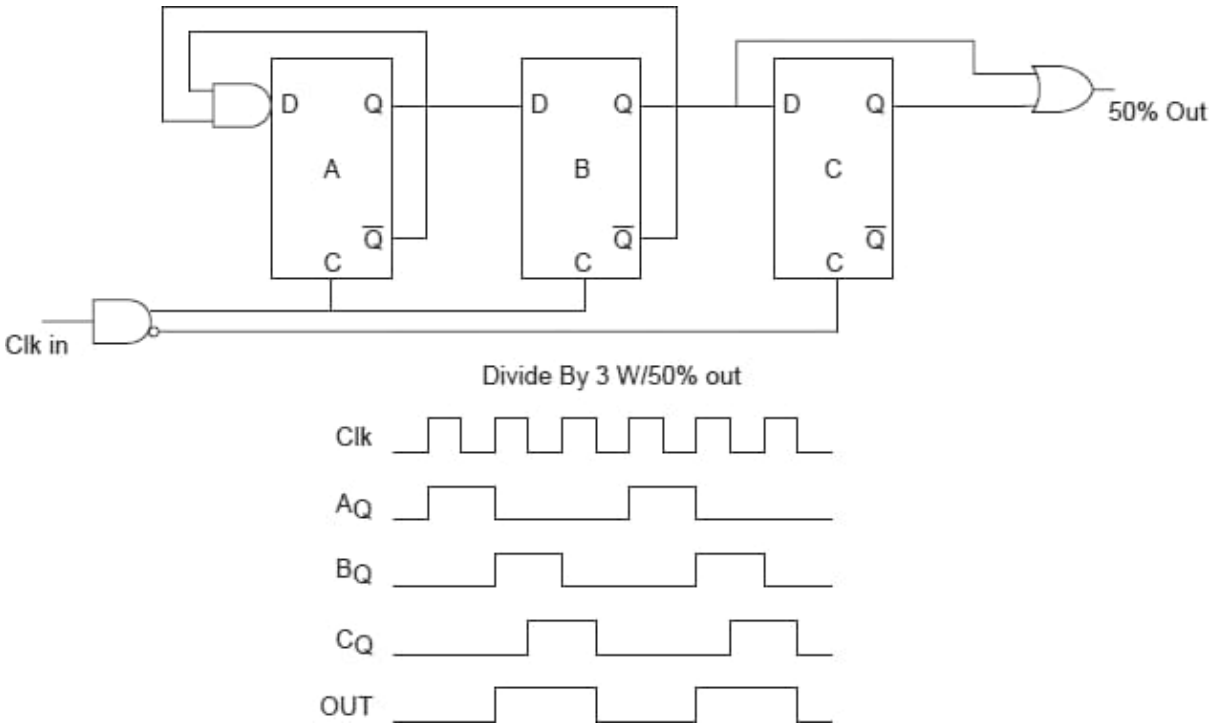

OFF
OFF
D
Q
E
Q̄

Input		Output	
D	E	Q	$\overline{Q}$
0	1	0	1
1	1	1	0
0	0	Q	$\overline{Q}$
1	0	Q	$\overline{Q}$



Divide By 3





Divide By 3 W/50% out

Imperial College
London

Lecture 6
Instruction Set Architecture
(RISC-V ISA)

Peter Cheung
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/EE2_CAS/
E-mail: p.cheung@imperial.ac.uk

In this lecture, the instruction set architecture (ISA) of the RISC-V processor will be introduced. We will only consider the base instruction set for the 32-bit integer version of the ISA. Focus will be on the six different instruction types with emphasis each instruction's functionality and encoding of its machine code.

ISA of a processor does not dictate how the processor is implemented. It only defines how to the processor is programmed. The actual hardware architecture will be covered in the next lecture.

RISC-V

- Developed by Krste Asanovic, David Patterson and colleagues at UC Berkeley in 2010
- First widely accepted **open-source** computer architecture
- Underlying design principles:
  1. **Simplicity favours regularity**
  2. **Make the common case fast**
  3. **Smaller is faster**
  4. **Good design demands good compromises**

RISC-V is by no means the first or the only Reduced Instruction Set Computer that enjoys widespread adoption. The early RISC processor from Berkeley, the MIPS, was used for over four decades in industry. The UK's ARM processor (original stands for Acorn Risc Machine) is the most manufactured CPU in history with an estimated 200 billions being shipped to date. However, RISC-V is the first widely accepted open-source RISC processor. Opening the ISA to the public (royalty free) is a new business model and has captured much attention in the past 5 years. Together with the free toolchain for development and many open-source design freely available, RISC-V is expected to pose real competition to x86 and ARM architecture to become at least one of the dominant play in the sector.

RISC-V was developed by Krste Asanovic and Dave Patterson (and others) in Berkeley in early 2010's. The ISA was first published in 2011, and its future development and ratification are under the control of RISC-V Foudation and RISC-V International located in Switzerland.

According to Patterson and Hennessy's textbook, the underpinning design principles are shown on the slides.

Design Principles

Principle 1: Simplicity favors regularity

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Easier to encode and handle in hardware

Principle 2: Make the common case fast

- RISC-V includes only simple, commonly used instructions
- Hardware to decode and execute instructions can be simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions
- RISC-V is a ***reduced instruction set computer*** **(RISC)**, with a small number of simple instructions
- Other architectures, such as Intel's x86, are ***complex instruction set computers*** **(CISC)**

Principle 3: Smaller is Faster

H&H p301-303

As shown in later slides, the instruction set has highly consistent format. The base instruction set is known as RV32I (RISC-V 32-bit integer only) only has 40 instructions. The ISA has two sources and one destination operands. The format of the instructions are divided into only six different types.

The second principle is the RISC-V only has commonly used instructions. Yet, it is Turing Complete, meaning that it can be used to implement any computer algorithms. This makes the RISC-V implementation both small and fast. Complex operations are achieved by stringing together multiple instructions.

Instructions: Addition & Subtraction

C Code	RISC-V assembly code
`a = b + c;`	`add a, b, c`
`a = b - c;`	`sub a, b, c`

- **`Add/sub:`** mnemonic indicates operation to perform
- **`b, c:`** source operands (on which the operation is performed)
- **`a:`** destination operand (to which the result is written)

More complex code is handled by multiple RISC-V instructions.

C Code	RISC-V assembly code
`a = b + c - d;`	`add t, b, c  # t = b + c` `sub a, t, d  # a = t - d`

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

H&H p303

To understand the RV32I ISA, we start with the add and subtract instructions as shown in this slide. The diea is very simple – it shows that for add instructions, we need two source operands (b and c) and one destimation (a).

RISC-V Operands

- **Operand location:** physical location in computer
  - Registers
  - Memory
  - Constants (also called *immediates*)

- RISC-V has 32 32-bit registers
- Registers are faster than memory
- RISC-V called “32-bit architecture” because it operates on 32-bit data

H&H p304

Where do operands come from and where does the destination operand go?

There are three possibilities. The fastest and most often used operand is from or to **registers** on the CPU chip itself. RISC-V RV32I has 32 32-bit registers. They form an integral part of the CPU design and accessing them is easy and fast. 32 registers means the instruction must use 3 x 5-bit = 15 bit of the 32-bit instruction to specify a register-only (R-type) instruction.

The second possibility is from data memory. To access this, the instruction must specify the data memory address, using a register as a pointer. In RISC-V, the register content is often specified with an associated offset constant value as part of the instruction.

The third possibility is from instruction memory, i.e. the operand is a constant within the instruction itself. In ”RISC-V speak”, this is called an immediate.

32-bit RISC-V Instruction Types

Instruction Type	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate (I-type)	imm[11:0]												rs1					funct3			rd					opcode						
Upper (U-type)	imm[31:12]																				rd					opcode						
Store (S-type)	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch (B-type)	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump (J-type)	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- **opcode (7 bit):** partially specifies which of the 6 types of *instruction formats*
- **funct7 + funct3 (10 bit):** combined with **opcode**, these two fields describe what operation to perform
- **rs1 (5 bit):** specifies register containing first operand
- **rs2 (5 bit):** specifies second register operand
- **rd (5 bit):**: Destination register specifies register which will receive result of computation

RISC-V RV32I has six types of instructions.

R-type (**Register/register**) instructions use only registers as source and destiantions. This instruction type is mostly used for arithmetic and logic operations involving the ALU.

I-type (**Immediate**) instructions has one of the two source operands specified within the 32-bit instruction word as a 12-bit constant (or immediate). This constant is regards as 12-bit signed 2's complement number, which is always sign extended to form a 32-bit operand.

S-type (**Store**) instructions are exclusively used for storing contents of a register to data memory.

B-type (**Branch**) instructions are used to control program flow. It compares two operands stored in registers and branch to a destination address relative to the current Program Counter value.

J-type (**Jump**) instructions are used for subroutine calls.

U-type (**Upper immediate**) instructions are used to specify the upper 20 bits immediate value of a register.

RISC-V Registers

Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporaries

H&H p305

RISC-V RV32 has 32 registers designated as **x0** to **x31**. They are "general purpose" registers in the sense that the ISA allows them to be used for any purpose with the exception of **x0**, which ALWAYS contain the value 32'b0. Writing to x0 does not change its content.

Having **x1** to **x31** for any general use can be confusing. Common good practice is included in a guideline where specific registers are used for special functions. For example x1 is used to store the **return address** (of a subroutine) and therefore **x1** is also called **ra**.

The table above shows the various aliases for all 32 registers. You are recommend o use the given name of these registers to make the program more readable. For example instead of using **x0**, you should always refer to it as **zero**.

RISC-V operand from Registers

Name	Register Number	Usage
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
s2-11	x18-27	Saved registers

C Code

```
a = b + c;

a = b + 6;
```

RISC-V assembly code

```
# s0 = a, s1 = b, s2 = c
add s0, s1, s2

# s0 = a, s1 = b
addi s0, s1, 6
```

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						

H&H p305

Consider again the add instructions. ADD is a typical ALU instruction in the class of arithematic and logic operations. It needs two source operands and one destination operands to store the results. Shown here is the instruction: `add s0, s1, s2` which uses three registers. Consider the encode of this instructions (slide 21).

op	funct3	funct7	Type	Instruction	Description	Operation
0110011 (51)	000	0000000	R	add rd, rs1, rs2	add	rd = rs1 + rs2

The operation is specified with the opcode, funct3 and funct7 fields of the instructions.
opcode = 7'h38 (51), funct3 = 3'b0, funct7 = 7'b0.
rd: s0 = x8 = 5'b01000, rs1 = s1 = x9 = 5'b01001, rs2 = s2 = x18 = 5'b10010
If we fill in the fields with these values according the diagram here, we get:

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0	1	1	0	0	1	1	0	0	1

Therefore this instruction has a machine code of 32'h01248433.

Similar, for: `addi s0, s1, 6`

op	funct3	funct7	Type	Instruction	Description	Operation
0010011 (19)	000	-	I	addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)

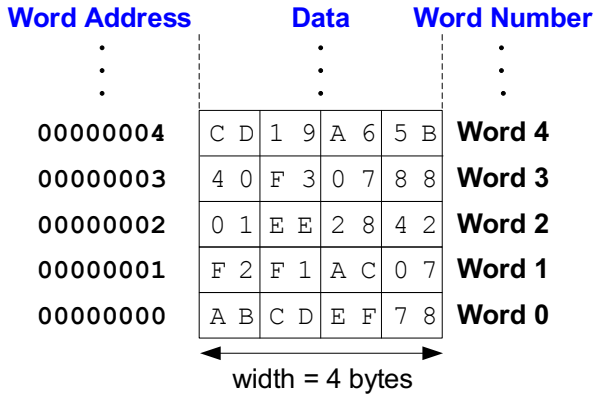
opcode = 7'h13 (19), funct3 = 3'b0.
rd: s0 = x8 = 5'b01000, rs1 = s1 = x9 = 5'b01001 as before.
Imm_{12} = 12'h6.

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
	0	0	0	0	0	0	0	0	0	0	0	6	0	1	0	0	0	0	0	0	1	0	0	1	1	0	0	1	0	0	1	1

Therefore this instruction has a machine code of 32’h00648413.

RISC-V operands from memory

- Each 32-bit data word has a unique address



RISC-V uses **byte-addressable** memory (i.e. byte has a unique address), so each 32-bit word uses 4 byte addresses

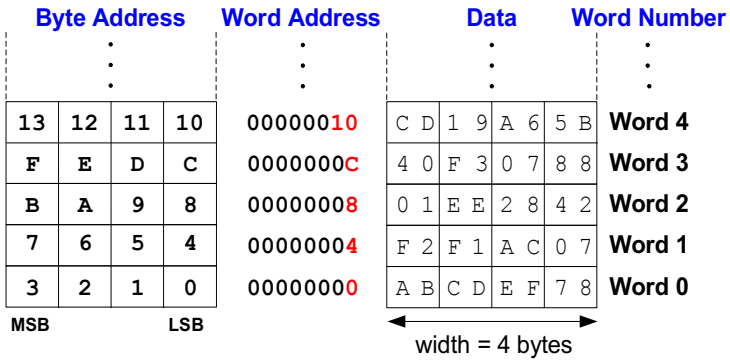
H&H p307

32-bit operands in memory occupies 4 bytes. Some processor uses one unique address for each 32-bit words. MIPS processor is one such example. Everything instruction or data word has a unique address – it is "word addressable" processor.

RISC-V uses byte-addressable to access memory, where EVERY BYTE has a unique address.

RISC-V Byte-addressable Memory

- Each data byte has a unique address
- Load/store words or single bytes: load byte (lb) and store byte (sb)
- 32-bit word = 4 bytes, so word address **increments by 4**



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Therefore, in RISC-V, every 32-bit occupies four unique addresses. If the least significant byte has an address of base = 4, then the most significant byte has an address of base + 3 = 7 as shown above.

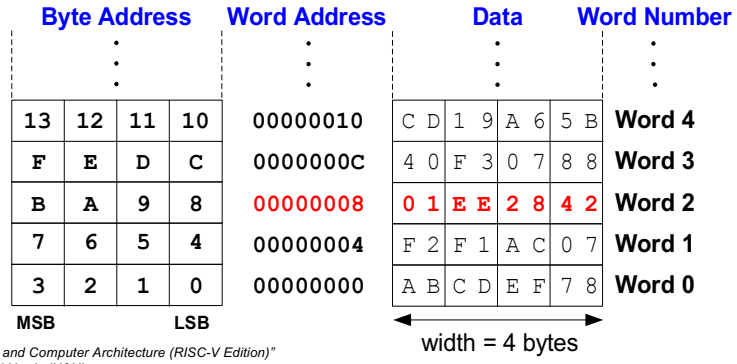
Since all RISC-V instructions are 32-bit, addresses of the instruction memory are all aligned to an increment of 4.

Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 8 into s3.
- s3 holds the value 0x1EE2842 after load

RISC-V assembly code

```
lw s3, 8(zero)  # read word at address 8 into s3
```



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

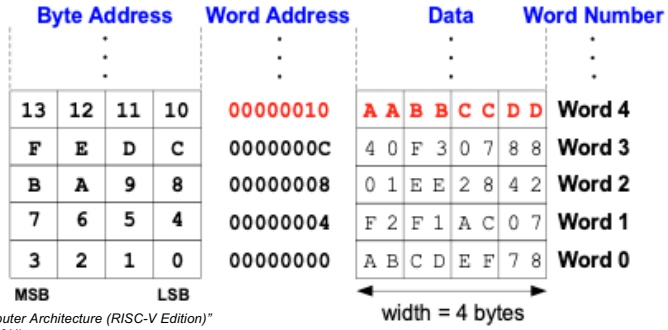
This shows an example of how reading from data memory into register s3 at data memory address 32'h8.

Writing Byte-Addressable Memory

- **Example:** store the value held in t7 into memory address 0x10 (16)
  - if t7 holds the value 0xAABBCCDD, then after the sw completes, word 4 (at address 0x10) in memory will contain that value

RISC-V assembly code

```
SW t7, 0x10(zero)  # write t7 into address 16
```



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

This is an example of storing from register t7 to memory address 32'h10.

RISC-V: Operands from Constants

- 12-bit signed constants (immediates) using addi:

C Code

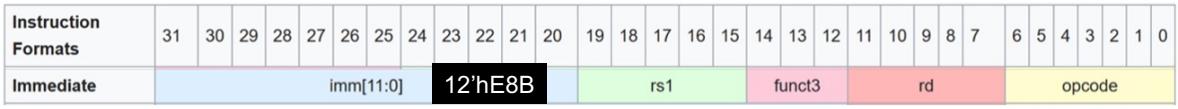
```
// int is a 32-bit signed word
int a = -372;
int b = a + 6;
```

RISC-V assembly code

```
# s0 = a, s1 = b
addi s0, zero, -372
addi s1, s0, 6
```

372 = 12'h174 = 12'b0001_0111_0100
-372 = 12'b1110_1000_1100 = 12'hE8B

- Form 32-bit constant using **sign extension**



Any immediate that needs **more than 12 bits** cannot use this method.

H&H p306

An operand can also be a constant encoded within the instruction itself. Here comes a problem: since RISC-V instructions are all **single** 32-bit words, and an operand is also 32-bit wide, how can an immediate constant operand be embedded in a 32-bit instruction?

If the constant operand has a value of -2048 to -2047 (12'hFFF to 12'h8FF), the operand can be fully specified with a 12-bit binary number in 2's complement form. As it turns out, most constants in computer programs are small. For example, to refer to an offset index of an array, the index often falls within this range of numbers.

In RV32I, I-type instructions have 12 bits reserved for such a constant operand as shown in the slide here. The constant is always **sign exterended** before being used as an operand.

RISC-V: Operand with 32-bit Constants

- Use load upper immediate (lui) and addi
- lui: puts an immediate in the upper 20 bits of destination register and 0's in lower 12 bits

C Code

```
int a = 0xFEDC8765;
```

RISC-V assembly code

```
# s0 = a
lui  s0, 0xFEDC8
addi s0, s0, 0x765
```

Remember that addi **sign-extends** its 12-bit immediate constant

H&H p306

Using a 12-bit immediate constant works most of the time. However, there are times when a program requires to load a register (say) with a 32-bit constant value.

In RV32I, this is achieved by splitting the constants into two parts – the upper 20 bit, which can be loaded into a register, using the instruction "load upper immediate" `lui`.

For example, the instruction: `lui  s0, 0xFEDC8` load into s0 the value 32'hFEDC8000. This is then added to the bottom 12 bits of the constant with the "add immediate" `addi` instruction:

```
addi  s0, s0 0x765.
```

This works perfectly if the MSB of the 12-bit immediate operand is 0. Unfortunately, if the MSB of the 12-bit constant (i.e. bit 11) is a 1, the constant is then sign extended. When added to the upper 20-bits previously loaded value in s0, the answer will be wrong because the upper 20-bit will be modified. This is because in 2's complement representation, a 20-bit value of 20'hFFFFF is equivalent to -1. Therefore the upper 20-bit, after the addi instruction with be 1 lower than what it should be.

RISC-V: 32-bit Constants (bit 11 is 1)

- If **bit 11** of the constant is **1**, increment upper 20 bits by **1** in `lui`

C Code

```
int a = 0xFEDC8EAB;
```

Note: -341 = 0xEAB

RISC-V assembly code

```
# s0 = a
lui  s0, 0xFEDC9     # s0 = 0xFEDC9000
addi s0, s0, -341    # s0 = 0xFEDC9000 + 0xFFFFFEAB
                     #    = 0xFEDC8EAB
```

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Therefore, if bit 11 of the 32-bit constant is 1, we load the upper 20-bit with a constant that is 1 larger than the constant.

In this example, the constant is 32'hFEDC8EAB. Bit 11 is 1. Upper 20-bit is 20'hFEDC8, and lower 12-bit is 12'hEAB, which is -341 in 2's complement representation after sign extension.

We first load s0 with 0xFEDC9 (1 larger than the upper value). After the addi instruction, s0 will have the correct 32-bit constant value.

Fortunately the assembly and compiler for RISC-V take care of this automatically.

RISC-V: Psuedoinstruction

- Load immediate 32-bit word is tedious.
- Pseudoinstruction – Assembler program translate "Load Immediate" instruction "li" to two real RISC-V instructions: "lui" and "addi"

C Code

```
int a = 0xFEDC8EAB;
```

Note: -341 = 0xEAB

RISC-V pseudoinstructions

```
# s0 = a
li  s0, 0xFEDC8EAB
```

RISC-V real instructions

```
# s0 = a
lui  s0, 0xFEDC9
addi s0, s0, 0xEAB
```

- RISC-V has many pseudoinstructions (see later lectures)

RISC-V has many instructions missing deliberately to make is small and fast. More complex operations are accomplished by multiple instructions or by an instruction that result in the same operation.

For example there is no instruction to load a register with a constant value. To load s0 with the small constant 6, we use the instruction:

```
addi  s0, zero, 6
```

To load s0 with a large constant 0xFEDC8EAB, we use the two instructions:

```
lui   s0, 0xFEDC9
addi  s0, s0, 0xEAB
```

This makes the assembly language program of RISC-V much harder to read and understand. Fortunately, RISC-V assembler understand a number of pseudo instructions. These instructions do not exist in the RISC-V ISA, but are translated into equivalent RV32I instructions.

To load a register with a constant of any size constant (up to 32 bits), one can use the "load immediate" `li` pseudoinstruction.

```
li    s0, 6
li    s0, 0xFEDC9
```

Slide 29 shows all the pseudo instructions that RISC-V assembler accepts.

RISC-V: Addressing Modes

How do we address the operands?

- Register Only
- Immediate
- Base Addressing
- PC-Relative

Register Only

- Operands found in registers
  - **Example:** add s0, t2, t3
  - **Example:** sub t6, s1, 0

Immediate

- 12-bit signed immediate used as an operand
  - **Example:** addi s4, t5, -73
  - **Example:** ori t3, t7, 0xFF

H&H p340

Specifying where the operand comes from is called “addressing modes” of an ISA. We have already discussed the two of the four addressing modes found in RISC-V ISA: Register addressing and Immediate addressing. We will now consider the remain two other addressing modes: Base addressing (with offset) and Program Counter Relative addressing.

RISC-V: Base + Offset Addressing

Base Addressing

- Loads and Stores
- Address of operand is:

  base address + immediate

  – **Example:** lw s4, 72(zero)
    - address = 0 + 72

  – **Example:** sw t2, -25(t1)
    - address = t1 - 25

Base addressing mode uses one of the registers content as the address into memory. What stored in the register is not the actual operand, but it stores the address of the operand. In C++, we call this a **pointer** - it points to the place where the operand is stored.

In RISC-V, Base addressing is always used with an offset value which must be a 12-bit 2's complement immediate constant. The "load" and "store" instrutions use this mode of addressing.

RISC-V: PC-relative Addressing

PC-Relative Addressing: branches and jal

Example:

Address		**Instruction**
0x354	L1:	addi s1, s1, 1
0x358		sub t0, t1, s7
...		...
0xEB0		bne s8, s9, L1

The label is (0xEB0-0x354) = 0xB5C (**2908**) instructions **before** bne

The final addressing mode is the Program Counter, or PC-relative addressing. The operand is derived from the PC value by adding a 13-bit (not 12-bit) 2's complement offset. This type of addressing is ONLY used by the branch and jump instructions.

For example, the above "branch if not equal" instruction compares s8 and s9 contents. If they are NOT the same, then the PC counter is load with the address of L1, which is 0x354.

How is the value 0x354 encoded in the instruction? The immediate constant is calculated with the value of PC for the bne instruction, which ix 0xEB0. The offset is calculated by 0xEB0 – 0x354 = 0xB5C. Therefore the stored immediate value is therefore the value -2908.

RISC-V: Instruction coding for Branch offset

Assembly

```
beq s8,  s9,  L1
(beq x24, x25, L1)
```

Relative offset = -2908

$imm_{12:0}$ = -2908 1 0 1 0 0 1 0 1 0 0 1 0 0

bit number 12 11 10 9 8 7 6 5 4 3 2 1 0

Instruction Formats	31	30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8	7	6 5 4 3 2 1 0
Branch	[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	[11]	opcode

Field Values

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
1100 101	24	25	1	0010 0	99
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Machine Code

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	
1100 101	11000	11001	001	0010 0	110 0011	(0xCB8C9263)
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

The way that RISC-V encodes the relative offset of -2908 is complicated and appears illogical. In fact the design decision for this instruction is very clever and is aimed at making the hardware implementation as simple as possible.

Here are the design constraints that determine how the instruction is encoded:

1. It uses the same fields for opcode (7 bits), funct3 (3 bits), rs1 and rs2 (5 bits) as other instructions. This means that 20 bits of the 32-bit instructions are already used. So there are 12 bits left for encoding the offset.
2. Since the branch destination is ALWAYS an instruction address, and that RISC-V uses byte-addressable memory, the instructions for RV32I is ALWAYS aligned to 4. In other words, there is no need to store the bottom 2 bits of the offset – they are always zero. However, there is a variant of RISC-V ISA which targets microcontroller, where the among of program memory is limited. The "Compressed" extension of RISC-V ISA includes 16-bit instructions (i.e. packing two instructions into a 32-bit word). Therefore, the instruction address can be an increment of 2 instead of 4, meaning that only bit 0 is always 0.
3. It is convenient in hardware that the bits used for encoding B-type immediate values should be similar to that used for I-type and S-type instructions. Therefore the locations of bits are the same for imm[4:1], imm[10:5]. However, the branch immediate is 13 bits instead of 12 bits, therefore imm[12] now takes the place of imm[11] in other case. They are both sign bits.
4. Since imm[0] is always 0, there is no need to store it. Instead imm[11] is

stored here!

R-type Instructions: 3 register instructions

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						

op	funct3	funct7	Type	Instruction	Description	Operation
0110011 (51)	000	0000000	R	add rd, rs1, rs2	add	rd = rs1 + rs2
0110011 (51)	000	0100000	R	sub rd, rs1, rs2	sub	rd = rs1 − rs2
0110011 (51)	001	0000000	R	sll rd, rs1, rs2	shift left logical	rd = rs1 << $rs2_{4:0}$
0110011 (51)	010	0000000	R	slt rd, rs1, rs2	set less than	rd = (rs1 < rs2)
0110011 (51)	011	0000000	R	sltu rd, rs1, rs2	set less than unsigned	rd = (rs1 < rs2)
0110011 (51)	100	0000000	R	xor rd, rs1, rs2	xor	rd = rs1 ^ rs2
0110011 (51)	101	0000000	R	srl rd, rs1, rs2	shift right logical	rd = rs1 >> $rs2_{4:0}$
0110011 (51)	101	0100000	R	sra rd, rs1, rs2	shift right arithmetic	rd = rs1 >>> $rs2_{4:0}$
0110011 (51)	110	0000000	R	or rd, rs1, rs2	or	rd = rs1 | rs2
0110011 (51)	111	0000000	R	and rd, rs1, rs2	and	rd = rs1 & rs2

This an the next few slides are summary of ALL the 40 instructions in RISC-V RV32I ISA.

Here is the R-type instructions that perform arithmetic and logical operations using three registers. They all share the opcode of 51 decimal (or 0x33). The funct3 and funct7 fields defines the specific operation.

I & S-type Instructions: All involve imm constants

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address]$_{7:0}$)
0000011 (3)	001	–	I	lh rd, imm(rs1)	load half	rd = SignExt([Address]$_{15:0}$)
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word	rd = [Address]$_{31:0}$
0000011 (3)	100	–	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
0000011 (3)	101	–	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)
0010011 (19)	000	–	I	addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)
0010011 (19)	001	0000000*	I	slli rd, rs1, uimm	shift left logical immediate	rd = rs1 << uimm
0010011 (19)	010	–	I	slti rd, rs1, imm	set less than immediate	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	sltiu rd, rs1, imm	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	xori rd, rs1, imm	xor immediate	rd = rs1 ^ SignExt(imm)
0010011 (19)	101	0000000*	I	srli rd, rs1, uimm	shift right logical immediate	rd = rs1 >> uimm
0010011 (19)	101	0100000*	I	srai rd, rs1, uimm	shift right arithmetic imm.	rd = rs1 >>> uimm
0010011 (19)	110	–	I	ori rd, rs1, imm	or immediate	rd = rs1 | SignExt(imm)
0010011 (19)	111	–	I	andi rd, rs1, imm	and immediate	rd = rs1 & SignExt(imm)
0100011 (35)	000	–	S	sb rs2, imm(rs1)	store byte	[Address]$_{7:0}$ = rs2$_{7:0}$
0100011 (35)	001	–	S	sh rs2, imm(rs1)	store half	[Address]$_{15:0}$ = rs2$_{15:0}$
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word	[Address]$_{31:0}$ = rs2

This group includes two instruction types which both require TWO register operands and one 12-bit immediate operands.

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						

The I-type instructions specify either a load instruction or a ALU instructions. Here we specify a destination register rd to store the result of an memory read of the ALU operation, and a source register rs1 to specify an operand for the ALU operation or the address of the data to fetch.

Two opcodes are used for I-type instructions: 3 for load instructions and 19 for ALU immediate instruction. Note that some I-type instructions (shift instructions) do not use sign-extension to the immediate values.

The S-type instructions does not require a destination register because the destination is data memory. However they require two source registers, one contain the value to write to memory, and a second has the base address of the destination. The 12-bit immediate offset is split into two parts, using the funct7 field of instr[31:25] and the rd field of instr[11:7], combined to form imm[11:0].

B-type Instructions: PC-relative Branches

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						

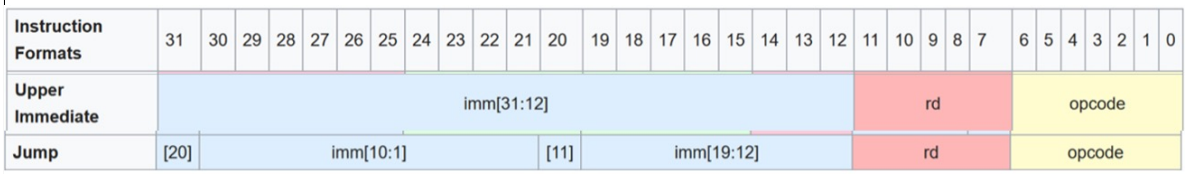
op	funct3	funct7	Type	Instruction	Description	Operation
1100011 (99)	000	–	B	beq rs1, rs2, label	branch if =	if (rs1 == rs2) PC = BTA
1100011 (99)	001	–	B	bne rs1, rs2, label	branch if ≠	if (rs1 ≠ rs2) PC = BTA
1100011 (99)	100	–	B	blt rs1, rs2, label	branch if <	if (rs1 < rs2) PC = BTA
1100011 (99)	101	–	B	bge rs1, rs2, label	branch if ≥	if (rs1 ≥ rs2) PC = BTA
1100011 (99)	110	–	B	bltu rs1, rs2, label	branch if < unsigned	if (rs1 < rs2) PC = BTA
1100011 (99)	111	–	B	bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA

H&H p311

We have discussed the encoding of branch instructions in details in slide 19 & 20. Note that the opcode for B-type instructions is 99 or 0x63. funct3 defines the conditions under which branch takes place.

When implementing B-type instruction in hardware, one could use the ALU to perform the comparison, or create special branch unit which provides performs ONLY the comparison and no other operations and generates all the required conditions. The second option makes the design cleaner.

U & I -type Instructions: Upper & Jump/Link



Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Upper Immediate	imm[31:12]																				rd					opcode							
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode							

op	funct3	funct7	Type	Instruction	Description	Operation
0010111 (23)	–	–	U	`auipc rd, upimm`	add upper immediate to PC	`rd = {upimm, 12'b0} + PC`
0110111 (55)	–	–	U	`lui rd, upimm`	load upper immediate	`rd = {upimm, 12'b0}`
1100111 (103)	000	–	I	`jalr rd, rs1, imm`	jump and link register	`PC = rs1 + SignExt(imm), rd = PC + 4`
1101111 (111)	–	–	J	`jal rd, label`	jump and link	`PC = JTA, rd = PC + 4`

- We will discuss auipc, jalr and jal instructions in another lecture

Finally there are four special instructions that are not in the other category. We have already discussed the lui instruction previously.

The U-type instructions are used to manipulate the upper 20-bit of a register to handle 32-bit immediate constants.

The J-type instructions are for function or subroutine calls. They will be discussed in a later lecture.

RISC-V Arithmetic instructions

Mnemonic	Instruction	Type	Description
ADD rd, rs1, rs2	Add	R	rd ← rs1 + rs2
SUB rd, rs1, rs2	Subtract	R	rd ← rs1 - rs2
ADDI rd, rs1, imm12	Add immediate	I	rd ← rs1 + imm12
SLT rd, rs1, rs2	Set less than	R	rd ← rs1 < rs2 ? 1 : 0
SLTI rd, rs1, imm12	Set less than immediate	I	rd ← rs1 < imm12 ? 1 : 0
SLTU rd, rs1, rs2	Set less than unsigned	R	rd ← rs1 < rs2 ? 1 : 0
SLTIU rd, rs1, imm12	Set less than immediate unsigned	I	rd ← rs1 < imm12 ? 1 : 0
LUI rd, imm20	Load upper immediate	U	rd ← imm20 << 12
AUIP rd, imm20	Add upper immediate to PC	U	rd ← PC + imm20 << 12

The next few slides provide a catalogue of all the RISC-V RV32I instructions in various groups.

All these instructions involve arithmetic operation.

RISC-V Logic instructions

Mnemonic	Instruction	Type	Description
AND rd, rs1, rs2	AND	R	rd ← rs1 & rs2
OR rd, rs1, rs2	OR	R	rd ← rs1 | rs2
XOR rd, rs1, rs2	XOR	R	rd ← rs1 ^ rs2
ANDI rd, rs1, imm12	AND immediate	I	rd ← rs1 & imm12
ORI rd, rs1, imm12	OR immediate	I	rd ← rs1 | imm12
XORI rd, rs1, imm12	XOR immediate	I	rd ← rs1 ^ imm12
SLL rd, rs1, rs2	Shift left logical	R	rd ← rs1 << rs2
SRL rd, rs1, rs2	Shift right logical	R	rd ← rs1 >> rs2
SRA rd, rs1, rs2	Shift right arithmetic	R	rd ← rs1 >> rs2
SLLI rd, rs1, shamt	Shift left logical immediate	I	rd ← rs1 << shamt
SRLI rd, rs1, shamt	Shift right logical imm.	I	rd ← rs1 >> shamt
SRAI rd, rs1, shamt	Shift right arithmetic immediate	I	rd ← rs1 >> shamt

These instructions perform logical operations.

RISC-V Load/Store instructions

Mnemonic	Instruction	Type	Description
LW rd, imm12(rs1)	Load word	I	rd ← mem[rs1 + imm12]
LH rd, imm12(rs1)	Load halfword	I	rd ← mem[rs1 + imm12]
LB rd, imm12(rs1)	Load byte	I	rd ← mem[rs1 + imm12]
LWU rd, imm12(rs1)	Load word unsigned	I	rd ← mem[rs1 + imm12]
LHU rd, imm12(rs1)	Load halfword unsigned	I	rd ← mem[rs1 + imm12]
LBU rd, imm12(rs1)	Load byte unsigned	I	rd ← mem[rs1 + imm12]
SW rs2, imm12(rs1)	Store word	S	rs2(31:0) → mem[rs1 + imm12]
SH rs2, imm12(rs1)	Store halfword	S	rs2(15:0) → mem[rs1 + imm12]
SB rs2, imm12(rs1)	Store byte	S	rs2(7:0) → em[rs1 + imm12]

These instructions perform data memory read and write operations using pointer address in register and an immediate offset.

RISC-V Branch & Jump instructions

Mnemonic	Instruction	Type	Description
`BEQ rs1, rs2, imm12`	Branch equal	SB	`if rs1 == rs2 pc ← pc + imm12`
`BNE rs1, rs2, imm12`	Branch not equal	SB	`if rs1 != rs2 pc ← pc + imm12`
`BGE rs1, rs2, imm12`	Branch greater than or equal	SB	`if rs1 >= rs2 pc ← pc + imm12`
`BGEU rs1, rs2, imm12`	Branch greater than or equal unsigned	SB	`if rs1 >= rs2 pc ← pc + imm12`
`BLT rs1, rs2, imm12`	Branch less than	SB	`if rs1 < rs2 pc ← pc + imm12`
`BLTU rs1, rs2, imm12`	Branch less than unsigned	SB	`if rs1 < rs2 pc ← pc + imm12 << 1`
`JAL rd, imm20`	Jump and link	UJ	`rd ← pc + 4` `pc ← pc + imm20`
`JALR rd, imm12(rs1)`	Jump and link register	I	`rd ← pc + 4` `pc ← rs1 + imm12`

These are the branch and jump instructions involving offset to the Program Counter.

RISC-V Psuedoinstructions

Mnemonic	Instruction	Base instruction(s)
`LI rd, imm12`	Load immediate (near)	`ADDI rd, zero, imm12`
`LI rd, imm`	Load immediate (far)	`LUI rd, imm[31:12]` `ADDI rd, rd, imm[11:0]`
`LA rd, sym`	Load address (far)	`AUIPC rd, sym[31:12]` `ADDI rd, rd, sym[11:0]`
`MV rd, rs`	Copy register	`ADDI rd, rs, 0`
`NOT rd, rs`	One's complement	`XORI rd, rs, -1`
`NEG rd, rs`	Two's complement	`SUB rd, zero, rs`
`BGT rs1, rs2, offset`	Branch if rs1 > rs2	`BLT rs2, rs1, offset`
`BLE rs1, rs2, offset`	Branch if rs1 ≤ rs2	`BGE rs2, rs1, offset`
`BGTU rs1, rs2, offset`	Branch if rs1 > rs2 (unsigned)	`BLTU rs2, rs1, offset`
`BLEU rs1, rs2, offset`	Branch if rs1 ≤ rs2 (unsigned)	`BGEU rs2, rs1, offset`

Mnemonic	Instruction	Base instruction(s)
`BEQZ rs1, offset`	Branch if rs1 = 0	`BEQ rs1, zero, offset`
`BNEZ rs1, offset`	Branch if rs1 ≠ 0	`BNE rs1, zero, offset`
`BGEZ rs1, offset`	Branch if rs1 ≥ 0	`BGE rs1, zero, offset`
`BLEZ rs1, offset`	Branch if rs1 ≤ 0	`BGE zero, rs1, offset`
`BGTZ rs1, offset`	Branch if rs1 > 0	`BLT zero, rs1, offset`
`J offset`	Unconditional jump	`JAL zero, offset`
`CALL offset12`	Call subroutine (near)	`JALR ra, ra, offset12`
`CALL offset`	Call subroutine (far)	`AUIPC ra, offset[31:12]` `JALR ra, ra, offset[11:0]`
`RET`	Return from subroutine	`JALR zero, 0(ra)`
`NOP`	No operation	`ADDI zero, zero, 0`

These are all the pseudo instructions accepted by the RISC-V assembler but are not really RISC-V instructions in the ISA. They are translated by the RISC-V assembler to one or more RISC-V instructions to make the program more readable.

Lecture 4:
Tomasulo Algorithm and Dynamic Branch Prediction

Professor David A. Patterson

Computer Science 252

Fall 1996

Review: Summary

- Instruction Level Parallelism (ILP) in SW or HW
- Loop level parallelism is easiest to see
- SW parallelism dependencies defined for program, hazards if HW cannot resolve
- SW dependencies/compiler sophistication determine if compiler can unroll loops
  - Memory dependencies hardest to determine
- HW exploiting ILP
  - Works when can't know dependence at run time
  - Code for one machine runs well on another
- Key idea of Scoreboard: Allow instructions behind stall to proceed (Decode => Issue instr & read operands)
  - Enables out-of-order execution => out-of-order completion
  - ID stage checked both for structural & data dependencies

Review: Three Parts of the Scoreboard

1. **Instruction status—which of 4 steps the instruction is in**

2. **Functional unit status—Indicates the state of the functional unit (FU). 9 fields for each functional unit**
   - **Busy—Indicates whether the unit is busy or not**
   - **Op—Operation to perform in the unit (e.g., + or –)**
   - **Fi—Destination register**
   - **Fj, Fk—Source-register numbers**
   - **Qj, Qk—Functional units producing source registers Fj, Fk**
   - **Rj, Rk—Flags indicating when Fj, Fk are ready**

3. **Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions will write that register**

Review: Scoreboard Example Cycle 3

Instruction status

Instruction		j	k	Issue	Read operand	Execution complete	Write Result
LD	F6	34+	R2	1	2	3	
LD	F2	45+	R3				
MULT	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for Qj	FU for Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU				Integer					

- **Issue MULT? No, stall on structural hazard**

Review: Scoreboard Example Cycle 9

Instruction status

Instruction	j	k	Issue	Read operanc	Executi comple	Write Result
LD F6	34+	R2	1	2	3	4
LD F2	45+	R3	5	6	7	8
MULT F0	F2	F4	6	9		
SUBD F8	F6	F2	7	9		
DIVD F10	F0	F6	8			
ADDD F6	F8	F2				

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	No								
10	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
2	Add	Yes	Sub	F8	F6	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
9	FU	Mult1				Add	Divide			

- **Read operands for MULT & SUBD? Issue ADDD?**

Review: Scoreboard Example Cycle 17

Instruction status

Instruction		j	k	Issue	Read operanc	Executi complei	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for Qj	FU for k Qk	Fj? Rj	Fk? Rk
	Integer	No								
2	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
17	FU	Mult1			Add		Divide			

- **Write result of ADDD? No, WAR hazard**

Review: Scoreboard Example Cycle 62

Instruction status

Instruction		j	k	Issue	Read operand	Execution complete	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	62
ADDD	F6	F8	F2	13	14	16	22

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for Qj	FU for Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
0	Divide	No								

Register result status

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
62	FU								

- **In-order issue; out-of-order execute & commit**

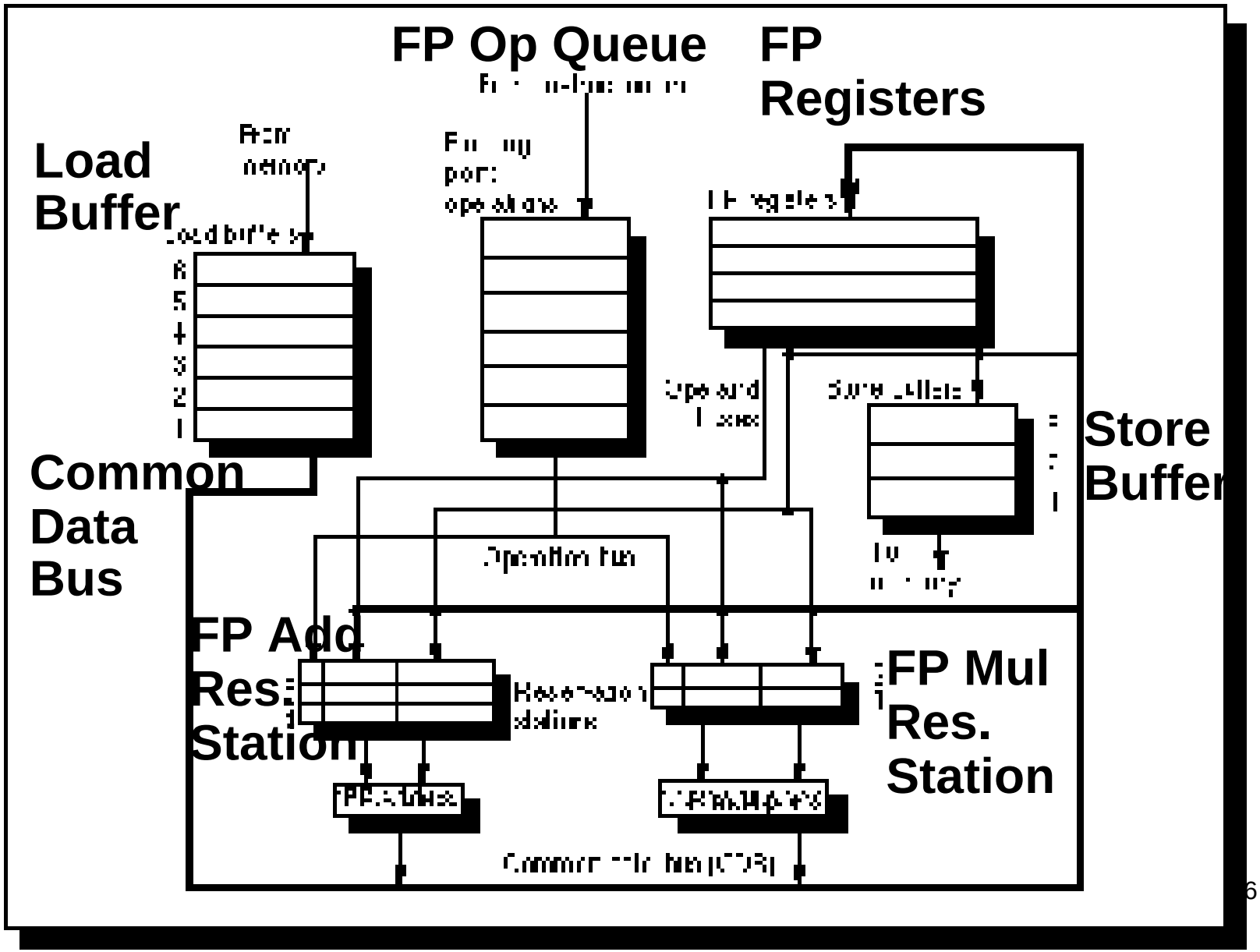
Review: Scoreboard Summary

- **Speedup 1.7 from compiler; 2.5 by hand BUT slow memory (no cache)**
- **Limitations of 6600 scoreboard**
  - **No forwarding (First write regsiter then read it)**
  - **Limited to instructions in basic block (small *window*)**
  - **Number of functional units(structural hazards)**
  - **Wait for WAR hazards**
  - **Prevent WAW hazards**

Another Dynamic Algorithm: Tomasulo Algorithm

- For IBM 360/91 about 3 years after CDC 6600 (1966)
- Goal: High Performance without special compilers
- Differences between IBM 360 & CDC 6600 ISA
  - IBM has only 2 register specifiers/instr vs. 3 in CDC 6600
  - IBM has 4 FP registers vs. 8 in CDC 6600
- Differences between Tomasulo Algorithm & Scoreboard
  - Control & buffers *distributed* with Function Units vs. centralized in scoreboard; called "reservation stations"
  - Registers in instructions replaced by pointers to reservation station buffer
  - HW *renaming* of registers to avoid WAR, WAW hazards
  - Common Data Bus broadcasts results to all FUs
  - Load and Stores treated as FUs as well
- Why Study? It lead to Pentium Pro, PowerPC 604, …

Tomasulo Organization



Reservation Station Components

Op—Operation to perform in the unit (e.g., + or –)

Qj, Qk—Reservation stations producing source registers (value to be written)

Vj, Vk—Value of Source operands

Rj, Rk—Flags indicating when Vj, Vk are ready

Busy—Indicates reservation station and FU is busy

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Three Stages of Tomasulo Algorithm

1. **Issue—get instruction from FP Op Queue**

   If reservation station free (no structural hazard), the scoreboard issues instr & sends operands (renames registers).

2. **Execution—operate on operands (EX)**

   When both operands ready then execute;
   if not ready, watch CDB for result

3. **Write result—finish execution (WB)**

   Write on Common Data Bus to all awaiting units;
   mark reservation station available

- **Normal bus: data + destination**
  Common Data Bus: data + *source*:
  Normal = "Go To" bus; CDB = "Come From" bus

Tomasulo Example Cycle 0

Instruction status												
Instruction		*j*	*k*	*Issue*	*Execution complete*	*Write Result*			Busy	Address		
LD	F6	34+	R2					Load1	No			
LD	F2	45+	R3					Load2	No			
MULT	F0	F2	F4					Load3	No			
SUBD	F8	F6	F2									
DIVD	F10	F0	F6									
ADDD	F6	F8	F2									

Reservation Stations												
	Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS for j Qj*	*RS for k Qk*				
	0	Add1	No									
	0	Add2	No									
	0	Add3	No									
	0	Mult1	No									
	0	Mult2	No									

Register result status												
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
0			*FU*									

CS 252 Administrivia

- **Reading Assignments for Lectures 3 to 6**
  - **Computer Architecture: A Quantitative Approach, Second Edition (1996)**
  - **Chapter 4, Appendix B**
- **Exercises for Lectures 3 to 6**
  - **4.14, all parts (a - k)**
  - **4.25, table of pros and cons + short essay**
  - **B.3, all parts (a - g)**
  - **B.15, table of pros and cons + short essay**
  - **Due Monday September 16 at 5PM homework box in 283 Soda (building is locked at 6:45 PM)**
  - **Done in pairs, but both need to understand whole assignment**
  - **Study groups encouraged, but pairs do own work**

Tomasulo Example Cycle 1

Instruction status

Instruction		j	k	Issue	Execution complete	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULT	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	No	
Load3	No	

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk
0	Add1	No					
0	Add2	No					
	Add3	No					
0	Mult1	No					
0	Mult2	No					

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
1	FU				Load1					

Tomasulo Example Cycle 2

Instruction status

Instruction		j	k	Issue	Execution complete	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3	2		
MULT	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk
0	Add1	No					
0	Add2	No					
	Add3	No					
0	Mult1	No					
0	Mult2	No					

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
2	FU		Load2		Load1					

Tomasulo Example Cycle 3

Instruction status

Instruction		j	k	Issue	Execution complete	Write Result
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULT	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk
0	Add1	No					
0	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD		R(F4)	Load2	
0	Mult2	No					

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU	Mult1	Load2		Load1					

- **Issue MULT now vs in scoreboard?**

Tomasulo Example Cycle 4

Instruction status					Execution	Write					
Instruction		j	k	Issue	complete	Result			Busy	Address	
LD	F6	34+	R2	1	3	4		Load1	No		
LD	F2	45+	R3	2	4			Load2	Yes	45+R3	
MULT	F0	F2	F4	3				Load3	No		
SUBD	F8	F6	F2	4							
DIVD	F10	F0	F6								
ADDD	F6	F8	F2								

Reservation Stations					S1	S2	RS for j	RS for k				
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk				
	0	Add1	Yes	SUBD	M(34+R2)			Load2				
	0	Add2	No									
		Add3	No									
	0	Mult1	Yes	MULTD		R(F4)	Load2					
	0	Mult2	No									

Register result status											
Clock			F0	F2	F4	F6	F8	F10	F12	...	F30
4		FU	Mult1	Load2		M(34+R2)	Add1				

Tomasulo Example Cycle 5

Instruction status												
Instruction		*j*	*k*	*Issue*	*Execution complete*	*Write Result*			Busy	Address		
LD	F6	34+	R2	1	3	4		Load1	No			
LD	F2	45+	R3	2	4	5		Load2	No			
MULT	F0	F2	F4	3				Load3	No			
SUBD	F8	F6	F2	4								
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2									

Reservation Stations												
	Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS for j Qj*	*RS for k Qk*				
	2	Add1	Yes	SUBD	M(34+R2)	M(45+R3)						
	0	Add2	No									
		Add3	No									
	10	Mult1	Yes	MULTD	M(45+R3)	R(F4)						
	0	Mult2	Yes	DIVD		M(34+R2)	Mult1					

Register result status												
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
5			*FU*	Mult1	M(45+R3)		M(34+R2)	Add1	Mult2			

Tomasulo Example Cycle 6

Instruction status												
Instruction		*j*	*k*	*Issue*	*Execution complete*	*Write Result*			Busy	Address		
LD	F6	34+	R2	1	3	4		Load1	No			
LD	F2	45+	R3	2	4	5		Load2	No			
MULT	F0	F2	F4	3				Load3	No			
SUBD	F8	F6	F2	4								
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6								

Reservation Stations					*S1*	*S2*	*RS for j*	*RS for k*				
	Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*				
	1	Add1	Yes	SUBD	M(34+R2)	M(45+R3)						
	0	Add2	Yes	ADDD		M(45+R3)	Add1					
		Add3	No									
	9	Mult1	Yes	MULTD	M(45+R3)	R(F4)						
	0	Mult2	Yes	DIVD		M(34+R2)	Mult1					

Register result status												
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
6			*FU*	Mult1	M(45+R3)		Add2	Add1	Mult2			

- **Issue MULT vs. scoreboard?**

Tomasulo Example Cycle 7

Instruction status												
Instruction		*j*	*k*	*Issue*	*Execution complete*	*Write Result*			Busy	Address		
LD	F6	34+	R2	1	3	4		Load1	No			
LD	F2	45+	R3	2	4	5		Load2	No			
MULT	F0	F2	F4	3				Load3	No			
SUBD	F8	F6	F2	4	7							
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6								

Reservation Stations					*S1*	*S2*	*RS for j*	*RS for k*				
	Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*				
	0	Add1	Yes	SUBD	M(34+R2)	M(45+R3)						
	0	Add2	Yes	ADDD		M(45+R3)	Add1					
		Add3	No									
	8	Mult1	Yes	MULTD	M(45+R3)	R(F4)						
	0	Mult2	Yes	DIVD		M(34+R2)	Mult1					

Register result status												
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
7			*FU*	Mult1	M(45+R3)		Add2	Add1	Mult2			

Tomasulo Example Cycle 8

Instruction status												
Instruction		*j*	*k*	*Issue*	*Execution complete*	*Write Result*			Busy	Address		
LD	F6	34+	R2	1	3	4		Load1	No			
LD	F2	45+	R3	2	4	5		Load2	No			
MULT	F0	F2	F4	3				Load3	No			
SUBD	F8	F6	F2	4	7	8						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6								

Reservation Stations					*S1*	*S2*	*RS for j*	*RS for k*				
	Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*				
	0	Add1	Yes	SUBD	M(34+R2)	M(45+R3)						
	2	Add2	Yes	ADDD	M()-M()	M(45+R3)						
	0	Add3	No									
	7	Mult1	Yes	MULTD	M(45+R3)	R(F4)						
	0	Mult2	Yes	DIVD		M(34+R2)	Mult1					

Register result status												
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
8			*FU*	Mult1	M(45+R3)		Add2	M()-M()	Mult2			

Tomasulo Example Cycle 9

Instruction status												
Instruction		*j*	*k*	*Issue*	*Execution complete*	*Write Result*			Busy	Address		
LD	F6	34+	R2	1	3	4		Load1	No			
LD	F2	45+	R3	2	4	5		Load2	No			
MULT	F0	F2	F4	3				Load3	No			
SUBD	F8	F6	F2	4	7	8						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6								

Reservation Stations												
	Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS for j Qj*	*RS for k Qk*				
	0	Add1	No									
	1	Add2	Yes	ADDD	M()–M()	M(45+R3)						
	0	Add3	No									
	6	Mult1	Yes	MULTD	M(45+R3)	R(F4)						
	0	Mult2	Yes	DIVD		M(34+R2)	Mult1					

Register result status												
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
9			*FU*	Mult1	M(45+R3)		Add2	M()–M()	Mult2			

Tomasulo Example Cycle 10

Instruction status												
Instruction		*j*	*k*	*Issue*	*Execution complete*	*Write Result*			Busy	Address		
LD	F6	34+	R2	1	3	4		Load1	No			
LD	F2	45+	R3	2	4	5		Load2	No			
MULT	F0	F2	F4	3				Load3	No			
SUBD	F8	F6	F2	4	7	8						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6	10							

Reservation Stations					*S1*	*S2*	*RS for j*	*RS for k*				
	Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*				
	0	Add1	No									
	0	Add2	Yes	ADDD	M()–M()	M(45+R3)						
	0	Add3	No									
	5	Mult1	Yes	MULTD	M(45+R3)	R(F4)						
	0	Mult2	Yes	DIVD		M(34+R2)	Mult1					

Register result status												
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
10			*FU*	Mult1	M(45+R3)		Add2	M()–M()	Mult2			

Tomasulo Example Cycle 11

Instruction status												
Instruction		*j*	*k*	*Issue*	*Execution complete*	*Write Result*			Busy	Address		
LD	F6	34+	R2	1	3	4		Load1	No			
LD	F2	45+	R3	2	4	5		Load2	No			
MULT	F0	F2	F4	3				Load3	No			
SUBD	F8	F6	F2	4	7	8						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6	10	11						

Reservation Stations												
	Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS for j Qj*	*RS for k Qk*				
	0	Add1	No									
	1	Add2	No									
	0	Add3	No									
	4	Mult1	Yes	MULTD	M(45+R3)	R(F4)						
	0	Mult2	Yes	DIVD		M(34+R2)	Mult1					

Register result status												
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
11			*FU*	Mult1	M(45+R3)		(M-M)+M()	M()–M()	Mult2			

- **Write result of ADDD vs. scoreboard?**

Tomasulo Example Cycle 12

Instruction status												
Instruction		*j*	*k*	*Issue*	*Execution complete*	*Write Result*			Busy	Address		
LD	F6	34+	R2	1	3	4		Load1	No			
LD	F2	45+	R3	2	4	5		Load2	No			
MULT	F0	F2	F4	3				Load3	No			
SUBD	F8	F6	F2	4	6	7						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6	10	11						

Reservation Stations												
	Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS for j Qj*	*RS for k Qk*				
	0	Add1	No									
	0	Add2	No									
	0	Add3	No									
	3	Mult1	Yes	MULTD	M(45+R3)	R(F4)						
	0	Mult2	Yes	DIVD		M(34+R2)	Mult1					

Register result status												
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
12			*FU*	Mult1	M(45+R3)		(M-M)+M()	M()–M()	Mult2			

Tomasulo Example Cycle 13

Instruction status												
Instruction		*j*	*k*	*Issue*	*Execution complete*	*Write Result*			Busy	Address		
LD	F6	34+	R2	1	3	4		Load1	No			
LD	F2	45+	R3	2	4	5		Load2	No			
MULT	F0	F2	F4	3				Load3	No			
SUBD	F8	F6	F2	4	7	8						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6	10	11						

Reservation Stations					*S1*	*S2*	*RS for j*	*RS for k*				
	Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*				
	0	Add1	No									
	0	Add2	No									
		Add3	No									
	2	Mult1	Yes	MULTD	M(45+R3)	R(F4)						
	0	Mult2	Yes	DIVD		M(34+R2)	Mult1					

Register result status												
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
13			*FU*	Mult1	M(45+R3)		(M–M)+M()	M()–M()	Mult2			

Tomasulo Example Cycle 14

Instruction status												
Instruction		*j*	*k*	*Issue*	*Execution complete*	*Write Result*			Busy	Address		
LD	F6	34+	R2	1	3	4		Load1	No			
LD	F2	45+	R3	2	4	5		Load2	No			
MULT	F0	F2	F4	3				Load3	No			
SUBD	F8	F6	F2	4	7	8						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6	10	11						

Reservation Stations												
	Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS for j Qj*	*RS for k Qk*				
	0	Add1	No									
	0	Add2	No									
	0	Add3	No									
	1	Mult1	Yes	MULTD	M(45+R3)	R(F4)						
	0	Mult2	Yes	DIVD		M(34+R2)	Mult1					

Register result status												
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
14			*FU*	Mult1	M(45+R3)		(M–M)+M()	M()–M()	Mult2			

Tomasulo Example Cycle 15

Instruction status												
Instruction		*j*	*k*	*Issue*	*Execution complete*	*Write Result*			Busy	Address		
LD	F6	34+	R2	1	3	4		Load1	No			
LD	F2	45+	R3	2	4	5		Load2	No			
MULT	F0	F2	F4	3	15			Load3	No			
SUBD	F8	F6	F2	4	7	8						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6	10	11						

Reservation Stations					*S1*	*S2*	*RS for j*	*RS for k*				
	Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*				
	0	Add1	No									
	0	Add2	No									
		Add3	No									
	0	Mult1	Yes	MULTD	M(45+R3)	R(F4)						
	0	Mult2	Yes	DIVD		M(34+R2)	Mult1					

Register result status												
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
15			*FU*	Mult1	M(45+R3)		(M–M)+M()	M()–M()	Mult2			

Tomasulo Example Cycle 16

Instruction status												
Instruction		*j*	*k*	*Issue*	*Execution complete*	*Write Result*			Busy	Address		
LD	F6	34+	R2	1	3	4		Load1	No			
LD	F2	45+	R3	2	4	5		Load2	No			
MULT	F0	F2	F4	3	15	16		Load3	No			
SUBD	F8	F6	F2	4	7	8						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6	10	11						

Reservation Stations												
	Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS for j Qj*	*RS for k Qk*				
	0	Add1	No									
	0	Add2	No									
		Add3	No									
	0	Mult1	No									
	40	Mult2	Yes	DIVD	M*F4	M(34+R2)						

Register result status												
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
16			*FU*	M*F4	M(45+R3)		(M–M)+M()	M()–M()	Mult2			

Tomasulo Example Cycle 55

Instruction status												
Instruction		*j*	*k*	*Issue*	*Execution complete*	*Write Result*			Busy	Address		
LD	F6	34+	R2	1	3	4		Load1	No			
LD	F2	45+	R3	2	4	5		Load2	No			
MULT	F0	F2	F4	3	15	16		Load3	No			
SUBD	F8	F6	F2	4	7	8						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6	10	11						

Reservation Stations												
	Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS for j Qj*	*RS for k Qk*				
	0	Add1	No									
	0	Add2	No									
		Add3	No									
	0	Mult1	No									
	1	Mult2	Yes	DIVD	M*F4	M(34+R2)						

Register result status												
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
55			*FU*	M*F4	M(45+R3)		(M–M)+M()	M()–M()	Mult2			

Tomasulo Example Cycle 56

Instruction status												
Instruction		*j*	*k*	*Issue*	*Execution complete*	*Write Result*			Busy	Address		
LD	F6	34+	R2	1	3	4		Load1	No			
LD	F2	45+	R3	2	4	5		Load2	No			
MULT	F0	F2	F4	3	15	16		Load3	No			
SUBD	F8	F6	F2	4	7	8						
DIVD	F10	F0	F6	5	56							
ADDD	F6	F8	F2	6	10	11						

Reservation Stations												
	Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS for j Qj*	*RS for k Qk*				
	0	Add1	No									
	0	Add2	No									
		Add3	No									
	0	Mult1	No									
	0	Mult2	Yes	DIVD	M*F4	M(34+R2)						

Register result status												
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
56			*FU*	M*F4	M(45+R3)		(M–M)+M()	M()–M()	Mult2			

Tomasulo Example Cycle 57

Instruction status												
Instruction		*j*	*k*	*Issue*	*Execution complete*	*Write Result*			Busy	Address		
LD	F6	34+	R2	1	3	4		Load1	No			
LD	F2	45+	R3	2	4	5		Load2	No			
MULT	F0	F2	F4	3	15	16		Load3	No			
SUBD	F8	F6	F2	4	7	8						
DIVD	F10	F0	F6	5	56	57						
ADDD	F6	F8	F2	6	10	11						

Reservation Stations												
	Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS for j Qj*	*RS for k Qk*				
	0	Add1	No									
	0	Add2	No									
		Add3	No									
	0	Mult1	No									
	0	Mult2	No									

Register result status												
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
57			*FU*	M*F4	M(45+R3)		(M–M)+M()	M()–M()	M*F4/M			

Compare to Scoreboard Cycle 62

Instruction status

Instruction		j	k	Issue	Read operand	Execution complete	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	62
ADDD	F6	F8	F2	13	14	16	22

Functional unit status

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU for Qj	FU for Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
0	Divide	No								

Register result status

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
62	FU								

- **In-order issue; out-of-order execute & commit**

Tomasulo Loop Example

```
Loop: LD      F0    0     R1
      MULTD   F4    F0    F2
      SD      F4    0     R1
      SUBI    R1    R1    #8
      BNEZ    R1    Loop
```

- Multiply takes 4 clocks
- Assume loads talke 8 clocks (cache miss)

Loop Example Cycle 0

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1			
MULT F4	F0	F2	1			
SD F4	0	R1	1			
LD F0	0	R1	2			
MULT F4	F0	F2	2			
SD F4	0	R1	2			

	Busy	Address	Qi
Load1	No		
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	No					
0	Mult2	No					

Code:

```
LD    F0   0  R1
MULT  F4   F0 F2
SD    F4   0  R1
SUBI  R1   R1 #8
BNEZ  R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
0	80	Qi								

5 minute Class Break

- **Lecture Format:**
  - ≈ 1 minute: review last time & motivate this lecture
  - ≈ 20 minute lecture
  - ≈ 3 minutes: discuss class manangement
  - ≈ 25 minutes: lecture
  - 5 minutes: break
  - ≈25 minutes: lecture
  - ≈1 minute: summary of today's important topics

Loop Example Cycle 1

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1		
MULT F4	F0	F2	1			
SD F4	0	R1	1			
LD F0	0	R1	2			
MULT F4	F0	F2	2			
SD F4	0	R1	2			

	Busy	Address	Qi
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	No					
0	Mult2	No					

Code:

```
LD   F0   0  R1
MULT F4   F0 F2
SD   F4   0  R1
SUBI R1   R1 #8
BNEZ R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
1	80	Qi	Load1							

Loop Example Cycle 2

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1		
MULT F4	F0	F2	1	2		
SD F4	0	R1	1			
LD F0	0	R1	2			
MULT F4	F0	F2	2			
SD F4	0	R1	2			

	Busy	Address	Qi
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	No					

Code:

```
LD    F0   0  R1
MULT  F4   F0 F2
SD    F4   0  R1
SUBI  R1   R1 #8
BNEZ  R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
2	80	Qi	Load1		Mult1					

Loop Example Cycle 3

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1		
MULT F4	F0	F2	1	2		
SD F4	0	R1	1	3		
LD F0	0	R1	2			
MULT F4	F0	F2	2			
SD F4	0	R1	2			

	Busy	Address	Qi
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	No					

Code:

```
LD   F0   0  R1
MULT F4   F0 F2
SD   F4   0  R1
SUBI R1   R1 #8
BNEZ R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
3	80	Qi	Load1		Mult1					

Loop Example Cycle 4

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1		
MULT F4	F0	F2	1	2		
SD F4	0	R1	1	3		
LD F0	0	R1	2			
MULT F4	F0	F2	2			
SD F4	0	R1	2			

	Busy	Address	Qi
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	No					

Code:

```
LD   F0   0  R1
MULT F4   F0 F2
SD   F4   0  R1
SUBI R1   R1 #8
BNEZ R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
4	72	Qi	Load1		Mult1					

Loop Example Cycle 5

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1		
MULT F4	F0	F2	1	2		
SD F4	0	R1	1	3		
LD F0	0	R1	2			
MULT F4	F0	F2	2			
SD F4	0	R1	2			

	Busy	Address	Qi
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	No					

Code:

```
LD   F0   0  R1
MULT F4   F0 F2
SD   F4   0  R1
SUBI R1   R1 #8
BNEZ R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
5	72	Qi	Load1		Mult1					

Loop Example Cycle 6

Instruction status

Instruction		j	k	iteration	Issue	Execution complete	Write Result
LD	F0	0	R1	1	1		
MULT	F4	F0	F2	1	2		
SD	F4	0	R1	1	3		
LD	F0	0	R1	2	6		
MULT	F4	F0	F2	2			
SD	F4	0	R1	2			

Load1	Yes	80	
Load2	Yes	72	
Load3	No		Qi
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations

	Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for k Qk
	0	Add1	No					
	0	Add2	No					
	0	Add3	No					
	0	Mult1	Yes	MULTD		R(F2)	Load1	
	0	Mult2	No					

Code:

LD	F0	0	R1
MULT	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
6	72	Qi	Load2		Mult1						

Loop Example Cycle 7

Instruction status

Instruction		j	k	iteration	Issue	Execution complete	Write Result
LD	F0	0	R1	1	1		
MULT	F4	F0	F2	1	2		
SD	F4	0	R1	1	3		
LD	F0	0	R1	2	6		
MULT	F4	F0	F2	2	7		
SD	F4	0	R1	2			

	Busy	Address	Qi
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations

	Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for k Qk
	0	Add1	No					
	0	Add2	No					
	0	Add3	No					
	0	Mult1	Yes	MULTD		R(F2)	Load1	
	0	Mult2	Yes	MULTD		R(F2)	Load2	

Code:			
LD	F0	0	R1
MULT	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
7	72	Qi	Load2		Mult2						

Loop Example Cycle 8

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1		
MULT F4	F0	F2	1	2		
SD F4	0	R1	1	3		
LD F0	0	R1	2	6		
MULT F4	F0	F2	2	7		
SD F4	0	R1	2	8		

	Busy	Address	Qi
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	Yes	MULTD		R(F2)	Load2	

Code:

```
LD   F0   0  R1
MULT F4   F0 F2
SD   F4   0  R1
SUBI R1   R1 #8
BNEZ R1   Loop
```

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12 ...	F30
8	R1								
	72 Qi	Load2		Mult2					

Loop Example Cycle 9

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1	9	
MULT F4	F0	F2	1	2		
SD F4	0	R1	1	3		
LD F0	0	R1	2	6		
MULT F4	F0	F2	2	7		
SD F4	0	R1	2	8		

	Busy	Address	Qi
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	Yes	MULTD		R(F2)	Load2	

Code:

```
LD    F0   0  R1
MULT  F4   F0 F2
SD    F4   0  R1
SUBI  R1   R1 #8
BNEZ  R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
9	64	Qi	Load2		Mult2					

Loop Example Cycle 10

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1	9	10
MULT F4	F0	F2	1	2		
SD F4	0	R1	1	3		
LD F0	0	R1	2	6	10	
MULT F4	F0	F2	2	7		
SD F4	0	R1	2	8		

	Busy	Address	Qi
Load1	No		
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
4	Mult1	Yes	MULTD	M(80)	R(F2)		
0	Mult2	Yes	MULTD		R(F2)	Load2	

Code:

```
LD   F0   0  R1
MULT F4   F0 F2
SD   F4   0  R1
SUBI R1   R1 #8
BNEZ R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
10	64	Qi	Load2		Mult2					

Loop Example Cycle 11

Instruction status

Instruction		j	k	iteration	Issue	Execution complete	Write Result
LD	F0	0	R1	1	1	9	10
MULT	F4	F0	F2	1	2		
SD	F4	0	R1	1	3		
LD	F0	0	R1	2	6	10	11
MULT	F4	F0	F2	2	7		
SD	F4	0	R1	2	8		

Load1	No		
Load2	No		
Load3	Yes	64	Qi
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations

	Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for k Qk
	0	Add1	No					
	0	Add2	No					
	0	Add3	No					
	3	Mult1	Yes	MULTD	M(80)	R(F2)		
	4	Mult2	Yes	MULTD	M(72)	R(F2)		

Code:			
LD	F0	0	R1
MULT	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
11	**64**	Qi	Load3		Mult2						

Loop Example Cycle 12

Instruction status

Instruction		j	k	iteration	Issue	Execution complete	Write Result
LD	F0	0	R1	1	1	9	10
MULT	F4	F0	F2	1	2		
SD	F4	0	R1	1	3		
LD	F0	0	R1	2	6	10	11
MULT	F4	F0	F2	2	7		
SD	F4	0	R1	2	8		

Load1	No		
Load2	No		
Load3	Yes	64	Qi
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations

	Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for k Qk
	0	Add1	No					
	0	Add2	No					
	0	Add3	No					
	2	Mult1	Yes	MULTD	M(80)	R(F2)		
	3	Mult2	Yes	MULTD	M(72)	R(F2)		

Code:			
LD	F0	0	R1
MULT	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
12	64	Qi	Load3		Mult2						

Loop Example Cycle 13

Instruction status

Instruction		j	k	iteration	Issue	Execution complete	Write Result		Busy	Address	
LD	F0	0	R1	1	1	9	10	Load1	No		
MULT	F4	F0	F2	1	2			Load2	No		
SD	F4	0	R1	1	3			Load3	Yes	64	Qi
LD	F0	0	R1	2	6	10	11	Store1	Yes	80	Mult1
MULT	F4	F0	F2	2	7			Store2	Yes	72	Mult2
SD	F4	0	R1	2	8			Store3	No		

Reservation Stations

	Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for k Qk	Code:			
	0	Add1	No						LD	F0	0	R1
	0	Add2	No						MULT	F4	F0	F2
	0	Add3	No						SD	F4	0	R1
	1	Mult1	Yes	MULTD	M(80)	R(F2)			SUBI	R1	R1	#8
	2	Mult2	Yes	MULTD	M(72)	R(F2)			BNEZ	R1	Loop	

Register result status

Clock		R1		F0	F2	F4	F6	F8	F10	F12	...	F30
13		**64**	Qi	Load3		Mult2						

Loop Example Cycle 14

Instruction status

Instruction		j	k	iteration	Issue	Execution complete	Write Result
LD	F0	0	R1	1	1	9	10
MULT	F4	F0	F2	1	2	14	
SD	F4	0	R1	1	3		
LD	F0	0	R1	2	6	10	11
MULT	F4	F0	F2	2	7		
SD	F4	0	R1	2	8		

	Busy	Address	Qi
Load1	No		
Load2	No		
Load3	Yes	64	Qi
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations

	Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for k Qk
	0	Add1	No					
	0	Add2	No					
	0	Add3	No					
	0	Mult1	Yes	MULTD	M(80)	R(F2)		
	1	Mult2	Yes	MULTD	M(72)	R(F2)		

Code:			
LD	F0	0	R1
MULT	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
14	64	Qi	Load3		Mult2					

Loop Example Cycle 15

Instruction status

Instruction		j	k	iteration	Issue	Execution complete	Write Result
LD	F0	0	R1	1	1	9	10
MULT	F4	F0	F2	1	2	14	15
SD	F4	0	R1	1	3		
LD	F0	0	R1	2	6	10	11
MULT	F4	F0	F2	2	7	15	
SD	F4	0	R1	2	8		

	Busy	Address	
Load1	No		
Load2	No		
Load3	Yes	64	Qi
Store1	Yes	80	M(80)*R(
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations

	Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for k Qk
	0	Add1	No					
	0	Add2	No					
	0	Add3	No					
	0	Mult1	No					
	0	Mult2	Yes	MULTD	M(72)	R(F2)		

Code:			
LD	F0	0	R1
MULT	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
15	64	Qi	Load3		Mult2					

- **Update to regsiter F0?**

Loop Example Cycle 16

Instruction status

Instruction		j	k	iteration	Issue	Execution complete	Write Result
LD	F0	0	R1	1	1	9	10
MULT	F4	F0	F2	1	2	14	15
SD	F4	0	R1	1	3		
LD	F0	0	R1	2	6	10	11
MULT	F4	F0	F2	2	7	15	16
SD	F4	0	R1	2	8		

	Busy	Address	Qi
Load1	No		
Load2	No		
Load3	Yes	64	Qi
Store1	Yes	80	M(80)*R(
Store2	Yes	72	M(72)*R(
Store3	No		

Reservation Stations

	Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for k Qk
	0	Add1	No					
	0	Add2	No					
	0	Add3	No					
	0	Mult1	Yes	MULTD		R(F2)	Load3	
	0	Mult2	No					

Code:			
LD	F0	0	R1
MULT	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
16	64	Qi	Load3		Mult1					

Loop Example Cycle 17

Instruction status												
Instruction		*j*	*k*	*iteration*	*Issue*	*Execution complete*	*Write Result*		Busy	Address		
LD	F0	0	R1	1	1	9	10	Load1	No			
MULT	F4	F0	F2	1	2	14	15	Load2	No			
SD	F4	0	R1	1	3			Load3	Yes	64	Qi	
LD	F0	0	R1	2	6	10	11	Store1	Yes	80	M(80)*R(	
MULT	F4	F0	F2	2	7	15	16	Store2	Yes	72	M(72)*R(	
SD	F4	0	R1	2	8			Store3	Yes	64	Mult1	

Reservation Stations												
	Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS for Qj*	*RS for k Qk*	*Code:*			
	0	Add1	No						LD	F0	0	R1
	0	Add2	No						MULT	F4	F0	F2
	0	Add3	No						SD	F4	0	R1
	0	Mult1	Yes	MULTD		R(F2)	Load3		SUBI	R1	R1	#8
	0	Mult2	No						BNEZ	R1	Loop	

Register result status											
Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
17	**64**	*Qi*	Load3		Mult1						

Loop Example Cycle 18

Instruction status												
Instruction		*j*	*k*	*iteration*	*Issue*	*Execution complete*	*Write Result*		Busy	Address		
LD	F0	0	R1	1	1	9	10	Load1	No			
MULT	F4	F0	F2	1	2	14	15	Load2	No			
SD	F4	0	R1	1	3	18		Load3	Yes	64	Qi	
LD	F0	0	R1	2	6	10	11	Store1	Yes	80	M(80)*R(	
MULT	F4	F0	F2	2	7	15	16	Store2	Yes	72	M(72)*R(	
SD	F4	0	R1	2	8			Store3	Yes	64	Mult1	

Reservation Stations												
	Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS for Qj*	*RS for k Qk*	*Code:*			
	0	Add1	No						LD	F0	0	R1
	0	Add2	No						MULT	F4	F0	F2
	0	Add3	No						SD	F4	0	R1
	0	Mult1	Yes	MULTD		R(F2)	Load3		SUBI	R1	R1	#8
	0	Mult2	No						BNEZ	R1	Loop	

Register result status											
Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
18	**56**	*Qi*	Load3		Mult1						

Loop Example Cycle 19

Instruction status

Instruction		j	k	iteration	Issue	Execution complete	Write Result
LD	F0	0	R1	1	1	9	10
MULT	F4	F0	F2	1	2	14	15
SD	F4	0	R1	1	3	18	19
LD	F0	0	R1	2	6	10	11
MULT	F4	F0	F2	2	7	15	16
SD	F4	0	R1	2	8		

Load1	No		
Load2	No		
Load3	Yes	64	Qi
Store1	No		
Store2	Yes	72	M(72)*R(
Store3	Yes	64	Mult1

Reservation Stations

	Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for k Qk
	0	Add1	No					
	0	Add2	No					
	0	Add3	No					
	0	Mult1	Yes	MULTD		R(F2)	Load3	
	0	Mult2	No					

Code:			
LD	F0	0	R1
MULT	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
19	56	Qi	Load3		Mult1						

Loop Example Cycle 20

Instruction status

Instruction		j	k	iteration	Issue	Execution complete	Write Result
LD	F0	0	R1	1	1	9	10
MULT	F4	F0	F2	1	2	14	15
SD	F4	0	R1	1	3	18	19
LD	F0	0	R1	2	6	10	11
MULT	F4	F0	F2	2	7	15	16
SD	F4	0	R1	2	8	20	

Load1	No		
Load2	No		
Load3	Yes	64	Qi
Store1	No		
Store2	Yes	72	M(72)*R(
Store3	Yes	64	Mult1

Reservation Stations

	Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for k Qk
	0	Add1	No					
	0	Add2	No					
	0	Add3	No					
	0	Mult1	Yes	MULTD		R(F2)	Load3	
	0	Mult2	No					

Code:			
LD	F0	0	R1
MULT	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
20	**56**	Qi	Load3		Mult1						

Loop Example Cycle 21

Instruction status

Instruction		j	k	iteration	Issue	Execution complete	Write Result
LD	F0	0	R1	1	1	9	10
MULT	F4	F0	F2	1	2	14	15
SD	F4	0	R1	1	3	18	19
LD	F0	0	R1	2	6	10	11
MULT	F4	F0	F2	2	7	15	16
SD	F4	0	R1	2	8	20	21

	Busy	Address	Qi
Load1	No		
Load2	No		
Load3	Yes	64	Qi
Store1	No		
Store2	No		
Store3	Yes	64	Mult1

Reservation Stations

	Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for Qj	RS for k Qk
	0	Add1	No					
	0	Add2	No					
	0	Add3	No					
	0	Mult1	Yes	MULTD		R(F2)	Load3	
	0	Mult2	No					

Code:

LD	F0	0	R1
MULT	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
21	56	Qi	Load3		Mult1						

Tomasulo Summary

- Prevents Register as bottleneck
- Avoids WAR, WAW hazards of Scoreboard
- Allows loop unrolling in HW
- Not limited to basic blocks (provided branch prediction)
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation
- 360/91 descendants are PowerPC 604, 620; MIPS R10000; HP-PA 8000; Intel Pentium Pro

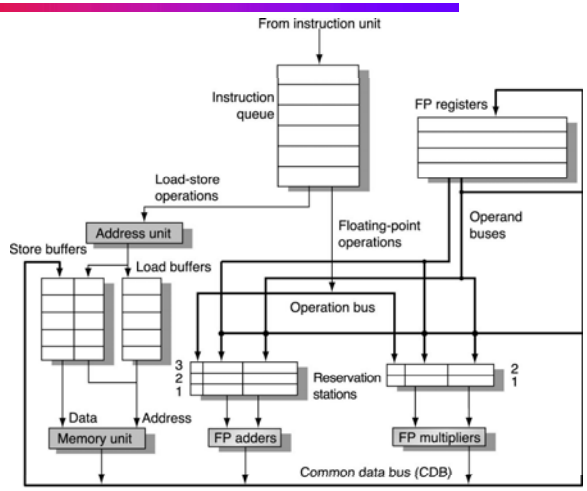
Another Dynamic Algorithm: Tomasulo Algorithm

- For IBM 360/91 about 3 years after CDC 6600
- Goal: High Performance without special compilers
- Differences between IBM 360 & CDC 6600 ISA
  - IBM has only 2 register specifiers/instr vs. 3 in CDC 6600
  - IBM has 4 FP registers vs. 8 in CDC 6600
  - Implications?

Differences between Tomasulo Algorithm & Scoreboard

- Control & buffers distributed with Function Units vs. centralized in scoreboard; called "reservation stations"
  => instrs schedule themselves
- Registers in instructions replaced by pointers to reservation station buffer
  scoreboard => registers primary operand storage
  Tomasulo => reservation stations as operand storage
- HW renaming of registers to avoid WAR, WAW hazards
  Scoreboard => both source registers read together (thus one could not be overwritten while we wait for the other).
  Tomasulo => each register read as soon as available.
- Common Data Bus broadcasts results to all FUs
  RS's (FU's), registers, etc. responsible for collecting own data off CDB
- Load and Store Queues treated as FUs as well

Tomasulo Organization



Reservation Station Components

Op—Operation to perform in the unit (e.g., + or –)

Qj, Qk—Reservation stations producing source registers

Vj, Vk—Value of Source operands

Rj, Rk—Flags indicating when Vj, Vk are ready

Busy—Indicates reservation station is busy

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Three Stages of Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue

   If reservation station free, the scoreboard issues instr & sends operands (renames registers).

2. Execution—operate on operands (EX)

   When both operands ready then execute; if not ready, watch CDB for result

3. Write result—finish execution (WB)

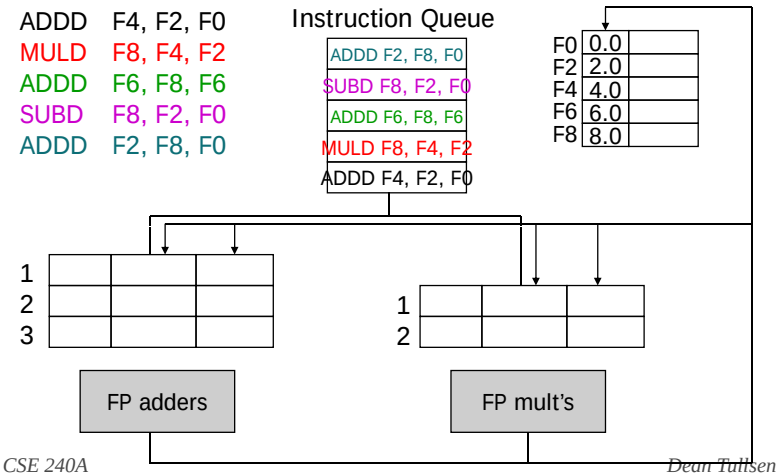
   Write on Common Data Bus to all waiting units; mark reservation station available.

Tomasulo Example

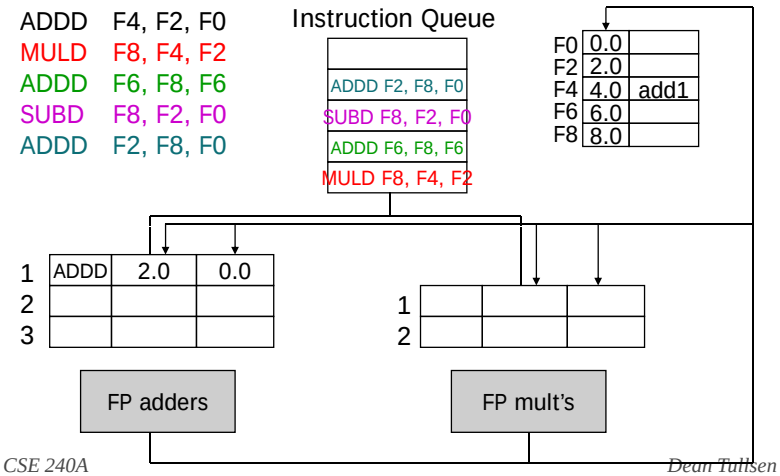
```
ADDD   F4, F2, F0
MULD   F8, F4, F2
ADDD   F6, F8, F6
SUBD   F8, F2, F0
ADDD   F2, F8, F0
```

Multiply takes 10 clocks, add/sub take 4

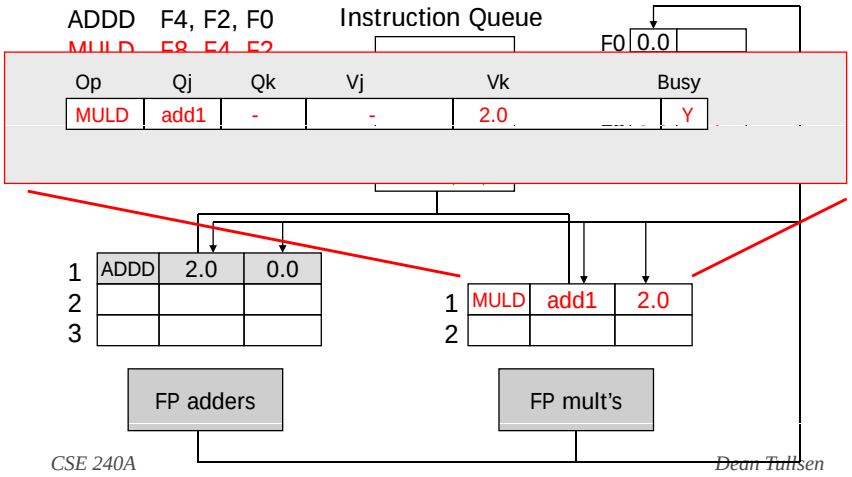
Tomasulo – cycle 0



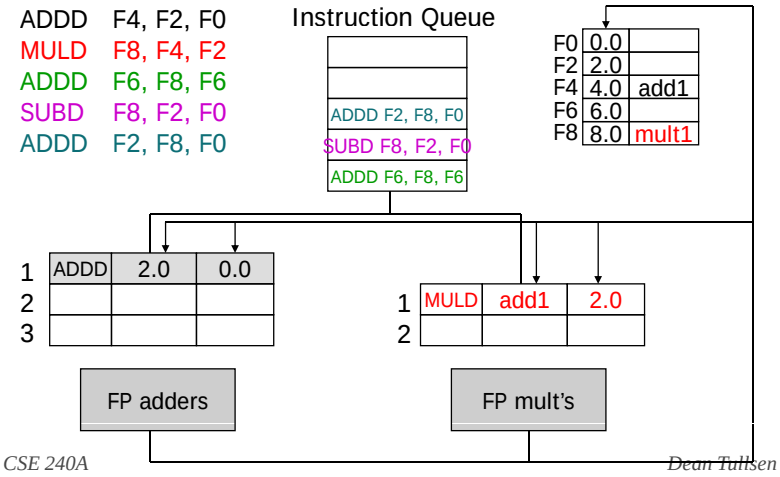
Tomasulo – cycle 1



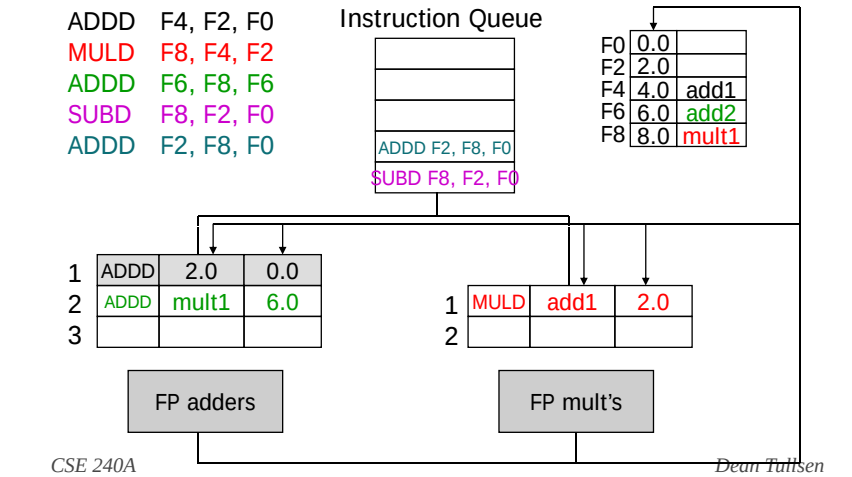
Tomasulo – cycle 2



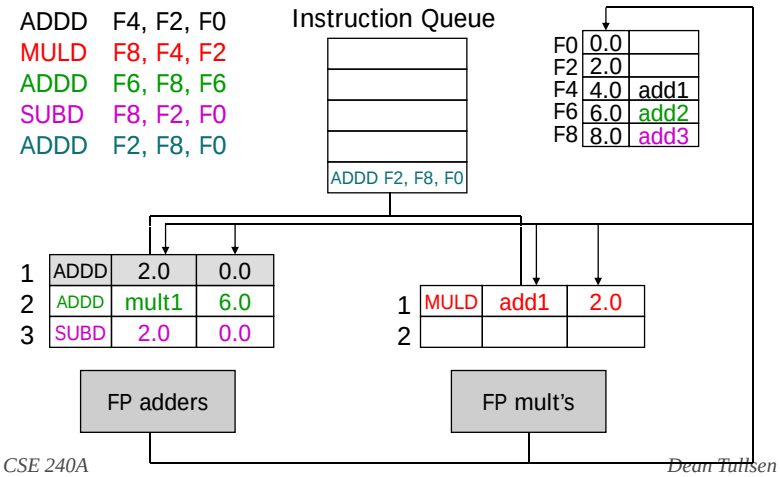
Tomasulo – cycle 2



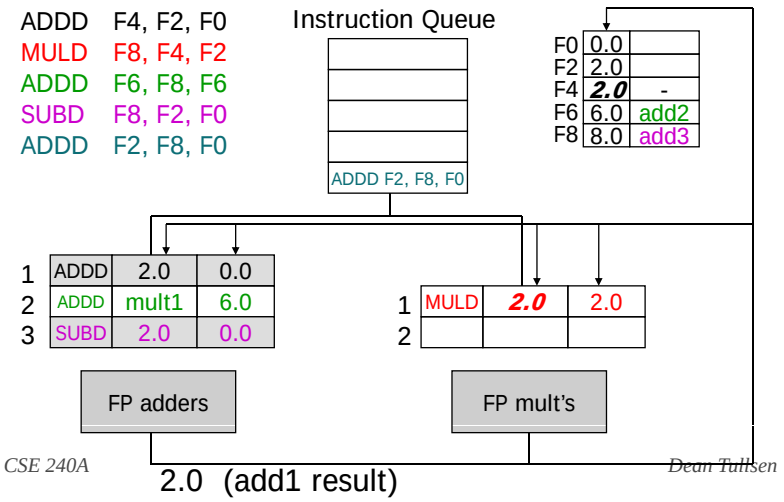
Tomasulo – cycle 3



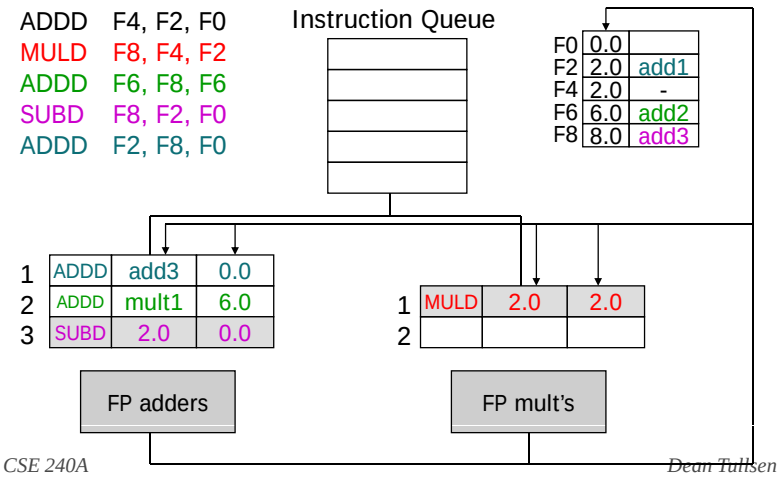
Tomasulo – cycle 4



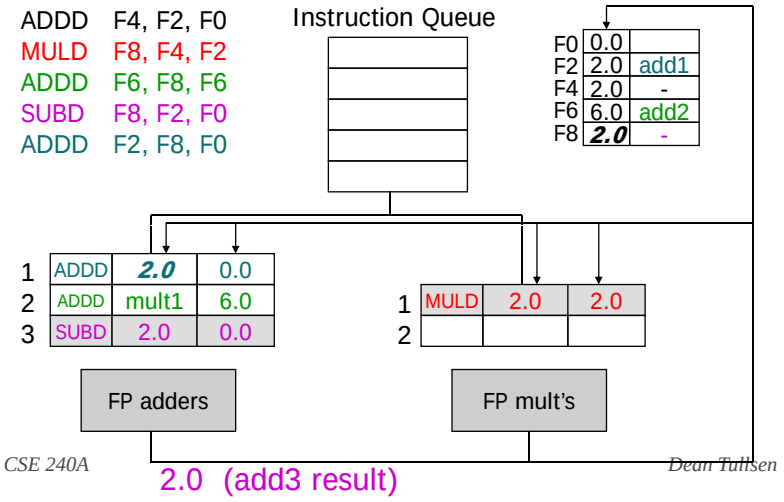
Tomasulo – cycle 5



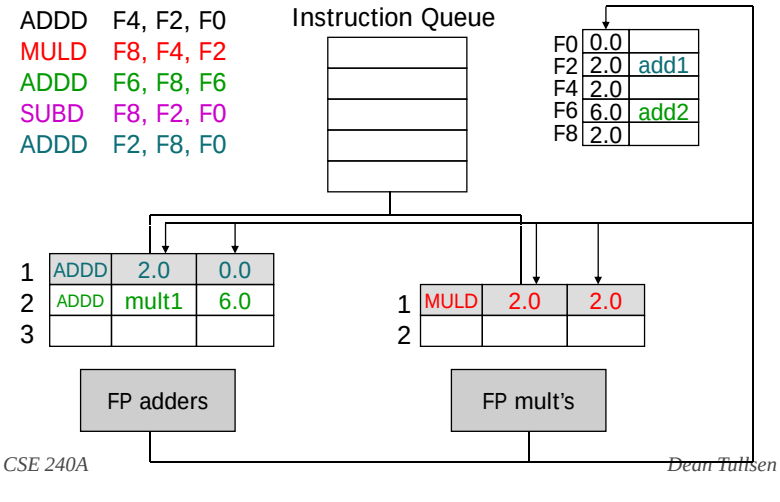
Tomasulo – cycle 6



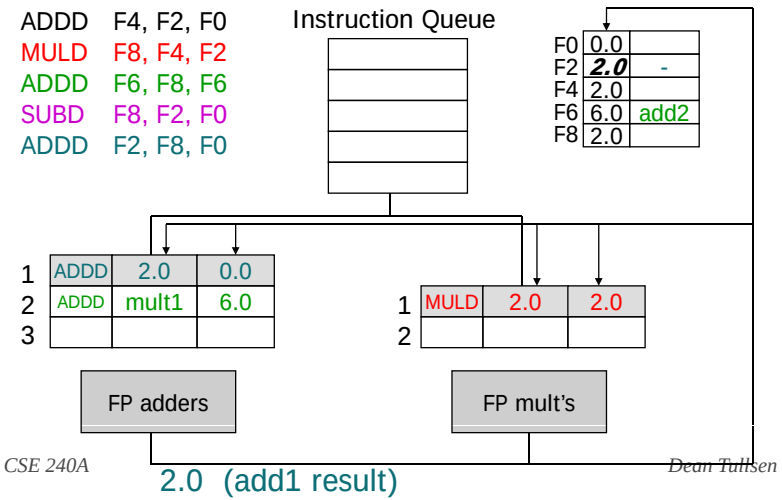
Tomasulo – cycle 8



Tomasulo – cycle 9

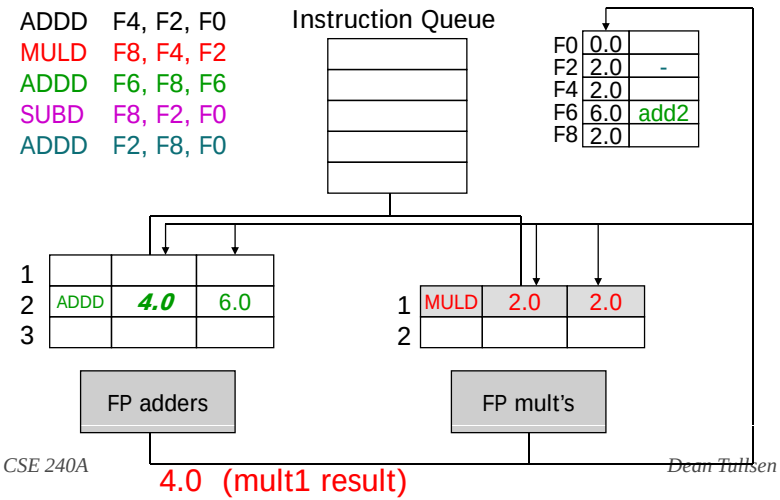


Tomasulo – cycle 12



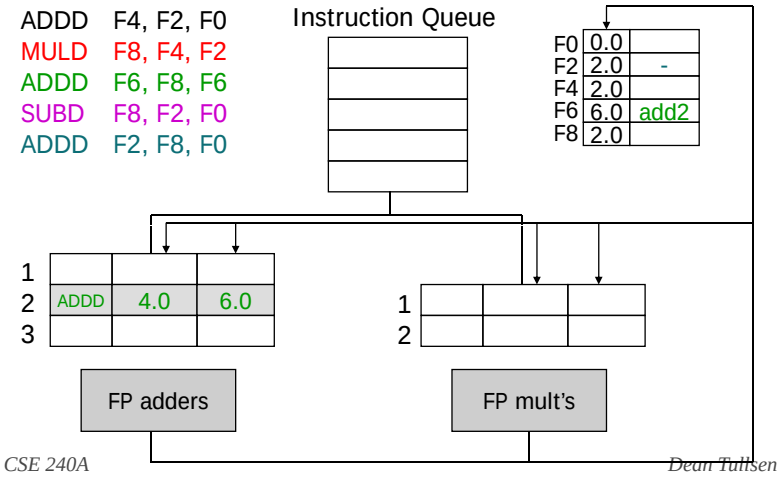
2.0 (add1 result)

Tomasulo – cycle 15

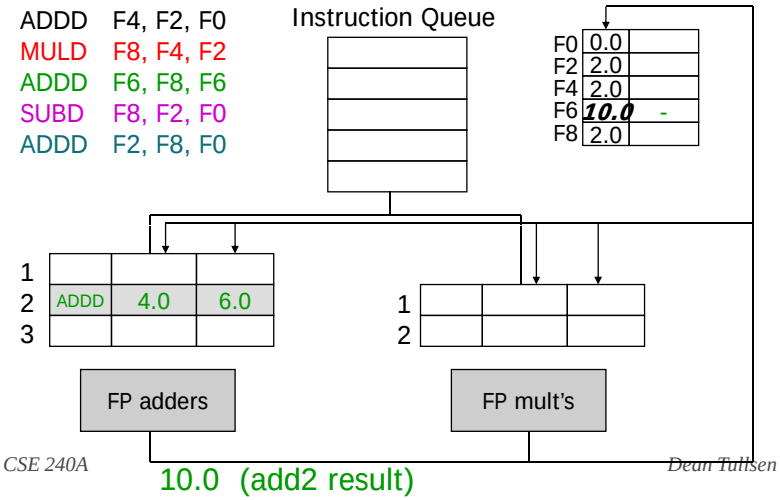


4.0 (mult1 result)

Tomasulo – cycle 16



Tomasulo – cycle 19



10.0 (add2 result)

Tomasulo Summary

- Prevents Register as bottleneck
- Avoids WAR, WAW hazards of Scoreboard
- Allows loop unrolling in HW
- Not limited to basic blocks (provided branch prediction)
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming (in what way does the register name *change*?)
  - Load/store disambiguation

Scoreboard vs. Tomasulo, the score

	Scoreboard	Tomasulo
issue	when FU free	when RS free
read operands	from reg file	from reg file, CDB
write operands	to reg file	to CDB
structural hazards	functional units	reservation stations
WAW, WAR hazards	problem	no problem
register renaming	no	yes
instructions completing	no limit	1 / cycle (per CDB)
instructions beginning ex.	1 (per set of read ports)	no limit

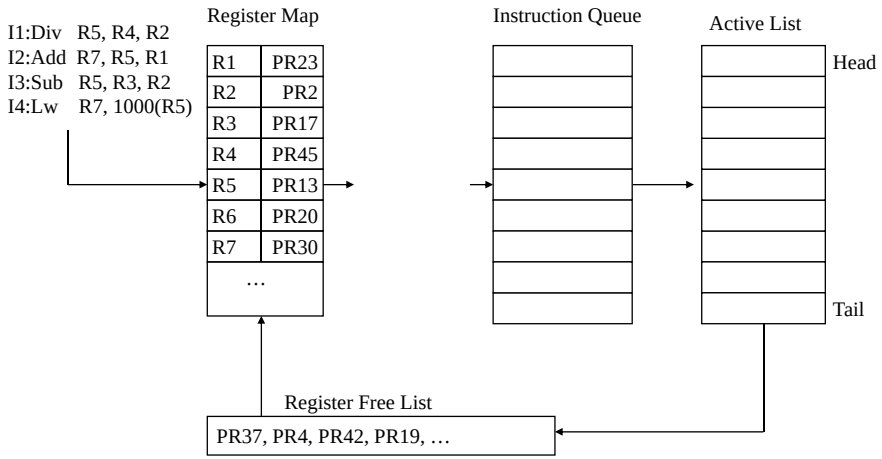
Modern Architectures

- Alpha 21264+, MIPS R10K+, Pentium 4 use an ***instruction queue***.
- Uses explicit register renaming. Registers are not read until instruction dispatches (begins execution). Register renaming ensures no conflicts.

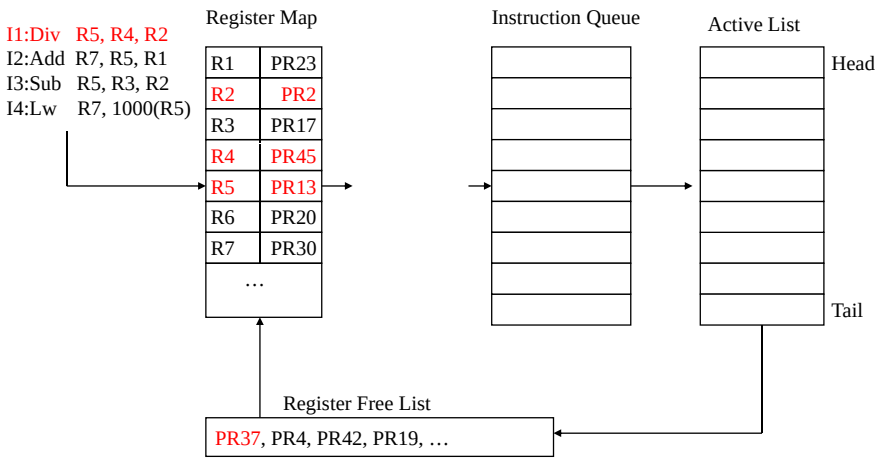
```
Div  R5, R4, R2          Div  PR37, PR45, PR2
Add  R7, R5, R1          Add  PR4, PR37, PR23
Sub  R5, R3, R2          Sub  PR42, PR17, PR2
Lw   R7, 1000(R5)        Lw   PR19, 1000(PR42)
```

R1	PR23
R2	PR2
R3	PR17
R4	PR45
R5	**PR42**
R6	PR20
R7	**PR19**
…	

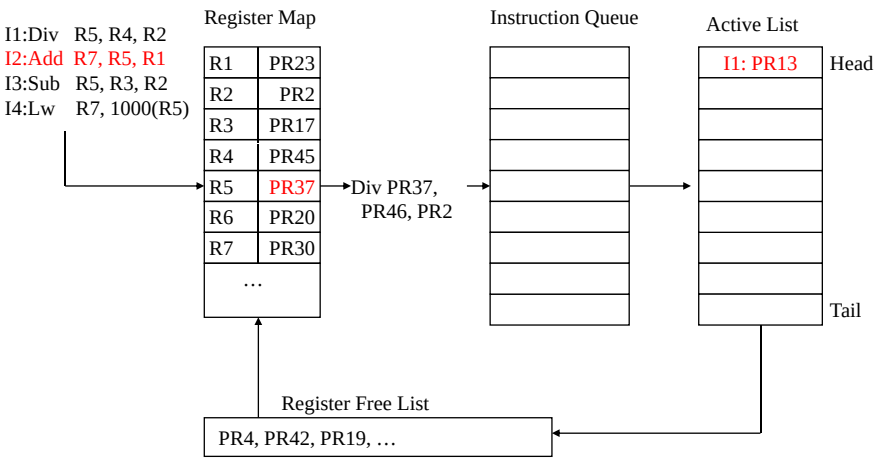
MIPS R10000, some detail



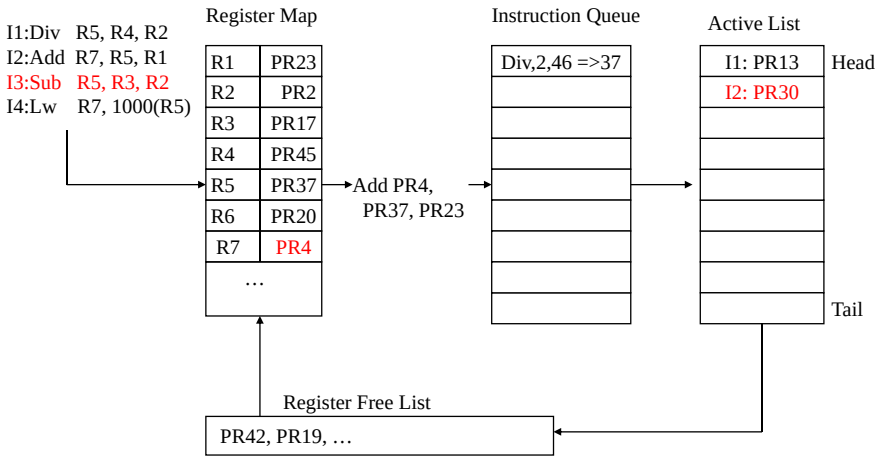
MIPS R10000, some detail



MIPS R10000, some detail

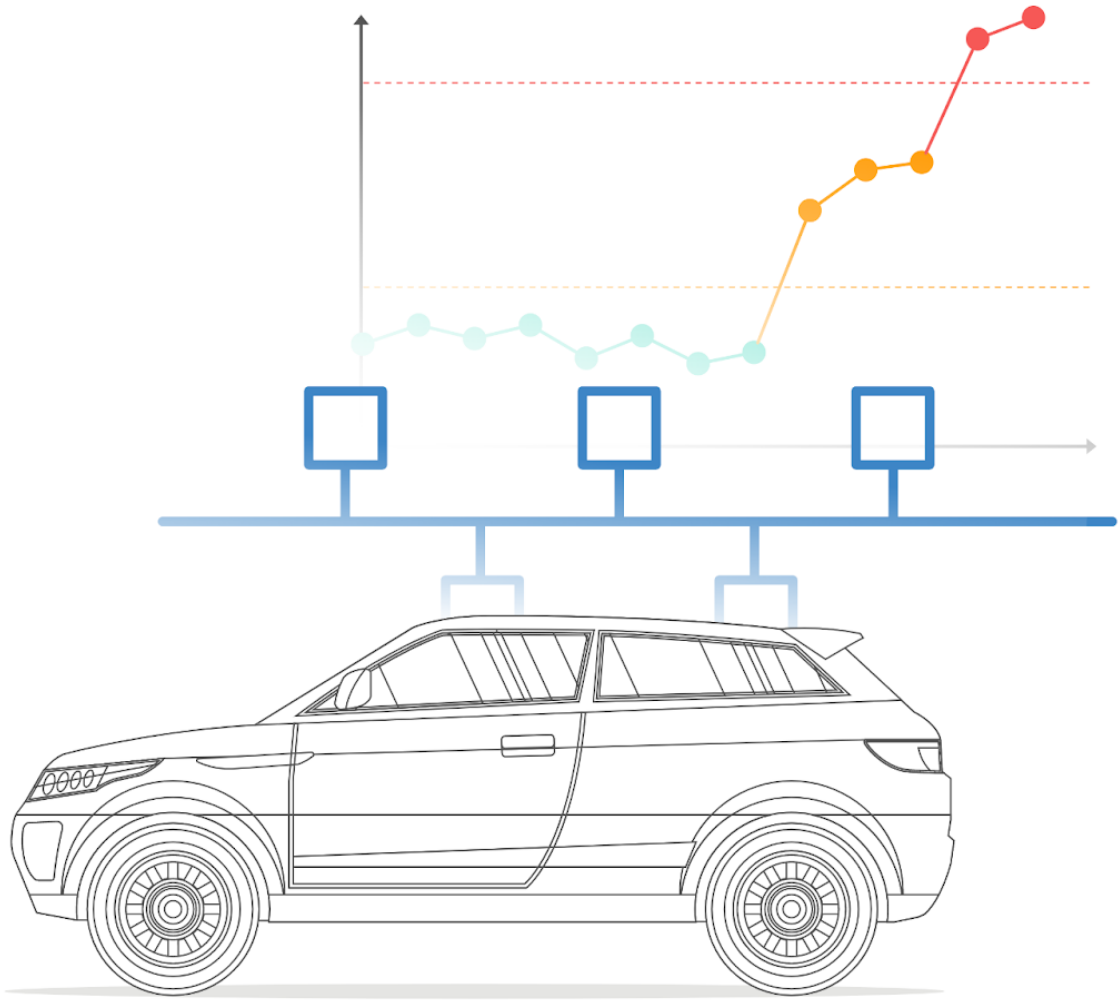


MIPS R10000, some detail



Dynamic Scheduling Key Points

- Dynamic scheduling is code motion in HW.
- Dynamic scheduling can do things SW scheduling (static scheduling) cannot.
- Scoreboard, Tomasulo have various tradeoffs
- Register renaming eliminates WAW, WAR dependencies.
- To get cross-iteration parallelism, we need to eliminate WAW, WAR dependencies.



CAN bus

the ultimate guide

CAN | J1939 | OBD2 | UDS | CANopen | CAN FD | LIN | DBC | CAN Errors

NMEA 2000 | ISOBUS | CCP/XCP on CAN

CSS Electronics | www.csselectronics.com | contact@csselectronics.com
Last modified: June 21, 2023

Table of Contents

CANedge - 2 x CAN/LIN Bus Data Logger **2**

CAN Bus Explained - A Simple Intro **2**

What is CAN bus? 2

Top 4 benefits of CAN bus 6

The CAN bus history in short 7

The future of CAN bus 7

What is a CAN frame? 8

Logging CAN data - example use cases 9

How to log CAN bus data 9

How to decode raw CAN data to 'physical values' 11

What is the link between CAN, J1939, OBD2, CANopen ...? 13

J1939 Explained - A Simple Intro **14**

What is J1939? 14

J1939 history & future 14

4 key characteristics of J1939 15

The J1939 connector (9-pin) 17

The J1939 PGN and SPN 18

J1939 truck sample data: Raw & physical values 20

J1939 request messages 23

J1939 transport protocol (TP) 24

Logging J1939 data - example use cases 26

6 practical tips for J1939 data logging 26

OBD2 Explained - A Simple Intro **28**

What is OBD2? 28

The OBD2 connector 28

Does my car have OBD2? 29

Link between OBD2 and CAN bus 29

OBD2 history & future 30

OBD2 parameter IDs (PID) 32

How to log OBD2 data? 33

Raw OBD2 frame details 34

OBD2 data logging - use case examples 36

UDS Explained (Unified Diagnostic Services) **36**

What is the UDS protocol? 36

UDS message structure 37

UDS vs CAN bus: Standards & OSI model 42

CAN ISO-TP - Transport Protocol (ISO 15765-2) 46

UDS vs. OBD2 vs. WWH-OBD vs. OBDonUDS 47

FAQ: How to request/record UDS data 51

Example 1: Record single frame UDS data (Speed via WWH-OBD) 55

Example 2: Record & decode multi frame UDS data (SoC) 56

Example 3: Record the Vehicle Identification Number 59

UDS data logging - applications 60
CANopen Explained - A Simple Intro 61
What is CANopen? 61
Six core CANopen concepts 63
CANopen communication basics 63
CANopen Object Dictionary 68
SDO - configuring the CANopen network 69
PDO - operating the CANopen network 70
CANopen data logging - use case examples 72
CAN FD Explained - A Simple Intro 73
Why CAN FD? 73
What is CAN FD? 73
How does CAN FD work? 74
Overhead and data efficiency of CAN FD vs. CAN 76
Examples: CAN FD applications 78
Logging CAN FD data - use case examples 79
CAN FD - outlook 80
LIN Bus Explained - A Simple Intro 80
What is LIN bus? 80
LIN bus applications 81
How does LIN bus work? 83
Six LIN frame types 85
Advanced LIN topics 86
LIN Description File (LDF) vs. DBC files 87
LIN bus data logging - use case examples 89
Practical considerations for LIN data logging 90
CAN DBC File Explained - A Simple Intro 91
What is a CAN DBC file? 91
Example: Extract physical value of EngineSpeed signal 92
CAN DBC editor playground 96
J1939/OBD2 data & DBC samples 96
Advanced: Meta info, attributes & multiplexing 96
DBC software tools (editing & processing) 98
CAN Bus Errors Explained - A Simple Intro 99
What are CAN bus errors? 99
The CAN bus error frame 100
CAN error types 103
CAN node states & error counters 106
Examples: Generating & logging error frames 109
LIN bus errors 112
Example use cases for CAN error frame logging 113
FAQ 114
NMEA 2000 Explained - A Simple Intro 114
What is NMEA 2000? 114
NMEA 2000 vs NMEA 0183 115
NMEA history and certification 118
NMEA 2000 OSI model & standards 119

NMEA 2000 connectors & network topology 121
NMEA 2000 Fast Packet 122
N2K PGN & data fields 122
Logging NMEA 2000 maritime data 127
ISOBUS (ISO 11783) Explained - A Simple Intro 128
What is ISOBUS? 128
ISOBUS history and AEF 128
ISOBUS OSI model & standards 130
ISOBUS Functionalities 132
ISOBUS vs SAE J1939 134
The ISOBUS connectors 135
ISOBUS PGN and SPN [+ DBC] 136
Logging tractor/implement data 139
Data logging use case examples 141
CCP / XCP on CAN Explained - A Simple Intro 141
What is CCP/XCP? 142
CCP message types 144
How to record ECU data via CCP 148
Decoding CCP signal data from ECUs 154
A2L - ECU Description Files 157
Seed & key authorization 159
XCP on CAN - the basics 160
Using the CANedge for CCP/XCP data acquisition 164
CCP/XCP data logging - applications 165

CANedge - 2 x CAN/LIN Bus Data Logger

Interested in CAN bus and related protocols? Then make sure to check out our CANedge series as well!

The CANedge is a 2 x CAN/LIN data logger with an 8-32 GB industrial SD card. The series offers optional GPS/IMU, WiFi and/or 3G/4G - ideal for OEM diagnostics, vehicle fleet telematics, predictive maintenance and more:

PLUG & PLAY
Log CAN data out-the-box. Standalone. Bit-rate auto detect. Power safe

PRO SPECS
Extractable 8-32 GB SD. 2xCAN/LIN. CAN FD. Zero data loss. 50 Âµs RTC. Error frames. MF4

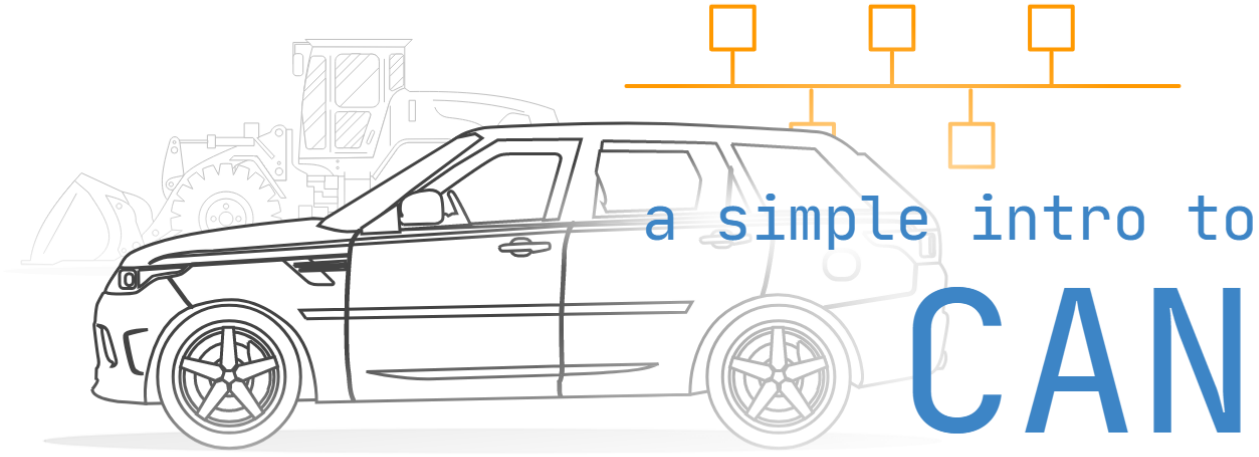
COMPACT
Only 8 x 5 x 2 CM. 100G. Robust alu enclosure. 5+ LEDs. 5V power out (CH2)

WIFI/LTE
Push data via WiFi or 3G/4G to your server. E2E security. OTA updates

GPS + 3D IMU
Built-in GPS/IMU. 3x accuracy via sensor fusion. Position, speed, distance & more

INTEROPERABLE
Free open source software/APIs. MF4 to ASC/CSV. DBC decoding. Dashboards

Check out the **5 min CANedge intro video**



CAN Bus Explained - A Simple Intro

Need a simple, practical intro to CAN bus? In this tutorial we explain the Controller Area Network (CAN bus) 'for dummies' incl. message interpretation, CAN logging - and the link to OBD2, J1939 and CANopen.

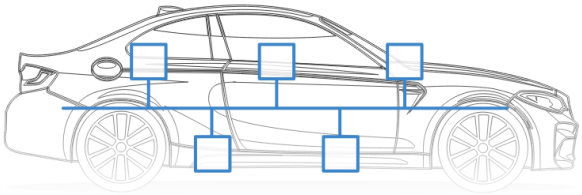
Read on to learn why this has become the #1 guide on CAN bus.

What is CAN bus?

Your car is like a human body:

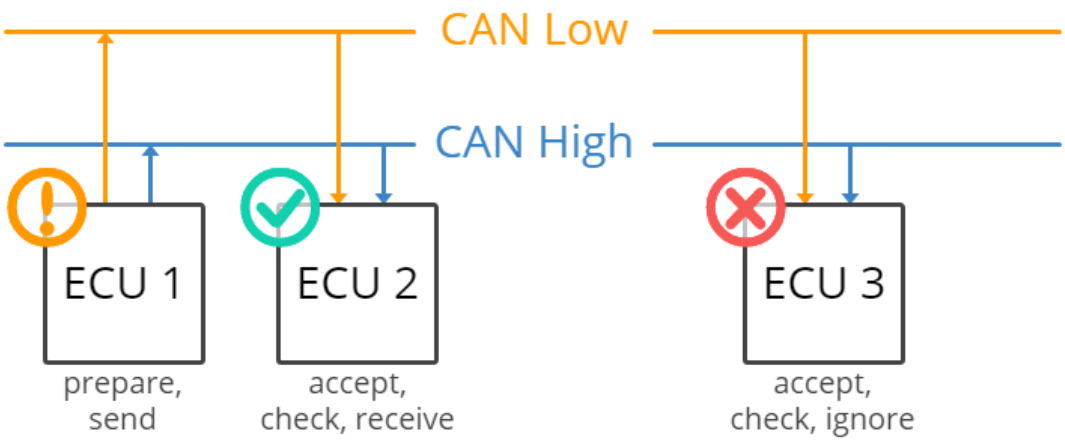
The Controller Area Network (CAN bus) is the nervous system, enabling communication.

In turn, 'nodes' or 'electronic control units' (ECUs) are like parts of the body, interconnected via the CAN bus. Information sensed by one part can be shared with another.



So what is an ECU?

In an automotive CAN bus system, ECUs can e.g. be the engine control unit, airbags, audio system etc. A modern car may have up to 70 ECUs - and each of them may have information that needs to be shared with other parts of the network.



This is where the CAN standard comes in handy:

The CAN bus system enables each ECU to communicate with all other ECUs - without complex dedicated wiring.

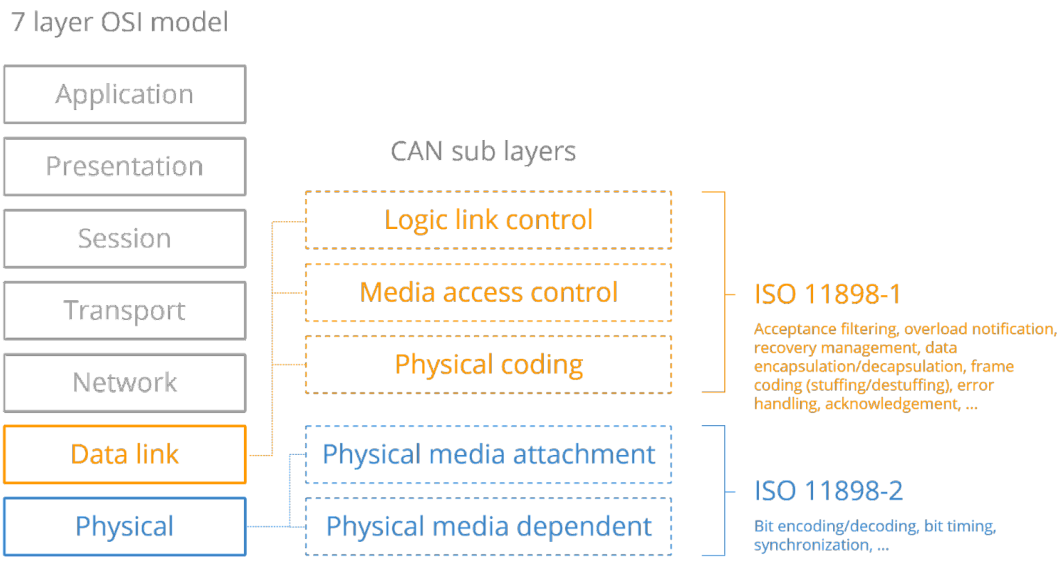
Specifically, an ECU can prepare and broadcast information (e.g. sensor data) via the CAN bus (consisting of two wires, CAN low and CAN high). The broadcasted data is accepted by all other ECUs on the CAN network - and each ECU can then check the data and decide whether to receive or ignore it.

CAN bus physical & data link layer (OSI)

In more technical terms, the controller area network is described by a data link layer and physical layer. In the case of high speed CAN, ISO 11898-1 describes the data link layer, while ISO 11898-2 describes the physical layer. The role of CAN is often presented in the 7 layer OSI model as per the illustration.

The CAN bus physical layer defines things like cable types, electrical signal levels, node requirements, cable impedance etc. For example, ISO 11898-2 dictates a number of things, including below:

- Baud rate: CAN nodes must be connected via a two wire bus with baud rates up to 1 Mbit/s (Classical CAN) or 5 Mbit/s (CAN FD)
- Cable length: Maximal CAN cable lengths should be between 500 meters (125 kbit/s) and 40 meters (1 Mbit/s)
- Termination: The CAN bus must be properly terminated using a 120 Ohms CAN bus termination resistor at each end of the bus



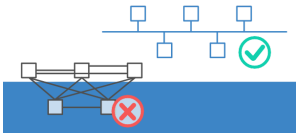
High speed CAN, low speed CAN, LIN bus, ...

In the context of automotive vehicle networks, you'll often encounter a number of different types of network types. Below we provide a very brief outline:

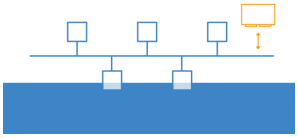
- **High speed CAN bus:** The focus of this article is on high speed CAN bus (ISO 11898). It is by far the most popular CAN standard for the physical layer, supporting bit rates from 40 kbit/s to 1 Mbit/s (Classical CAN). It provides simple cabling and is used in practically all automotive applications today. It also serves as the basis for several higher layer protocols such as OBD2, J1939, NMEA 2000, CANopen etc. The second generation of CAN is referred to as CAN FD (CAN with Flexible Data-rate)
- **Low speed CAN bus:** This standard enables bit rates from 40 kbit/s to 125 kbit/s and allows the CAN bus commmunication to continue even if there is a fault on one of the two wires - hence it is also referred to as 'fault tolerant CAN'. In this system, each CAN node has it's own CAN termination
- **LIN bus:** LIN bus is a lower cost supplement to CAN bus networks, with less harness and cheaper nodes. LIN bus clusters typically consist of a LIN master acting as gateway and up to 16 slave nodes. Typical use cases include e.g. non-critical vehicle functions like aircondition, door functionality etc. - for details see our LIN bus intro or LIN bus data logger article
- **Automotive ethernet:** This is increasingly being rolled out in the automotive sector to support the high bandwidth requirements of ADAS (Advanced Driver Assistance Systems), infotainment systems, cameras etc. Automotive ethernet offers much higher data transfer rates vs. CAN bus, but lacks some of the safety/performance features of Classical CAN and CAN FD. Most likely, the coming years will see both automotive ethernet, CAN FD and CAN XL being used in new automotive and industrial development

Top 4 benefits of CAN bus

The CAN bus standard is used in practically all vehicles and many machines due to below key benefits:

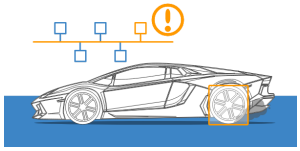


Simple & low cost
ECUs communicate via a single CAN system instead of via direct complex analogue signal lines - reducing errors, weight, wiring and costs



Fully centralized
The CAN bus provides 'one point-of-entry' to communicate with all network ECUs - enabling central diagnostics, data logging and configuration

Extremely robust
The system is robust towards electric disturbances and electromagnetic interference - ideal for safety critical applications (e.g. vehicles)

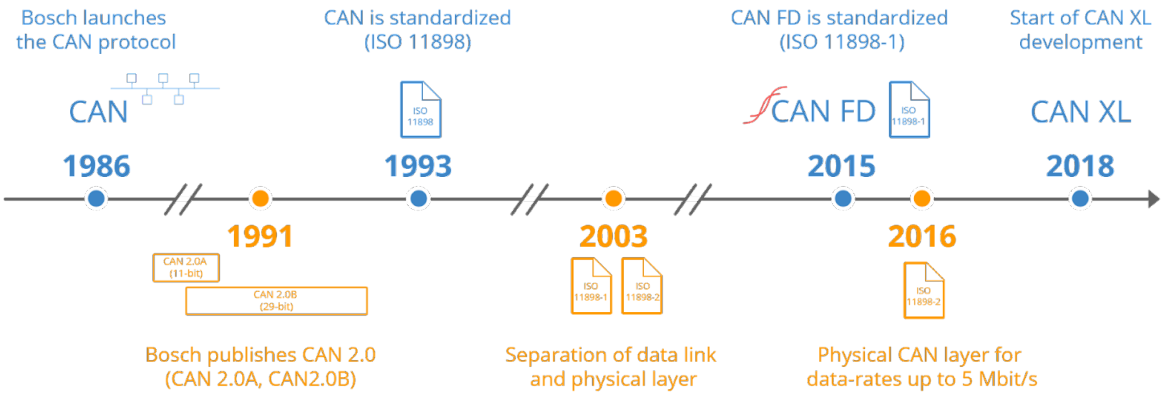


Efficient
CAN frames are prioritized by ID so that top priority data gets immediate bus access, without causing interruption of other frames

The CAN bus history in short

- Pre CAN: Car ECUs relied on complex point-to-point wiring
- 1986: Bosch developed the CAN protocol as a solution
- 1991: Bosch published CAN 2.0 (CAN 2.0A: 11 bit, 2.0B: 29 bit)
- 1993: CAN is adopted as international standard (ISO 11898)
- 2003: ISO 11898 becomes a standard series
- 2012: Bosch released the CAN FD 1.0 (flexible data rate)
- 2015: The CAN FD protocol is standardized (ISO 11898-1)
- 2016: The physical CAN layer for data-rates up to 5 Mbit/s standardized in ISO 11898-2

Today, CAN is standard in automotives (cars, trucks, buses, tractors, ...), ships, planes, EV batteries, machinery and more.



The future of CAN bus

Looking ahead, the CAN bus protocol will stay relevant - though it will be impacted by major trends:

- A need for increasingly advanced vehicle functionality
- The rise of cloud computing
- Growth in Internet of Things (IoT) and connected vehicles
- The impact of autonomous vehicles

In particular, the rise in connected vehicles (V2X) and cloud will lead to a rapid growth in vehicle telematics and IoT CAN loggers. In turn, bringing the CAN bus network 'online' also exposes vehicles to security risks - and may require a shift to new CAN protocols like CAN FD.

The rise of CAN FD

As vehicle functionality expands, so does the load on the CANbus. To support this, CAN FD (Flexible Data Rate) has been designed as the 'next generation' CAN bus. Specifically, CAN FD offers three benefits (vs Classical CAN):

- It enables data rates up to 8 Mbit/s (vs 1 Mbit/s)
- It allows data payloads of up to 64 bytes (vs 8 bytes)
- It enables improved security via authentication

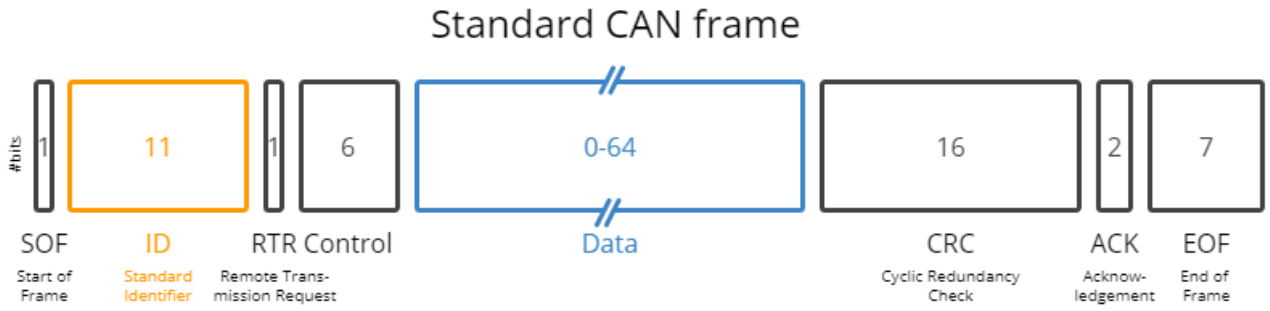
In short, CAN FD boosts speed and efficiency - and it is therefore being rolled out in newer vehicles. This will also drive an increasing need for IoT CAN FD data loggers.

"The first cars using CAN FD will appear in 2019/2020 and CAN FD will replace step-by-step Classical CAN"
- CAN in Automation (CiA), "CAN 2020: The Future of CAN Technology"

What is a CAN frame?

Communication over the CAN bus is done via CAN frames.

Below is a standard CAN frame with 11 bits identifier (CAN 2.0A), which is the type used in most cars. The extended 29-bit identifier frame (CAN 2.0B) is identical except the longer ID. It is e.g. used in the J1939 protocol for heavy-duty vehicles. Note that the CAN ID and Data are highlighted - these are important when recording CAN bus data, as we'll see below.



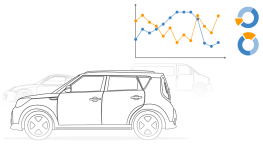
The 8 CAN bus protocol message fields

- **SOF**: The Start of Frame is a 'dominant 0' to tell the other nodes that a CAN node intends to talk
- **ID**: The ID is the frame identifier - lower values have higher priority

- **RTR**: The Remote Transmission Request indicates whether a node sends data or requests dedicated data from another node
- **Control**: The Control contains the Identifier Extension Bit (IDE) which is a 'dominant 0' for 11-bit. It also contains the 4 bit Data Length Code (DLC) that specifies the length of the data bytes to be transmitted (0 to 8 bytes)
- **Data**: The Data contains the data bytes aka payload, which includes CAN signals that can be extracted and decoded for information
- **CRC**: The Cyclic Redundancy Check is used to ensure data integrity
- **ACK**: The ACK slot indicates if the node has acknowledged and received the data correctly
- **EOF**: The EOF marks the end of the CAN frame

Logging CAN data - example use cases

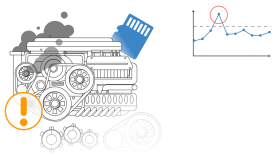
There are several common use cases for recording CAN bus data frames:



Logging/streaming data from cars
OBD2 data from cars can e.g. be used to reduce fuel costs, improve driving, test prototype parts and insurance
learn more

Heavy duty fleet telematics
J1939 data from trucks, buses, tractors etc. can be used in fleet management to reduce costs or improve safety
learn more

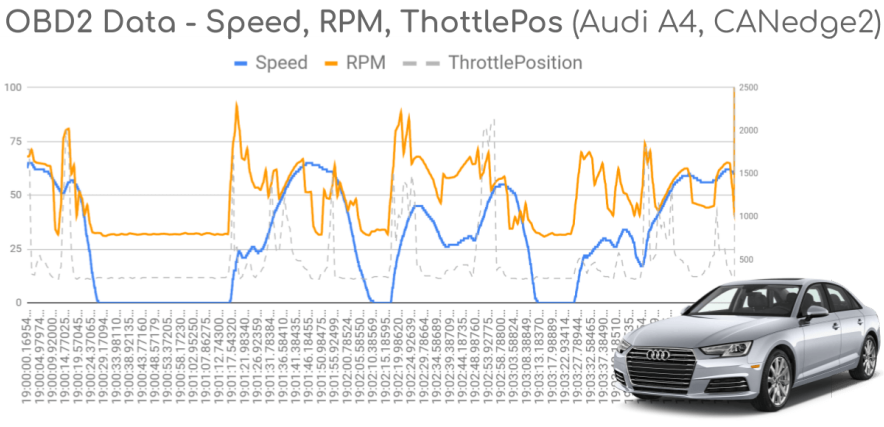
Predictive maintenance
Vehicles and machinery can be monitored via IoT CAN loggers in the cloud to predict and avoid breakdowns
learn more



Vehicle/machine blackbox
A CAN logger can serve as a 'blackbox' for vehicles or equipment, providing data for e.g. disputes or diagnostics
learn more

How to log CAN bus data

As mentioned, two CAN fields are important for CAN logging: The CAN ID and the Data. To record CAN data you need a CAN logger. This lets you log timestamped CAN data to an SD card. In some cases, you need a CAN interface to stream data to a PC - e.g. for car hacking.



Connecting to the CAN bus

This first step is to connect your CAN logger to your CAN bus. Typically this involves using an adapter cable:

- **Cars**: In most cars, you simply use an OBD2 adapter to connect. In most cars, this will let you log raw CAN data, as well as perform requests to log OBD2 or UDS (Unified Diagnostic Services) data
- **Heavy duty vehicles**: To log J1939 data from trucks, excavators, tractors etc you can typically connect to the J1939 CAN bus via a standard J1939 connector cable (deutsch 9-pin)
- **Maritime**: Most ships/boats use the NMEA 2000 protocol and enable connection via an M12 adapter to log marine data
- **CANopen**: For CANopen logging, you can often directly use the CiA 303-1 DB9 connector (i.e. the default connector for our CAN loggers), optionally with a CAN bus extension cable
- **Contactless**: If no connector is available, a typical solution is to use a contactless CAN reader - e.g. the CANCrocodile. This lets you log data directly from the raw CAN twisted wiring harness, without direct connection to the CAN bus (often useful for warranty purposes)
- **Other**: In practice, countless other connectors are used and often you'll need to create a custom CAN bus adapter - here a generic open-wire adapter is useful

When you've identified the right connector and verified the pin-out, you can connect your CAN logger to start recording data. For the CANedge/CLX000, the CAN baud rate is auto-detected and the device will start logging raw CAN data immediately.

Example: Raw CAN sample data (J1939)

You can optionally download raw OBD2 and J1939 samples from the CANedge2 in our intro docs. You can e.g. load this data in the free CAN bus decoder software tools. Data from the CANedge is recorded in the popular binary format, MF4, but can be converted to any file format via our simple MF4 converters (e.g. to CSV, ASC, TRC, ...).

Below is a CSV example of raw CAN frames logged from a heavy-duty truck using the J1939 protocol. Notice that the CAN IDs and data bytes are in hexadecimal format:

```
TimestampEpoch;BusChannel;ID;IDE;DLC;DataLength;Dir;EDL;BRS;DataBytes
1578922367.777150;1;14FEF131;1;8;8;0;0;0;CFFFFFF300FFFF30
1578922367.777750;1;10F01A01;1;8;8;0;0;0;2448FFFFFFFFFFF
1578922367.778300;1;CF00400;1;8;8;0;0;0;107D82BD1200F482
1578922367.778900;1;14FF0121;1;8;8;0;0;0;FFFFFFFFFFFFFCFF
...
```

Example: CANedge CAN logger

The CANedge1 lets you easily record data from any CAN bus to an 8-32 GB SD card. Simply connect it to e.g. a car or truck to start logging - and decode the data via free software/APIs. Further, the CANedge2 (WiFi) and CANedge3 (3G/4G) let you auto-transfer data to your own server - and update devices over-the-air.

How to decode raw CAN data to 'physical values'

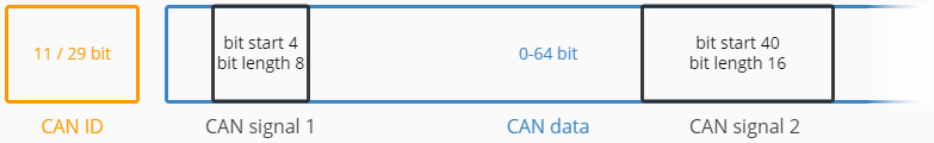
If you review the raw CAN bus data sample above, you will probably notice something: Raw CAN bus data is not human-readable.

To interpret it, you need to decode the CAN frames into scaled engineering values aka physical values (km/h, degC, ...).

Below we show step-by-step how this works.

Extracting CAN signals from raw CAN frames

Each CAN frame on the bus contains a number of CAN signals (parameters) within the CAN databytes. For example, a CAN frame with a specific CAN ID may carry data for e.g. 2 CAN signals.



To extract the physical value of a CAN signal, the following information is required:

- Bit start: Which bit the signal starts at
- Bit length: The length of the signal in bits
- Offset: Value to offset the signal value by
- Scale: Value to multiply the signal value by

To extract a CAN signal, you 'carve out' the relevant bits, take the decimal value and perform a linear scaling: physical_value = offset + scale * raw_value_decimal

The challenge of proprietary CAN data

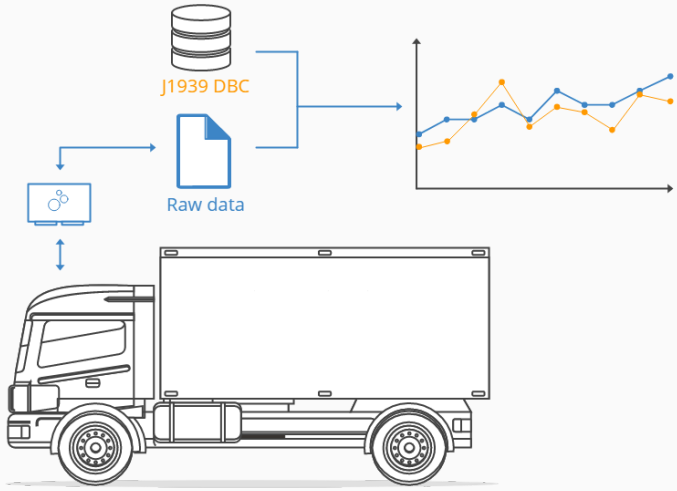
Most often, the CAN bus "decoding rules" are proprietary and not easily available (except to the OEM, i.e. Original Equipment Manufacturer). There are a number of solutions to this when you're not the OEM:

- **Record J1939 data**: If you're logging raw data from heavy duty vehicles (trucks, tractors, ...), you're typically recording J1939 data. This is standardized across brands - and you can use our J1939 DBC file to decode data. See also our J1939 data logger intro
- **Record OBD2/UDS data**: If you need to log data from cars, you can typically request OBD2/UDS data, which is a standardized protocol across cars. For details see our OBD2 data logger intro and our free OBD2 DBC file
- **Use public DBC files**: For cars, online databases exist where others have reverse engineered proprietary some of the CAN data. We keep a list of such databases in our DBC file intro
- **Reverse engineer data**: You can also attempt to reverse engineer data yourself by using a CAN bus sniffer, though it can be time consuming and difficult

- **Use sensor modules**: In some cases you may need sensor data that is not available on the CAN bus (or which is difficult to reverse engineer). Here, sensor-to-CAN modules like the CANmod series can be used. You can integrate such modules with your CAN bus, or use them as add-ons for your CAN logger to add data such as GNSS/IMU or temperature data
- **Partner with the OEM**: In some cases the OEM will provide decoding rules as part of the CAN bus system technical specs. In other cases you may be able to get the information through e.g. a partnership

CAN database files (DBC) - J1939 example

In some cases, conversion rules are standard across manufacturers - e.g. in the J1939 protocol for heavy-duty.
This means that you can use the J1939 parameter conversion rules on practically any heavy-duty vehicle to convert a large share of your data. To make this practical, you need a format for storing the conversion rules. Here, the CAN database (DBC) format is the industry standard - and is supported by most CAN bus decoder software incl. the supporting tools for our CAN loggers, asammdf and CANvas. We also offer a low cost J1939 DBC file, which you can purchase as a digital download. With this, you can get quickly from raw J1939 data to human-readable form. Learn more!



Example: Decoded CAN sample data (physical values)

To illustrate how you can extract CAN signals from raw CAN data frames, we include below the previous J1939 sample data - but now decoded via a J1939 DBC file using the asammdf GUI tool.
As evident, the result is timeseries data with parameters like oil temperature, engine speed, GPS, fuel rate and speed:

```
timestamps,ActualEnginePercentTorque,EngineSpeed,EngineCoolantTemperature,EngineOilTemperature1,EngineFu
elRate,EngineTotalIdleHours,FuelLevel1,Latitude,Longitude,WheelBasedVehicleSpeed
2020-01-13 16:00:13.259449959+01:00,0,1520.13,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.268850088+01:00,0,1522.88,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.270649910+01:00,0,1523.34,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.271549940+01:00,0,1523.58,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.278949976+01:00,0,1525.5,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
…
```

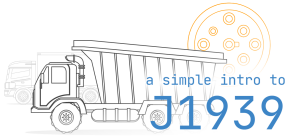
For more on logging J1939 data, see our J1939 data logger and mining telematics articles. You can also learn how to analyze/visualize your CAN data via the free asammdf GUI tool or telematics dashboards.

What is the link between CAN, J1939, OBD2, CANopen ...?

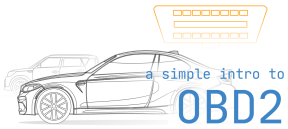
The Controller Area Network provides the basis for communication - but not a lot more.

For example, the CAN standard does not specify how to handle messages larger than 8 bytes - or how to decode the raw data. Therefore a set of **standardized protocols** exist to further specify how data is communicated between CAN nodes of a given network.

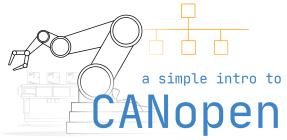
Some of the most common standards include SAE J1939, OBD2 and CANopen. Further, these higher-layer protocols will increasingly be based on the 'next generation' of CAN, CAN FD (e.g. CANopen FD and J1939-17/22).



SAE J1939
J1939 is the standard in-vehicle network for heavy-duty vehicles (e.g. trucks & buses). J1939 parameters (e.g. RPM, speed, ...) are identified by a suspect parameter number (SPN), which are grouped in parameter groups identified by a PG number (PGN).
j1939 intro
j1939 telematics

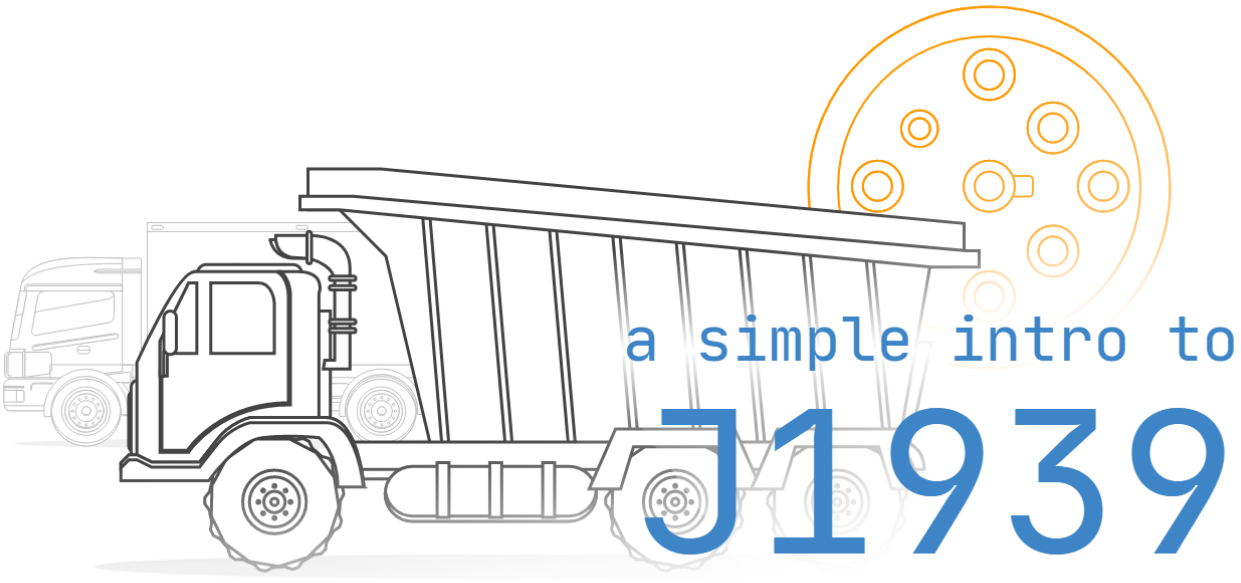


OBD2
On-board diagnostics (OBD, ISO 15765) is a self-diagnostic and reporting capability that e.g. mechanics use to identify car issues. OBD2 specifies diagnostic trouble codes (DTCs) and real-time data (e.g. speed, RPM), which can be recorded via OBD2 loggers.
obd2 intro
obd2 logging



CANopen
CANopen is used widely in embedded control applications, incl. e.g. industrial automation. It is based on CAN, meaning that a CAN bus data logger is also able to log CANopen data. This is key in e.g. machine diagnostics or optimizing production.
canopen intro
canopen logger

CAN FD
CAN bus with flexible data-rate (CAN FD) is an extension of the Classical CAN data link layer. It increases the payload from 8 to 64 bytes and allows for a higher data bit rate, dependent on the CAN transceiver. This enables increasingly data-intensive use cases like EVs.
can fd intro
can fd logger



J1939 Explained - A Simple Intro

In this guide we introduce the J1939 protocol basics incl. PGNs and SPNs. This is a practical intro so you will also learn how to decode J1939 data via DBC files, how J1939 logging works, key use cases and practical tips.

What is J1939?

In short, SAE J1939 is a set of standards that define how ECUs communicate via the CAN bus in heavy-duty vehicles.
As explained in our CAN bus intro, most vehicles today use the Controller Area Network (CAN) for ECU communication.
However, CAN bus only provides a "basis" for communication (like a telephone) - not a "language" for conversation.

In most heavy-duty vehicles, this language is the SAE J1939 standard defined by the Society of Automotive Engineers (SAE). In more technical terms, J1939 provides a higher layer protocol (HLP) based on CAN as the "physical layer".
What does that mean, though?

One standard across heavy-duty vehicles

In simple terms, J1939 offers a standardized method for communication across ECUs, or in other words:
J1939 provides a common language across manufacturers.
In contrast, e.g. cars use proprietary OEM specific protocols.

J1939 application examples

Heavy-duty vehicles (e.g. trucks and buses) is one of the most well-known applications. However, several other key industries leverage SAE J1939 today either directly or via derived standards (e.g. ISO 11783, MilCAN, NMEA 2000, FMS):

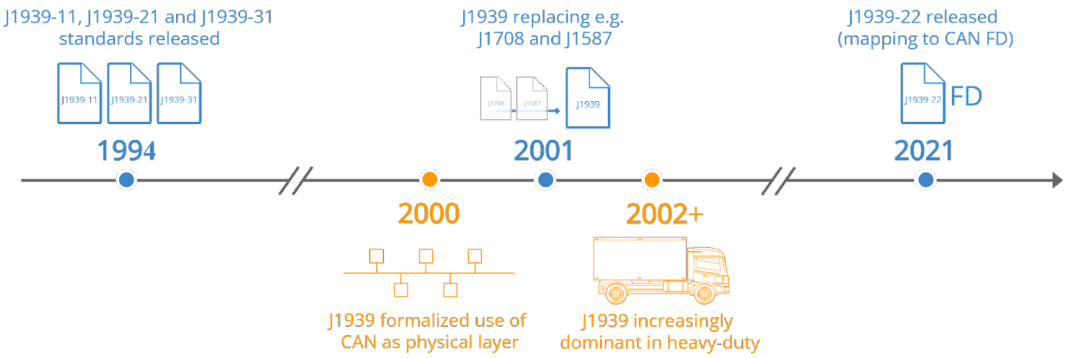
- Foresting machines (e.g. delimbers, forwarders, skidders)
- Mining vehicles (e.g. bulldozers, draglines, excavators, ...)
- Military vehicles (e.g. tanks, transport vehicles, ...)
- Agriculture (e.g. tractors, harvesters, ...)

- Construction (e.g. mobile hydraulics, cranes, ...)
- Fire & Rescue (e.g. ambulances, fire trucks, ...)
- Other (e.g. ships, pumping, e-buses, power generation, ...)

J1939 history & future

History

- 1994: First docs were released (J1939-11, J1939-21, J1939-31)
- 2000: The initial top level document was published
- 2000: CAN formally included as part of J1939 standard
- 2001: J1939 starts replacing former standards SAE J1708/J1587



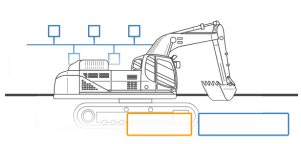
Future

With the rise of heavy-duty telematics, J1939 will increasingly play a role in the market for connected vehicles. In turn, this will increase the need for secure J1939 IoT loggers. In parallel, OEMs will increasingly shift from Classical CAN to CAN FD as part of the transition to J1939 with flexible data-rate. In turn, this will increase the need for J1939 FD data loggers.

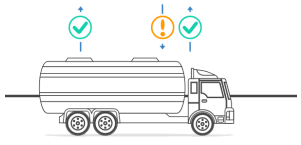
"The market for in-vehicle connectivity - the hardware and services bringing all kinds of new functionality to drivers and fleet owners - is expected to reach EUR 120 billion by 2020."
- Boston Consulting Group, Connected Vehicles and the Road to Revenue

4 key characteristics of J1939

The J1939 protocol has a set of defining characteristics outlined below:

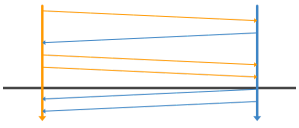


250K baud rate & 29-bit extended ID
The J1939 baud rate is typically 250K (though recently with support for 500K) - and the identifier is extended 29-bit (CAN 2.0B)



Broadcast + on-request data
Most J1939 messages are broadcast on the CAN-bus, though some data is only available by requesting the data via the CAN bus

PGN identifiers & SPN parameters
J1939 messages are identified by 18-bit Parameter Group Numbers (PGN), while J1939 signals are called Suspect Parameter Numbers (SPN)



Multibyte variables & Multi-packets
Multibyte variables are sent least significant byte first (Intel byte order). PGNs with up to 1785 bytes are supported via J1939 transport protocol

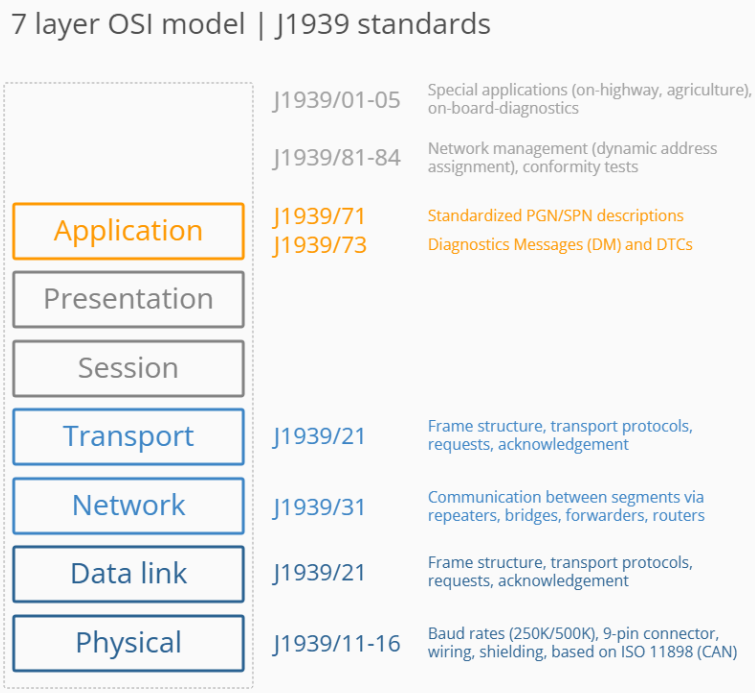
Additional J1939 characteristics

Below are a set of additional characteristics of the J1939 protocol:

- Reserved: J1939 includes a large range of standard PGNs, though PGNs 00FF00 through 00FFFF are reserved for proprietary use
- Special Values: A data byte of 0xFF (255) reflects N/A data, while 0xFE (254) reflects an error
- J1939 address claim: The SAE J1939 standard defines a procedure for assigning source addresses to J1939 ECUs after network initialization via an 8-bit address in a dynamic way

Technical: J1939 'higher layer protocol' explained

J1939 is based on CAN, which provides the basic "physical layer" and "data link layer", the lowest layers in the OSI model. Basically, CAN allows the communication of small packets on the CAN bus, but not a lot more than that. Here, J1939 serves as a higher layer protocol on top, enabling more complex communication.



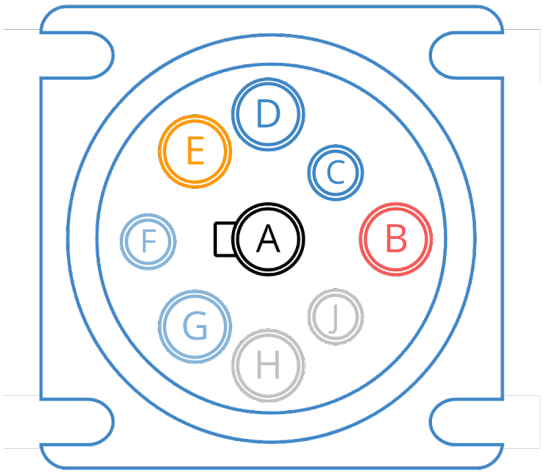
A higher layer protocol enables communication across the large complex networks of e.g. vehicle manufacturers.

For example, the SAE J1939 protocol specifies how to handle "multi-packet messages", i.e. when data larger than 8 bytes needs to be transferred. Similarly, it specifies how data is to be converted into human-readable data.
It does so by providing a family of standards. For example, J1939-71 is a document detailing the information required to convert a large set of cross-manufacturer standardized J1939 messages into human-readable data (more on this below). Many other CAN based higher layer protocols exist, e.g. CANopen, DeviceNet, Unified Diagnostic Services. These typically offer some level of standardization within their respective industries - though all of them can be extended by manufacturers.

In comparison, the aforementioned passenger cars have unique standards per manufacturer. In other words, you can use the same J1939 database file to convert e.g. engine speed across two trucks from different manufacturers - but you cannot e.g. read the data from an Audi A4 using the same IDs & scaling parameters as for a Peugeot 207.

The J1939 connector (9-pin)

The J1939-13 standard specifies the 'off-board diagnostic connector' - also known as the J1939 connector or 9-pin deutsch connector. This is a standardized method for interfacing with the J1939 network of most heavy duty vehicles - see the illustration for the J1939 connector pinout.



A	Ground
B	Battery power
C	CAN 1 H
D	CAN 1 L
E	CAN shield
F	J1708 (+) / CAN 2 H
G	J1708 (-) / CAN 2 L
H	OEM specific
J	OEM specific

Type 1 (black): CAN 1 = 250K
Type 2 (green): CAN 1 = 500K

Black type 1 vs green type 2

Note that the J1939 deutsch connector comes in two variants: The original black connector (type 1) and the newer green connector (type 2), which started getting rolled out around 2013-14.

J1939 type 2 female connectors are physically backwards compatible, while type 1 female connectors only fit type 1 male sockets. The type 2 connector was designed for the SAE J1939-14 standard, which adds support for 500K bit rates. The purpose of "blocking" type 1 connectors is to ensure that older hardware (presumably using 250K bit rates) is not connected to type 2 500K bit rate J1939 networks. Specifically, the physical block is through a smaller hole for pin F in the type 2 male connectors. See also the example of a DB9-J1939 adapter cable (type 2).

Multiple J1939 networks

As evident, the J1939 deutsch connector provides access to the J1939 network through pins C (CAN high) and D (CAN low). This makes it easy to interface with the J1939 network across most heavy duty vehicles.

In some cases, however, you may also be able to access a secondary J1939 network through pins F and G or pins H and J (with H being CAN H and J being CAN L).

Many of today's heavy duty vehicles have 2 or more parallel CAN bus networks and in some cases at least two of these will be available through the same J1939 connector. This also means that you will not necessarily have gained access to all the available J1939 data if you've only attempted to interface through the 'standard' pins C and D.

Other heavy duty connectors

While the J1939 deutsch connector is the most common way to interface with the J1939 network of heavy duty vehicles, other connectors of course exist. Below are some examples:

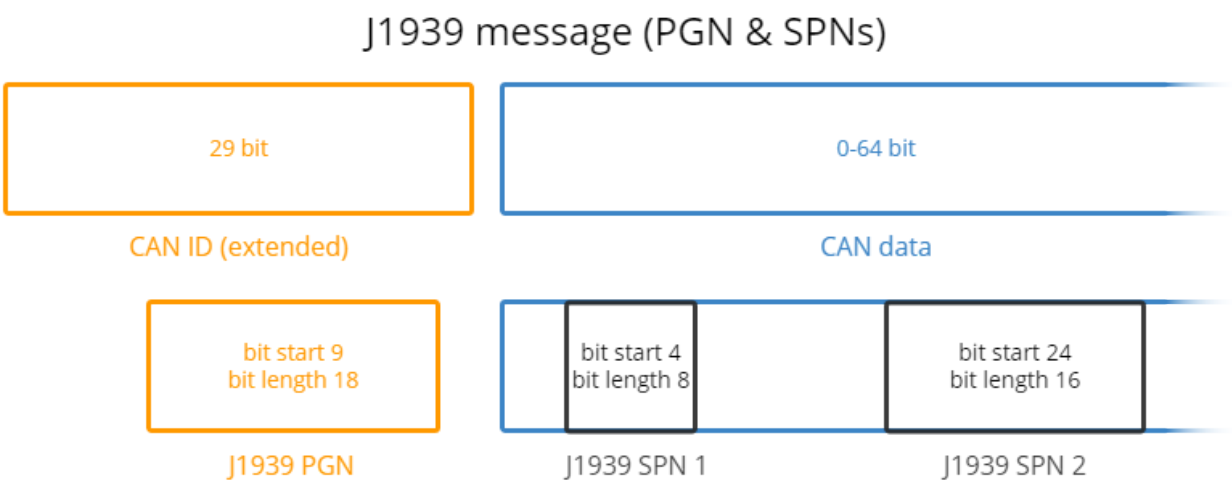
- J1939 Backbone Connector: This 3-pin deutsch connector provides pins for CAN H/L a CAN shield (no power/ground)
- CAT connector: The Caterpillar industrial connector is a grey 9-pin deutsch connector. However, the pin-out differs from the J1939 connector (A: Power, B: Ground, F: CAN L, G: CAN H) and the connector physically blocks access from standard type 1 and 2 J1939 connectors
- OBD2 type B connector: The type B OBD2 connector (SAE J1962) in heavy duty vehicles sometimes provide direct access to the J1939 network through pins 6 and 14
- Volvo 2013 OBD2 connector: This variant matches the type B OBD2 connector, but also adds access to the J1939 high via pin 3 and J1939 low via pin 11

The J1939 PGN and SPN

In the following section we explain the J1939 PGNs and SPNs.

Parameter Group Number (PGN)

The J1939 PGN comprises an 18-bit subset of the 29-bit extended CAN ID. In simple terms, the PGN serves as a unique frame identifier within the J1939 standard. For example, you can look this up in the J1939-71 standard documentation, which lists PGNs/SPNs.



Example: J1939 PGN 61444 (EEC1)

Assume you recorded a J1939 message with HEX ID 0CF00401. Here, the PGN starts at bit 9, with length 18 (indexed from 1). The resulting PGN is 0F004 or in decimal 61444. Looking this up in the SAE J1939-71 documentation, you will find that it is the "Electronic Engine Controller 1 - EEC1". Further, the document will have details on the PGN including priority, transmission rate and a list of the associated SPNs - cf. the illustration. For this PGN, there are seven SPNs (e.g. Engine Speed, RPM), each of which can be looked up in the J1939-71 documentation for further details.

PGN61444 - Electronic Engine Controller 1 - EEC1

Transmission Repetition Rate: Engine speed dependent
Data Length: 8 bytes
Data Page: 0
PDU Format: 240
PDU Specific: 4
Default Priority: 3
Parameter Group Number: 61444 (0x00F004)

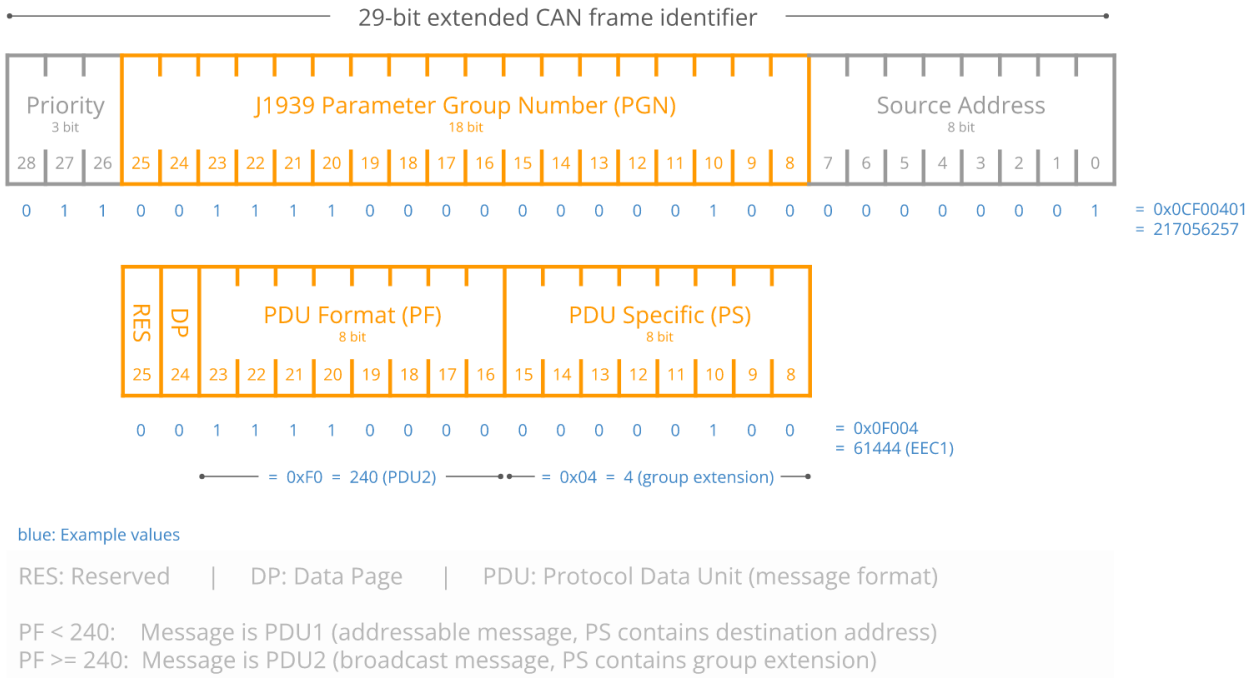
Bit Start/Byte	Length	SPN ID	SPN Description
1.1	4 bits	899	Engine Torque Mode
2	1 byte	512	Driver's Demand Engine - % Torque
3	1 byte	513	Actual Engine - Percent Torque
4-5	2 bytes	190	Engine Speed
6	1 byte	1483	SA of Controlling Device for Engine Control
7.1	4 bits	1675	Engine Starter Mode
8	1 byte	2432	Engine Demand - Percent Torque

Detailed breakdown of the J1939 PGN

Let's look at the CAN ID to PGN transition in detail. Specifically, the 29 bit CAN ID comprises the Priority (3 bits), the J1939 PGN (18 bits) and the Source Address (8 bits). In turn, the PGN can be split into the Reserved Bit (1 bit), Data Page (1 bit), PDU format (8 bit) and PDU Specific (8 bit).

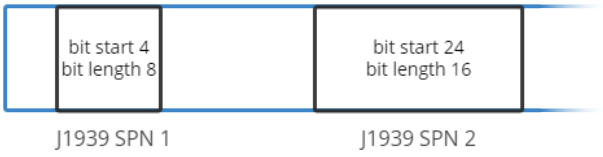
The detailed PGN illustration also includes example values for each field in binary, decimal and hexadecimal form.

To learn more about the transition from 29 bit CAN ID to 18 bit J1939 PGN, see also our online CAN ID to J1939 PGN converter. The converter also includes a full J1939 PGN list for PGNs included in our J1939 DBC file.



Suspect Parameter Number (SPN)

The J1939 SPN serves as the identifier for the CAN signals (parameters) contained in the databytes. SPNs are grouped by PGNs and can be described in terms of their bit start position, bit length, scale, offset and unit - information required to extract and scale the SPN data to physical values.



Example: Extracting J1939 SPN 190 (Engine Speed)

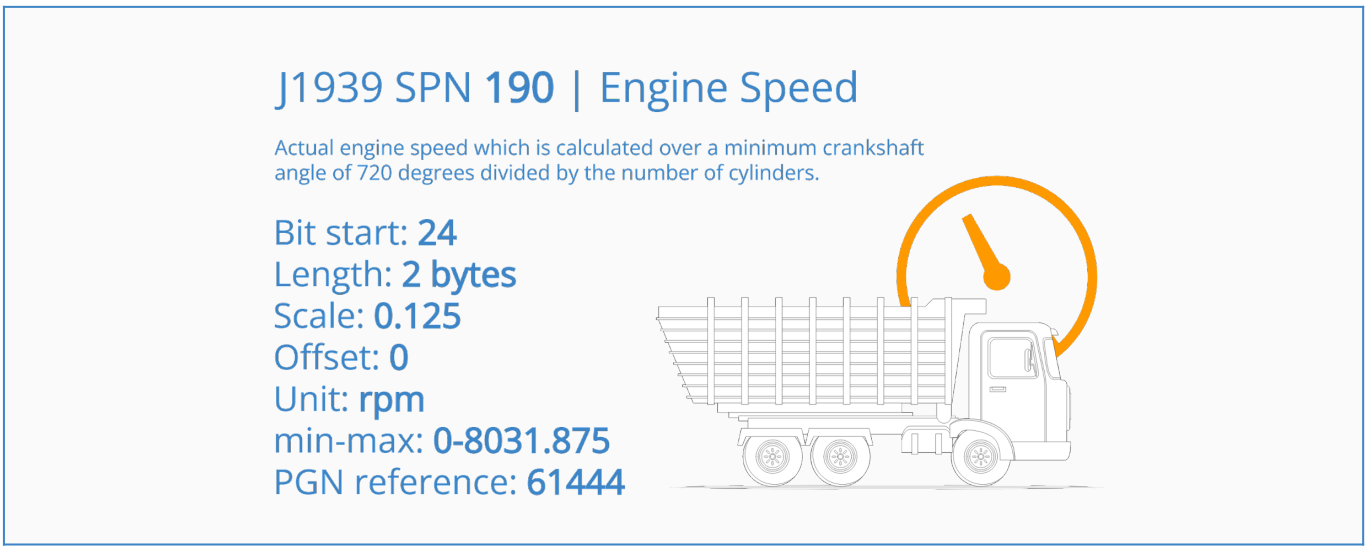
Assume you have recorded a raw J1939 frame as below:

CAN ID	Data bytes
0CF00401	FF FF FF 68 13 FF FF FF

By converting the CAN ID to the J1939 PGN you identify that this is the PGN 61444 from before. From the J1939-71 document, you observe that one of the SPNs within this PGN is Engine Speed (SPN 190) with details as in the illustration below.

Using these details, it is possible to extract the Engine Speed physical value data e.g. for plot purposes. To do so, note from the SPN info that the relevant data is in bytes 4 and 5, i.e. the HEX data bytes 68 and 13. Taking the decimal form of the HEX value 1368 (Intel byte order), we get 4968 in decimal. To arrive at the RPM, we conduct a scaling of this value using the offset 0 and the scale 0.125 RPM/bit. The physical value (aka scaled engineering value) is 621 RPM.

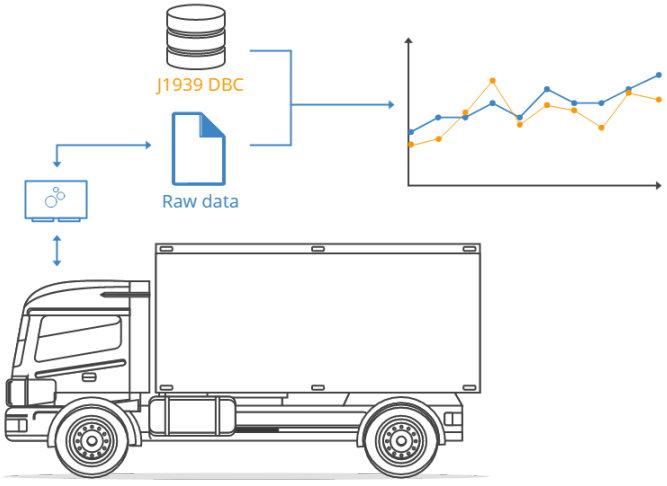
Note how some data bytes in the above are FF or 255 decimal, i.e. not available. While the PGN may theoretically support SPNs in this range, the FF padding means that this particular application does not support these parameters.



In practice, you will not 'PDF-lookup' rules for J1939 data - instead, this info is stored in a CAN database file (DBC).

Example: J1939 DBC file

A J1939 DBC file can be used to decode data across most heavy-duty vehicles. For example, raw J1939 data can be recorded with a CAN bus data logger and analyzed in a CAN software tool that supports DBC conversion (e.g. asammdf). This will typically result in a conversion of 40-60% of the vehicle data - with the rest being OEM specific proprietary data that requires reverse engineering.



J1939 truck sample data: Raw & physical values

Below we illustrate what real J1939 data looks like. The 'raw' J1939 data was recorded from a heavy duty truck using a CANedge2, while the 'physical values' reflect the output after decoding the raw data via the free asammdf software and the J1939 DBC.

Sample: Raw J1939 truck data (CSV)

Data from the CANedge is recorded in a standardized binary format, MDF4, which can be converted to any file format via our MDF4 converters (e.g. to CSV, ASC, TRC, ...). Below is a CSV version of the raw J1939 frames. Notice that the CAN IDs and data bytes are in hexadecimal format:

```
TimestampEpoch;BusChannel;ID;IDE;DLC;DataLength;Dir;EDL;BRS;DataBytes
1578922367.777150;1;14FEF131;1;8;8;0;0;0;CFFFFFF300FFFF30
1578922367.777750;1;10F01A01;1;8;8;0;0;0;2448FFFFFFFFFFF
1578922367.778300;1;CF00400;1;8;8;0;0;0;107D82BD1200F482
1578922367.778900;1;14FF0121;1;8;8;0;0;0;FFFFFFFFFFFFFCFF
1578922367.779500;1;18F0000F;1;8;8;0;0;0;007DFFFF0F7DFFFF
1578922367.780050;1;18FFA03D;1;8;8;0;0;0;2228240019001AFF
1578922367.780600;1;10FCFD01;1;8;8;0;0;0;FFFFFFFF1623FFFF
1578922367.781200;1;18FD9401;1;8;8;0;0;0;A835FFFFA9168F03
1578922367.781750;1;18FDA101;1;8;8;0;0;0;1224FFFFFFFF00FF
1578922367.782350;1;18F00E3D;1;8;8;0;0;0;741DFFFFFFFFFFF
1578922367.782950;1;18F00F3D;1;8;8;0;0;0;B40FFFFFFFFFFFF
1578922367.783500;1;10FDA301;1;8;8;0;0;0;FFFFFFFFFFFFFFFF
...
```

You can optionally download full raw J1939 MDF4 samples from the CANedge2 in our intro docs. The sample data also includes a demo J1939 DBC so that you can replicate the conversion steps via asammdf.

Sample: Decoded physical values J1939 truck data (CSV)

Once the raw J1939 data is decoded and exported, the result is timeseries data with parameters like oil temperature, engine speed, GPS, fuel rate and speed:

```
timestamps,ActualEnginePercentTorque,EngineSpeed,EngineCoolantTemperature,EngineOilTemperature1,EngineFu
elRate,EngineTotalIdleHours,FuelLevel1,Latitude,Longitude,WheelBasedVehicleSpeed
2020-01-13 16:00:13.259449959+01:00,0,1520.13,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.268850088+01:00,0,1522.88,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.270649910+01:00,0,1523.34,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.271549940+01:00,0,1523.58,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.278949976+01:00,0,1525.5,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.289050102+01:00,0,1527.88,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.299000025+01:00,0,1528.13,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.308300018+01:00,0,1526.86,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.309099913+01:00,0,1526.75,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.317199945+01:00,0,1526.45,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
...
```

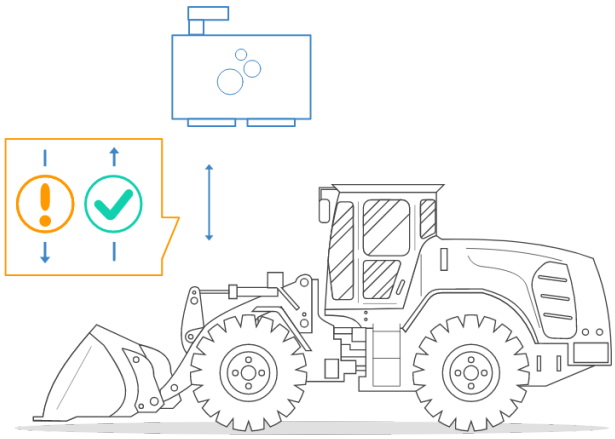
For more on logging J1939 data, see our J1939 data logger and mining telematics articles.

About the CANedge J1939 logger

The CANedge lets you easily record J1939 data to an 8-32 GB SD card. Simply connect it to e.g. a truck to start logging - and decode the data via free software/APIs and our J1939 DBC. Learn more.

J1939 request messages

Most J1939 messages are broadcast via the CAN bus, but some are only sent "on-request" (e.g. when polled by a J1939 data logger). On-request data often includes J1939 diagnostic trouble codes (DTCs), making it important in vehicle diagnostics. Below we briefly outline how it works:



Sending J1939 request messages

To send a J1939 request via the CAN bus, a special 'request message' is used (PGN 59904), which is the only J1939 message with only 3 bytes of data. It has priority 6, a variable transmit rate and can either be sent as a global or specific address request. The data bytes 1-3 should contain the requested PGN (Intel byte order). Examples of requested J1939 messages include the diagnostic messages (e.g. J1939 DM2).

A CAN bus data logger like the CANedge can be set up to send J1939 request messages - see e.g. our CANedge Intro for a detailed step-by-step guide.

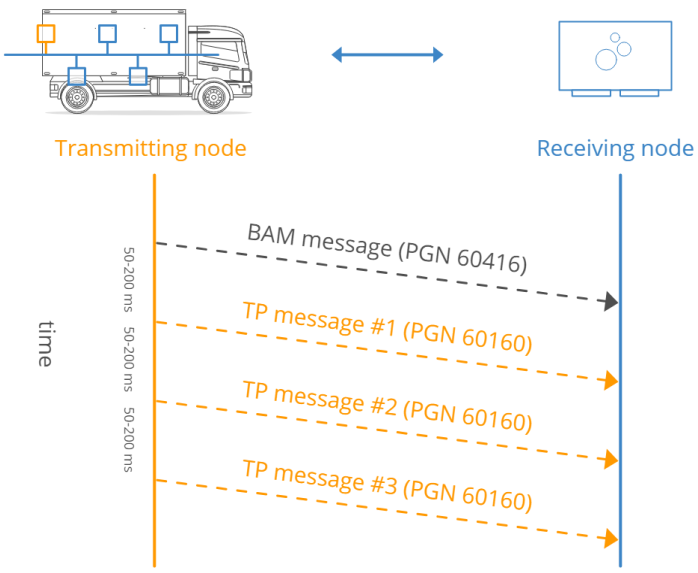
J1939 code requests vs warranty compliance
Sending request messages is typically key to requesting J1939 codes and thus J1939 diagnostics. One challenge, however, is that J1939 loggers are often required to have a contactless connection to the J1939 data link, meaning that they're unable to interact with the CAN bus and transmit J1939 request frames. The restriction is often related to warranty compliance as some vehicle manufacturers do not allow direct access by 3rd party devices via the J1939 connector.

In some cases, it is required that the J1939 analyzer is "physically" contactless, e.g. via a CANcrocodile adapter. In other cases, it is sufficient that the J1939 logger operates in "configurable" silent mode. The latter makes it easier to perform ad hoc requests for J1939 fault codes, either via a manual configuration update or via an over-the-air update for the CANedge2.

J1939 transport protocol (TP)

The previous PGN and SPN examples are based on J1939 messages with 8 data bytes. While these are most common, J1939 multi-frame messages also exist with >8 data bytes - sent via the J1939 transport protocol.

Below we outline how the J1939 transport protocol works, a practical J1939 TP data example and how to decode multi-frame J1939 messages via DBC files:



How does the J1939 transport protocol work?

The J1939 protocol specifies how to deconstruct, transfer and reassemble packets across multiple frames - a process referred to as the J1939 Transport Protocol (see J1939-21). Two types of the J1939 TP exist:

1. The Connection Mode (intended for a specific device)
2. The BAM (Broadcast Announce Message) which is intended for the entire network

For example, a transmitting ECU may send an initial BAM packet to set up a data transfer. The BAM specifies the PGN identifier for the multi-packet message as well as the number of data bytes and packets to be sent. It is then followed by up to 255 packets/frames of data. Each of the 255 packets use the first data byte to specify the sequence number (1 up to 255), followed by 7 bytes of data. The max number of bytes per multi-packet message is therefore 7 bytes x 255 = 1785 bytes.

The final packet contains at least one byte of data, followed by unused bytes set to FF. In the BAM type scenario, the time between messages is 50-200 ms. In post processing, a conversion software tool can reassemble the multiple entries of 7 data bytes into a single payload and handle it according to the multi-packet PGN and SPN specifications as found in e.g. a J1939 DBC file.

A practical J1939 transport protocol example

Decoding J1939 multiframe data is more complex than decoding standard J1939 frames. To understand why, consider the below example of a J1939 transport protocol response, recorded with the CANedge2:

```
TimestampEpoch;BusChannel;ID;IDE;DLC;DataLength;Dir;EDL;BRS;DataBytes
1605078459.438750;1;1CECFF00;1;8;8;0;0;0;20270006FFE3FE00
1605078459.498750;1;1CEBFF00;1;8;8;0;0;0;013011B2A041B240
1605078459.559750;1;1CEBFF00;1;8;8;0;0;0;021FB2102CB2603B
1605078459.618750;1;1CEBFF00;1;8;8;0;0;0;03B230430000D309
1605078459.678750;1;1CEBFF00;1;8;8;0;0;0;04C0441E37967DE1
1605078459.738750;1;1CEBFF00;1;8;8;0;0;0;05E02E7B02FFFF80
1605078459.799850;1;1CEBFF00;1;8;8;0;0;0;06E0FFFFFFFFFFFF
```

The above sequence consists of two J1939 message types:

PGN 60416 - J1939 Transport Protocol BAM (Connection Management)

Data Length: 8 bytes
Default Priority: 7
Parameter Group Number: 60416 (0xEC00)

Byte	Description
1	Fixed at 32
2-3	Message size in bytes
4	Number of packets
5	Reserved (filled with FF)
6-8	PGN

PGN 60160 - J1939 Transport Protocol (Data Transfer)

Data Length: 8 bytes
Default Priority: 7
Parameter Group Number: 60160 (0xEB00)

Byte	Description
1	Sequence number (1 to 255)
2-8	Data payload (unused locations in the last frame are set to FF)

A J1939 BAM message with ID 1CECFF00 (PGN 60416 or EC00), which contains the response data length and J1939 PGN - and a J1939 data transfer messages with ID 1CEBFF00 (PGN 60160 or EB00). These contain the payload across multiple frames.

Below we break down the J1939 transport protocol example with focus on the data byte interpretation:

Timestamp (Epoch)	CAN ID	PGN (HEX)	PGN (DEC)	DataBytes
1605078459.438750	1CECFF00	EC00	60416	20270006FFE3FE00
1605078459.498750	1CEBFF00	EB00	60160	013011B2A041B240
1605078459.559750	1CEBFF00	EB00	60160	021FB2102CB2603B
1605078459.618750	1CEBFF00	EB00	60160	03B230430000D309
1605078459.678750	1CEBFF00	EB00	60160	04C0441E37967DE1
1605078459.738750	1CEBFF00	EB00	60160	05E02E7B02FFFF80
1605078459.799850	1CEBFF00	EB00	60160	06E0FFFFFFFFFFFF
1605078459.438750	18FEE3FE	FEE3	65251	3011B2A041B240 ... E0FFFFFF

legend
control byte (0x20=BAM)
#data bytes (0x0027=39)
#packets (0x06=6)
reserved
J1939 PGN (0x00FEE3=65251)
sequence numbers
payload data (39 bytes)
unused (3 bytes)

Generally, a J1939 transport protocol response sequence can be processed as follows:

- Identify the BAM frame, indicating a new sequence being initiated (via the PGN 60416)
- Extract the J1939 PGN from bytes 6-8 of the BAM payload to use as the identifier of the new frame
- Construct the new data payload by concatenating bytes 2-8 of the data transfer frames (i.e. excl. the 1st byte)

Above, the last 3 bytes of the BAM equal E3FE00. When reordered, these equal the PGN FEE3 aka Engine Configuration 1 (EC1). Further, the payload is found by combining the the first 39 bytes across the 6 data transfer packets/frames.
Note: The last 3 data payload bytes in this practical example happen to be FF, yet we still include these in the payload as the BAM message specifies the data length to be 39. The final 3 FF bytes in the 6th packet are unused.

How to decode a J1939 transport protocol message

With the method explained above, we have created a 'constructed' J1939 data frame with a data length exceeding 8 bytes. This frame can be decoded using a J1939 DBC file, just like a regular J1939 data frame. For the PGN EC1, the J1939 DBC specifies a data length of 40 with signals defined for the full payload.

As such, once the J1939 software/API has reconstructed the multiframe response into a single J1939 frame, the DBC decoding can be done as usual. One minor tweak is that most J1939 DBC files expects that the raw log file of J1939 data will contain 29-bit CAN IDs (not 18-bit J1939 PGNs). As such, if the software embeds the reconstructed J1939 TP frame in the original raw data, it may need to convert the extracted J1939 PGN into a 29-bit CAN ID first. You can also see our J1939 google sheet, which breaks down how a J1939 PGN can be converted to a 29-bit CAN ID.

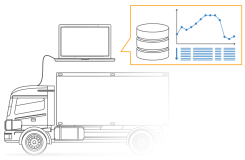
J1939 TP data & Python API decoding
The CANedge lets you request and record J1939 transport protocol data. To decode the TP data, you can either convert the raw log files to another format (like Vector ASC), or you can use our Python API. In our api-examples library on github, we provide a basic example of how J1939 transport protocol data can be reconstructed and DBC decoded, incl.

sample data. Since the CANedge Python API enables decoding of J1939 transport protocol data, J1939 signals from multiframe messages can e.g. be visualized in J1939 telematics dashboards.

Logging J1939 data - example use cases

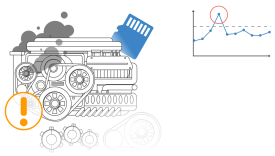
There are several common use cases for recording J1939 data:

Heavy duty fleet telematics
J1939 data from trucks, buses, tractors etc. can be used in fleet management to reduce costs or improve safety
learn more



Live stream diagnostics
By streaming decoded J1939 data to a PC, technicians can perform real-time J1939 diagnostics on vehicles
learn more

Predictive maintenance
Vehicles can be monitored via WiFi CAN loggers in the cloud to predict breakdowns based on the J1939 data
learn more



Heavy-duty vehicle blackbox
A CAN logger can serve as a 'blackbox' for heavy-duty vehicles, providing data for e.g. disputes or J1939 diagnostics
learn more

6 practical tips for J1939 data logging

Many of our end users work with J1939 logging in the field - and below we share 6 practical logging tips:

J1939 logger vs J1939 streaming interface

Standalone J1939 data loggers with SD cards are ideal for logging data from e.g. vehicle fleets over weeks or months. A WiFi J1939 logger also enables telematics use cases. In contrast, a J1939 USB-PC interface requires a PC to stream data from the CAN bus in real-time. This is e.g. useful for diagnostic purposes - or analysing physical events. The CLX000 enables both modes of operation, while the CANedge2 is perfect for telematics.

Direct adapter cable vs contactless reading

To connect your CAN analyzer to a J1939 asset (e.g. a truck) you can typically use the 9-pin J1939 connector. We offer a DB9-J1939 adapter which fits the 9-pin deutsch connector found in many heavy duty vehicles. Alternatively, you may prefer to connect your CAN logger directly to the CAN bus via e.g. a CANCrocodile. This method uses induction to record data silently without cutting any CAN wiring.

WiFi vs. cellular (3G/4G) data upload

For vehicle fleet management & telematics you will typically upload the data via either WiFi or 3G/4G. The CANedge2 lets you transfer data by connecting to a WiFi access point - which can both be a WLAN router or a 3G/4G hotspot. If you need data from a truck on-the-road, you can install the CANedge2 and use it to power a 3G/4G USB hotspot. The benefit to this is that you'll have continuous access to the device - unless it is out-of-coverage. However, in cases where

data only needs to be periodically uploaded an alternative can be to upload data via WLAN routers when the vehicles visit e.g. specific areas (garages, repair shops etc) - letting you reduce data transfer costs.

Software selection & J1939 DBC file

When logging or streaming J1939 data, software for post processing is key. In particular, the software should support DBC-based J1939 conversion to allow easy conversion to human-readable data. The free supporting softwares/APIs for our CAN loggers support this. For USB streaming, our free Wireshark plugin enables live DBC conversion. Further, we offer a digital download J1939 DBC file in collaboration with SAE.

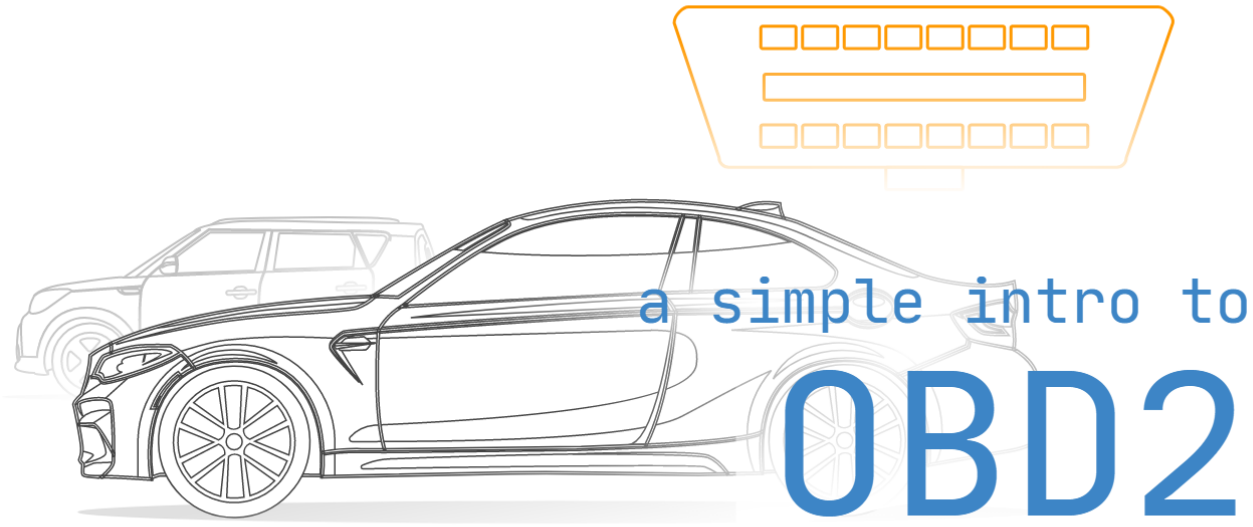
Consider the need for request PGNs

Some J1939 PGNs are only available on-request, meaning that you need to "poll" the CAN bus to log these. The CANedge and CLX000 are able to transmit custom CAN messages, which can be used to send periodic PGN requests. Note that this is not possible in "silent mode" (i.e. it is not possible if the logger is connected via e.g. a CANCrocodile).

Filter, compress and encrypt the data

To optimize your J1939 data logging, a number of advanced configurations can be helpful. In particular, the CANedge advanced filters and sampling rate options help optimize the amount of data logged - key for e.g. minimizing cellular bandwidth usage. Other options include silent mode and cyclical logging, with the latter enabling the logger to always prioritize the latest data (useful in e.g. blackbox logging).

Since J1939 is standardized, it is critical to encrypt your data 'at rest' (e.g. on an SD card) and 'in transit' (during upload). Not doing so exposes your data processing to various security risks, incl. GDPR/CCPA fines and loss of confidentiality and data integrity. For details on securing your J1939 data logging, see our intro to secure CAN logging.

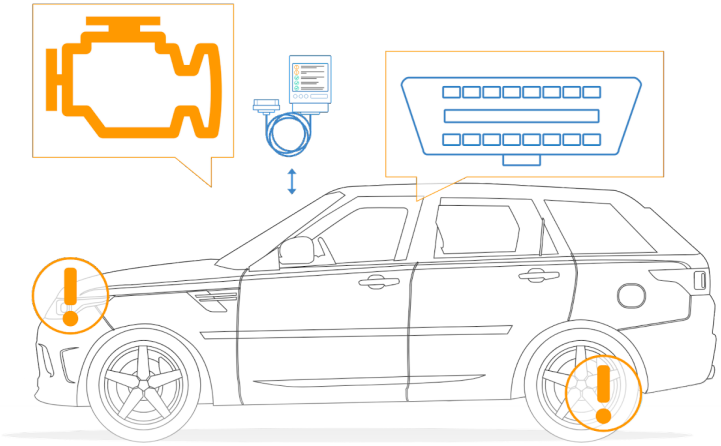


OBD2 Explained - A Simple Intro

In this guide we introduce the On Board Diagnostic (OBD2) protocol incl. the OBD2 connector, OBD2 parameter IDs (PID) and the link to CAN bus. This is a practical intro so you will also learn how to request and decode OBD2 data, key logging use cases and practical tips.

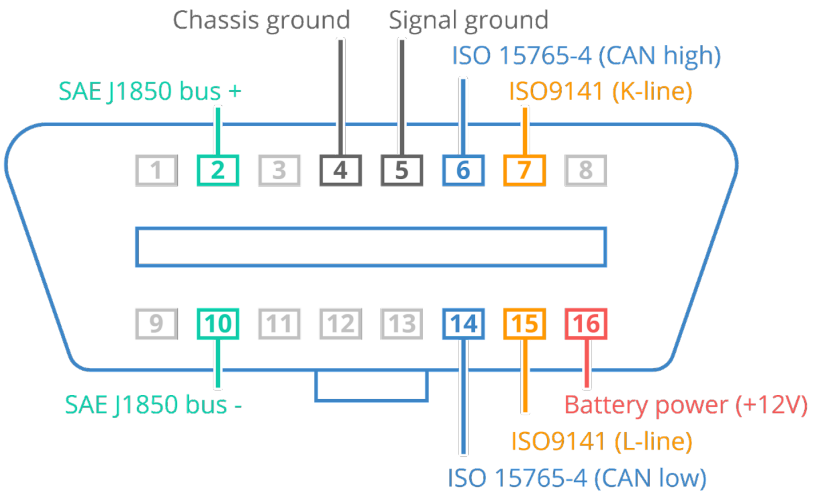
What is OBD2?

In short, OBD2 is your vehicle's built-in self-diagnostic system. You've probably encountered OBD2 already: Ever noticed the malfunction indicator light on your dashboard? That is your car telling you there is an issue. If you visit a mechanic, he will use an OBD2 scanner to diagnose the issue. To do so, he will connect the OBD2 reader to the OBD2 16 pin connector near the steering wheel. This lets him read OBD2 codes aka Diagnostic Trouble Codes (DTCs) to review and troubleshoot the issue.



The OBD2 connector

The OBD2 connector lets you access data from your car easily. The standard SAE J1962 specifies two female OBD2 16-pin connector types (A & B). In the illustration is an example of a Type A OBD2 pin connector (also sometimes referred to as the Data Link Connector, DLC).

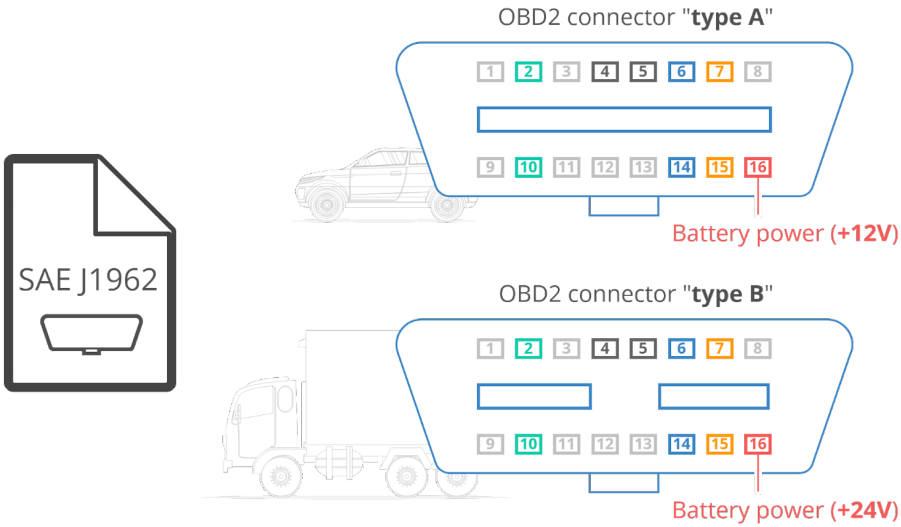


A few things to note:

- The OBD2 connector is near your steering wheel, but may be hidden behind covers/panels
- Pin 16 supplies battery power (often while the ignition is off)
- The OBD2 pinout depends on the communication protocol
- The most common protocol is CAN (via ISO 15765), meaning that pins 6 (CAN-H) and 14 (CAN-L) will typically be connected

OBD2 connector - type A vs. B

In practice, you may encounter both the type A and type B OBD2 connector. Typically, type A will be found in cars, while type B is common in medium and heavy duty vehicles.



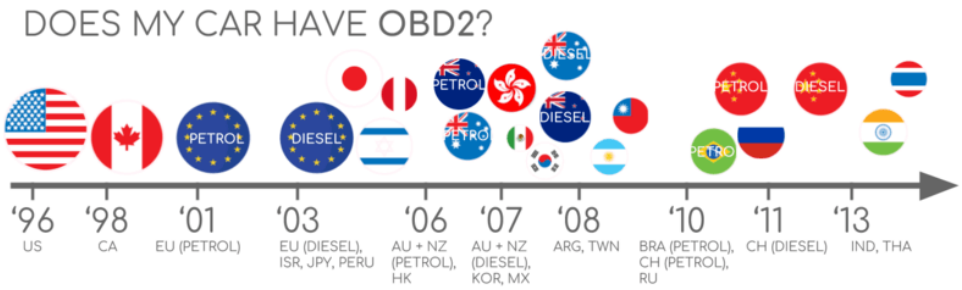
Type A & B connector differences

As evident from the illustration, the two types share similar OBD2 pinouts, but provide two different power supply outputs (12V for type A and 24V for type B). Often the baud rate will differ as well, with cars typically using 500K, while most heavy duty vehicles use 250K (more recently with support for 500K).

To help physically distinguish between the two types of OBD2 sockets, note that the type B OBD2 connector has an interrupted groove in the middle. As a result, a type B OBD2 adapter cable will be compatible with both types A and B, while a type A will not fit into a type B socket.

Does my car have OBD2?

In short: Probably! Almost all newer cars support OBD2 and most run on CAN (ISO 15765). For older cars, be aware that even if a 16 pin OBD2 connector is present, it may still not support OBD2. One way to determine compliance is to identify where & when it was bought new:



Link between OBD2 and CAN bus

On board diagnostics, OBD2, is a 'higher layer protocol' (like a language). CAN is a method for communication (like a phone). In particular, the OBD2 standard specifies the OBD2 connector, incl. a set of five protocols that it can run on (see below). Further, since 2008, CAN bus (ISO 15765) has been the mandatory protocol for OBD2 in all cars sold in the US.

What is the ISO 15765 standard?

ISO 15765 refers to a set of restrictions applied to the CAN standard (which is itself defined in ISO 11898). One might say that ISO 15765 is like "CAN for cars". In particular, ISO 15765-4 describes the physical, data link layer and network layers, seeking to standardize the CAN bus interface for external test equipment.

ISO 15765-2 in turn describes the transport layer (ISO TP) for sending CAN frames with payloads that exceed 8 bytes. This sub standard is also sometimes referred to as Diagnostic Communication over CAN (or DoCAN). See also the 7 layer OSI model illustration. OBD2 can also be compared to other higher layer protocols (e.g. J1939, CANopen).

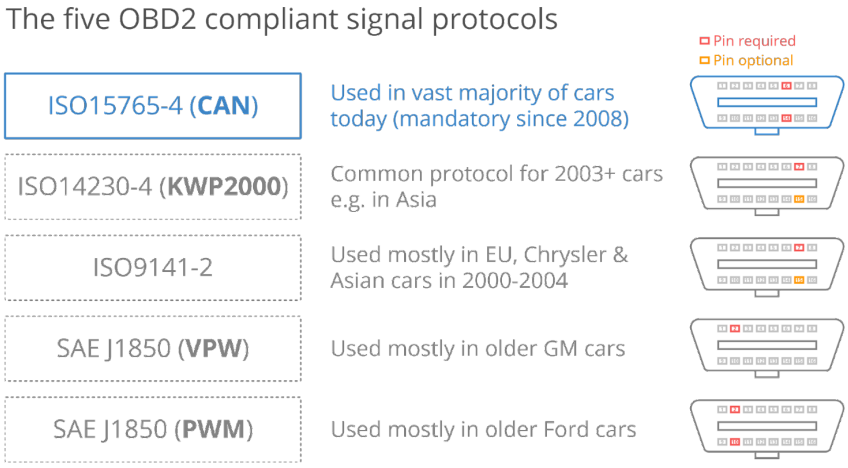
7 layer OSI model

Application	ISO 14229-1 (UDS) ISO 15765-3 (UDS) SAE J1979 / ISO 15031
Presentation	
Session	ISO 15765-3 (UDS)
Transport	ISO 15765-2 (ISO-TP)
Network	ISO 15765-4 ('OBD via CAN') ISO 15765-2 (ISO-TP)
Data link	ISO 15765-4 ('OBD via CAN') ISO 11898-1 (CAN)
Physical	ISO 15765-4 ('OBD via CAN') ISO 11898-2 (CAN)

The five OBD2 protocols

As explained above, CAN bus today serves as the basis for OBD2 communication in the vast majority of cars through ISO 15765. However, if you're inspecting an older car (pre 2008), it is useful to know the other four protocols that have been used as basis for OBD2. Note also the pinouts, which can be used to determine which protocol may be used in your car.

- ISO 15765 (CAN bus): Mandatory in US cars since 2008 and is today used in the vast majority of cars
- ISO14230-4 (KWP2000): The Keyword Protocol 2000 was a common protocol for 2003+ cars in e.g. Asia
- ISO9141-2: Used in EU, Chrysler & Asian cars in 2000-04
- SAE J1850 (VPW): Used mostly in older GM cars
- SAE J1850 (PWM): Used mostly in older Ford cars

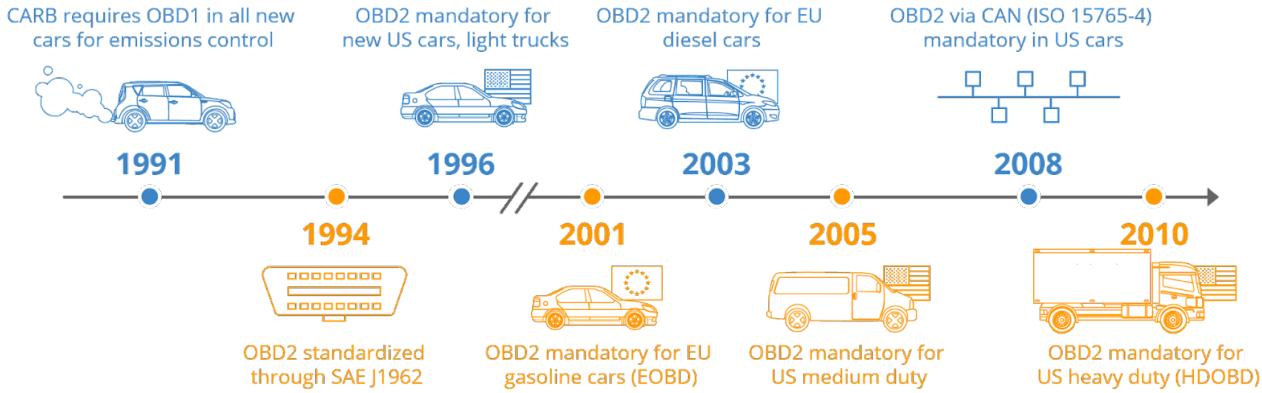


OBD2 history & future

History

OBD2 originates from California where the California Air Resources Board (CARB) required OBD in all new cars from 1991+ for emission control purposes. The OBD2 standard was recommended by the Society of Automotive Engineers (SAE) and standardized DTCs and the OBD connector across manufacturers (SAE J1962). From there, the OBD2 standard was rolled out step-by-step:

- 1996: OBD2 made mandatory in USA for cars/light trucks
- 2001: Required in EU for gasoline cars
- 2003: Required in EU also for diesel cars (EOBD)
- 2005: OBD2 was required in US for medium duty vehicles
- 2008: US cars must use ISO 15765-4 (CAN) as OBD2 basis
- 2010: Finally, OBD2 was required in US heavy duty vehicles



Future

OBD2 is here to stay - but in what form? Two potential routes may radically change OBD2:

OBD3/OBD-III - wireless emission testing

In today's world of connected cars, OBD2 tests can seem cumbersome: Manually doing emission control checks is time-consuming and expensive. The solution? OBD3 - adding telematics to all cars. Basically, OBD3 adds a small radio transponder (as in e.g. bridge tolls) to all cars. Using this, the car vehicle identification number (VIN) and DTCs can be sent via WiFi to a central server for checks.

Many devices today already facilitate transfer of CAN or OBD2 data via WiFi/cellular - e.g. the CANedge2 WiFi CAN logger. This saves cost and is convenient, but it is also politically a challenge due to surveillance concerns.

Eliminating 3rd party OBD2 services

The OBD2 protocol was originally designed for stationary emission controls. Yet, today OBD2 is used extensively for generating real-time data by 3rd parties - via OBD2 dongles, CAN loggers etc. However, the German car industry is looking to change this:

"OBD has been designed to service cars in repair shops. In no way has it been intended to allow third parties to build a form of data-driven economy on the access through this interface"

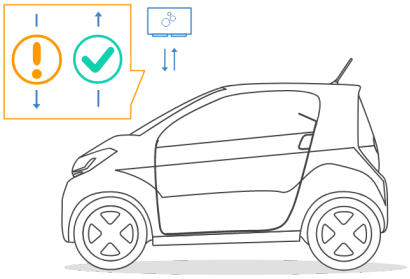
- Christoph Grote, SVP Electronics, BMW (2017)

The proposal is to "turn off" the OBD2 functionality while driving - and instead collect the data in a central server. This would effectively put the manufacturers in control of the automotive 'big data'. The argumentation is based in security (e.g. removing the risk of car hacking), though many see it as a commercial move. Whether this becomes a real trend is to be seen - but it may truly disrupt the market for OBD2 3rd party services.

OBD2 parameter IDs (PID)

Why should you care about OBD2 data? Mechanics obviously care about OBD2 DTCs (maybe you do too), while regulatory entities need OBD2 to control emission. But the OBD2 protocol also supports a broad range of standard parameter IDs (PIDs) that can be logged across most cars.

This means that you can easily get human-readable OBD2 data from your car on speed, RPM, throttle position and more. In other words, OBD2 lets you analyze data from you car easily - in contrast to the OEM specific proprietary raw CAN data.



Decoding OBD2 vs CAN bus data

In principle it is simple to log the raw CAN frames from your car. If you e.g. connect a CAN logger to the OBD2 connector, you'll start logging broadcasted CAN bus data out-the-box. However, the raw CAN messages need to be decoded via a database of conversion rules (DBC) and a suitable CAN software that supports DBC decoding (like e.g. asammdf). The challenge is that these CAN DBC files are typically proprietary, making the raw CAN data unreadable unless you're the automotive OEM.

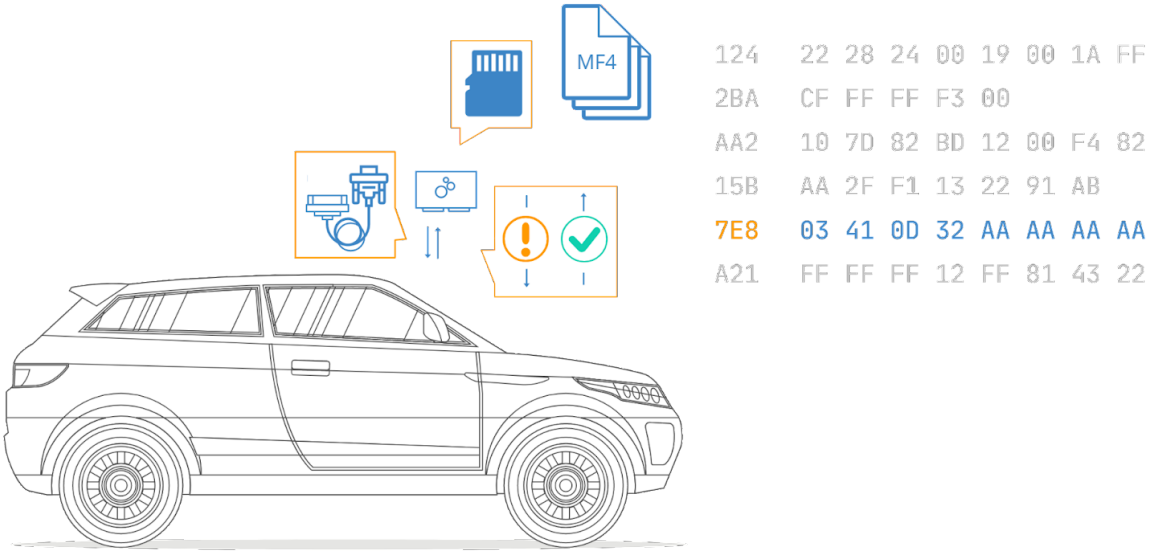
Car hackers may try to reverse engineer the rules, though this can be difficult. CAN is, however, still the only method to get "full access" to your car data - while OBD2 only provides access to a limited subset of data.

How to log OBD2 data?

OBD2 data logging works as follows:

- You connect an OBD2 logger to the OBD2 connector
- Using the tool, you send 'request frames' via CAN
- The relevant ECUs send 'response frames' via CAN
- Decode the raw OBD2 responses via e.g. an OBD2 DBC

In other words, a CAN logger that is able to transmit custom CAN frames can also be used as an OBD2 logger.
Note that cars differ by model/year in what OBD2 PIDs they support. For details, see our OBD2 data logger guide.

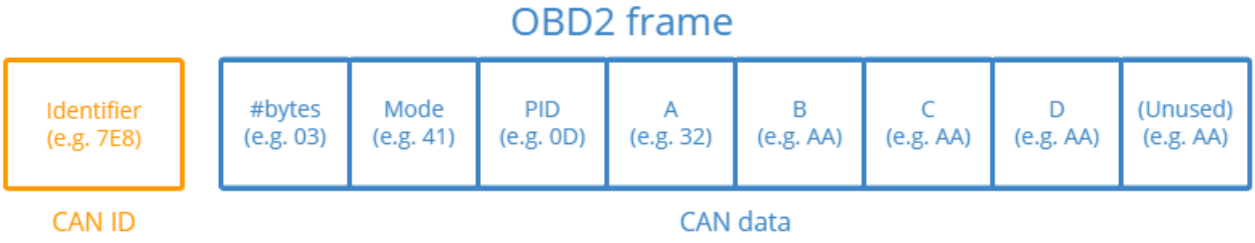


CANedge OBD2 data logger

The CANedge lets you easily log OBD2 data to an 8-32 GB SD card. Simply specify what OBD2 PIDs you wish to request, then connect it to your car via an OBD2 adapter to start logging. Process the data via free software/APIs and our OBD2 DBC.

Raw OBD2 frame details

To get started recording OBD2 data, it is helpful to understand the basics of the raw OBD2 message structure. In simplified terms, an OBD2 message consists of an identifier and data. Further, the data is split in Mode, PID and data bytes (A, B, C, D) as below.



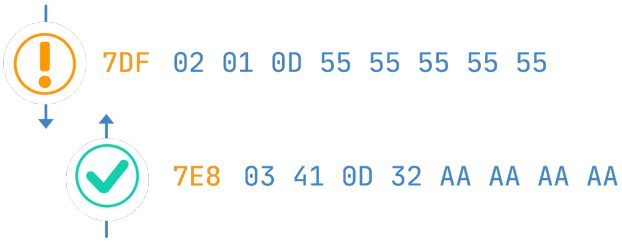
OBD2 message fields explained

- **Identifier**: For OBD2 messages, the identifier is standard 11-bit and used to distinguish between "request messages" (ID 7DF) and "response messages" (ID 7E8 to 7EF). Note that 7E8 will typically be where the main engine or ECU responds at.
- **Length**: This simply reflects the length in number of bytes of the remaining data (03 to 06). For the Vehicle Speed example, it is 02 for the request (since only 01 and 0D follow), while for the response it is 03 as both 41, 0D and 32 follow.
- **Mode**: For requests, this will be between 01-0A. For responses the 0 is replaced by 4 (i.e. 41, 42, ... , 4A). There are 10 modes as described in the SAE J1979 OBD2 standard. Mode 1 shows Current Data and is e.g. used for looking at real-time vehicle speed, RPM etc. Other modes are used to e.g. show or clear stored diagnostic trouble codes and show freeze frame data.

- **PID**: For each mode, a list of standard OBD2 PIDs exist - e.g. in Mode 01, PID 0D is Vehicle Speed. For the full list, check out our OBD2 PID overview. Each PID has a description and some have a specified min/max and conversion formula. The formula for speed is e.g. simply A, meaning that the A data byte (which is in HEX) is converted to decimal to get the km/h converted value (i.e. 32 becomes 50 km/h above). For e.g. RPM (PID 0C), the formula is (256*A + B) / 4.
- **A, B, C, D**: These are the data bytes in HEX, which need to be converted to decimal form before they are used in the PID formula calculations. Note that the last data byte (after Dh) is not used.

OBD2 request/response example

An example of a request/response CAN message for the PID 'Vehicle Speed' with a value of 50 km/h can be seen in the illustration. Note in particular how the formula for the OBD2 PID 0D (Vehicle Speed) simply involves taking the 4th byte (0x32) and converting it to decimal form (50).



Extended OBD2 PID request/response

In some vehicles (e.g. vans and light/medium/heavy duty vehicles), you may find that the raw CAN data uses extended 29-bit CAN identifiers instead of 11-bit CAN identifiers.

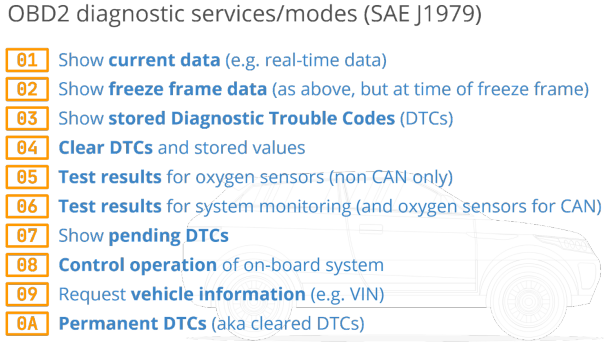
In this case, you will typically need to modify the OBD2 PID requests to use the CAN ID 18DB33F1 instead of 7DF. The data payload structure is kept identical to the examples for 11-bit CAN IDs.

If the vehicle responds to the requests, you'll typically see responses with CAN IDs 18DAF100 to 18DAF1FF (in practice, typically 18DAF110 and 18DAF11E). The response identifier is also sometimes shown in the 'J1939 PGN' form, specifically the PGN 0xDA00 (55808), which in the J1939-71 standard is marked as 'Reserved for ISO 15765-2'.

We provide an OBD2 DBC file for both the 11-bit and 29-bit responses, enabling simple decoding of the data in most CAN software tools.

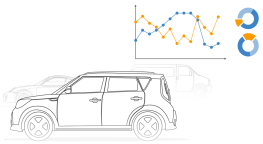
The 10 OBD2 services (aka modes)

There are 10 OBD2 diagnostic services (or modes) as described in the SAE J1979 OBD2 standard. Mode 1 shows Current Data and is used for looking at real-time parameters like vehicle speed, RPM, throttle position etc. Other modes are e.g. used to show/clear diagnostic trouble codes (DTCs) and show freeze frame data. Manufacturers do not have to support all diagnostic services - and they may support modes outside these 10 services (i.e. manufacturer specific OBD2 services).



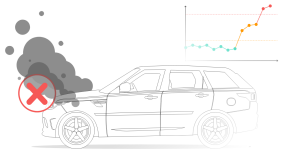
OBD2 data logging - use case examples

OBD2 data from cars and light trucks can be used in various use cases:

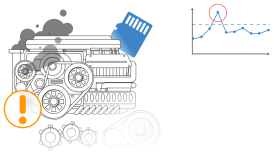


Logging data from cars
OBD2 data from cars can e.g. be used to reduce fuel costs, improve driving, test prototype parts and insurance
learn more

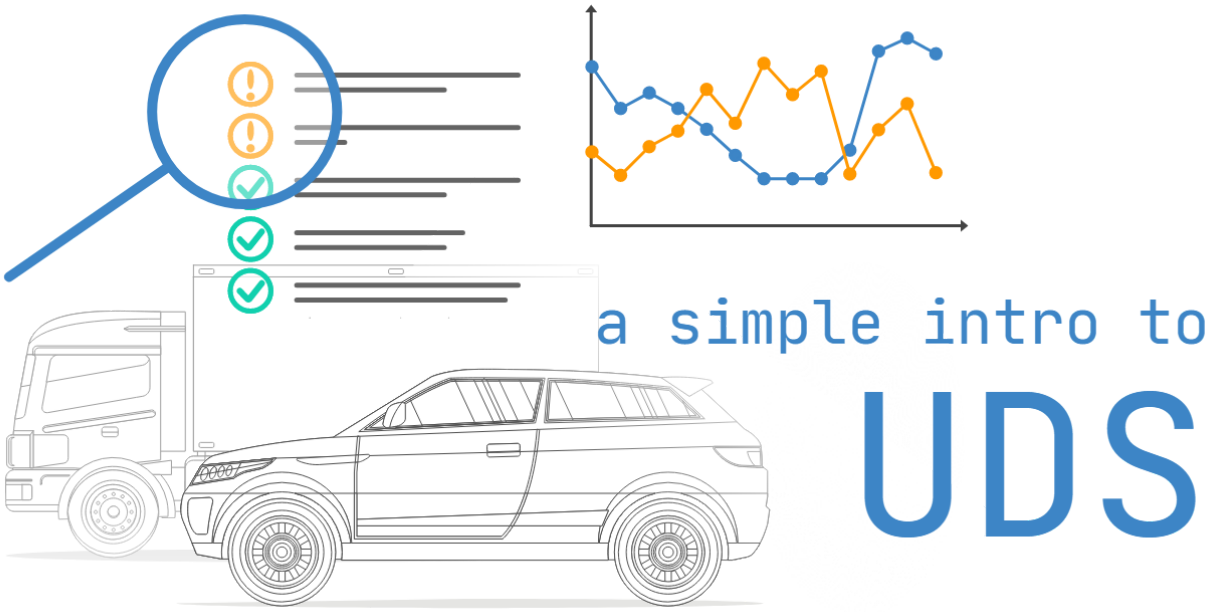
Real-time car diagnostics
OBD2 interfaces can be used to stream human-readable OBD2 data in real-time, e.g. for diagnosing vehicle issues
learn more



Predictive maintenance
Cars and light trucks can be monitored via IoT OBD2 loggers in the cloud to predict and avoid breakdowns
learn more



Vehicle blackbox logger
An OBD2 logger can serve as a 'blackbox' for vehicles or equipment, providing data for e.g. disputes or diagnostics
learn more



UDS Explained (Unified Diagnostic Services)

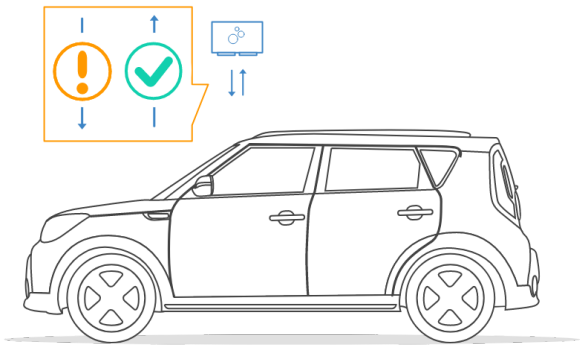
In this practical tutorial, we introduce the UDS basics with focus on UDS on CAN bus (UDSonCAN) and Diagnostics over CAN (DoCAN). We also introduce the ISO-TP protocol and explain the difference between UDS, OBD2, WWH-OBD and OBDonUDS. Finally, we'll explain how to request, record & decode UDS messages - with practical examples for logging EV State of Charge and the Vehicle Identification Number (VIN).

What is the UDS protocol?

Unified Diagnostic Services (UDS) is a communication protocol used in automotive Electronic Control Units (ECUs) to enable diagnostics, firmware updates, routine testing and more.

The UDS protocol (ISO 14229) is standardized across both manufacturers and standards (such as CAN, KWP 2000, Ethernet, LIN). Further, UDS is today used in ECUs across all tier 1 Original Equipment Manufacturers (OEMs).

In practice, UDS communication is performed in a client-server relationship - with the client being a tester-tool and the server being a vehicle ECU. For example, you can connect a CAN bus interface to the OBD2 connector of a car and send UDS requests into the vehicle. Assuming the targeted ECU supports UDS services, it will respond accordingly.



In turn, this enables various use cases:

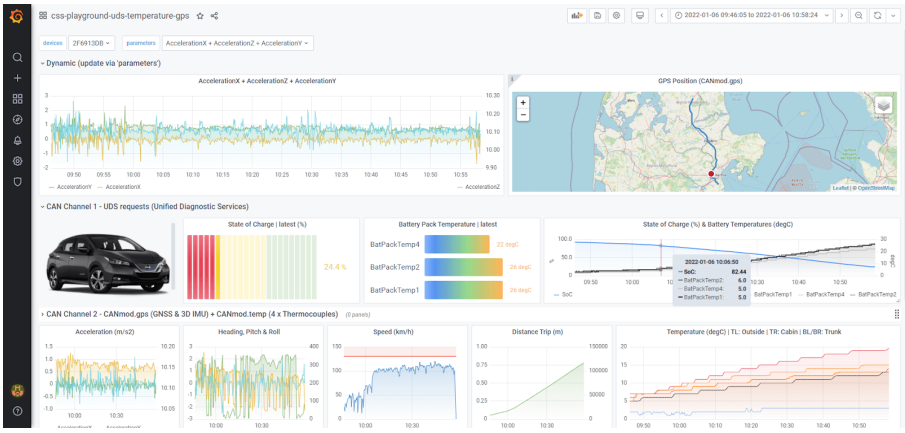
- Read/clear diagnostic trouble codes (DTC) for troubleshooting vehicle issues
- Extract parameter data values such as temperatures, state of charge, VIN etc
- Initiate diagnostic sessions to e.g. test safety-critical features
- Modify ECU behavior via resets, firmware flashing and settings modification

UDS is often referred to as 'vehicle manufacturer enhanced diagnostics' or 'enhanced diagnostics' - more on this below.

Example: Nissan Leaf SoC%

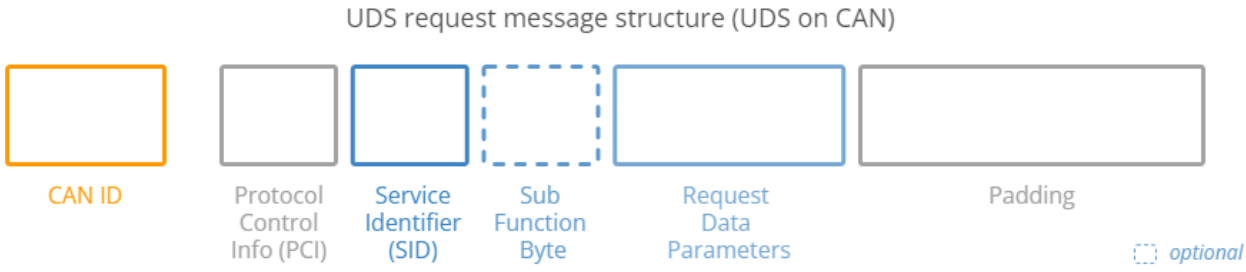
UDS and CAN ISO-TP are complex topics. As motivation, we've done a case study to show how it can be useful. Specifically, we use a CANedge2 to request data on State of Charge (SoC%) and battery temperatures from a Nissan Leaf EV. In the example, we also add a CANmod.gps and CANmod.temp to add GNSS, IMU and temperature data.

The multiframe ISO-TP responses and CANmod signals are DBC decoded via our Python API and written to a database for visualization in Grafana dashboards. Check it out: playground case study



UDS message structure

UDS is a request based protocol. In the illustration we've outlined an example of an UDS request frame (using CAN bus):



A diagnostic UDS request on CAN contains various fields that we detail below:

Protocol Control Information (PCI)

The PCI field is not per se related to the UDS request itself, but is required for diagnostic UDS requests made on CAN bus. In short, the PCI field can be 1-3 bytes long and contains information related to the transmission of messages that do not fit within a single CAN frame. We will detail this more in the section on the CAN bus transport protocol (ISO-TP).

UDS Service ID (SID)

The use cases outlined above relate to different UDS services. When you wish to utilize a specific UDS service, the UDS request message should contain the UDS Service Identifier (SID) in the data payload. Note that the identifiers are split between request SIDs (e.g. 0x22) and response SIDs (e.g. 0x62). As in OBD2, the response SIDs generally add 0x40 to the request SIDs. See also the overview of all standardized UDS services and SIDs. We will mainly focus on UDS service 0x22 in this article, which is used to read data (e.g. speed, SoC, temperature, VIN) from an ECU.

UDS service identifiers (SIDs)

	UDS SID (request)	UDS SID (response)	Service	Details
Diagnostic and Communications Management	0x10	0x50	Diagnostic Session Control	Control which UDS services are available
	0x11	0x51	ECU Reset	Reset the ECU (“hard reset”, “key off”, “soft reset”)
	0x27	0x67	Security Access	Enable use of security-critical services via authentication
	0x28	0x68	Communication Control	Turn sending/receiving of messages on/off in the ECU
	0x29	0x69	Authentication	Enable more advanced authentication vs. 0x27 (PKI based exchange)
	0x3E	0x7E	Tester Present	Send a “heartbeat” periodically to remain in the current session
	0x83	0xC3	Access Timing Parameters	View/modify timing parameters used in client/server communication
	0x84	0xC4	Secured Data Transmission	Send encrypted data via ISO 15764 (Extended Data Link Security)
	0x85	0xC5	Control DTC Settings	Enable/disable detection of errors (e.g. used during diagnostics)
	0x86	0xC6	Response On Event	Request that an ECU processes a service request if an event happens
	0x87	0xC7	Link Control	Set the baud rate for diagnostic access
Data Transmission	0x22	0x62	Read Data By Identifier	Read data from targeted ECU - e.g. VIN, sensor data values etc.
	0x23	0x63	Read Memory By Address	Read data from physical memory (e.g. to understand software behavior)
	0x24	0x64	Read Scaling Data By Identifier	Read information about how to scale data identifiers
	0x2A	0x6A	Read Data By Identifier Periodic	Request ECU to broadcast sensor data at slow/medium/fast/stop rate
	0x2C	0x6C	Dynamically Define Data Identifier	Define data parameter for use in 0x22 or 0x2A dynamically
	0x2E	0x6E	Write Data By Identifier	Program specific variables determined by data parameters
	0x3D	0x7D	Write Memory By Address	Write information to the ECU’s memory
DTCs	0x14	0x54	Clear Diagnostic Information	Delete stored DTCs
	0x19	0x59	Read DTC Information	Read stored DTCs, as well as related information
	0x2F	0x6F	Input Output Control By Identifier	Gain control over ECU analog/digital inputs/outputs
	0x31	0x71	Routine Control	Initiate/stop routines (e.g. self-testing, erasing of flash memory)
Upload/ Download	0x34	0x74	Request Download	Start request to add software/data to ECU (incl. location/size)
	0x35	0x75	Request Upload	Start request to read software/data from ECU (incl. location/size)
	0x36	0x76	Transfer Data	Perform actual transfer of data following use of 0x74/0x75
	0x37	0x77	Request Transfer Exit	Stop the transfer of data
	0x38	0x78	Request File Transfer	Perform a file download/upload to/from the ECU
		0x7F	Negative Response	Sent with a Negative Response Code when a request cannot be handled

UDS SIDs vs other diagnostic services

The standardized UDS services shown above are in practice a subset of a larger set of diagnostic services - see below overview. Note here that the SIDs 0x00 to 0x0F are reserved for legislated OBD services (more on this later).

Diagnostic service identifiers - overview (0x00 - 0xFF)

Service Identifier (SID)	Service type	Further details
0x00 - 0x0F	OBD service requests	ISO 15031-5
0x00 - 0x3E	ISO 14229 (requests)	ISO 14229
0x3F	Not applicable	Reserved
0x40 - 0x4F	OBD service responses	ISO 15031-5
0x50 - 0x7E	ISO 14229 (responses)	ISO 14229
0x7F	Negative response SID	ISO 14229
0x80	Not applicable	ISO 14229 (reserved)
0x81 - 0x82	Not applicable	ISO 14229 (reserved)
0x83 - 0x88	ISO 14229 (requests)	ISO 14229
0x89 - 0x9F	Service requests	Reserved
0xA0 - 0xB9	Service requests	Defined by vehicle OEM
0xBA - 0xBE	Service requests	Defined by systems OEM
0xBF	Not applicable	Reserved
0xC0	Not applicable	ISO 14229 (reserved)
0xC1 - 0xC2	Not applicable	ISO 14229 (reserved)
0xC3 - 0xC8	ISO 14229 (responses)	ISO 14229
0xC9 - 0xDF	Service responses	Reserved
0xE0 - 0xF9	Service responses	Defined by vehicle OEM
0xFA - 0xFE	Service responses	Defined by systems OEM
0xFF	Not applicable	Reserved

UDS request SIDs UDS positive response SIDs UDS negative response SID

UDS security via session-control (authentication)

As evident, UDS enables extensive control over the vehicle ECUs. For security reasons, critical UDS services are therefore restricted through an authentication process. Basically, an ECU will send a 'seed' to a tester, who in turn must produce a 'key' to gain access to security-critical services. To retain this access, the tester needs to send a 'tester present' message periodically.

In practice, this authentication process enables vehicle manufactures to restrict UDS access for aftermarket users and ensure that only designated tools will be able to utilize the security-critical UDS services.

Note that the switching between authentication levels is done through diagnostic session control, which is one of the UDS services available. Vehicle manufactures can decide which sessions are supported, though they must always support the 'default session' (i.e. which does not involve any authentication). With that said, they decide what services are supported within the default session as well. If a tester tool switches to a non-default session, it must send a 'tester present' message periodically to avoid being returned to the default session.

UDS Sub Function Byte

The sub function byte is used in some UDS request frames as outlined below. Note, however, that in some UDS services, like 0x22, the sub function byte is not used. Generally, when a request is sent to an ECU, the ECU may respond positively or negatively. In case the response is positive, the tester may want to suppress the response (as it may be irrelevant). This is done by setting the 1st bit to 1 in the sub function byte. Negative responses cannot be suppressed.

The remaining 7 bits can be used to define up to 128 sub function values. For example, when reading DTC information via SID 0x19 (Read Diagnostic Information), the sub function can be used to control the report type - see also below table.

UDS services - sub function types

UDS SID (request)	UDS SID (response)	Service	Sub function types
0x10	0x50	Diagnostic Session Control	Diagnostic session type
0x11	0x51	ECU Reset	Reset type
0x27	0x67	Security Access	Security access type
0x28	0x68	Communication Control	Control type
0x3E	0x7E	Tester Present	"Zero sub function"
0x83	0xC3	Access Timing Parameters	Timing parameter access type
0x85	0xC5	Control DTC Settings	DTC setting type
0x86	0xC6	Response On Event	Event type
0x87	0xC7	Link Control	Link control type
0x2C	0x6C	Dynamically Define Data Identifier	Definition type
0x19	0x59	Read DTC Information	Report type
0x31	0x71	Routine Control	Routine control type

Example: Service 0x19 sub functions

If we look specifically at service 0x19, we can see an example of the various sub functions below:

UDS service 0x19 - sub function byte values & types

UDS SID (request)	UDS SID (response)	Report type (sub function value)	Report
0x19	0x59	0x01	Number of DTC by status mask
		0x02	DTC by status mask
		0x03	DTC snapshot identification
		0x04	DTC snapshot record by DTC number
		0x05	DTC stored data by record number
		0x06	DTC extended data record by DTC number
		0x07	Number of DTC by severity mask record
		0x08	DTC by severity mask record
		0x09	Severity information of DTC
		0x0A	Supported DTC
		0x0B	First failed DTC
		0x0C	First confirmed DTC
		0x0D	Most recent failed DTC
		0x0E	Most recent confirmed DTC
		0x0¤	Mirror memory DTC by status mask
		0x10	Mirror memory DTC by DTC number
		0x11	Number of mirror memory DTC by status mask
		0x12	Number of emissions OBD DTC by status mask
		0x13	Emissions OBD DTC by status mask
		0x14	DTC fault detection counter
		0x15	DTC with permanent status
		0x16	DTC extended data record by record number
		0x17	User defined memory DTC by status mask
		0x18	User defined memory DTC snapshot by number
		0x19	User defined memory DTC record by number
		0x42	WWH-OBD DTC by status mask record
		0x55	WWH-OBD DTCs with permanent status

UDS 'Request Data Parameters' - incl. Data Identifier (DID)

In most UDS request services, various types of request data parameters are used to provide further configuration of a request beyond the SID and optional sub function byte. Here we outline two examples.

Service 0x19 (Read DTC Information) - request configuration

For example, service 0x19 lets you read DTC information. The UDS request for SID 0x19 includes a sub function byte - for example, 0x02 lets you read DTCs via a status mask. In this specific case, the sub function byte is followed by a 1-byte parameter called DTC Status Mask to provide further information regarding the request. Similarly, other types of sub functions within 0x19 have specific ways of configuring the request.

Service 0x22 (Read Data by Identifier) - Data Identifiers

Another example is service 0x22 (Read Data by Identifier). This service uses a Data Identifier (DID), which is a 2-byte value between 0 and 65535 (0xFFFF). The DID serves as a parameter identifier for both requests/responses (similar to how the parameter identifier, or PID, is used in OBD2).

For example, a request for reading data via a single DID in UDS over CAN would include the PCI field, the UDS service 0x22 and the 2-byte DID. Alternatively, one can request data for additional DIDs by adding them after the initial DID in the request. We will look further into this in the section on how to record and decode UDS communication.

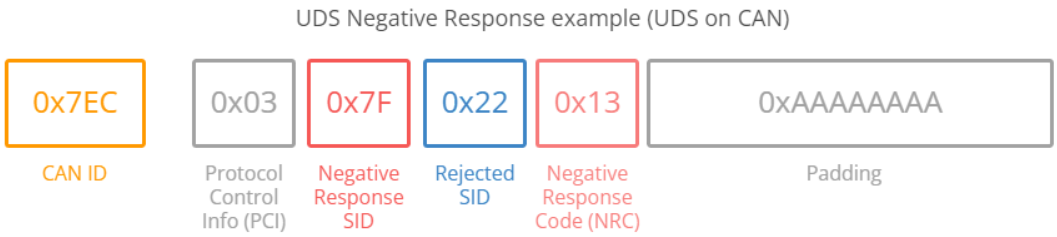
Data Identifiers can be proprietary and only known by OEMs, though some DIDs are standardized. This is for example the case for the WWH-OBD DIDs (more on this later) and the Vehicle Identification Number (VIN) is 0xF190. See the separate table for a list of standardized DIDs across UDS.

UDS - standardized data identifiers (DID)

UDS DID (data identifier)	Description
0xF180	Boot software identification
0xF181	Application software identification
0xF182	Application data identification
0xF183	Boot software fingerprint
0xF184	Application software fingerprint
0xF185	Application data fingerprint
0xF186	Active diagnostic session
0xF187	Manufacturer spare part number
0xF188	Manufacturer ECU software number
0xF189	Manufacturer ECU software version
0xF18A	Identifier of system supplier
0xF18B	ECU manufacturing date
0xF18C	ECU serial number
0xF18D	Supported functional units
0xF18E	Manufacturer kit assembly part number
0xF190	Vehicle Identification Number (VIN)
0xF192	System supplier ECU hardware number
0xF193	System supplier ECU hardware version number
0xF194	System supplier ECU software number
0xF195	System supplier ECU software version number
0xF196	Exhaust regulation/type approval number
0xF197	System name / engine type
0xF198	Repair shop code / tester serial number
0xF199	Programming date
0xF19D	ECU installation date
0xF19E	ODX file

Positive vs. negative UDS responses

When an ECU responds positively to an UDS request, the response frame is structured with similar elements as the request frame. For example, a 'positive' response to a service 0x22 request will contain the response SID 0x62 (0x22 + 0x40) and the 2-byte DID, followed by the actual data payload for the requested DID. Generally, the structure of a positive UDS response message depends on the service. However, in some cases an ECU may provide a negative response to an UDS request - for example if the service is not supported. A negative response is structured as in below CAN frame example:



Details + Negative Response Code table

Below we briefly detail the negative response frame with focus on the NRC:

- The 1st byte is the PCI field
- The 2nd byte is the Negative Response Code SID, 0x7F
- The 3rd byte is the SID of the rejected request
- The 4th byte is the Negative Response Code (NRC)

In the negative UDS response, the NRC provides information regarding the cause of the rejection as per the table below.

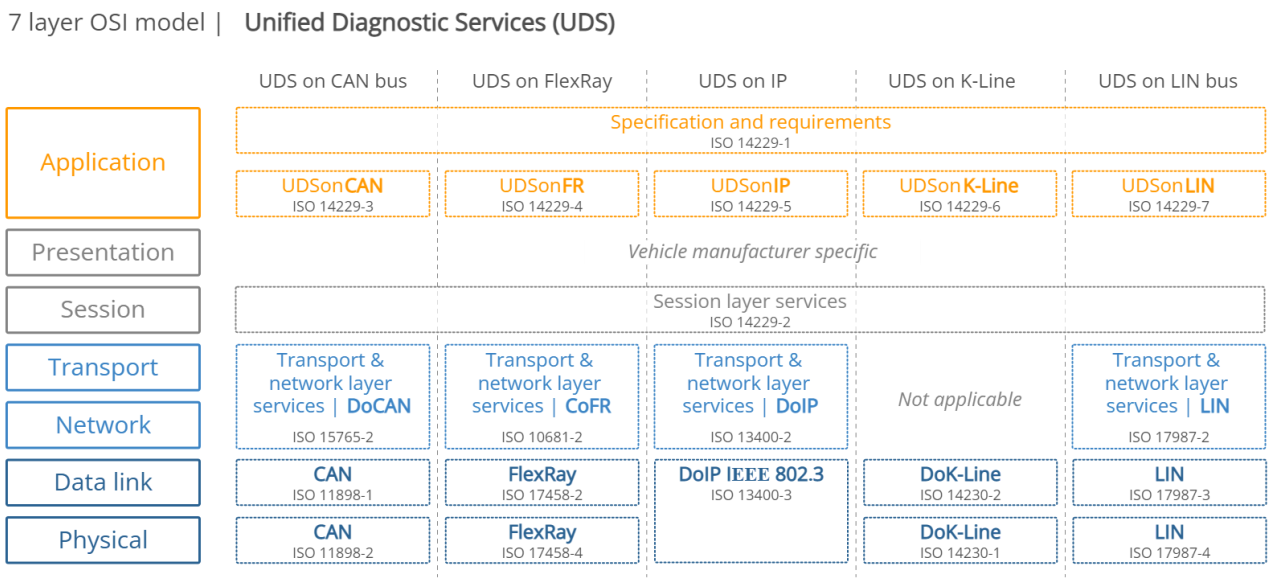
UDS SID 0x7F - Negative Response Codes (NRC)

UDS NRC	Description
0x10	General reject
0x11	Service not supported
0x12	Sub-function not supported
0x13	Invalid message length/format
0x14	Response too long
0x21	Busy-repeat request
0x22	Conditions not correct
0x24	Request sequence error
0x25	No response from subnet component
0x26	Failure prevents execution of requested action
0x31	Request out of range
0x33	Security access denied
0x35	Invalid key
0x36	Exceeded number of attempts
0x37	Required time delay has not expired
0x70	Upload/download not accepted
0x71	Transfer data suspended
0x72	Programming failure
0x73	Wrong block sequence counter
0x78	Request received - response pending
0x7E	Sub function not supported in active session
0x7F	Service not supported in active session
0x81/0x82	RPM too high/low
0x83/0x84	Engine is running/not running
0x85	Engine run time too low
0x86/0x87	Temperature too high/low
0x88/0x89	Speed too high/low
0x8A/0x8B	Throttle pedal too high/low
0x8C/0x8D	Transmission range not in neutral/dear
0x8F	Brake switches not closed
0x90	Shifter lever not in park
0x91	Torque converter clutch locked
0x92/0x93	Voltage too high/low
0xF0-0xFE	Manufacturer specific conditions not correct

UDS vs CAN bus: Standards & OSI model

To better understand UDS, we will look at how it relates to CAN bus and the OSI model.

As explained in our CAN bus tutorial, the Controller Area Network serves as a basis for communication. Specifically, CAN is described by a data-link layer and physical layer in the OSI model (as per ISO 11898). In contrast to CAN, UDS (ISO 14229) is a 'higher layer protocol' and utilizes both the session layer (5th) and application layer (7th) in the OSI model as shown below:



Overview of UDS standards & concepts

UDS refers to a large number of standards/concepts, meaning it can be a bit overwhelming. To give you an overview, we provide a high-level explanation of the most relevant ones below (with focus on CAN as the basis).

Quick overview of the UDS OSI model layers

In the following we provide a quick breakdown of each layer of the OSI model:

- Application: This is described by ISO 14229-1 (across the various serial data link layers). Further, separate ISO standards describe the UDS application layer for the various lower layer protocols - e.g. ISO 14229-3 for CAN bus (aka UDSonCAN)
- Presentation: This is vehicle manufacturer specific
- Session: This is described in ISO 14229-2
- Transport + Network: For CAN, ISO 15765-2 is used (aka ISO-TP)
- Data Link: In the case of CAN, this is described by ISO 11898-1
- Physical: In the case of CAN, this is described by ISO 11898-2

As illustrated, multiple standards other than CAN may be used as the basis for UDS communication - including FlexRay, Ethernet, LIN bus and K-line. In this tutorial we focus on CAN, which is also the most common lower layer protocol.

ISO 14229-1 (Application Layer)

The ISO 14229-1 standard describes the application layer requirements for UDS (independent of what lower layer protocol is used). In particular, it outlines the following:

- Client-server communication flows (requests, responses, ...)
- UDS services (as per the overview described previously)
- Positive responses and negative response codes (NRCs)
- Various definitions (e.g. DTCs, parameter data identifiers aka DIDs, ...)

ISO 14229-3 (Application Layer for CAN)

The purpose of 14229-3 is to enable the implementation of Unified Diagnostic Services (UDS) on Controller Area Networks (CAN) - also known as UDSonCAN. This standard describes the application layer requirements for UDSonCAN.

This standard does not describe any implementation requirements for the in-vehicle CAN bus architecture. Instead, it focuses on some additional requirements/restrictions for UDS that are specific to UDSonCAN.

Specifically, 14229-3 outlines which UDS services have CAN specific requirements. The affected UDS services are ResponseOnEvent and ReadDataByPeriodicIdentifier, for which the CAN specific requirements are detailed in 14229-3. All other UDS services are implemented as per ISO 14229-1 and ISO 14229-2.

ISO 14229-3 also describes a set of mappings between ISO 14229-2 and ISO 15765-2 (ISO-TP) and describes requirements related to 11-bit and 29-bit CAN IDs when these are used for UDS and legislated OBD as per ISO 15765-4.

ISO 14229-2 (Session Layer)

This describes the session layer in the UDS OSI model. Specifically, it outlines service request/confirmation/indication primitives. These provide an interface for the implementation of UDS (ISO 14229-1) with any of the communication protocols (e.g. CAN).

ISO 15765-2 (Transport + Network Layer for CAN)

For UDS on CAN, ISO 15765-2 describes how to communicate diagnostic requests and responses. In particular, the standard describes how to structure CAN frames to enable communication of multi-frame payloads. As this is a vital part of understanding UDS on CAN, we go into more depth in the next section.

ISO 11898 (Physical + Data Link Layer for CAN)

When UDS is based on CAN bus, the physical and data link layers are described in ISO 11898-1 and ISO 11898-2. When UDS is based on CAN, it can be compared to a higher layer protocol like J1939, OBD2, CANopen, NMEA 2000 etc. However, in contrast to these protocols, UDS could alternatively be based on other communication protocols like FlexRay, Ethernet, LIN etc.

UDSonCAN vs DoCAN

When talking about UDS based on CAN bus, you'll often see two terms used: UDSonCAN (UDS on CAN bus) and DoCAN (Diagnostics on CAN bus). Some UDS tutorials use these terms interchangeably, which may cause confusion.

In ISO 14229-1 the terms are used as in our OSI model illustration. In other words, UDSonCAN is used to refer to ISO 14229-3, while DoCAN is used to refer to ISO 15765-2 aka ISO-TP.

However, part of the confusion may arise because ISO 14229-3 also provides an OSI model where DoCAN is both used in relation to ISO 15765-2 and as an overlay across OSI model layers 2 to 7. In ISO 14229-2, DoCAN is referred to as the

communication protocol on which UDS (ISO 14229-1) is implemented. This is in sync with the illustration from ISO 14229-3. In this context, DoCAN can be viewed as a more overarching term for the implementation of UDS on CAN, whereas UDSonCAN seems consistently to refer to ISO 14229-3 only.

ISO 15765-3 vs. ISO 14229-3

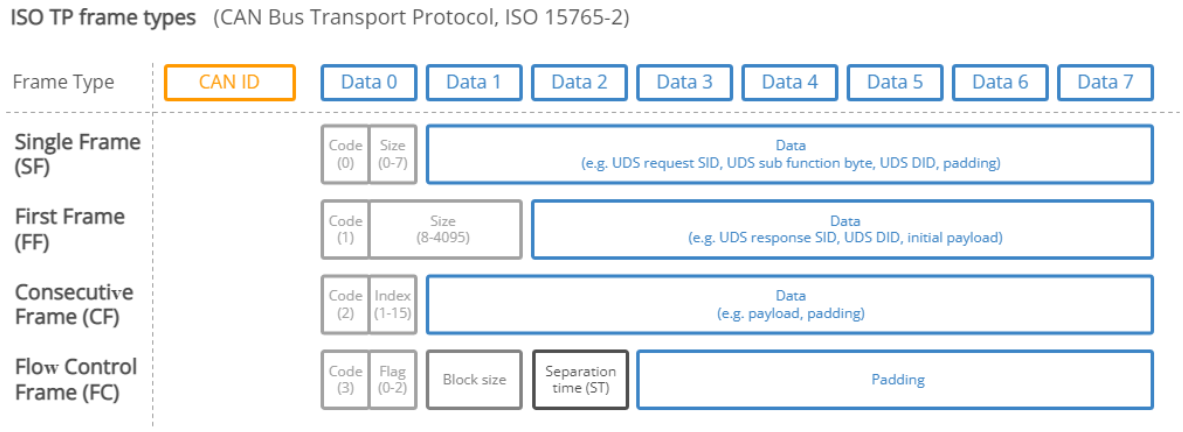
UDS on CAN bus (UDSonCAN) is sometimes referred to through ISO 15765-3. However, this standard is now obsolete and has been replaced by ISO 14229-3.

CAN ISO-TP - Transport Protocol (ISO 15765-2)

When implementing diagnostics on CAN, one challenge is the size of the CAN frame payload: For Classical CAN frames, this is limited to 8 bytes and for CAN FD the payload is limited to 64 bytes. Vehicle diagnostics often involve communication of far larger payloads.

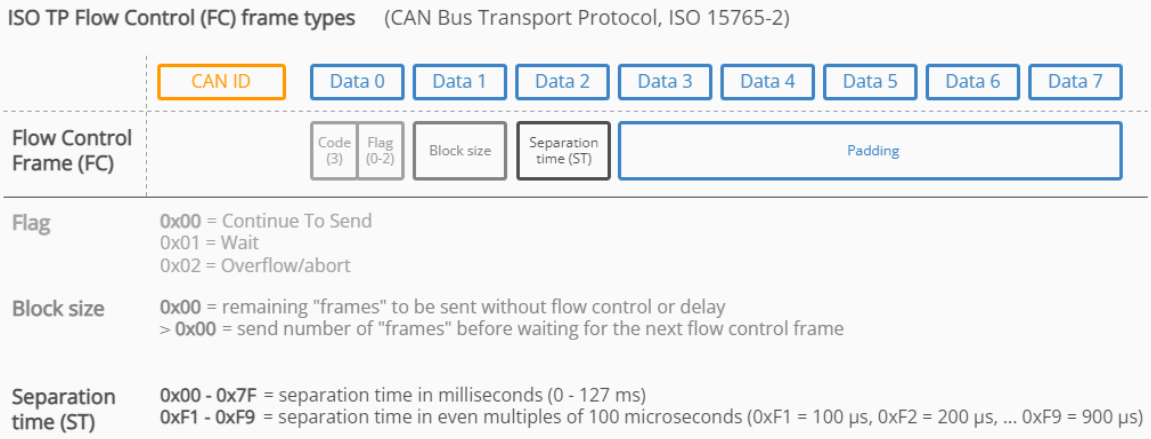
ISO 15765-2 was established to solve the challenge of large payloads for CAN based vehicle diagnostics.
The standard specifies a transport protocol and network layer services for use in CAN based vehicle networks. The most common use cases include UDS (ISO 14229-1), OBD (SAE J1979, ISO 15031-5) and world-wide harmonized OBD aka WWH-OBD (ISO 27145).

The ISO-TP standard outlines how to communicate CAN data payloads up to 4095 bytes through segmentation, flow control and reassembly. ISO-TP defines specific CAN frames for enabling this communication as shown below:



Regarding the Flow Control frame

The flow control frame is used to 'configure' the subsequent communication. It can be constructed as below:



A few comments:

- In the simplest case, the FC payload can be set to 30 00 00 00 00 00 00 00 (all remaining frames to be sent without delay)
- Alternatively, one can decide to perform more granular control over the communication by e.g. alternating between the Wait and Continue commands, as well as specifying a specific separation time (in milliseconds) between frames

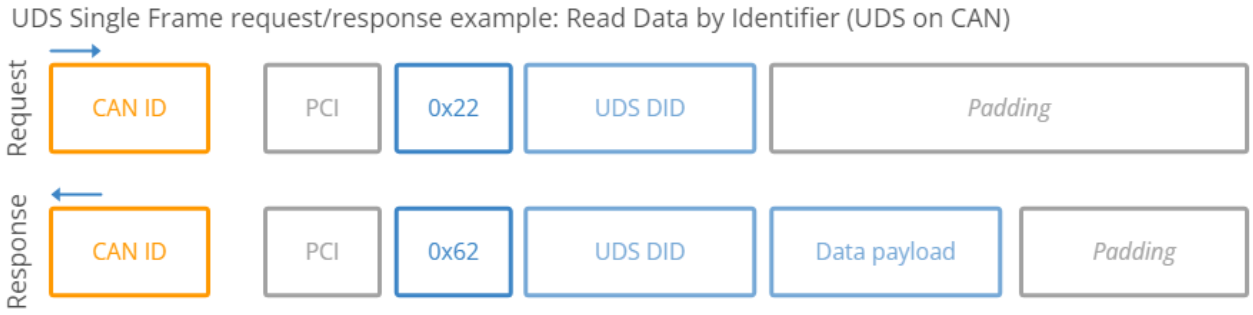
Other ISO-TP frame comments

- The ISO-TP frame type can be identified from the first nibble of the first byte (0x0, 0x1, 0x2, 0x3)
- The total frame size can be up to 4095 bytes (0xFFF) as evident from the FF frame
- The CF index runs from 1 to 15 (0xF) and is then reset if more data is to be sent
- Padding (e.g. 0x00, 0xAA, ...) is used to ensure the frame payloads equal 8 bytes in length

Below we outline how the ISO-TP protocol works for single-frame and multi-frame communication:

ISO-TP: Single-frame communication

In vehicle diagnostics, communication is initiated by a tester tool sending a request. This request frame is a Single Frame (SF). In the simplest case, a tester tool sends a Single Frame to request data from an ECU. If the response can be contained in a 7-byte payload, the ECU provides a Single Frame response.



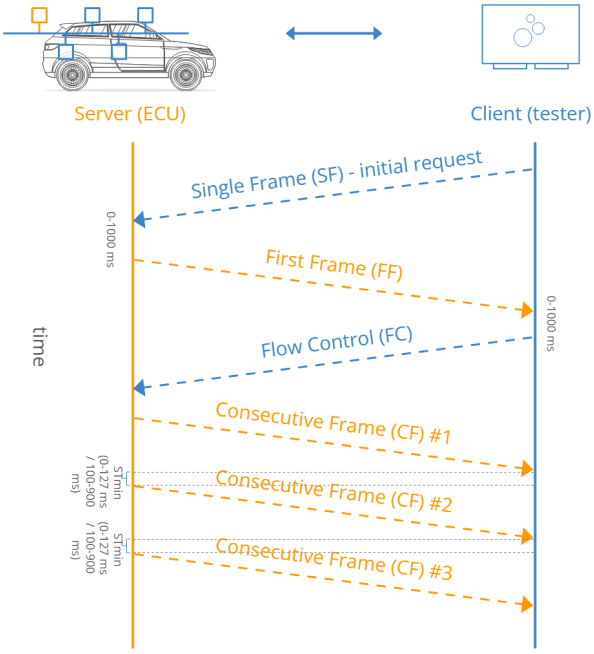
ISO-TP: Multi-frame communication

When the payload exceeds 7 bytes, it needs to be split across multiple CAN frames.

As before, a tester starts by sending a Single Frame (SF) request to an ECU (sender). However, in this case the response exceeds 7 bytes. Because of this, the ECU sends a First Frame (FF) that contains information on the total packet length (8 to 4095 bytes) as well as the initial chunk of data. When the tester receives the FF, it will send a Flow Control (FC) frame, which tells the ECU how the rest of the data transfer should be transmitted.

Following this, the ECU will send Consecutive Frames (CF) that contain the remaining data payload.

ISO-TP plays an important role in most CAN based diagnostics protocols. Before we show practical examples of such communication flows, it is useful to get an overview of the most common vehicle diagnostic protocols.

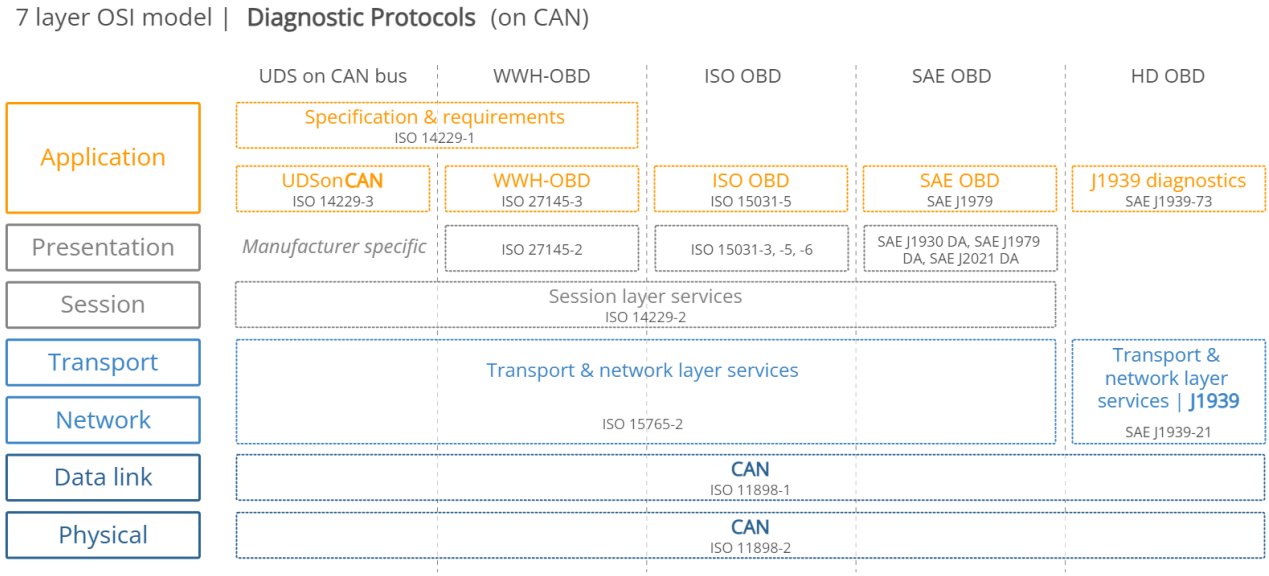


UDS vs. OBD2 vs. WWH-OBD vs. OBDonUDS

A common question is how UDS relates to On-Board Diagnostics (OBD2), World-Wide Harmonized OBD (WWH-OBD) and OBDonUDS. To understand this, it is important to first note the following:

OBD (On-Board Diagnostics) is today implemented in different ways across countries and vehicles.

This is illustrated via the below OSI model comprising CAN based vehicle diagnostic protocols in use today:



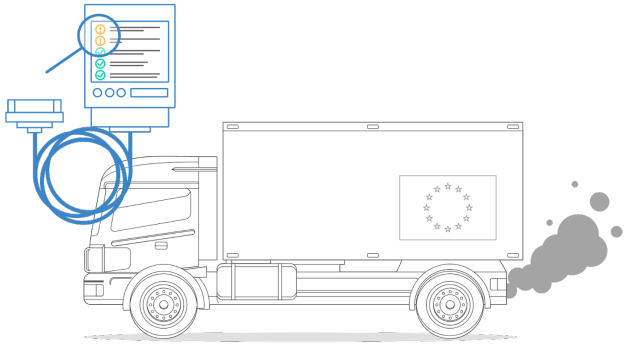
Let's look at each diagnostic protocol:

- ISO OBD (or EOBD) refers to the OBD protocol specification legislated for use in EU cars, while SAE OBD refers to the OBD protocol specification legislated for use in US. The two are technically equivalent and hence often referred to simply as OBD or OBD2
- HD OBD (SAE J1939) typically refers to heavy duty OBD and is commonly implemented through the J1939 protocol in both US and EU produced vehicles with J1939-73 specifying diagnostic messages
- UDS (ISO 14229) has been implemented by vehicle manufacturers to serve the need for richer diagnostics data/functionality - beyond the limits of the emissions-focused OBD protocols. It is implemented in most ECUs today, across markets and vehicle types - though in itself, UDS does not offer the necessary standardization required to serve as an alternative to OBD
- WWH-OBD (and/or possibly OBDonUDS) provide an updated version of OBD2 for emissions-related diagnostics - based on UDS

To understand UDS, it is useful to better understand WWH-OBD and OBDonUDS:

What is WWH-OBD (ISO 27145)?

WWH-OBD is a global standard for vehicle diagnostics, developed by the UN under the Global Technical Regulations (GTR) mandate. It aims to provide a single, future-proof alternative to the existing OBD protocols (ISO OBD, SAE OBD, HD OBD). Furthermore, WWH-OBD is based on UDS in order to suit the enhanced diagnostics functionality already deployed by most automotive OEMs today.



Advantages of WWH-OBD

Moving from OBD2 to WWH-OBD will involve a number of benefits, primarily derived from using the UDS protocol as the basis. First of all, the data richness can be increased. OBD2 parameter identifiers (PID) are limited to 1 byte, restricting the number of unique data types to 255, while the UDS data identifier (DID) is 2 bytes, enabling 65535 parameters.

For diagnostic trouble codes (DTCs), OBD2 would allow for 2-byte DTCs. Here, WWH-OBD allows for 'extended DTCs' of 3 bytes. This allows for grouping DTCs by 2-byte types and using the 3rd byte as a failure mode indicator to specify the DTC sub type. Further, WWH-OBD enables a classification of DTCs based on how severe an issue is in regards to the exhaust emissions quality.

WWH-OBD also seeks to take potential future requirements into account by allowing for the Internet Protocol (IP) to be used as an alternative to CAN, meaning that UDSonIP will also be possible in future implementations of WWH-OBD. One potential benefit from this will be the ability to one day perform remote diagnostics through the same protocol.

What is the status on WWH-OBD roll-out?

The intent of WWH-OBD is to serve as a global standard, across all markets and across all vehicle types (cars, trucks, buses, ...). Further, the aim is to potentially expand the standardized functionality beyond just emissions-control.

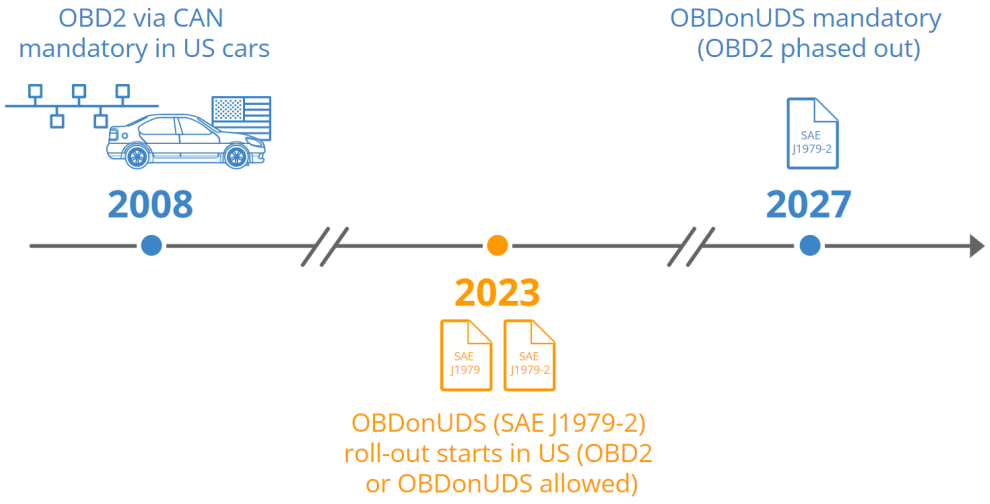
In practice, WWH-OBD has been required in EU since 2014 for newly developed heavy duty vehicles (as per the Euro-VI standards). Note in this case that HD OBD (as per J1939) remains allowed in these vehicles.

Beyond this, however, the roll-out of WWH-OBD has been limited. One challenge is that WWH-OBD is currently not accepted by EPA/CARB in USA. See e.g. this discussion for potential motivations. However, recently OBDonUDS (SAE J1979-2) is being adopted in US.

What is OBDonUDS (SAE J1979-2)?

Similar to how OBD2 has been split into 'SAE OBD' (SAE J1979) for US and 'ISO OBD' (ISO 15031) for EU, the 'next generation' of OBD2 may again be regionally split.

Specifically, WWH-OBD (ISO 21745) has been adopted in EU for heavy duty vehicles already - but not yet in the US. Instead, it has recently been decided to adopt OBD on UDS in US vehicles in the form of the SAE J1979-2 standard, which serves as an update to the SAE J1979. The new SAE J1979-2 standard is also referred to as OBDonUDS. The aim is to initiate a transition phase starting in 2023, where ECUs are allowed to support either OBD2 or OBDonUDS. From 2027, OBDonUDS will be a mandatory requirement for all vehicles produced in the US.



Looking ahead: WWH-OBD vs. OBDonUDS

To recap, WWH-OBD and OBDonUDS both serve as possible solutions for creating a 'next generation" protocol for emissions-related on-board diagnostics. It remains to be seen if the two will exist in parallel (like ISO/SAE OBD), or if one of the protocols will become the de facto standard across both US, EU and beyond.

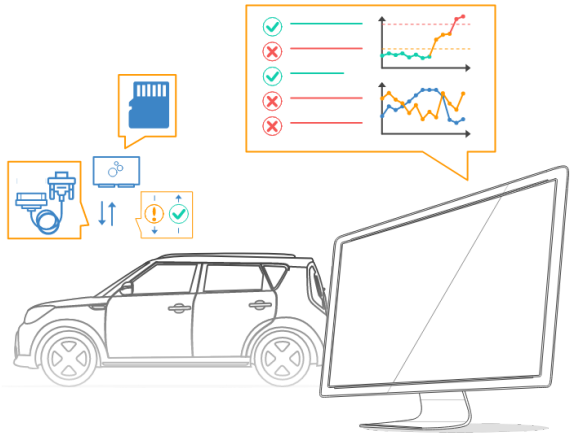
In either case, the basis for emissions-related diagnostics will be UDS, which will serve to simplify ECU programming as the emissions-related diagnostics can increasingly be implemented within the same UDS based structure as the manufacturer specific enhanced diagnostics.

FAQ: How to request/record UDS data

We have now gone through the basics of UDS and the CAN based transport protocol.

With this in place, we can provide some concrete guidance on how you can work with UDS data in practice. In particular, we will focus on how UDS can be used to log various data parameters - like state of charge (SoC) in electric vehicles.

Before the examples, we'll cover frequently asked questions on UDS data logging:

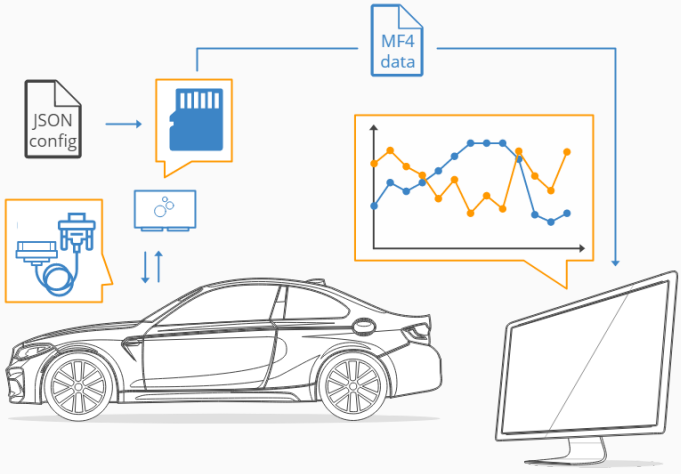


Can the CANedge record UDS data?

Yes, as we'll show further below, the CANedge can be configured to request UDS data. Effectively, the device can be configured to transmit up to 64 custom CAN frames as periodic or single shot frames. You can control the CAN ID, CAN data payload, period time and delay.

For single-frame UDS communication, you simply configure the CANedge with the request frame, which will trigger a single response frame from the ECU.

For multi-frame communication, you can again configure the CANedge with a request frame and then add the Flow Control frame as a separate frame to be transmitted X milliseconds after the request frame. By adjusting the timing, you can set this up so that the Flow Control is sent after the ECU has sent the First Frame response.



Note that the CANedge will record the UDS responses as raw CAN frames. You can then process and decode this data via your preferred software (e.g. Vector tools) or our CAN bus Python API to reassemble and decode the frames.

Note: In future firmware updates, we may enhance the transmit functionality to enable the CANedge to transmit custom CAN frames based on a trigger condition, such as receiving a specific frame. This would allow for sending the Flow Control frame with a set delay after receiving the First Frame, providing a simpler and more robust

implementation. With that said, the current functionality will serve to support most UDS communicated related to services 0x22 (Read Data by Identifier) and 0x19 (Read DTC Information).

What data can I log via UDS?

A very common use case for recording UDS data via 'standalone' data loggers will be to acquire diagnostic trouble code values, e.g. for use in diagnosing issues.

In addition to the trouble codes, UDS lets you request the 'current value" of various sensors related to each ECU. This allows e.g. vehicle fleet managers, researchers etc. to collect data of interest such as speed, RPM, state of charge, temperatures etc. from the car (assuming they know how to request/decode the data, as explained below).

Beyond the above, UDS can of course also be used for more low-level control of ECUs, e.g. resets and flashing of firmware, though such use cases are more commonly performed using a CAN bus interface - rather than a 'standalone' device.

Is UDS data proprietary - or are there public parameters?

Importantly, UDS is a manufacturer specific diagnostic protocol. In other words, while UDS provides a standardized structure for diagnostic communication, the actual 'content" of the communication remains proprietary and is in most cases only known to the manufacturer of a specific vehicle/ECU.

For example, UDS standardizes how to request parameter data from an ECU via the 'Read Data By Identifier" service (0x22). But it does not specify a list of standardized identifiers and interpretation rules. In this way, UDS differs from OBD2, where a public list of OBD2 PIDs enable almost anyone to interpret OBD2 data from their car.

With that said, vehicles that support WWH-OBD or OBDonUDS may support some of the usual OBD2 PIDs like speed, RPM etc via the usual PID values - but with a prefix of 0xF4 as shown in Example 1 below.

Generally, only the manufacturer (OEM) will know how to request proprietary parameters via UDS - and how to interpret the result. Of course, one exception to this rule is cases where companies or individuals successfully reverse engineer this information. Engaging in such reverse engineering is a very difficult task, but you can sometimes find public information and DBC files where others have done this exercise. Our intro to DBC files contain a list of public DBC/decoding databases.

Who benefits from logging UDS data?

Because of the proprietary nature of UDS communication, it is typically most relevant to automotive engineers working at OEMs. Tools like the CANedge CAN bus data logger allow these users to record raw CAN traffic from a vehicle - while at the same time requesting diagnostic trouble codes and specific parameter values via UDS.

Further, some after market users like vehicle fleet managers and even private persons can benefit from UDS assuming they are able to identify the reverse engineered information required to request and decode the UDS parameters of interest.

Logging UDS data will also become increasingly relevant assuming WWH-OBD gets rolled out as expected. Already today, WWH-OBD is used in EU heavy duty vehicles produced after 2014, meaning UDS communication will be relevant for use cases related to on-board diagnostics in these vehicles.

Central Gateway (CGW): Why log sensor data via UDS vs. CAN?

If you're looking to request UDS-based diagnostic trouble codes (DTC), you'll of course have to use UDS communication for this purpose. However, if your aim is to record current sensor values it is less clear.
Typically, data parameters of interest for e.g. vehicle telematics (speed, state of charge etc) will already be communicated between ECUs on the CAN bus in the form of raw CAN frames - without the need for a diagnostic tool requesting this information. That is because ECUs rely on communicating this information as part of their operation (as explained in our intro to CAN bus).
If you have direct access to the CAN bus, it would thus appear easier to simply log this raw CAN traffic and process it. If you are the vehicle manufacturer, you will know how to interpret this raw CAN data either way - and it'll be simpler to perform your device configuration and post processing in most cases. If you're in the aftermarket, it'll also be simpler to reverse engineer the raw CAN frames as you can focus on single frames - and avoid the request/response layer of complexity.

However, one key reason why UDS is frequently used for extracting sensor values despite the above is due to 'gateways'. Specifically, an increasing share of modern cars have started to block the access to the raw CAN bus data via the OBD2 connector. This is particularly often the case for German vehicles, as well as electric vehicles.
To record the existing CAN traffic in such a car, you would need to e.g. use a CANCrocodile adapter and 'snap' it onto the CAN low/high wiring harness. This in turn will require exposing the wiring harness by removing a panel, which is often prohibitive for many use cases. In contrast, the OBD2 connector gateways typically still allow for UDS communication - incl. sensor value communication.

A secondary - and more subtle - reason is that most reverse engineering work is done by 'OBD2 dongle manufacturers'. These develop tools that let you extract data across many different cars through the OBD2 connector. Increasingly, the only way for these dongles to get useful information through the OBD2 connector is through UDS communication, which drives a proportionally higher availability of information/databases related to UDS parameters vs. raw CAN parameters.

Does my vehicle support UDS?

Since most ECUs today support UDS communication, the answer is in "yes, most likely".

If you're the vehicle manufacturer, you will in most cases have the information required to construct UDS requests for whatever data you need - and you'll also know how to decode it.

For the specific case of recording WWH-OBD data in EU trucks, standardized DID information can be recorded by both OEMs and after-market users - similar to how you can record public OBD2 PIDs from cars/trucks.

Beyond the above, if you are in the after market and wish to record proprietary UDS information from a car/truck, it will be difficult. In this case, you would either have to contact the OEM (e.g. as a system integrator/partner) or identify the request/decoding information through reverse engineering. The latter is in practice impossible for most people.

In select cases you may be able to utilize public information shared on e.g. github to help in constructing UDS requests - and decoding the responses. Just keep in mind that public resources are based on reverse engineering efforts - and may risk being incorrect or outdated. You should therefore take all information with a significant grain of salt.

Why is UDS increasingly important?

If your use case involves recording data from cars produced between 2008 and 2018, you will most often be interested in data that can be collected via OBD2 data logging. This is because most ICE cars after 2008 support a large share of the public OBD2 parameter identifiers like speed, RPM, throttle position, fuel level etc.
However, the availability of OBD2 data is expected to decrease over time for multiple reasons.

First of all, as we explained in the previous section, WWH-OBD (based on UDS) or OBDonUDS are expected to gradually replace OBD2 as the de facto standard for vehicle diagnostics.

Second, with the rise of electric vehicles, legislated OBD2 is not necessarily supported at all. And even if an EV supports some OBD2 PIDs, you'll note from our OBD2 PID list that some of the most relevant EV parameters like State of Charge (SoC) and State of Health (SoH) are not available via OBD2. In contrast, UDS remains supported in most EVs and will provide access to a far broader range of data - although without the after-market convenience of a public list of UDS parameters (at least yet). It is expected that EV sales will overtake ICE car sales between 2030 and 2040 - and thus UDS communication will become increasingly relevant.

About the CANedge CAN logger

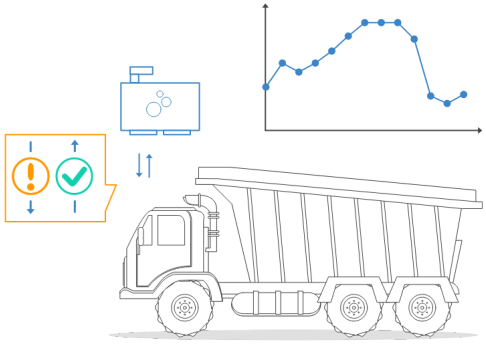
The CANedge lets you easily record CAN/UDS data to an 8-32 GB SD card. You can customize what CAN frames to send, incl. custom UDS requests and flow control frames. Data can be processed via free software/API tools.

The CANedge2 (WiFi) and CANedge3 (3G/4G) also let you auto-upload data to your own server/cloud, while all CANedge devices are available with internal GPS/IMU for adding 40+ signals like position, speed, trip distance and more to your logs.

Example 1: Record single frame UDS data (Speed via WWH-OBD)

To show how UDS works in practice, we will start with a basic example. As outlined before, WWH-OBD is based on UDS - and is mandated in all EU trucks after 2014.

As part of this, many EU heavy duty trucks will let you request parameters like speed, RPM, fuel level etc in a way similar to how you'd request this information via OBD2 PID requests in a car - see our OBD2 intro and OBD2 data logger intro for details. However, under WWH-OBD (ISO 21745-2), the OBD2 PIDs are replaced by the WWH-OBD DIDs.



For service 01, WWH-OBD PIDs are equivalent to the OBD2 PIDs, except that 0xF4 is added in front. For example, the OBD2 PID Vehicle Speed is 0x0D - which becomes 0xF40D in the WWH-OBD context.

In this case, we will use a CANedge2 CAN bus data logger as our "tester" tool. This tool lets you record raw CAN bus data, as well as transmit custom CAN frames. To request WWH-OBD Vehicle Speed, we will use the UDS service 0x22 (Read Data by Identifier) and the Data Identifier 0xF40D. The request/response looks as below:

UDS on CAN - single frame communication example

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
4.0435	18DB33F1	03 22 F4 0D AA AA AA AA	CANedge (client)	Single Frame (SF)
4.0468	18DAF100	04 62 F4 0D 32 AA AA AA	ECU (server)	First Frame (FF)

Legend
PCI field
UDS SID (request)
UDS DID
UDS SID (response)
padding/unused
payload (1 byte)

Communication flow details

Note how the request is sent with CAN ID 0x18DB33F1. This is the same 29-bit CAN ID that would be used to perform a functionally addressed OBD2 PID request in heavy duty vehicles and can be compared with the 11-bit 0x7DF CAN ID used for OBD2 requests in cars.

The response has CAN ID 0x18DAF100, an example of a physical response ID matching the IDs you'd see in regular OBD2 responses from heavy duty vehicles.

Let's break down the communication flow message payloads:

First, the CANedge2 sends a Single Frame (SF) request:

- The initial 4 bits of the PCI field equal the frame type (0x0 for SF)
- The next 4 bits of the PCI field equal the the length of the request (3 bytes)
- The 2nd byte contains the Service Identifier (SID), in this case 0x22
- The 3rd and 4th bytes contain the DID for Vehicle Speed in WWH-OBD (0xF40D)
- The remaining bytes are padded

In response to this request, the truck responds with a Single Frame (SF) response:

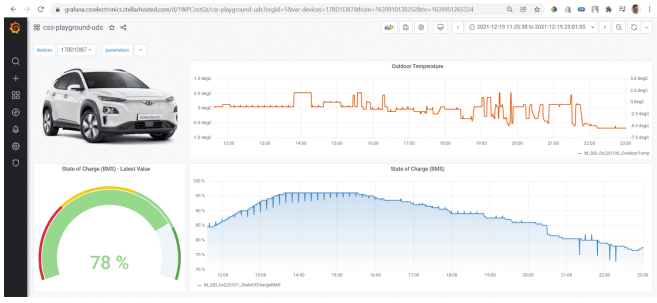
- The 1st byte again reflects the PCI field (now with a length of 4 bytes)
- The 2nd byte is the response SID for Read Data by Identifier (0x62, i.e. 0x22 + 0x40)
- The 3rd and 4th bytes again contain the DID 0xF40D
- The 5th byte contains the value of Vehicle Speed, 0x32

Here we can use the same decoding rules as for ISO/SAE OBD2, meaning that the physical value of Vehicle Speed is simply the decimal form of 0x32 - i.e. 50 km/h. See also our OBD2 PID conversion tool.
If you are familiar with logging OBD2 PIDs, it should be evident that WWH-OBD requests are very similar, except for using the UDSonCAN payload structure for requests/responses.

Example 2: Record & decode multi frame UDS data (SoC)

In this section, we illustrate how multi frame UDS communication works in the context of CAN ISO-TP.

Specifically, we will use the CANedge2 and the UDS service SID 0x22 (Read Data By Identifier) to request the current value of State of Charge (SoC%) from a Hyundai Kona electric vehicle.



First, the CANedge is configured to send two CAN frames:

1. A Single Frame (SF) request (period: 5000 ms, delay: 0 ms)
2. A Flow Control (FC) frame (period: 5000 ms, delay: 100 ms)

A subset of the resulting communication flow looks as below:

UDS on CAN - multiframe communication example

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.0135	7E4	03 22 01 01 AA AA AA AA	CANedge (client)	Single Frame (SF)
1.0228	7EC	10 3E 62 01 01 FF F7 E7	ECU (server)	First Frame (FF)
1.0235	7E4	30 00 00 00 00 00 00 00	CANedge (client)	Flow Control (FC)
1.0426	7EC	21 FF 96 3E 11 42 90 83	ECU (server)	Consecutive Frame (CF)
1.0486	7EC	22 00 12 0E F3 17 16 16	ECU (server)	Consecutive Frame (CF)
1.0586	7EC	23 16 17 16 00 00 18 C3	ECU (server)	Consecutive Frame (CF)
1.0687	7EC	24 02 C2 0B 00 00 92 00	ECU (server)	Consecutive Frame (CF)
1.0787	7EC	25 01 04 50 00 01 02 0E	ECU (server)	Consecutive Frame (CF)
1.0886	7EC	26 00 00 63 F3 00 00 60	ECU (server)	Consecutive Frame (CF)
1.0986	7EC	27 74 00 3D A5 07 0D 01	ECU (server)	Consecutive Frame (CF)
1.1086	7EC	28 7F 00 00 00 00 03 E8	ECU (server)	Consecutive Frame (CF)
Reassembled UDS frame				
1.0135	7EC	62 01 01 FF F7 E7 00 00 00 ... 7F 00 00 00 00 03 E8		

Legend
PCI field
UDS SID (request)
UDS DID
UDS SID (response)
padding/unused
FC block size, ST
payload (59 bytes)

Communication flow details

In the following we explore this communication between the CANedge and ECU in detail. First of all, the initial Single Frame (SF) request is constructed via the same logic as in our previous example - containing the PCI field, the SID 0x22 and the DID. In this case, we use the DID 0x0101. In response to the initial SF request, the targeted ECU sends a First Frame (FF) response:

- The initial 4 bits equal the frame type (0x1 for FF)
- The next 12 bits equal the data payload size, in this case 62 bytes (0x03E)
- The 3rd byte is the response SID for Read Data by Identifier (0x62, i.e. 0x22 + 0x40)
- The 4th and 5th bytes contain the Data Identifier (DID) 0x0101
- The remaining bytes contain the initial part of the data payload for the DID 0x0101

Following the FF, the tester tool now sends the Flow Control (FC) frame:

- The initial 4 bits equal the frame type (0x3 for FC)
- The next 4 bits specifies that the ECU should "Continue to Send" (0x0)
- The 2nd byte sets remaining frames to be sent without flow control or delay
- The 3rd byte sets the minimum consecutive frame separation time (ST) to 0

Once the ECU receives the FC, it sends the remaining Consecutive Frames (CF):

- The initial 4 bits equal the frame type (0x2 for CF)
- The next 4 bits equal the index counter, incremented from 1 up to 8 in this case
- The remaining 7 bytes of each CF contain the rest of the payload for the DID

Regarding proprietary UDS data

Part of the information used here is proprietary. In particular, it is generally not known what Data Identifier (DID) to use in order to request e.g. State of Charge from a given electric vehicle, unless you're the vehicle manufacturer (OEM). Further, as explained in the next section, it is not known how to decode the response payload.

However, various online resources exist e.g. on github, where enthusiasts create open source databases for specific parameters and certain cars (based on reverse engineering). The information we use for this specific communication is taken from one such database.

Regarding the CAN IDs used

In this case we use the CAN ID 0x7E4 to request data from a specific ECU, which in turn responds with CAN ID 0x7EC. This is known as a physically addressed request.

In contrast, functionally addressed request would use the CAN ID 0x7DF (or 0x18DB33F1 in heavy duty vehicles). Generally, request/response CAN IDs are paired (as per the table below) and you can identify the physical request ID corresponding to a specific physical response ID by subtracting the value 8 from the response ID. In other words, if an ECU responds via CAN ID 0x7EC, the physical request ID targeting that ECU would be 0x7E4 (as in our EV example).

Since you may not know what address to target initially, you can in some cases start by sending out a functional request using the CAN ID 0x7DF, in which case the relevant ECU should provide a positive First Frame response if the initial request payload is structured correctly. In some vehicles, you may be able to also send the subsequent Flow Control frame using the same CAN ID, 0x7DF, in order to trigger the ECU to send the remaining Consecutive Frames. However, some implementations may require that you instead utilize the physical addressing request ID for the Flow Control frame.

Implementing a request structure with dynamically updating CAN IDs may be difficult. If you're the manufacturer, you will of course know the relevant CAN IDs to use for sending physically addressed service requests. If not, you may perform an analysis using e.g. a CAN bus interface to identify what response CAN IDs appear when sending functionally addressed service requests - and using this information to construct your configuration.

On a separate note, ISO 15765-4 states that enhanced diagnostics requests/responses may utilize the legislated OBD2 CAN ID range as long as it does not interfere - which is what we are seeing in this specific Hyundai Kona example where the IDs 0x7EC/0x7E4 are used for proprietary data. See also the table from ISO 15765-4 for an overview of the legislated OBD CAN identifiers for use in functional and physical OBD PID requests.

Legislated OBD CAN identifiers (11-bit)

CAN ID	Description
0x7DF	Functionally addressed request sent by tester
0x7E0 to 0x7E7	Physical request from tester to ECU #1 to #8
0x7E8 to 0x7EF	Physical response from ECU #1 to #8 to tester

Legislated OBD CAN identifiers (29-bit)

CAN ID	Description
0x18DB33F1	Functionally addressed request sent by tester
0x18DAxxF1	Physical request from tester to ECU #xx
0x18DAF1xx	Physical response from ECU #xx to tester

Regarding timing parameters

In the above example, we generally focus on the sequence of CAN frames. The sequence is important: For example, if your tester tool sends the Flow Control frame before receiving the First Frame, the Flow Control frame will either be ignored (thus not triggering the Consecutive Frames) or cause an error.

However, in addition to this, certain timing thresholds will also need to be satisfied. For example, if your tester tool receives the First Frame from an ECU of a multi frame response, the ECU will 'time out' if the Flow Control frame is not sent within a set time period.

As a rule of thumb, you should configure your tester (e.g. the CANedge) so that the Flow Control frame is always sent after the First Frame response is received from the ECU (typically this happens within 10-50 ms from sending the initial request) - but in a way so that it is sent within a set time after receiving the First Frame (e.g. within 0-50 ms). For details on this, feel free to contact us.

How to reassemble and decode multi-frame UDS data?

We've now shown how you can request/record a multi-frame UDS response to collect proprietary ECU sensor data. In order to extract 'physical values' like State of Charge, you need to know how to interpret the response CAN frames. As explained, the 'decoding' information is typically proprietary and only known to the OEM. However, in the specific case of the Hyundai Kona EV, we know the following about the SoC signal from online resources:

- The signal is in the 8th byte of the data payload
- The signal is Unsigned
- The signal has a scale 0.5, offset 0 and unit "%"

So how do we use this knowledge to decode the signal?

First, we need to reassemble the segmented CAN frames. The result of this is shown in the previous communication example. Via reassembly, we get a "CAN frame" with ID 0x7EC and a payload exceeding 8 bytes. The payload in this case contains the SID in the 1st byte and DID in the 2nd and 3rd bytes.

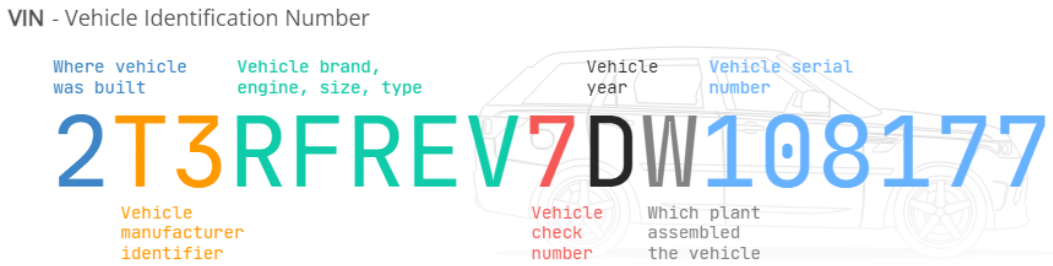
You could process the reassembled CAN frame manually in e.g. Excel. However, we generally recommend to use CAN databases (DBC files) to store decoding rules. In this particular case, you can treat the reassembled CAN frame as a case of extended multiplexing. We provide an example UDS DBC file for the Hyundai Kona incl. State of Charge and temperature signals, which can be useful as inspiration.

Our CAN bus Python API enables reassembly & DBC decoding of multi-frame UDS responses - see our API examples repository for more details incl. the Hyundai Kona sample data.

```
BO_ 2028 Battery: 62 Vector__XXX
 SG_ M_SID_0x220101_StateOfChargeBMS m257 : 56|8@1+ (0.5,0) [0|0] "%" Vector__XXX
 SG_ response m98M : 15|16@0+ (1,0) [0|0] "unit" Vector__XXX
 SG_ service M : 7|8@0+ (1,0) [0|0] "" Vector__XXX

BO_ 1979 Temperature: 54 Vector__XXX
 SG_ M_SID_0x220100_IndoorTemp m256 : 64|8@1+ (0.5,-40) [0|0] "degC" Vector__XXX
 SG_ response m98 : 15|16@0+ (1,0) [0|0] "unit" Vector__XXX
 SG_ service M : 7|8@0+ (1,0) [0|0] "" Vector__XXX
 SG_ M_SID_0x220100_OutdoorTemp : 79|8@0+ (0.5,-40) [0|0] "" Vector__XXX
 SG_ M_SID_0x220100_Speed : 255|8@0+ (0.5,0) [0|0] "" Vector__XXX


BA_DEF_ BO_  "VFrameFormat" ENUM  "StandardCAN","ExtendedCAN","StandardCAN_FD","ExtendedCAN_FD","J1939PG";
BA_DEF_  "ProtocolType" STRING ;
BA_DEF_DEF_  "VFrameFormat" "";
BA_DEF_DEF_  "ProtocolType" "";
BA_ "ProtocolType" "";
BA_ "VFrameFormat" BO_ 2028 0;

SG_MUL_VAL_ 2028 M_SID_0x220101_StateOfChargeBMS response 257-257;
SG_MUL_VAL_ 2028 response service 98-98;
```

Example 3: Record the Vehicle Identification Number

The Vehicle Identification Number (aka VIN, chassis number, frame number) is a unique identifier code used for road vehicles. The number has been standardized and legally required since the 1980s - for details see the VIN page on Wikipedia.



A VIN consists of 17 ASCII characters and can be extracted on-request from a vehicle. This is useful in e.g. data logging or telematics use cases where a unique identifier is required for association with e.g. CAN bus log files.

CAN data bytes can be converted from HEX to ASCII via tables, online HEX to ASCII converters, Python packages etc. For example, the byte 0x47 corresponds to the letter "G". Since a VIN is 17 bytes (17 ASCII characters) long, it does not fit into a single CAN frame, but has to be extracted via a multi frame diagnostic request/response as in Example 2. Further, the VIN is extracted differently depending on the protocol used.

Below we provide three examples on how to record the VIN.

3.1: How to record the VIN via OBD2 (SAE J1979)

To extract the Vehicle Identification Number from e.g. a passenger car using OBD2 requests, you use Service 0x09 and the PID 0x02:

VIN - Vehicle Identification Number request/response (OBD2)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.0135	7E0	02 09 02 AA AA AA AA AA	CANedge (client)	Single Frame (SF)
1.0228	7E8	10 14 49 02 01 32 54 33	ECU (server)	First Frame (FF)
1.0235	7E0	30 00 00 00 00 00 00 00	CANedge (client)	Flow Control (FC)
1.0426	7E8	21 52 46 52 45 56 37 44	ECU (server)	Consecutive Frame (CF)
1.0486	7E8	22 57 31 30 38 31 37 37	ECU (server)	Consecutive Frame (CF)

Reassembled OBD2 frame

1.0135	7E8	49 02 01 32 54 33 52 46 52 45 56 37 44 57 31 30 38 31 37 37

VIN = 2T3RFREV7DW108177

Legend
- PCI field
- OBD service (request)
- OBD PID
- OBD service (response)
- NODI
- padding/unused
- FC block size, ST
- payload (17 bytes)

Communication flow details

The logic of the frame structure is identical to Example 2, with the tester tool sending a Single Frame request with the PCI field (0x02), request service identifier (0x09) and data identifier (0x02).

The vehicle responds with a First Frame containing the PCI, length (0x014 = 20 bytes), response SID (0x49, i.e. 0x09 + 0x40) and data identifier (0x02). Following the data identifier is the byte 0x01 which is the Number Of Data Items (NODI), in this case 1 (see SAE J1979 or ISO 15031-5 for details).

The remaining 17 bytes equal the VIN and can be translated from HEX to ASC via the methods previously discussed.

3.2: How to record the VIN via UDS (ISO 14229-2)

To read the Vehicle Identification Number via UDS, you can use the UDS SID 0x22 and the DID 0xF190:

VIN - Vehicle Identification Number request/response (UDS on CAN)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.0135	7E0	03 22 F1 90 AA AA AA AA	CANedge (client)	Single Frame (SF)
1.0228	7E8	10 14 62 F1 90 32 54 33	ECU (server)	First Frame (FF)
1.0235	7E0	30 00 00 00 00 00 00 00	CANedge (client)	Flow Control (FC)
1.0426	7E8	21 52 46 52 45 56 37 44	ECU (server)	Consecutive Frame (CF)
1.0486	7E8	22 57 31 30 38 31 37 37	ECU (server)	Consecutive Frame (CF)

Reassembled UDS frame

1.0135	7E8	62 F1 90 32 54 33 52 46 52 45 56 37 44 57 31 30 38 31 37 37

VIN = 2T3RFREV7DW108177

Legend
PCI field
UDS SID (request)
UDS DID
UDS SID (response)
padding/unused
FC block size, ST
payload (17 bytes)

Communication flow details

As evident, the request/response communication flow looks similar to the OBD2 case above. The main changes relate to the use of the UDS service 0x22 instead of the OBD2 service 0x09 - and the use of the 2-byte UDS DID 0xF190 instead of the 1 byte OBD2 PID 0x02. Further, the UDS response frame does not include the Number of Data Items (NODI) field after the DID, in contrast to what we saw in the OBD2 case.

3.3: How to record the VIN via WWH-OBD (ISO 21745-3)

If you need to request the Vehicle Identification Number from an EU truck after 2014, you can use the WWH-OBD protocol. The structure is identical to the UDS example, except that WWH-OBD specifies the use of the DID 0xF802 for the VIN.

VIN - Vehicle Identification Number request/response (WWH-OBD on CAN)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.0135	18DB33F1	03 22 F8 02 AA AA AA AA	CANedge (client)	Single Frame (SF)
1.0228	18DAF100	10 14 62 F8 02 32 54 33	ECU (server)	First Frame (FF)
1.0235	18DB33F1	30 00 00 00 00 00 00 00	CANedge (client)	Flow Control (FC)
1.0426	18DAF100	21 52 46 52 45 56 37 44	ECU (server)	Consecutive Frame (CF)
1.0486	18DAF100	22 57 31 30 38 31 37 37	ECU (server)	Consecutive Frame (CF)

Reassembled WWH-OBD frame

1.0135	18DAF100	49 F8 02 32 54 33 52 46 52 45 56 37 44 57 31 30 38 31 37 37

VIN = 2T3RFREV7DW108177

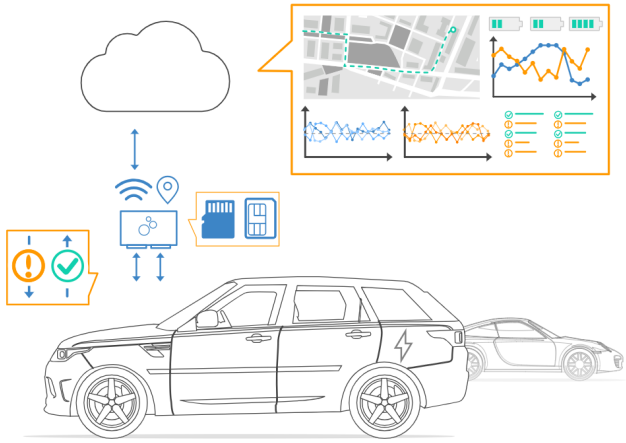
Legend
PCI field
UDS SID (request)
UDS DID
UDS SID (response)
padding/unused
FC block size, ST
payload (17 bytes)

UDS data logging - applications

In this section, we outline example use cases for recording UDS data.

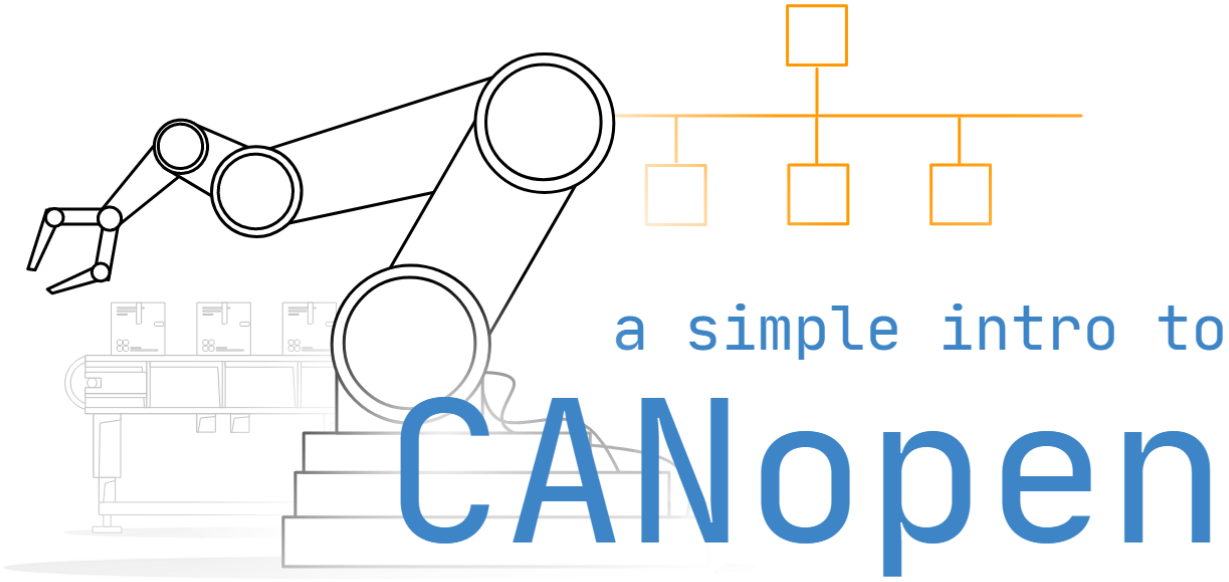
UDS telematics for prototype EVs

As an OEM, you may need to get data on various sensor parameters from prototype EVs while they are operating in the field. Here, the CANedge3 can be deployed to request data on e.g. state of charge, state of health, temperatures and more by transmitting UDS request frames and flow control frames periodically. The data is combined with the internally generated GPS/IMU data and sent via a 3G/4G to your own cloud server for analysis via Vector tools, Python or MATLAB.



Training a predictive maintenance model

If you're looking to implement predictive maintenance across fleets of heavy duty vehicles, the first step is typically to "train your model". This requires large amounts of training data to be collected, including both sensor data (speed, RPM, throttle position, tire pressures etc) and "classification results" (fault / no fault). One way to obtain the latter is by periodically requesting diagnostic trouble codes from the vehicle, providing you with log files that combine both types of data over time. You can use the CANedge1 to collect this data offline onto SD cards, or the CANedge2 to automatically offload the data - e.g. when the vehicles return to stationary WiFi routers in garages.



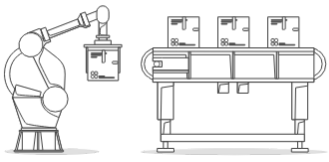
CANopen Explained - A Simple Intro

In this guide we introduce the CANopen protocol basics incl. the object dictionary, services, SDO, PDO and master/slave nodes. CANopen can seem complex - so this tutorial is a visual intro in layman's terms.

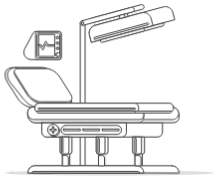
What is CANopen?

CANopen is a CAN based communication protocol.

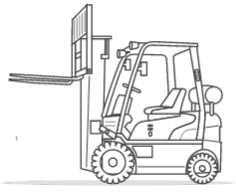
The CANopen standard is useful as it enables off-the-shelf interoperability between devices (nodes) in e.g. industrial machinery. Further, it provides standard methods for configuring devices - also after installation. CANopen was originally designed for motion-oriented machine control systems. Today, CANopen is extensively used in motor control (stepper/servomotors) - but also a wide range of other applications:



Robotics
Automated robotics, conveyor belts & other industrial machinery



Medical
X-ray generators, injectors, patient tables & dialysis devices



Automotive
Agriculture, railway, trailers, heavy duty, mining, marine & more

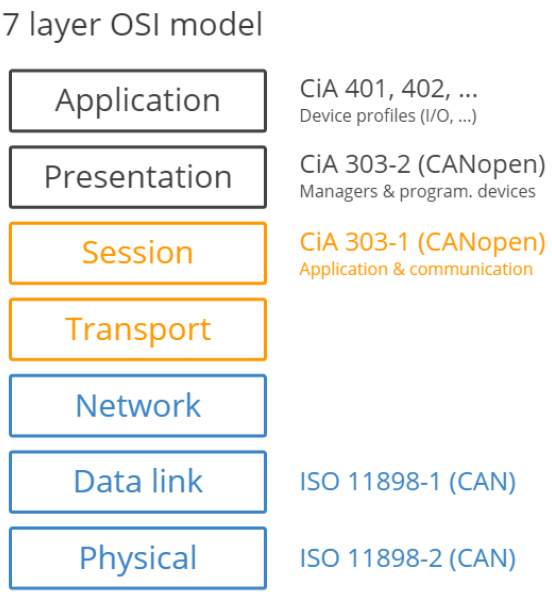
CANopen - higher layer protocol

The following is important to understand:

CANopen is a "higher layer protocol" based on CAN bus.

This means that CAN bus (ISO 11898) serves as the 'transport vehicle' (like a truck) for CANopen messages (like containers).

You can view CANopen from a 7-layer OSI model.



CANopen in OSI model context

The OSI model is a conceptual model standardizing communication functions across diverse communication technologies. Lower layers describe basic communication (e.g. raw bit streams), while higher layers describe things like segmentation of long messages and services like initiating, indicating, responding, and confirming of messages.
CAN bus represents the two lowest layers (1: Physical, 2: Data Link). This means that CAN simply enables the transmission of frames with an 11 bit CAN ID, a remote transmission (RTR) bit and 64 data bits (fields relevant to higher-layer protocols). In other words, CAN bus plays the same role in CANopen as it does in e.g. the J1939 protocol.
As evident above, CANopen implements the 7th layer of the OSI model (Application) via a set of standards. As part of this, it adds several important concepts that we detail below. It's worth noting, that CANopen could also be adapted to other data link layer protocols than CAN (e.g. EtherCAT, Modbus, Powerlink). To fully understand CAN bus vs. CANopen, see also our CAN bus intro tutorial.

CANopen FD

In this article we primarily focus on CANopen based on Classical CAN. However, it is worth noting that as CAN FD is being rolled out, CANopen FD may play an increasingly important role as the next generation of CANopen. For details, see the overview by CAN in Automation on CANopen FD.

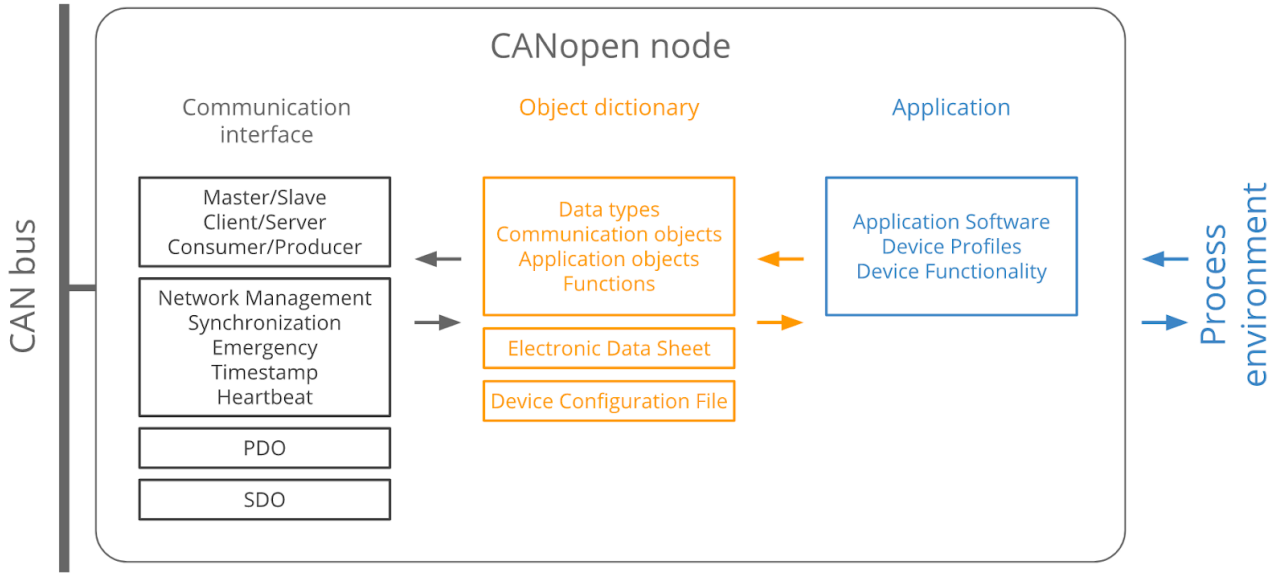
Six core CANopen concepts

Even if you're familiar with CAN bus and e.g. J1939, CANopen adds a range of important new concepts:

Communication Models | **Communication Protocols** | **Device States** | **Object Dictionary** | **Electronic Data Sheet** | **Device Profiles Standards** describe e.g. I/O

There are 3 models for device/node communication: Master/slave, client/server and producer/consumer	Protocols are used in communication, e.g. configuring nodes (SDOs) or transmitting real-time data (PDOs)	A device supports different states. A 'master' node can change state of a 'slave' node - e.g. resetting it	Each device has an OD with entries that specify e.g. the device config. It can be accessed via SDOs	The EDS is a standard file format for OD entries - allowing e.g. service tools to update devices	modules (CiA 401) and motion-control (CiA 402) for vendor independence

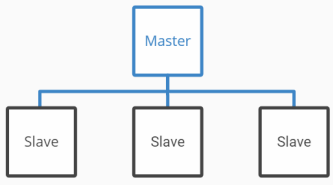
The below illustration shows how the CANopen concepts link together - and we will detail each below:



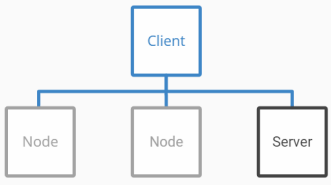
CANopen communication basics

In a CANopen network, several devices need to communicate. For example, in an industrial automation setup you may have a robot arm with multiple servomotor nodes and a control interface/PC node. To facilitate communication, three models exist within CANopen - each closely linked to the CANopen protocols that we look at shortly. See below for a brief introduction:

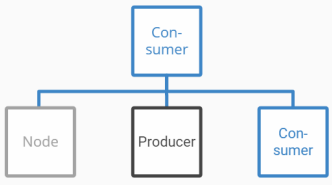
CANopen communication models



Master/Slave
One node (e.g. the control interface) acts as application master or host controller. It sends/requests data from the slaves (e.g. the servo motors). This is used in e.g. diagnostics or state management.



Client/Server
A client sends a data request to a server, which replies with the requested data. Used e.g. when an application master needs data from the OD of a slave. A read from a server is an "upload", while a write is



Consumer/Producer
Here, the producer node broadcasts data to the network, which is consumed by the consumer node. The producer either sends this data on request (pull model) or without a specific request (push model).

There can be 0-127 slaves in standard applications. Note that in a single CANopen network, there can be different host controllers sharing the same data link layer.
Service example: NMT

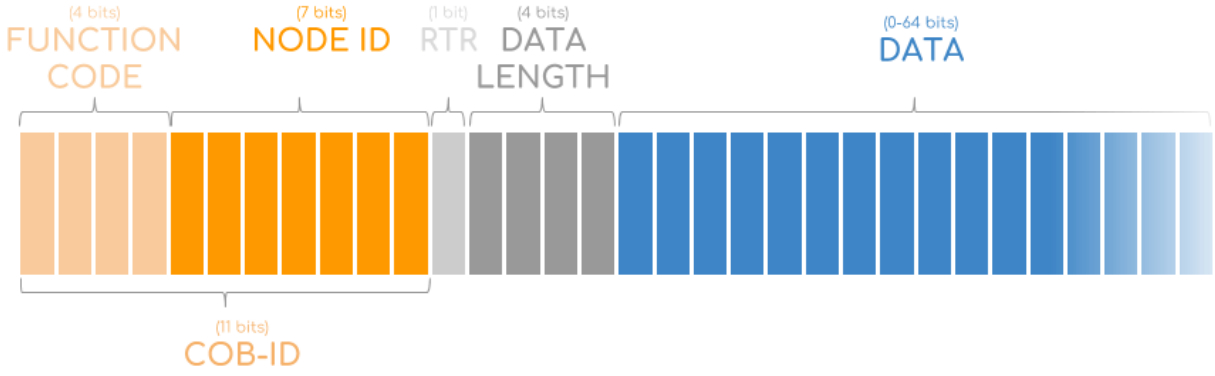
a "download" (the terminology takes a "server side" perspective).
Service example: SDO

Service example: Heartbeat

As evident, the models are practically identical, but we distinguish between them for terminology consistency.

The CANopen frame

To understand CANopen communication, it is necessary to break down the CANopen CAN frame:



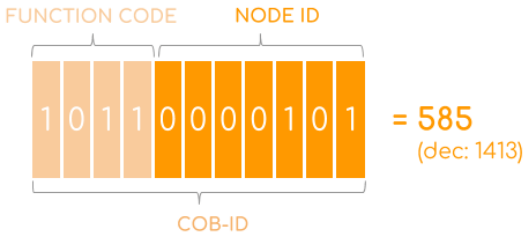
The 11-bit CAN ID is referred to as the Communication Object Identifier (COB-ID) and is split in two parts: By default, the first 4 bits equal a function code and the next 7 bits contain the node ID.

To understand how the COB-ID works, let's take outset in the pre-defined allocation of identifiers used in simple CANopen networks (see the table). Note: We'll refer to COB-IDs and Node IDs in HEX below. As evident, the COB-IDs (e.g. 381, 581, ...) are linked to the communication services (transmit PDO 3, transmit SDO, ...). As such, the COB-ID details which node is sending/receiving data - and what service is used.

	COMMUNICATION OBJECT	FUNCTION CODE (4 bit, bin)	NODE IDs (7 bit, bin)	COB-IDs (hex)	COB-IDs (dec)	#
1	NMT	0000	0000000	0	0	1
2	SYNC	0001	0000000	80	128	1
3	EMCY	0001	0000001-1111111	81 - FF	129 - 255	127
4	TIME	0010	0000000	100	256	1
5	Transmit PDO 1	0011	0000001-1111111	181 - 1FF	385 - 511	127
	Receive PDO 1	0100	0000001-1111111	201 - 27F	513 - 639	127
	Transmit PDO 2	0101	0000001-1111111	281 - 2FF	641 - 767	127
	Receive PDO 2	0110	0000001-1111111	301 - 37F	769 - 895	127
	Transmit PDO 3	0111	0000001-1111111	381 - 3FF	897 - 1023	127
	Receive PDO 3	1000	0000001-1111111	401 - 47F	1025 - 1151	127
	Transmit PDO 4	1001	0000001-1111111	481 - 4FF	1153 - 1279	127
	Receive PDO 4	1010	0000001-1111111	501 - 57F	1281 - 1407	127
6	Transmit SDO	1011	0000001-1111111	581 - 5FF	1409 - 1535	127
	Receive SDO	1100	0000001-1111111	601 - 67F	1537 - 1693	127
7	HEARTBEAT	1110	0000001-1111111	701 - 77F	1793 - 1919	127

Example

A CANopen device with node ID 5 would transmit an SDO via the 11-bit CAN ID 585.
This corresponds to a binary function code of 1011 and a node ID of 5 (binary: 0000101) - see the illustration.



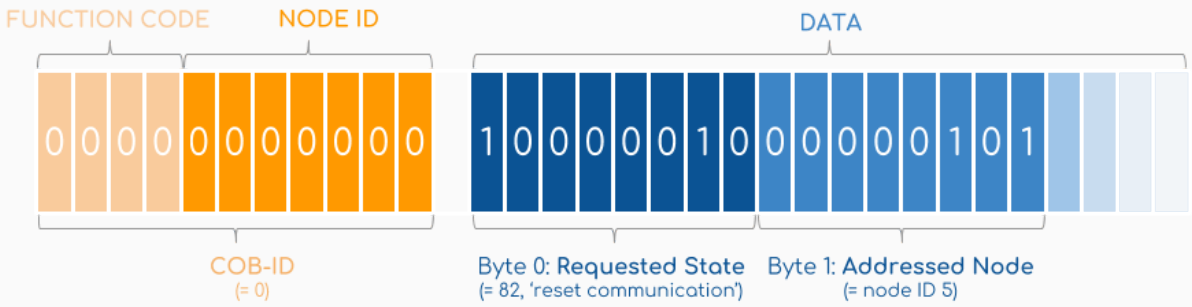
CANopen communication protocols/services

Below we briefly outline the 7 service types mentioned, incl. how they utilize the 8 CAN frame data bytes.

#1 Network Management (NMT)

What is it? The NMT service is used for controlling the state of CANopen devices (e.g. pre-operational, operational, stopped) by means of NMT commands (e.g. start, stop, reset).

How does it work? To change state, the NMT master sends a 2-byte message with CAN ID 0 (i.e. function code 0 and node ID 0). All slave nodes process this message. The 1st CAN data byte contains the requested state - while the 2nd CAN data byte contains the node ID of the targeted node. The node ID 0 indicates a broadcast command.



Possible commands include transition to operational (state 01), to stopped (state 02), pre-operational (state 80) as well as reset application (81) and reset communication (82).

#2 Synchronization (SYNC)

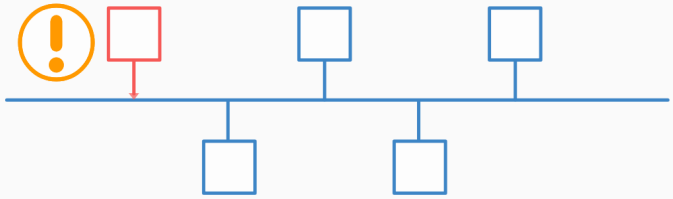
What is it? The SYNC message is used e.g. to synchronize the sensing of inputs and actuation of several CANopen devices - typically triggered by the application master.

How does it work? The application master sends the SYNC message (COB ID 080) to the CANopen network (with or without SYNC counter). Multiple slave nodes may be configured to react to the SYNC and respond by transmitting input data captured at the very same time or by setting the output at the very same time as the nodes participating in the synchronous operation. Using the SYNC counter several groups of synchronously operating devices can be configured.

#3 Emergency (EMCY)

What is it? The emergency service is used in case a device experiences a fatal error (e.g. a sensor failure), allowing it to indicate this to the rest of the network.

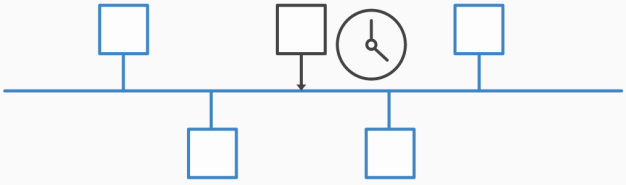
How does it work? The affected node sends a single EMCY message (e.g. with COB-ID 085 for node 5) to the network with high priority. The data bytes contain information about the error, which can be looked up for details.



#4 Timestamp (TIME) [PDO]

What is it? With this communication service a global network time can be distributed. The TIME service contains a 6-byte date & time information.

How does it work? An application master sends out the TIME message with CAN ID 100, where the initial 4 data bytes contain time in ms after midnight and the next 2 bytes contain the number of days since January 1, 1984.



#5 Process Data Object [PDO]

What is it? The PDO service is used to transmit real-time data between devices - e.g. measured data such as position or command data such as torque requests. In this respect it is similar to e.g. broadcasted data parameters in J1939.

How does it work? We'll deep dive on this further below.

#6 Service Data Object [SDO]

What is it? The SDO services are used to access/change values in the object dictionary of a CANopen device - e.g. when an application master needs to change certain configurations of a CANopen device.

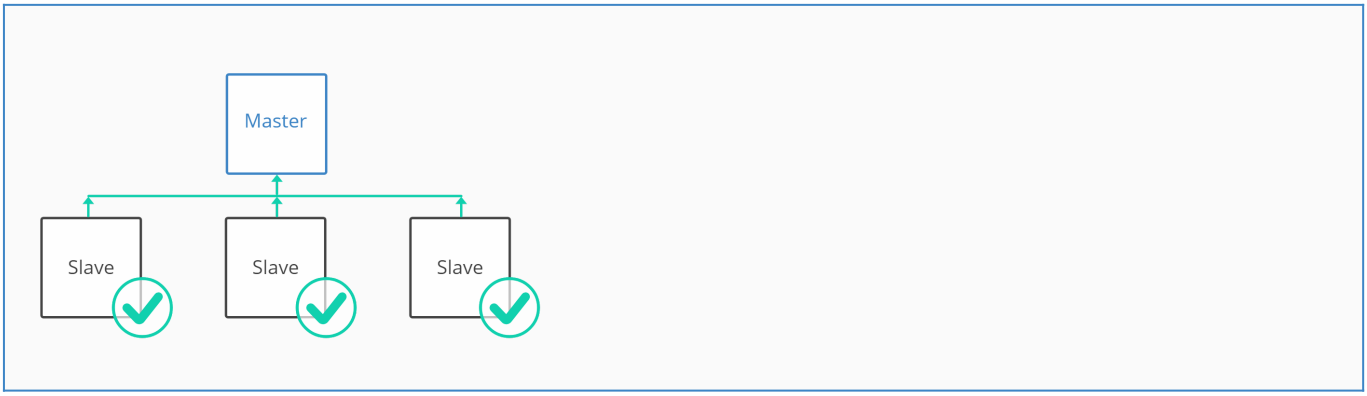
How does it work? We'll deep dive on this further below.

#7 Node monitoring (Heartbeat) [SDO]

What is it? The Heartbeat service has two purposes: To provide an 'alive' message and to confirm the NMT command.

How does it work? An NMT slave device periodically sends (e.g. every 100ms) the Heartbeat message (e.g. with CAN ID 705 for node 5) with the node's "state" in the 1st data byte

The "consumer" of the Heartbeat message (e.g. the NMT master and optionally any other device) then reacts if no message is received in a certain time limit.



The PDO and SDO services are particularly important as they form the basis for most CANopen communication. Below we deep-dive on each of these - but first we need to introduce a core concept of CANopen: The object dictionary.

CANopen Object Dictionary

All CANopen nodes must have an object dictionary (OD) - but what is it? The object dictionary is a standardized structure containing all parameters describing the behavior of a CANopen node. OD entries are looked up via a 16-bit index and 8-bit subindex. For example, index 1008 (subindex 0) of a CANopen-compliant node OD contains the node *device name*.

Specifically, an entry in the object dictionary is defined by attributes:

- Index: 16-bit base address of the object
- Object name: Manufacturer device name
- Object code: Array, variable, or record
- Data type: E.g. VISIBLE_STRING, or UNSIGNED32 or Record Name
- Access: rw (read/write), ro (read-only), wo (write-only)
- Category: Indicates if this parameter is mandatory/optional (M/O)

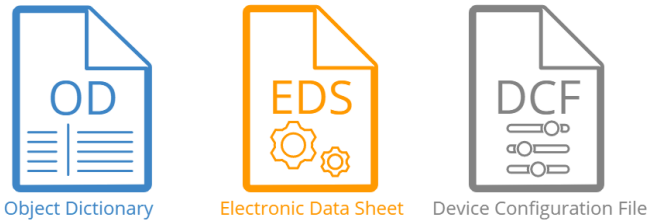
OD standardized sections

The object dictionary is split into standardized sections where some entries are mandatory and others are fully customizable. Importantly, OD entries of a device (e.g. a slave) can be accessed by another device (e.g. a master) via CAN using e.g. SDOs. For example, this might let an application master change whether a slave node logs data via a specific input sensor - or how often the slave sends a heartbeat.

OD INDEX (16 bits, hex)	DESCRIPTION
0000	Reserved
0001 - 025F	Data types
0260 - 0FFF	Reserved
1000 - 1FFF	Communication object area
2000 - 5FFF	Manufacturer specific area
6000 - 9FFF	Device profile specific area
A000 - BFFF	Interface profile specific area
C000 - FFFF	Reserved

Link to Electronic Data Sheet and Device Configuration File

To understand the OD, it is helpful to look at the 'human-readable form': The Electronic Data Sheet and Device Configuration File.

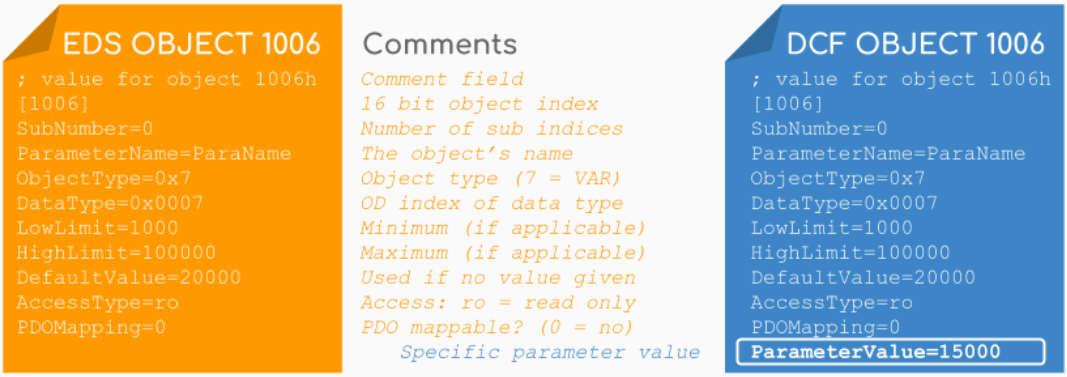


The Electronic Data Sheet (EDS)
In practice, configuring/managing complex CANopen networks will be done using adequate software tools.
To simplify this, the CiA 306 standard defines a human-readable (and machine friendly) INI file format, acting as a "template" for the OD of a device - e.g. the "ServoMotor3000". This EDS is typically provided by the vendor and contains info on all device objects (but not values).

Device Configuration File (DCF)
Assume a factory has bought a ServoMotor3000 to integrate into their conveyor belt. In doing so, the operator edits the device EDS and adds specific parameter values and/or changes the names of each object described in the EDS. In doing so, the operator effectively creates what is known as a Device Configuration File (DCF). With this in place, the ServoMotor3000 is ready for integration into the specific CANopen network on-site.

EDS and DCF example

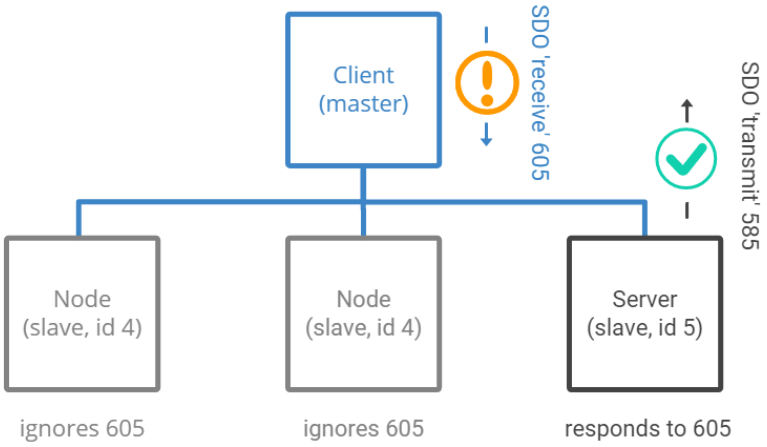
Reviewing real EDS/DCF examples is one of the best ways to really understand the object dictionary of CANopen - see e.g. the difference between an EDS and DCF object entry below. We recommend checking out the CiA 306 standard to gain a deeper understanding of the OD, EDS and DCF with practical examples.



As mentioned, the DCF is typically created upon device integration. However, often it will be necessary to read and/or change the object values of a node after initial configuration - this is where the CANopen SDO service comes into play.

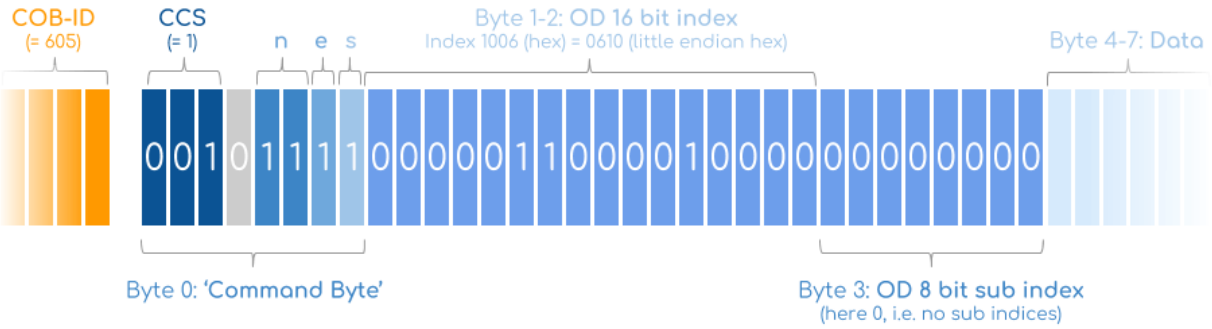
SDO - configuring the CANopen network

What is the SDO service? The SDO service allows a CANopen node to read/edit values of another node's object dictionary over the CAN network. As mentioned under 'communication models', the CANopen SDO services utilize a "client/server" behavior. Specifically, an SDO "client" initiates the communication with one dedicated SDO "server". The purpose can be to update an OD entry (called an "SDO download") or read an entry ("SDO upload"). In simple master/slave networks, the node with NMT master functionality acts as the client for all NMT slave nodes reading or writing to their ODs.



Example: Client node SDO download

The client node can initiate an SDO download to node 5 by broadcasting below CAN frame - which will trigger node 5 (and be ignored by other nodes, see above illustration). The SDO 'receive' (i.e. request) CAN frame looks as below:



SDO message variables explained

- First, COB-ID 605 reflects the use of an 'SDO receive' (COB-ID 600 + node ID).
- The CCS (client command specifier) is the transfer type (e.g. 1: Download, 2: Upload)
- n is the #bytes in data bytes 4-7 that do not contain data (valid if e & s are set)
- If set, e indicates an "expedited transfer" (all data is in a single CAN frame)
- If set, s indicates that data size is shown in n
- Index (16 bits) and subindex (8 bits) reflect the OD address to be accessed
- Finally, bytes 4-7 contain the data to be downloaded to node 5

CANopen SDO example comments

Once the CAN frame is sent by the master (client), the slave node 5 (server) responds via an 'SDO transmit' with COB-ID 585. The response contains the index/subindex and 4 empty data bytes. Naturally, if the client node requested an upload instead (i.e. reading data from the node 5 OD), node 5 would respond with the relevant data contained in bytes 4-7. A few comments:

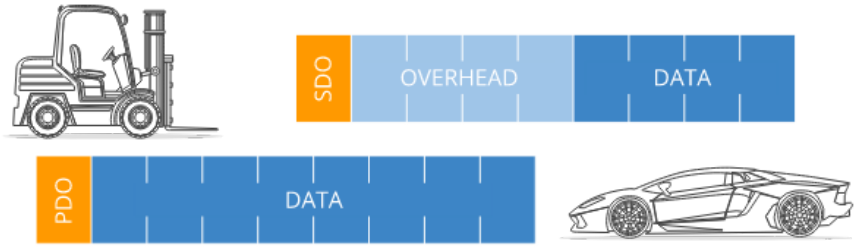
- As evident, each SDO uses 2 identifiers, creating an "SDO channel"
- The example is simplified as it's "expedited" (data is contained in the 4 bytes)
- For larger data scenarios, SDO segmentation/block transfers can be used

SDOs are flexible, but carry a lot of overhead - making them less ideal for real-time operational data. This is where the PDO comes in.

PDO - operating the CANopen network

First of all: What is the CANopen PDO service? The CANopen PDO service is used for effectively sharing real-time operational data across CANopen nodes. For example, the PDO would carry pressure data from a pressure transducer - or temperature data from a temperature sensor. But wait: Can't the SDO service just do this?

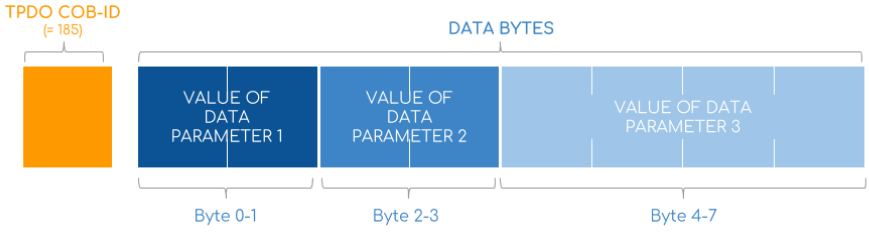
Yes, in principle the SDO service could be used for this. However, a single SDO response can only carry 4 data bytes due to overhead (command byte and OD addresses). Further, let's say a master node needs two parameter values (e.g. "SensTemp2" and "Torque5") from Node 5 - to get this via SDO, it would require 4 full CAN frames (2 requests, 2 responses).



In contrast, a PDO message can contain 8 full bytes of data - and it can contain multiple object parameter values within a single frame. Thus, what would require at least 4 frames with SDO could potentially be done with 1 frame in the PDO service. The PDO is often seen as the most important CANopen protocol as it carries the bulk of information.

How does the CANopen PDO service work?

For PDOs, the consumer/producer terminology is used. Thus, a producer 'produces data', which it transmits to a 'consumer' (master) using a transmit PDO (TPDO). Conversely, it may receive data from the consumer via a receive PDO (RPDO). Producer nodes may e.g. be configured to respond to a SYNC trigger broadcasted by the consumer every 100 ms. Node 5 may then e.g. broadcast below transmit PDO with COB-ID 185:



Note how the data bytes are packed with 3 parameter values. These values reflect real-time data of specific OD entries of node 5. The nodes that use this information (the consumers) of course need to know how to interpret the PDO data bytes.

PDO service vs. J1939 PGNs and SPNs

Isn't the PDO service a bit similar to J1939 PGNs & SPNs? Yes, to some extent this is similar to how a specific J1939 parameter group (PG) will contain multiple SPNs/signals (aka data parameters) in the 8 data bytes. The J1939 CAN frame does not need to waste data bytes on "decoding" information because this is known by the relevant nodes (and by external tools via e.g. J1939 DBC files or the J1939 PDF standards). The wrinkle is that in CANopen, these 'PDO mappings' are often configurable and can be changed during the creation of the DCF and/or via the SDO service. For more details on PDOs, see this article (p. 5).

CANopen data logging - use case examples

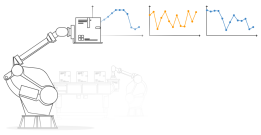
Since CANopen is a CAN-based protocol, it is possible to record raw CANopen frames using a CAN bus data logger. As an example, the CANedge lets you record CANopen data to an 8-32 GB SD card.

Simply connect it to your application to start logging - and process the data via free software/APIs. Learn more!

Raw data decoding via CANopen DBC files

As CANopen is based on CAN bus, you can store your CANopen decoding rules in the standardized CAN database format, a CAN DBC file. With this, you can directly decode your raw CANopen data in open source software/API tools for the CANedge, including the asammdf GUI, telematics dashboards and Python API tools. Note that you may not have a CANopen DBC file readily available, but rather your CANopen frame decoding rules may be stored in your EDS/DCF or in a PDF. In such a case you can take outset in our DBC intro to learn how to construct a CANopen DBC file from scratch using various free editor tools.

Solutions like the CANedge enable several CANopen logging use cases:



Logging CANopen node data

Generally, logging CANopen data can be used to e.g. analyze operational data. WiFi CAN loggers can also be used for e.g. over-the-air SDOs

learn more

Warehouse fleet management

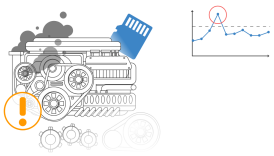
CANopen is often used in EV forklifts/AGVs in warehouses, where monitoring e.g. SoC helps reduce breakdowns and improve battery life

learn more

Predictive maintenance

Industrial machinery can be monitored via IIoT CAN loggers in the cloud to predict and avoid breakdowns based on the CANopen data

learn more



Machinery diagnostic blackbox

A CAN logger can serve as a 'blackbox' for industrial machinery, providing data for e.g. disputes or rare issue diagnostics

learn more

CAN FD Explained - A Simple Intro

In this guide we introduce CAN FD (CAN Flexible Data-rate) - incl. CAN FD frames, overhead & efficiency, example applications and logging use cases.

Why CAN FD?

The CAN protocol has been around since 1986 and it's popular: Practically any machine that moves utilizes CAN today - whether it's cars, trucks, boats, planes or robots.

But with the rise of modern technology, the "Classical" CAN protocol (official term used in ISO 11898-1:2015) is pressured:

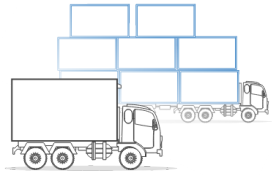
- A rise in vehicle functionality is driving an explosion in data
- Networks are increasingly limited by the 1-Mbit/s bandwidth
- To cope, OEMs create complex & costly workarounds

Specifically, Classical CAN struggles with substantial overhead (>50%) as each CAN data frame can only contain 8 data bytes. Further, the network speed is limited to 1 Mbit/s, restricting the implementation of data-producing features. CAN FD resolves these issues - making it future-proof.

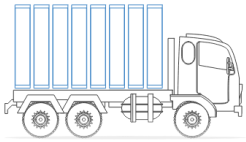
What is CAN FD?

The CAN FD protocol was pre-developed by Bosch (with industry experts) and released in 2012. It was improved through standardization and is now in ISO 11898-1:2015. The original Bosch CAN FD version (non-ISO CAN FD) is incompatible with ISO CAN FD.

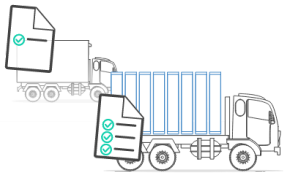
CAN FD offers four major benefits:



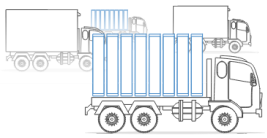
#1 Increased length
CAN FD supports up to 64 data bytes per data frame vs. 8 data bytes for Classical CAN. This reduces the protocol overhead and leads to an improved protocol efficiency.



#2 Increased speed
CAN FD supports dual bit rates: The nominal (arbitration) bit-rate limited to 1 Mbit/s as in Classical CAN - and the data bit-rate, which depends on the network topology/transceivers. In practice, data bit-rates up to 5 Mbit/s are achievable.



#3 Better reliability
CAN FD uses an improved cyclic redundancy check (CRC) and the "protected stuff-bit counter", which lower the risk of undetected errors. This is e.g. vital in safety-critical applications like vehicles and industrial automation.



#4 Smooth transition
CAN FD and Classical CAN only ECUs can be mixed under certain conditions. This allows for a gradual introduction of CAN FD nodes, greatly reducing costs and complexity for OEMs.

In practice, CAN FD can improve network bandwidth by 3-8x vs Classical CAN, creating a simple solution to the rise in data.

How does CAN FD work?

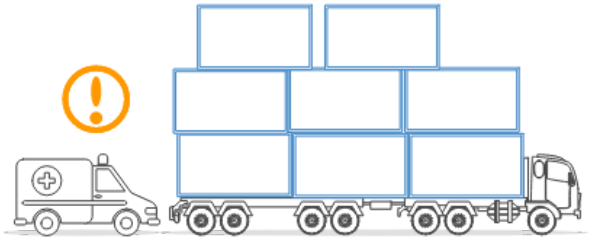
So CAN FD seems pretty simple: Speed up the data transmission and pack more data into each message, right? In practice, however, it's not that straightforward. Below we outline the main challenges that the CAN FD solution had to solve.

Two key challenges

Before looking at the CAN FD data frame, it's key to understand two core parts of Classical CAN that we want to maintain:

#1 Avoid critical message delays

Why not simply pack Classical CAN frames with 64 bytes of data? Doing so would reduce overhead and simplify message interpretation. However, if the bit-rate is unchanged, this would also block the CAN bus for longer, potentially delaying mission-critical higher-priority data frames.



#2 Maintain practical CAN wire lengths

So, more speed is needed to send more data per message. But why not speed up the entire CAN message (rather than just the data phase)? This is due to "arbitration": If 2+ nodes transmit data simultaneously, arbitration determines which node takes priority. The "winner" continues sending (without delay), while the other nodes "back off" during the data transmission.

Regarding bit time

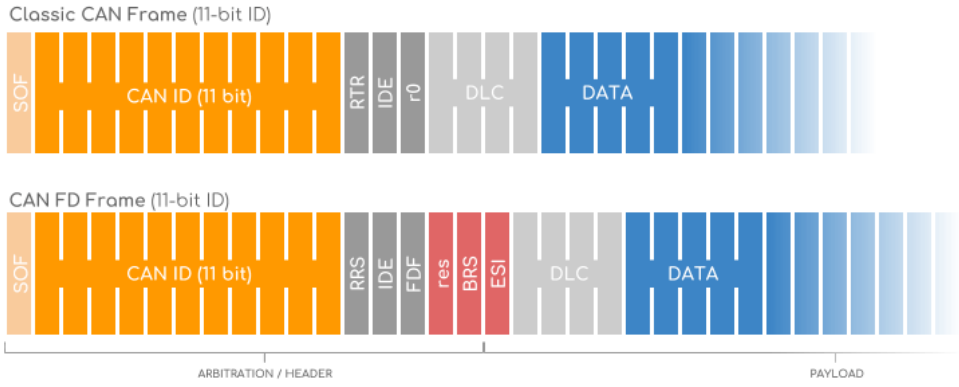
During arbitration, a "bit time" provides sufficient delay between each bit to allow every node on the network to react. To be certain that every node is reached within the bit time, a CAN network running at 1 Mbit/s needs to have a maximum length of 40 metres (in practice 25 metres). Speeding up the arbitration process would reduce the maximum length to unsuitable levels.

On the other hand, after arbitration there's an "empty highway" - enabling high speed during the data transmission (when there is just one node driving the bus-lines). Before the ACK slot - when multiple nodes acknowledge the correct reception of the data frame - the speed needs to be reduced to the nominal bit-rate.

In short, it is necessary to find a way to only increase the speed during the data transmission.

Solution: The CAN FD frame

The CAN FD protocol introduces an adjusted CAN data frame to enable the extra data bytes and flexible bit-rates. Below we compare an 11-bit Classical CAN frame vs. an 11-bit CAN FD frame (29-bit is also supported):



Below we go through the differences step-by-step:

CAN FD frame fields explained

RTR vs. RRS: The Remote Transmission Request (RTR) is used in Classical CAN to identify data frames and corresponding remote frames. In CAN FD, remote frames are not supported at all - the Remote Request Substitution (RRS) is always dominant (0).

r0 vs. FDF: In Classical CAN, r0 is reserved and dominant (0). In CAN FD, it's named FDF and recessive (1).
After the r0/FDF bit, the CAN FD protocol adds "3 new bits". Note that nodes that are not CAN FD capable produce an error frame after the FDF bit.

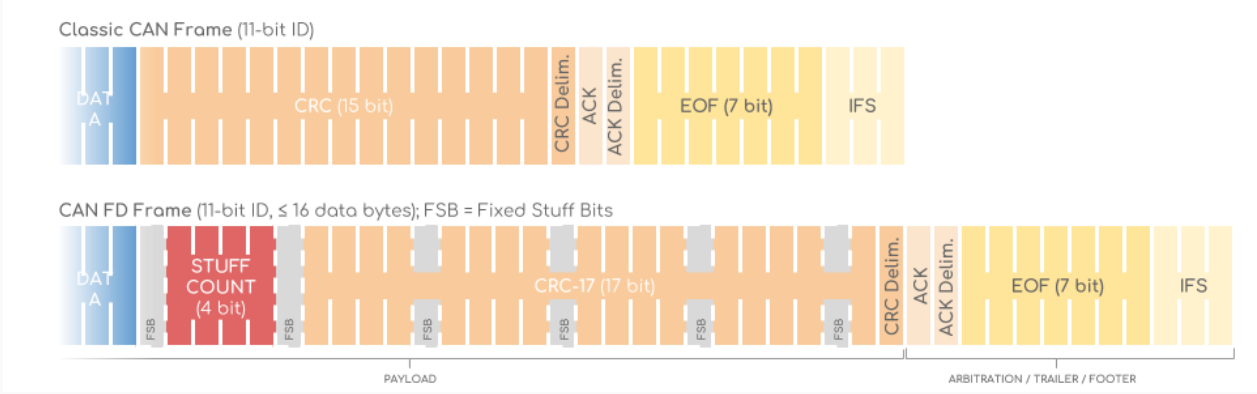
res: This new reserved bit plays the same role as r0 - i.e. it may in the future be set to recessive (1) to denote a new protocol.

BRS: The Bit Rate Switch (BRS) can be dominant (0), meaning that the CAN FD data frame is sent at the arbitration rate (i.e. up to max 1 Mbit/s). Setting it to recessive (1) means that the remaining part of the data frame is sent at a higher bit rate (up to 5 Mbit/s).

ESI: The Error Status Indicator (ESI) bit is by default dominant (0), i.e.'error active'. If the transmitter becomes 'error passive' it'll be recessive (1) to indicate it is in error passive mode.

DLC (bin)	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
DLC (dec)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Classic CAN	0	1	2	3	4	5	6	7	8	8	8	8	8	8	8	8
CAN FD	0	1	2	3	4	5	6	7	8	12	16	20	24	32	48	64

DLC: Like in Classical CAN, the CAN FD DLC is 4 bits and denotes the number of data bytes in the frame. The above table shows how the two protocols use the DLC consistently up to 8 data bytes. To maintain a 4-bit DLC, CAN FD uses the remaining 7 values from 9 to 15 to denote the number of data bytes used (12, 16, 20, 24, 32, 48, 64).



SBC: The Stuff Bit Count (SBC) precedes the CRC and consists of 3 gray-coded bits and a parity bit. The following Fixed Stuff-Bit can be regarded as a second parity bit. The SBC is added to improve communication reliability.

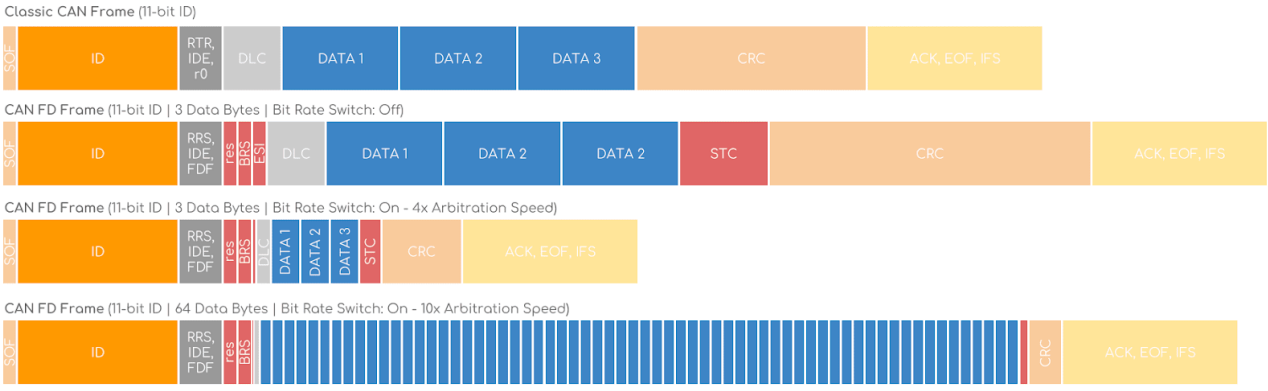
CRC: The Cyclic Redundancy Check (CRC) is 15 bit in the Classical CAN, while in CAN FD it's 17 bits (for up to 16 data bytes) or 21 bits (for 20-64 data bytes). In Classical CAN, there can be 0 to 3 stuff bits in the CRC, while in CAN FD there are always four fixed stuff bits to improve communication reliability.

ACK: The data phase (aka payload) of the CAN FD data frame stops at the ACK bit, which also marks the end of the potentially increased bit rate.

Overhead and data efficiency of CAN FD vs. CAN

As evident, the added functionality of CAN FD adds a lot of extra bits vs. Classical CAN - how can this lead to less overhead?

The answer is that it doesn't - see the below visualization of Classical CAN vs. CAN FD for 3 data bytes. In fact, the efficiency of CAN FD does not exceed Classical CAN until crossing 8 data bytes. However, by moving towards 64 data bytes, the efficiency can go from ~50% up towards ~90% (more on this below).



Need for speed: Turning on bit rate switching

As mentioned, sending 64 data bytes at regular speed would block the CAN bus, reducing the real-time performance. To solve this, bit rate switching can be enabled to allow the payload to be sent at a higher rate vs the arbitration rate (e.g. 5 Mbit/s vs 1 Mbit/s). Above we illustratively visualize the effect for the 3 data byte and 64 data byte scenarios. Note that the higher speed applies to the data frame section starting in the BRS bit and ending in the CRC delimiter. Further, most vehicles today use 0.25-0.5 Mbit/s, meaning that with 5 Mbit/s CAN FD would 10x the speed of the payload transmission.

Mixing Classical CAN and CAN FD nodes

As mentioned earlier, Classical CAN and CAN FD nodes can be mixed under certain conditions. This allows for a step-by-step migration towards CAN FD, rather than having to switch every ECU in one go. Two scenarios exist:

100% CAN FD system: Here, the CAN FD controllers can freely mix Classical CAN and CAN FD data frames.

Some nodes are legacy Classical CAN: Here, the CAN FD controllers can switch to Classical CAN communication to avoid error frame reactions from the Classical CAN nodes. Also, during e.g. ECU flashing, the Classical CAN nodes may be turned off to allow a temporary shift to CAN FD communication.

What is the max bit rate of CAN FD?

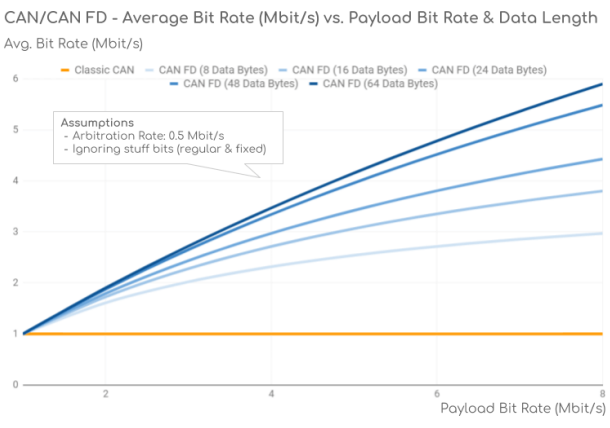
A confusing aspect of CAN FD is the max bit rate during the payload phase - as different articles mention different levels. Some state that practical applications enable up to 8 Mbit/s and theoretically 15 Mbit/s. Others state up to 12 Mbit/s. Further, Daimler state that beyond 5 Mbit/s is doubtful - both as there's no standard for this and because low-cost automotive Ethernet (10 BASE-T1) is expected to limit the demand for CAN FD at higher speed. So what is correct?

It depends. Looking at ISO 11898-2 (the transceiver chip standard), it specifies two symmetry parameter sets. It is recommended to use those with improved symmetry parameters, often advertised as 5 Mbit/s transceivers. The achievable data phase bit-rate depends on many things. One of the most important is the desired temperature range. Flashing of an ECU does not require the support of low temperatures. This implies that for ECU flashing, it is possible to go up to 12 Mbit/s. Another important bit-rate limitation is caused by the chosen topology. Bus-line topology with very short stubs allows significantly higher bit-rates versus hybrid topologies with long stubs or even stars. Most multi-drop bus-line networks are limited to 2 Mbit/s for a temperature range of -40 degC to +125 degC. CiA provides appropriate rules of thumb in the CiA 601-3 network design recommendation. This includes recommendations to set the sample-points in the data-phase.

CAN FD calculator: Efficiency & baud rate

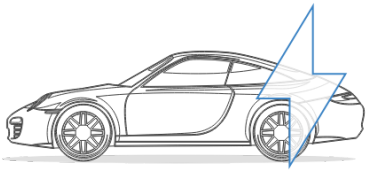
For a detailed understanding of CAN FD efficiency and bit rates, see our CAN FD calculator.

The calculator compares Classical CAN vs. CAN FD in regards to frames and visualizes the frame structures in response to the settings you provide.

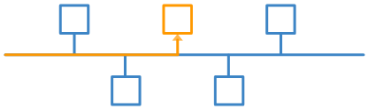


Examples: CAN FD applications

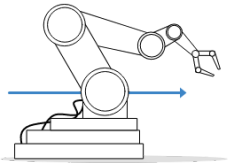
In short, CAN FD enables more data at faster rates. This is key for multiple increasingly relevant use cases:



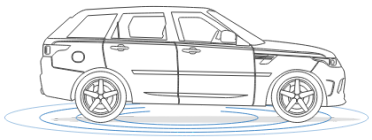
Electric vehicles
EVs and hybrids use new powertrain concepts that require far higher bit rates. Added complexity comes from new control units related to the DC/DC inverter, battery, charger, range extender etc. By 2025 it is expected that the required bit rate exceeds CAN - and with the explosive rise in EVs, this may be the spearhead of the CAN FD roll-out.



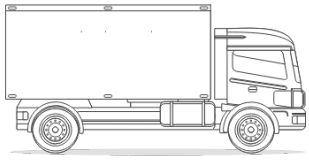
ECU flashing
Vehicle software is becoming increasingly complex. As such, performing ECU updates via e.g. the OBD2 port can take hours today. With CAN FD, such processes can be made >4x faster. This use case has been one of the original drivers behind the demand for CAN FD by automotive OEMs.



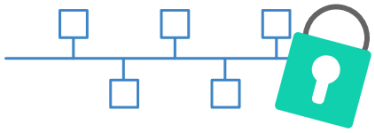
Robotics
Several applications rely on time-synced behaviour - e.g. robot arms with multiple axles. Such devices often use CANopen and require multiple CAN frames (PDOs) to be sent by each controller time-synced (without interruption from higher-priority frames). By shifting to CAN FD, the data can be sent in a single frame for efficiency.



ADAS & safe driving
Increasingly, advanced driver assistance systems (ADAS) are being introduced in passenger cars and commercial vehicles. This pressures the bus load of Classical CAN, yet ADAS is key to improving safety. Here, CAN FD will be key to enhancing safe driving in the near future.



Trucks & buses
Trucks & buses utilize long CAN buses (10-20 meters). As a result, they rely on slow bit rates (250 kbit/s or 500 kbit/s as per J1939-14). Here, the upcoming J1939 FD protocol is expected to enable a significant improvement in commercial vehicle features, incl. e.g. ADAS.



Secure CAN bus
As shown in recent CAN hacker attacks, Classical CAN is vulnerable. If hackers gain access to the CAN bus (e.g. over-the-air), they could e.g. turn off critical functions. CAN FD authentication via the Secure Onboard Communication (SecOC) module may be a key roll-out driver.

Logging CAN FD data - use case examples

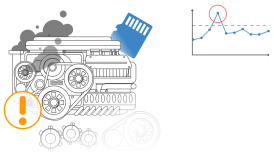
With the rise of CAN FD there will be several use cases for logging CAN FD data:

Logging data from cars
As CAN FD gets rolled out in new cars, CAN FD data loggers will be key for OEM R&D and diagnostics

learn more

Heavy duty fleet telematics
IoT CAN FD loggers compatible with J1939 FD (flexible data-rate) will be key to future heavy duty telematics
learn more

Predictive maintenance
As CANopen FD rolls out, new industrial machinery will need CAN FD IoT loggers to help predict and avoid breakdowns
learn more



Vehicle/machine blackbox
A CAN FD logger can serve as a 'blackbox' for e.g. new prototype vehicles, providing data for diagnostics and R&D
learn more

CAN FD logging - practical considerations

When logging CAN FD data the following considerations are relevant:

#1 ISO CAN FD vs. non-ISO CAN FD

Before the ISO 11898-1:2015 update, the CAN FD standard had a weakness related to error checking. Controllers adhering to the updated standard are sometimes referred to as "ISO CAN FD". When recording data from an early prototype CAN FD system you may therefore need to turn on "NON-ISO CAN FD" mode if your device supports this.

#2 Bit rate switch - on/off?

By default, your CAN FD data logger will be able to handle both Classical CAN and CAN FD data messages - without having to be pre-configured between them. Similarly, you won't have to pre-specify whether bit rate switching is on/off for pure logging purposes. However, when you transmit data to the CAN bus, you'll need to specify whether to use bit rate switching or not. If enabled, your data payloads are transmitted at the system's 2nd bit rate, which will typically be 2 or 4 Mbit/s.

#3 Conversion & CAN FD DBC files

CAN FD minimizes the need for handling multi-packet messages. This can greatly simplify the software development for converting raw CAN FD data to human-readable form, to the benefit of end users of CAN FD analyzers. Further, the standard CAN database format, DBC, also supports CAN FD conversion rules. As such, it's always recommended that you collect your scaling rules in a DBC file to enable easy transition between various CAN software like e.g. asammdf etc.

CAN FD - outlook

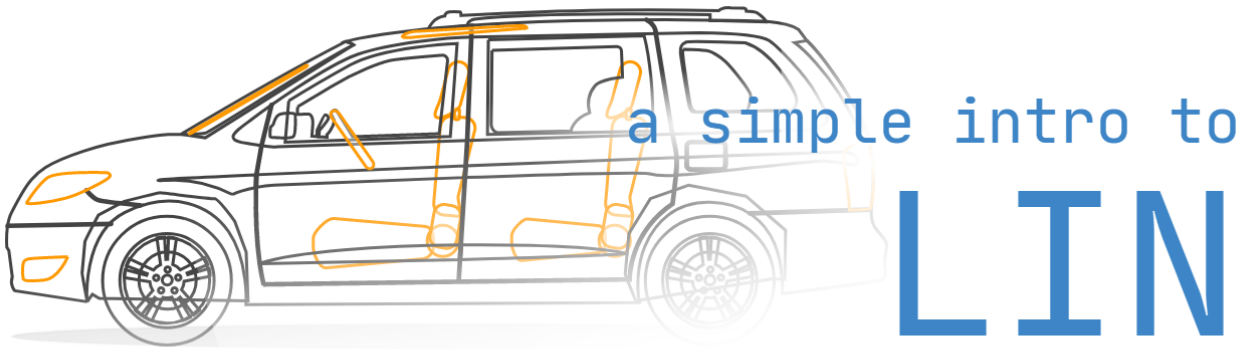
CAN FD is predicted to take over Classical CAN in the coming years:

- The first CAN FD capable cars are expected to be sold in 2019/20
- Initial roll-out will likely use 2 Mbit/s, gradually transitioning to 5 Mbit/s
- CANopen FD has been adapted via CiA 1301 1.0
- J1939-22 uses CAN FD data frames

- CAN is still a growing technology, but recently mainly due to CAN FD
- It's anticipated that in the future, CAN FD will be used in most new development

CAN FD vs. other protocols

Of course, legacy applications with no bandwidth and payload requirements will still use Classical CAN. Further, the CAN community is already developing the next generation of a CAN data link layer supporting payloads up to 2048 bytes. This approach can be regarded as an alternative to 10 Mbit/s Ethernet. As such, it is still to be determined exactly what role CAN FD will play in the future - but it will definitely be on the rise.



LIN Bus Explained - A Simple Intro

In this guide we introduce the Local Interconnect Network (LIN) protocol basics incl. LIN vs. CAN, use cases, how LIN works and the six LIN frame types.

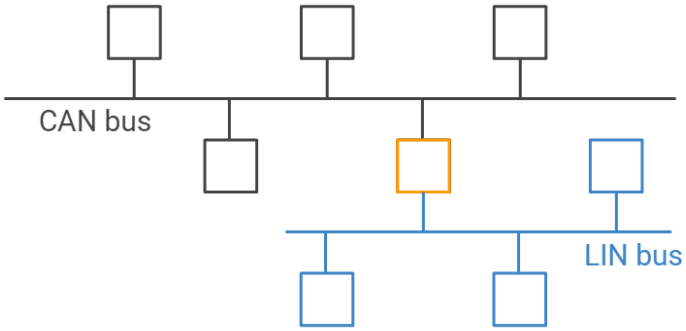
What is LIN bus?

LIN bus is a supplement to CAN bus. It offers lower performance and reliability - but also drastically lower costs. Below we provide a quick overview of LIN bus and a comparison of LIN bus vs. CAN bus.

- Low cost option (if speed/fault tolerance are not critical)
- Often used in vehicles for windows, wipers, air condition etc..
- LIN clusters consist of 1 master and up to 16 slave nodes
- Single wire (+ground) with 1-20 kbit/s at max 40 m bus length
- Time triggered scheduling with guaranteed latency time
- Variable data length (2, 4, 8 bytes)
- LIN supports error detection, checksums & configuration
- Operating voltage of 12V
- Physical layer based on ISO 9141 (K-line)
- Sleep mode & wakeup support
- Most newer vehicles have 10+ LIN nodes

LIN bus vs CAN bus

- LIN is lower cost (less harness, no license fee, cheap nodes)
- CAN uses twisted shielded dual wires 5V vs LIN single wire 12V
- A LIN master typically serves as gateway to the CAN bus
- LIN is deterministic, not event driven (i.e. no bus arbitration)
- LIN clusters have a single master - CAN can have multiple
- CAN uses 11 or 29 bit identifiers vs 6 bit identifiers in LIN
- CAN offers up to 1 Mbit/s vs. LIN at max 20 kbit/s



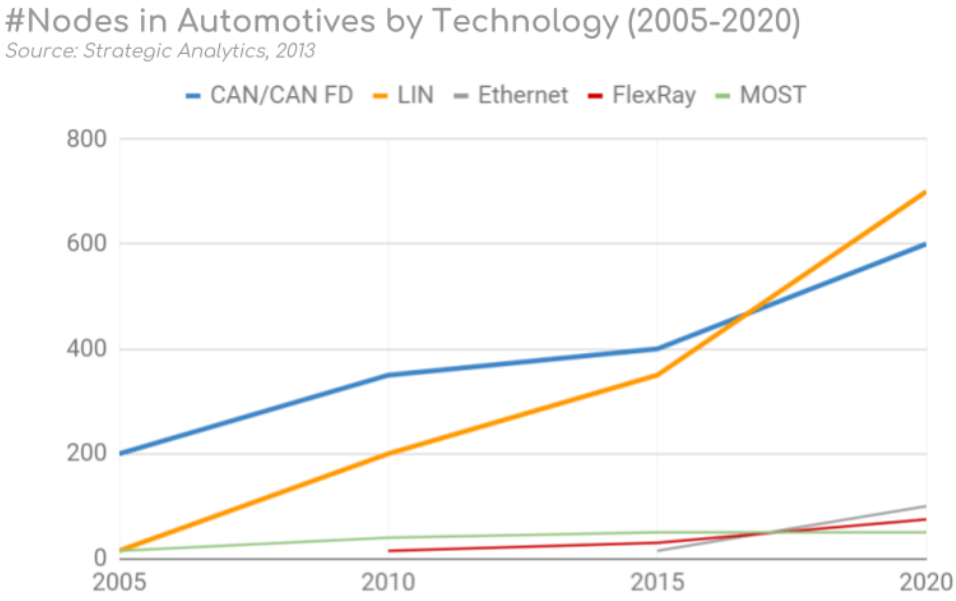
LIN bus history

Below we briefly recap the history of the LIN protocol:

- 1999: LIN 1.0 released by the LIN Consortium (BMW, VW, Audi, Volvo, Mercedes-Benz, Volcano & Motorola)
- 2000: The LIN protocol was updated (LIN 1.1, LIN 1.2)
- 2002: LIN 1.3 released, mainly changing the physical layer
- 2003: LIN 2.0 released, adding major changes (widely used)
- 2006: LIN 2.1 specification released
- 2010: LIN 2.2A released, now widely implemented versions
- 2010-12: SAE standardized LIN as SAE J2602, based on LIN 2.0
- 2016: CAN in Automation standardized LIN (ISO 17987:2016)

LIN bus future

The LIN protocol serves an increasingly important role in providing low cost feature expansion in modern vehicles. As such, LIN bus has exploded in popularity in the last decade with >700 million nodes expected in automotives by 2020 vs ~200 million in 2010.



Cybersecurity & new protocols

However, with the rise of LIN also comes increased scrutiny in regards to cyber security. LIN faces similar risk exposures as CAN - and since LIN plays a role in e.g. the seats and steering wheel, a resolution to these risks may be necessary. The future automotive vehicle networks are seeing a rise in CAN FD, FlexRay and automotive Ethernet. While there's uncertainty regarding the role each of these systems will play in future automotives, it's expected that LIN bus clusters will remain vital as the low cost solution for an ever increasing demand for features in modern vehicles.

Master/slave vs. commander/responder

As part of designing a more inclusive wording for LIN bus, the CiA/ISO/SAE agreed wording will transition to commander/responder. As such, this will be the de facto standard wording used in most guidelines and LIN bus specifications going forward.

LIN bus applications

Today, LIN bus is a de facto standard in practically all modern vehicles - with examples of automotive use cases below:

- Steering wheel: Cruise control, wiper, climate control, radio
- Comfort: Sensors for temperature, sun roof, light, humidity
- Powertrain: Sensors for position, speed, pressure
- Engine: Small motors, cooling fan motors
- Air condition: Motors, control panel (AC is often complex)
- Door: Side mirrors, windows, seat control, locks
- Seats: Position motors, pressure sensors
- Other: Window wipers, rain sensors, headlights, airflow

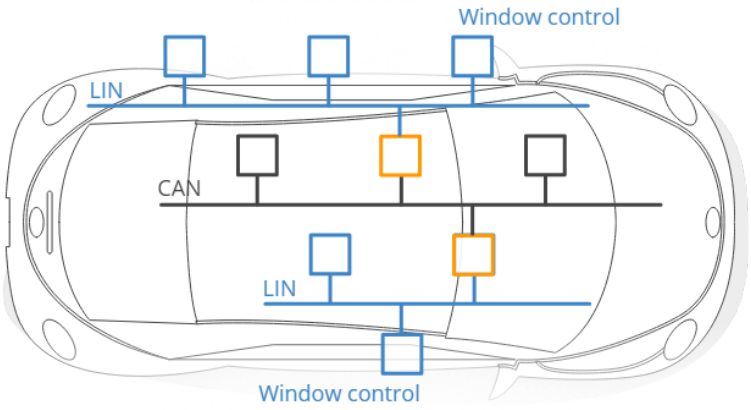
Further, LIN bus is also being used in other industries:

- Home appliances: Washing machines, refrigerators, stoves
- Automation: Manufacturing equipment, metal working

Example: LIN vs CAN window control

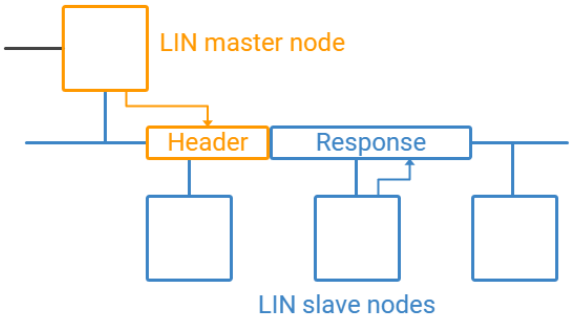
LIN nodes are typically bundled in clusters, each with a master that interfaces with the backbone CAN bus.

Example: In a car's right seat you can roll down the left seat window. To do so, you press a button to send a message via one LIN cluster to another LIN cluster via the CAN bus. This triggers the second LIN cluster to roll down the left seat window.



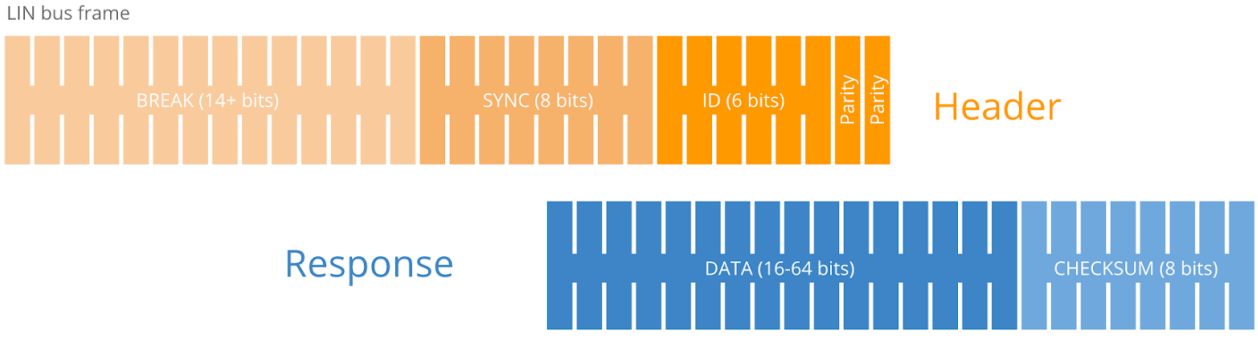
How does LIN bus work?

LIN communication at its core is relatively simple: A master node loops through each of the slave nodes, sending a request for information - and each slave responds with data when polled. The data bytes contain LIN bus signals (in raw form). However, with each specification update, new features have been added to the LIN specification - making it more complex. Below we cover the basics: The LIN frame & six frame types.



The LIN frame format

In simple terms, the LIN bus message frame consists of a header and a response. Typically, the LIN master transmits a header to the LIN bus. This triggers a slave, which sends up to 8 data bytes in response. This overall LIN frame format can be illustrated as below:



LIN frame fields

Break: The Sync Break Field (SBF) aka Break is minimum 13 + 1 bits long (and in practice most often 18 + 2 bits). The Break field acts as a “start of frame" notice to all LIN nodes on the bus.

Sync: The 8 bit Sync field has a predefined value of 0x55 (in binary, 01010101). This structure allows the LIN nodes to determine the time between rising/falling edges and thus the baud rate used by the master node. This lets each of them stay in sync.

Identifier: The Identifier is 6 bits, followed by 2 parity bits. The ID acts as an identifier for each LIN message sent and which nodes react to the header. Slaves determine the validity of the ID field (based on the parity bits) and act via below:

1. Ignore the subsequent data transmission
2. Listen to the data transmitted from another node
3. Publish data in response to the header

Typically, one slave is polled for information at a time - meaning zero collision risk (and hence no need for arbitration). Note that the 6 bits allow for 64 IDs, of which ID 60-61 are used for diagnostics (more below) and 62-63 are reserved.

Data: When a LIN slave is polled by the master, it can respond by transmitting 2, 4 or 8 bytes of data. The data length can be customized, but it is typically linked to the ID range (ID 0-31: 2 bytes, 32-47: 4 bytes, 48-63: 8 bytes). The data bytes contain the actual information being communicated in the form of LIN signals. The LIN signals are packed within the data bytes and may be e.g. just 1 bit long or multiple bytes.

Checksum: As in CAN, a checksum field ensures the validity of the LIN frame. The classic 8 bit checksum is based on summing the data bytes only (LIN 1.3), while the enhanced checksum algorithm also includes the identifier field (LIN 2.0).

Logging LIN bus data

The CANedge lets you easily log LIN bus data to an 8-32 GB SD card. Simply connect it to your LIN application to start logging - and process the data via free software/APIs.

For example, the free asammdf GUI/API lets you DBC decode your LIN data to physical values and e.g. plot your LIN signals. Learn more

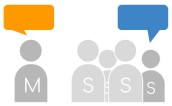
Six LIN frame types

Multiple types of LIN frames exist, though in practice the vast majority of communication is done via “unconditional frames".

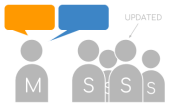
Note also that each of the below follow the same basic LIN frame structure - and only differ by timing or content of the data bytes. Below we briefly outline each LIN frame type:

ID (dec)	ID (hex)	LIN Frame Type
0-59	00-3B	Unconditional
0-59	00-3B	Event Triggered
0-59	00-3B	Sporadic
60-61	3C-3D	Diagnostic
62	3E	User Defined
63	3F	Reserved

Unconditional Frames
The default form of communication where the master sends a header, requesting information from a specific slave. The relevant slave reacts accordingly

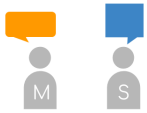


Event Trigger Frames
The master polls multiple slaves. A slave responds if its data has been updated, with its protected ID in the 1st data byte. If multiple respond, a collision occurs and the master defaults to unconditional frames

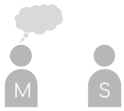


Sporadic Frames
Only sent by the master if it knows a specific slave has updated data. The master "acts as a slave" and provides the response to its own header - letting it provide slave nodes with "dynamic" info

Diagnostic Frames
Since LIN 2.0, IDs 60-61 are used for reading diagnostics from master or slaves. Frames always contain 8 data bytes. ID 60 is used for the master request, 61 for the slave response



User Defined Frames
ID 62 is a user-defined frame which may contain any type of information



Reserved Frames
Reserved frames have ID 63 and must not be used in LIN 2.0 conforming LIN networks

Advanced LIN topics

Below we include two advanced topics - click to expand.

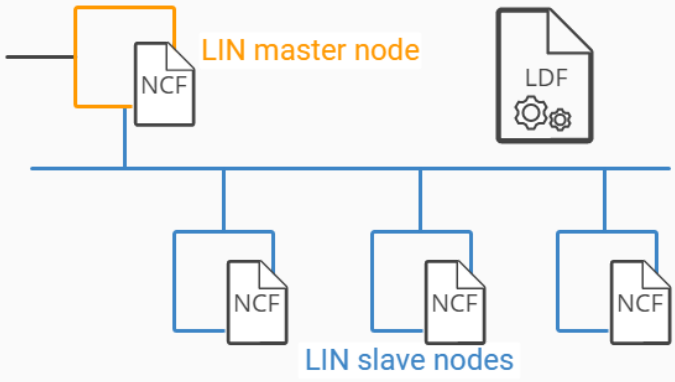
The LIN Node Configuration File (NCF) and LIN Description File (LDF)

To quickly set up LIN bus networks, off-the-shelf LIN nodes come with Node Configuration Files (NCF). The NCF details the LIN node capabilities and is a key part of the LIN topology.

An OEM will then combine these node NCFs into a cluster file, referred to as a LIN Description File (LDF). The master then sets up and manages the LIN cluster based on this LDF - e.g. the time schedule for headers.

Note that the LIN bus nodes can be re-configured by using the diagnostic frames described earlier. This type of configuration could be done during production - or e.g. everytime the network is started up. For example, this can be used to change node message IDs.

If you're familiar with CANopen, you may see parallels to the Device Configuration File used to pre-configure CANopen nodes - and the role of Service Data Objects in updating these configurations.

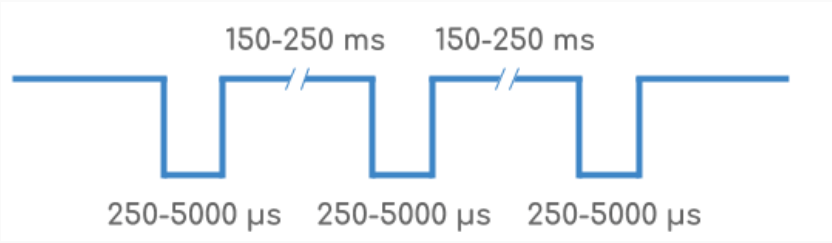


LIN Sleep & Wakeup

A key aspect of LIN is not only saving costs, but also power.

To achieve this, each LIN slave can be forced into sleep mode by the master sending a diagnostic request (ID 60) with the first byte equal to 0. Each slave also automatically sleeps after 4 seconds of bus inactivity.

The slaves can be woken up by either the master or slave nodes sending a wake up request. This is done by forcing the bus to be dominant for 250-5000 microseconds, followed by a pause for 150-250 ms. This is repeated up to 3 times if no header is sent by the master. After this, a pause of 1.5 seconds is required before sending a 4th wake up request. Typically nodes wake up after 1-2 pulses.

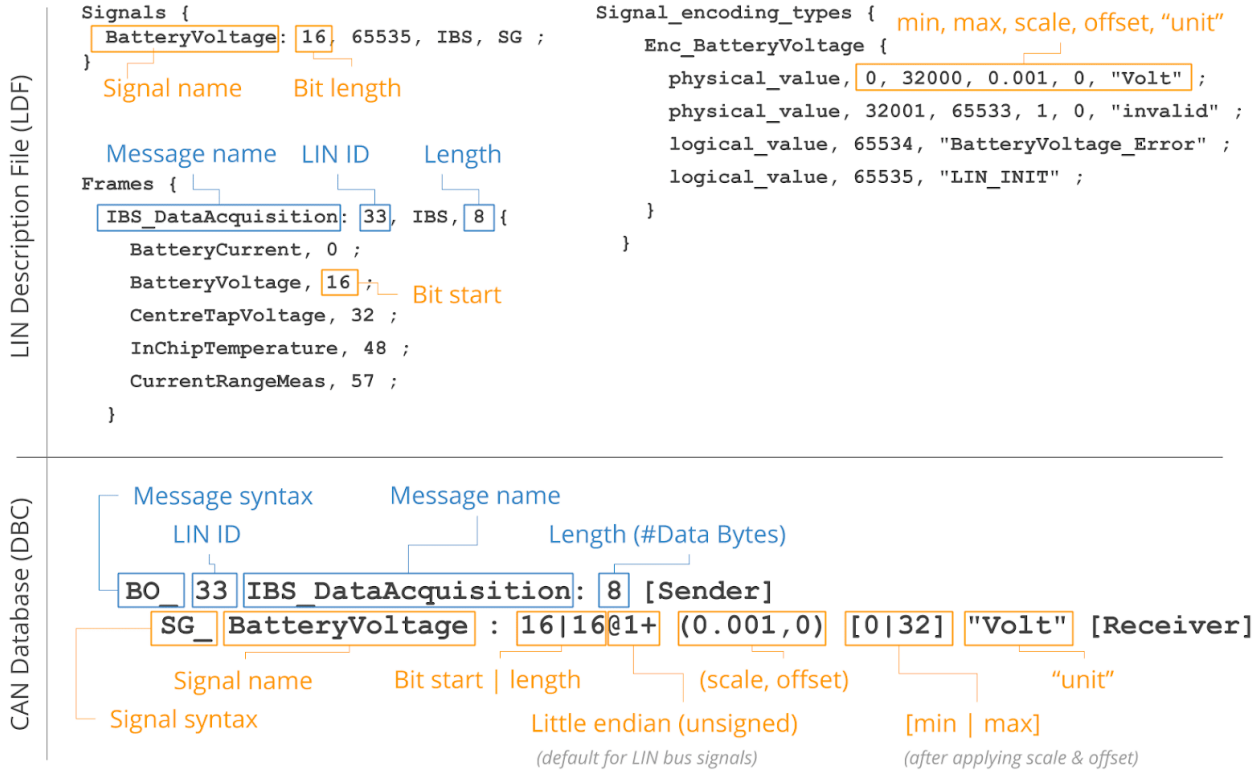


LIN Description File (LDF) vs. DBC files

As part of your LIN data logger workflow, you may need to decode your raw LIN bus data to physical values. Specifically, this involves extracting LIN signals from the LIN frame payload and decoding these to human-readable form.

This process of LIN bus decoding is similar to CAN bus decoding and requires the same information:

- ID: Which LIN frame ID contains the LIN bus signal
- Name: The LIN signal name should be known
- Start bit: Start position of the LIN signal in the payload
- Length: Length of the LIN bus signal
- Endianness: LIN signals are little endian (Intel byte order)
- Scale: How to multiply the decimal value of the LIN signal bits
- Offset: By what constant should the LIN signal value be offset
- Unit/Min/Max: Additional supporting information (optional)



This information is typically available as part of the LIN Description File (LDF) for a local interconnect network. However, since many software tools do not natively support the LDF format, we explain below how to use DBC files as an alternative.

LIN Description File (LDF) vs. DBC files

As evident from our CAN bus intro and DBC file intro, the above entries are equivalent to the information stored in a CAN DBC file. This means that a simple method for storing LIN bus decoding rules is to use the DBC file format, which is supported by many software and API tools (incl. the CANedge software tools like asammdf). For example, you can load a LIN DBC file and your raw LIN bus data from the CANedge in asammdf to extract LIN bus signals from the data, which you can then plot, analyze or export.

In many cases, you may not have a LIN DBC file directly available, but instead you may have a LIN description file (LDF). Below we therefore focus on how you can convert the relevant LIN signal information into the DBC format.

Note: The LDF typically contains various other information relevant to the operation of the LIN bus, which we do not focus on here. For a full deep-dive on the LIN protocol and the a detailed description of the LDF specification, see the LIN protocol PDF standard.

How to convert a LIN Description File (LDF) to DBC [incl. Examples]

Below we provide an example to showcase how you can extract LIN signal information from an LDF and enter it into a DBC file. We use a very simplified LIN description file (with only one signal and excluding some sections).
You can expand the below examples to see the LIN signal, BatteryVoltage, in the LDF format and in the DBC format.
You can also download a raw LIN bus log file (MF4) from the CANedge2 with data for this signal, which you can open and DBC decode in asammdf:

Example: BatteryVoltage signal (LDF)

download .ldf | download .dbc

Guide to LDF to DBC conversion

In short, to convert an LDF file to DBC, you'll go through the following steps for each LIN signal:

- Get the LIN signal name and length from the Signals section
- Get the LIN signal message name, ID and length from the Frames section
- Get the LIN signal bit start from the Frames section
- Go to the LDF Signal_encoding_types section and find "Enc_[signal_name]"
- Get remaining info via the syntax: 'physical_value, [min], [max], [scale], [offset], "[unit]" ;'

If you're looking to create your own LIN DBC file, we suggest you review our DBC file introduction for details on the syntax, as well as DBC editor tools.

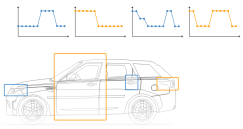
Minor pitfalls

The conversion from LDF to DBC is not entirely 1-to-1. In particular, note how the LIN signal BatteryVoltage has 2 entries for the physical value, one for the decimal range 0 to 32000 and one for 32001 to 65533. In this specific case, only the data in the first range are valid (the unit is "invalid" for the 2nd range). However, in some cases there can be multiple ranges that require separate scaling factors - something which is not possible to handle in the DBC file format. In this case, you will need to choose one of the ranges and e.g. treat results outside this range as invalid.

This is also the simplest way to handle the LIN signal 'logical_value' entries in the Signal_encoding_types section. These typically reflect how specific values of the LIN signal should be treated (e.g. as errors). One way of treating these entries would be to ignore them and possibly exclude them as part of your data post processing - similar to how FF byte values in CAN bus are often excluded as they represent invalid or N/A data.

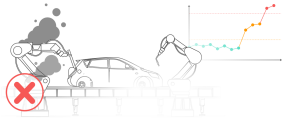
LIN bus data logging - use case examples

LIN bus data logging is relevant across various use cases:

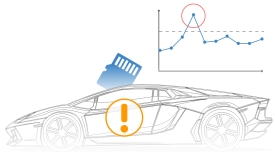


Vehicle CAN/LIN development
Logging CAN/LIN data via a hybrid logger is key to OEM vehicle development and can be used in optimization or diagnostics
learn more

Field prototype telematics
CAN/LIN data from automotive prototype equipment can be collected at scale using IoT CAN/LIN hybrid loggers to speed up R&D
learn more



Predictive maintenance
Industrial machinery can be monitored via IoT CAN/LIN loggers in the cloud to predict and avoid breakdowns via prediction models
learn more



Rare LIN issue diagnostics
A LIN bus logger can serve as a 'blackbox' for industrial machinery, providing data for e.g. disputes or rare issue diagnostics
learn more

Practical considerations for LIN data logging

Below we list key considerations for your LIN bus data logging:

LIN logger vs. LIN interface

To record LIN bus data, you'll need a LIN bus data logger and/or interface. A LIN bus data logger with SD card has the advantage of letting you record data in standalone mode - i.e. during actual usage of the vehicle. An interface, on the other hand, is helpful during e.g. dyno testing of the vehicle functionality.

For standalone LIN loggers, it's key that the device is plug & play, compact and low cost - so as to allow it to be used in scale applications across e.g. vehicle fleets.

CAN vs. LIN support

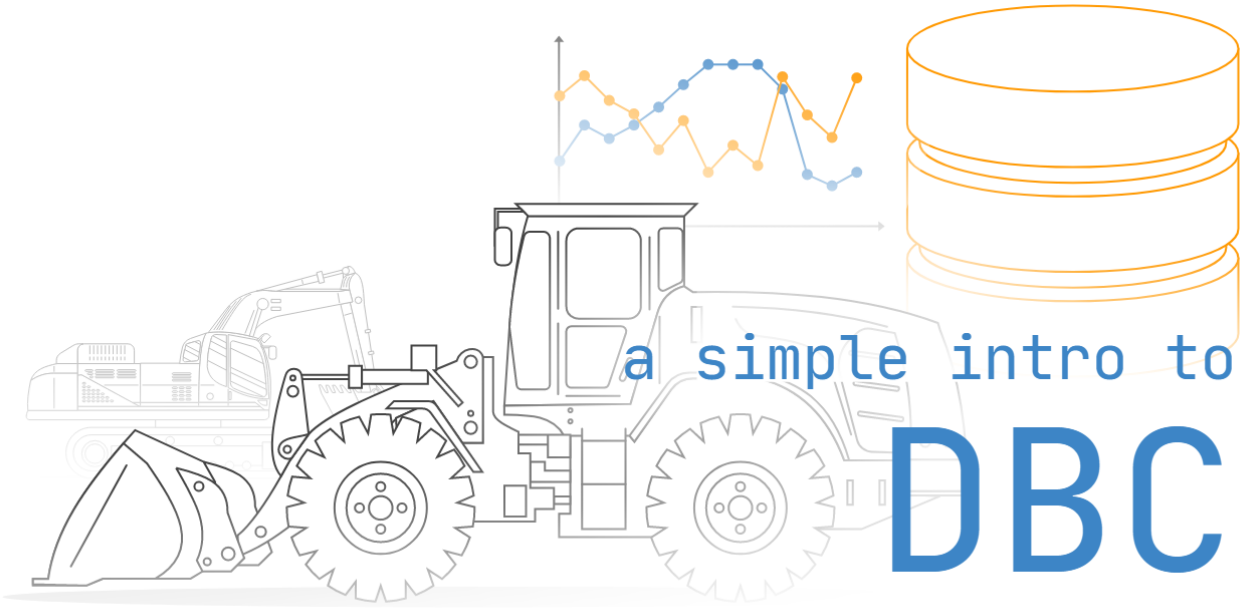
Often, you'll want to combine LIN data with CAN data to get a holistic perspective of the vehicle in use - for example:

- How is driving behavior correlated with use of various LIN bus features?
- Do issues arise in the interaction between LIN masters and the CAN bus?
- Are LIN related issues correlated with certain CAN based events?

To combine this data, you'll want a hybrid CAN/LIN logger with multiple channels. Further, CAN FD support is also key as it's expected to increasingly be rolled out in new vehicles.

WiFi

Collecting logged LIN bus data can be a hassle if you need to physically extract the data from e.g. large vehicle test fleets. Here, a WiFi enabled CAN/LIN logger can be a powerful solution. You simply specify a WiFi hotspot that the vehicle will get in range of from time-to-time - and the data will then be uploaded automatically from the SD card when in range. It's also possible to add a cellular hotspot within the vehicle for near real-time data transfer.



CAN DBC File Explained - A Simple Intro

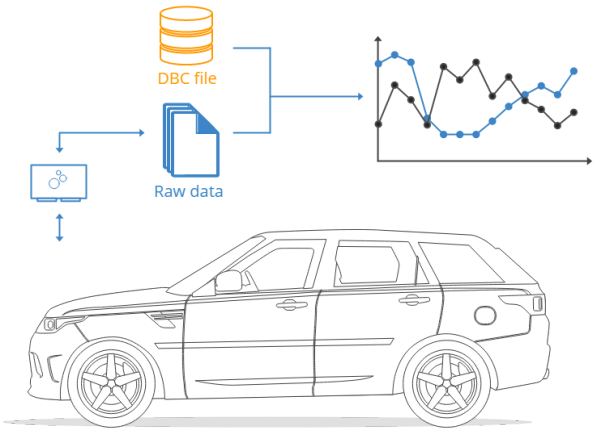
In this tutorial we explain DBC files (CAN bus databases), incl. structure, syntax, examples - and a cool DBC editor playground. To get practical, we also include real J1939/OBD2 data and DBC files - which you can load in free open source CAN tools.

What is a CAN DBC file?

A CAN DBC file (CAN database) is a text file that contains information for decoding raw CAN bus data to 'physical values'.

To understand what 'raw CAN data' looks like, see the below example CAN frame from a truck:

```
CAN ID    Data bytes
0CF00400  FF FF FF 68 13 FF FF FF
```



If you have a CAN DBC that contains decoding rules for the CAN ID, you can 'extract' parameters (signals) from the data bytes. One such signal could be EngineSpeed:

```
Message  Signal       Value  Unit
EEC1     EngineSpeed  621    rpm
```

To understand how DBC decoding works, we will explain the DBC syntax and provide step-by-step decoding examples.

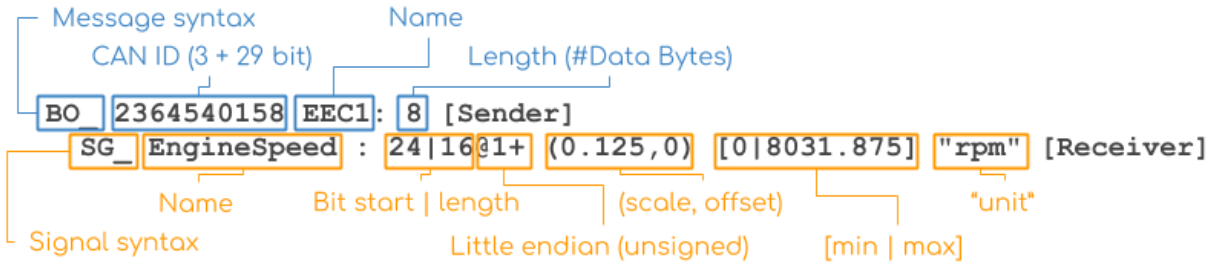
DBC message & signal syntax

Let us start with a real CAN DBC file example. Below is a demo J1939 DBC file that contains decoding rules for speed (km/h) and engine speed (rpm). You can copy this into a text file, rename it as e.g. j1939.dbc and use it to extract speed/RPM from trucks, tractors or other heavy-duty vehicles.

J1939 DBC demo example

```
VERSION ""


NS_ :
    CM_
    BA_DEF_
    BA_
    BA_DEF_DEF_

BS_:

BU_:


BO_ 2364540158 EEC1: 8 Vector_XXX
 SG_ EngineSpeed : 24|16@1+ (0.125,0) [0|8031.875] "rpm" Vector_XXX

BO_ 2566844926 CCVS1: 8 Vector_XXX
 SG_ WheelBasedVehicleSpeed : 8|16@1+ (0.00390625,0) [0|250.996] "km/h" Vector_XXX


CM_ BO_ 2364540158 "Electronic Engine Controller 1";
CM_ SG_ 2364540158 EngineSpeed "Actual engine speed which is calculated over a minimum crankshaft angle
of 720 degrees divided by the number of cylinders....";
CM_ BO_ 2566844926 "Cruise Control/Vehicle Speed 1";
CM_ SG_ 2566844926 WheelBasedVehicleSpeed "Wheel-Based Vehicle Speed: Speed of the vehicle as calculated
from wheel or tailshaft speed.";
BA_DEF_ SG_  "SPN" INT 0 524287;
BA_DEF_ BO_  "VFrameFormat" ENUM  "StandardCAN","ExtendedCAN","reserved","J1939PG";
BA_DEF_  "BusType" STRING ;
BA_DEF_  "ProtocolType" STRING ;
BA_DEF_DEF_  "SPN" 0;
BA_DEF_DEF_  "VFrameFormat" "J1939PG";
BA_DEF_DEF_  "BusType" "";
BA_DEF_DEF_  "ProtocolType" "";
BA_ "ProtocolType" "J1939";
BA_ "BusType" "CAN";
BA_ "VFrameFormat" BO_ 2364540158 3;
BA_ "VFrameFormat" BO_ 2566844926 3;
BA_ "SPN" SG_ 2364540158 EngineSpeed 190;
BA_ "SPN" SG_ 2566844926 WheelBasedVehicleSpeed 84;
```

At the heart of a DBC file are the rules that describe how to decode CAN messages and signals:



DBC message syntax explained

- A message starts with BO_ and the ID must be unique and in decimal (not hexadecimal)
- The DBC ID adds adds 3 extra bits for 29 bit CAN IDs to serve as an 'extended ID' flag
- The name must be unique, 1-32 characters and may contain [A-z], digits and underscores
- The length (DLC) must be an integer between 0 and 1785
- The sender is the name of the transmitting node, or Vector__XXX if no name is available

DBC signal syntax explained

- Each message contains 1+ signals that start with SG_
- The name must be unique, 1-32 characters and may contain [A-z], digits and underscores
- The bit start counts from 0 and marks the start of the signal in the data payload
- The bit length is the signal length
- The @1 specifies that the byte order is little-endian/Intel (vs @0 for big-endian/Motorola)
- The + informs that the value type is unsigned (vs - for signed signals)
- The (scale,offset) values are used in the physical value linear equation (more below)
- The [min|max] and unit are optional meta information (they can e.g. be set to [0|0] and "")
- The receiver is the name of the receiving node (again, Vector__XXX is used as default)

Example: Extract physical value of EngineSpeed signal

To understand how DBC decoding works, we'll show how to extract the signal EngineSpeed from the CAN frame in the intro:

```
CAN ID    Data bytes
0CF00400  FF FF FF 68 13 FF FF FF
```

1# Match the CAN ID vs DBC ID

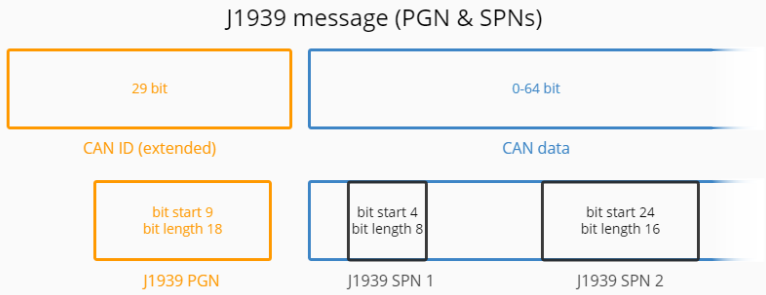
In practice, most raw CAN data log files contain 20-80 unique CAN IDs. As such, the first step is to map each CAN ID to the relevant conversion rules in the DBC. For regular 11-bit CAN IDs, this can simply be done by mapping the decimal value of the CAN ID to the DBC CAN IDs. For extended 29-bit CAN IDs, a mask (0x1FFFFFFF) needs to be applied to the 32-bit DBC ID to get the 29-bit CAN ID - which can then be mapped against the log file.

To easily convert between the IDs, see the 'DBC ID vs CAN ID' sheet in our DBC editor playground.

Regarding J1939 PGN conversion

In this example we directly map the 29-bit CAN ID to the 'masked' DBC ID. In practice, J1939 conversion is often done by extracting the 18-bit J1939 PGN from the CAN ID and the DBC ID and then comparing the PGNs.

The method depends on your software. In the CANedge tools like asammdf and our Python API tools, loading a J1939 DBC will automatically make the tools use J1939 PGN matching.



2# Extract the signal bits

Next, use the DBC bit start, length and endianness to extract the relevant bits from the CAN frame data payload. In this example, the start bit is 24 (meaning byte 3 when counting from 0) and the bit length is 16 (2 bytes):

```
FF FF FF 68 13 FF FF FF
```

Big endian vs. little-endian (advanced)

In the example, we use little-endian, meaning that the 'bit start' in the DBC file reflects the position of least-significant bit (LSB). If you add a DBC signal in a DBC file viewer (e.g. Vector CANDB++), the LSB is also used as the bit start in the DBC editor.

If you instead add a big-endian signal in a DBC editor GUI, you'll still see the LSB as the bit start - but when you save the DBC, the bit start is set to the most significant bit (MSB) in the signal. This approach makes GUI editing more intuitive, but can be a confusing if you switch between a GUI and text editor. To understand this fully, check out our CAN DBC editor playground.

Bit indices

7	6	5	4	3	2	1	0
7	6	5	4	3	2	1	0
15	14	13	12	11	10	9	8
23	22	21	20	19	18	17	16
31	30	29	28	27	26	25	24
39	38	37	36	35	34	33	32
47	46	45	44	43	42	41	40
55	54	53	52	51	50	49	48
63	62	61	60	59	58	57	56

3# Scale the extracted bits

The EngineSpeed signal is little-endian (@1) and we therefore need to reorder the byte sequence from 6813 to 1368. Next, we convert the HEX string to decimal and apply the linear conversion:

```
physical_value = offset + scale * raw_value_decimal
       621 rpm = 0 + 0.125 * 4968
```

In short, the EngineSpeed physical value (aka scaled engineering value) is 621 rpm.

4# DBC decode your full dataset

Steps 1-3 take outset in a single CAN frame. In practice, you'll decode raw data from e.g. vehicles, machines or boats with millions of CAN frames, timestamps, several unique CAN IDs and sometimes hundreds of signals.

To handle this complexity, specialized software is used to decode raw CAN data into human-readable form by loading the data log files and related DBC file(s). To illustrate what the output may look like, we've added a snippet of DBC decoded J1939 data logged with a CANedge. The data has been converted via asammdf, filtered to include a set of relevant signals and exported as CSV:

```
timestamps,ActualEnginePercentTorque,EngineSpeed,EngineCoolantTemperature,EngineOilTemperature1,EngineFu
elRate,EngineTotalIdleHours,FuelLevel1,Latitude,Longitude,WheelBasedVehicleSpeed
2020-01-13 16:00:13.259449959+01:00,0,1520.13,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.268850088+01:00,0,1522.88,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.270649910+01:00,0,1523.34,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.271549940+01:00,0,1523.58,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.278949976+01:00,0,1525.5,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.289050102+01:00,0,1527.88,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.299000025+01:00,0,1528.13,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.308300018+01:00,0,1526.86,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.309099913+01:00,0,1526.75,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.317199945+01:00,0,1526.45,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.319149971+01:00,0,1526.38,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.320349932+01:00,0,1526.15,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.23
2020-01-13 16:00:13.326800108+01:00,0,1524.93,92,106,3.8,1868.3,52,40.6440124,-76.1223603,86.25
…
```

CAN DBC editor playground

Adding or editing signals in your DBC file can be confusing. To help you better understand how this works, we've created a very basic online 'CAN DBC editor playground'. You can open it and edit the yellow cells - or make a copy for personal use. Feel free to bookmark the editor and share it!

J1939/OBD2 data & DBC samples

The best way to learn more about DBC conversion of raw CAN bus data is to try it out. Below you can download raw J1939/OBD2 data from the CANedge2. The zip also includes an OBD2 DBC and a demo J1939 DBC.

Via the link, you can also download the free asammdf GUI to open the data & DBC files. Try plotting EngineSpeed as an example. Download the sample data & dbc files.

Advanced: Meta info, attributes & multiplexing

In this section we briefly outline some of the more advanced topics of CAN DBC files, incl. meta info, attributes, value tables and multiplexing. If you're new to DBC files you can optionally skip this section.

Comments (message/signal names, ...)

The DBC file can contain various extra information and below we outline the most common types. This information is stored in the DBC file after all the messages. The comment attribute lets you map a 1-255 character comment to a message ID or signal name, e.g. to provide more information. Example for EEC1 message:

```
CM_ BO_ 2364540158 "Electronic Engine Controller 1"
```

Example for EngineSpeed signal:

```
CM_ SG_ 2364540158 EngineSpeed "Actual engine speed which is calculated over
a minimum crankshaft angle of 720 degrees divided by the number of cylinders....";
```

Attributes: Frame format, SPN IDs, ...

Custom attributes can be added to messages and signals in a similar way as the comment syntax. A typical attribute is the VFrameFormat, which can be used to describe the message frame type. It is initialized as below:
BA_DEF_ BO_ "VFrameFormat" ENUM "StandardCAN","ExtendedCAN","reserved","J1939PG";

Once initialized, a message can be mapped as follows (indexing from 0):

```
BA_ "VFrameFormat" BO_ 2364540158 3;
```

In this case, we inform that the message EEC1 is of the J1939 PGN type, which may result in specific display or handling in various DBC editor GUI tools, as well as data processing tools.

Similarly, you can add J1939 SPN IDs as an attribute as below:

```
BA_DEF_ SG_  "SPN" INT 0 524287;
BA_ "SPN" SG_ 2364540158 EngineSpeed 190;
```

Here, EngineSpeed is assigned the SPN 190. You might find it more natural to do this in the opposite way - i.e. use the SPN IDs in the message/signal section, then map the SPN names via attributes. While you can definitely do this, it is not the most common convention. Further, the first character in a CAN DBC signal name cannot be a number - so you'd need to write e.g. _190 or SPN_190.

Value tables

Some CAN DBC signals are 'state variables', such as gear-shift, diagnostic trouble codes, status codes etc. These take discrete values and will require a mapping table to enable interpretation.

The CAN DBC format enables this via value tables, which let you assign a state description to the physical decimal value of each state of a signal. The states should be in descending order.

Example:

```
VAL_ 2297441534 MaterialDropActiveStatus 3 "NotAvailable" 2 "Error" 1 "On" 0 "Off" ;
```

Multiplexed signals (OBD2 DBC example)

Multiplexing is sometimes used in CAN bus communication, with perhaps the most known example being within OBD2 communication. Consider for example the below OBD2 response frames:

```
7E8   03 41 11 30 FF FF FF FF
7E8   03 41 0D 37 FF FF FF FF
```

Here, both response frames carry 1 byte of data in byte 3, yet despite the CAN ID being identical, the interpretation of the data differs between the two frames. This is because byte 2 serves as a multiplexer, specifying which OBD2 PID the data is coming from. To handle this in a DBC file context, the below syntax can be applied:

```
BO_ 2024 OBD2: 8 Vector__XXX
 SG_ S1_PID_0D_VehicleSpeed m13 : 31|8@0+ (1,0) [0|255] "km/h" Vector__XXX
 SG_ S1_PID_11_ThrottlePosition m17 : 31|8@0+ (0.39216,0) [0|100] "%" Vector__XXX
 SG_ S1 m1M : 23|8@0+ (1,0) [0|255] "" Vector__XXX
 SG_ Service M : 11|4@0+ (1,0) [0|15] "" Vector__XXX
...
SG_MUL_VAL_ 2024 S1_PID_0D_VehicleSpeed S1 13-13;
SG_MUL_VAL_ 2024 S1_PID_11_ThrottlePosition S1 17-17;
SG_MUL_VAL_ 2024 S1 Service 1-1;
```

Here, the M in the Service signal serves as the 'multiplexor signal'. In this case, it toggles which OBD2 service mode is used (mode 01, 02, ...). The signal S1 is multiplexed by Service, which is evident from the SG_MUL_VAL_ field where the two are grouped. As evident, the signal S1 has the value m1 after the signal name, which means that if the Service signal takes the value 1, the data reflects the OBD2 service mode 01.

The above is referred to as simple multiplexing. But CAN DBC files also support extended multiplexing, where a multiplexed signal (in this case S1) can also be a multiplexor and thus control the interpretation of other parts of the data payload. To see this, note that S1 takes the same role as Service in the syntax, adding an M after m1 and being grouped with the two OBD2 PIDs, speed and throttle position.

Extended multiplexing works as before: If S1 takes the value 13 (HEX 0D), it means that only signals that are A) part of the S1 group and B) have a multiplexer switch value of 13 should be taken into account. In this example, it means that byte 4 reflects data for vehicle speed. If byte 3 equals 17 (HEX 11), byte 4 reflects data for the throttle position instead.

DBC multiplexing and extended multiplexing is an advanced topic and not supported by all data processing tools. However, you can use e.g. the CANDB++ DBC editor to more easily view and understand DBC files with multiplexing, like the OBD2 DBC: obd2 dbc file

The OBD2 DBC file can be used together with our CANedge CAN loggers for the purpose of decoding OBD2 frames in e.g. asammdf or our Python API modules. For more on this topic, see Vector's guide to extended multiplexing in DBC files here.

DBC software tools (editing & processing)

Software that uses CAN DBC files can be split in two groups: Editing & data processing.

DBC editor tools

- **Vector CANDB++:** The free demo version of Vector's CANalyzer includes a useful version of CANDB++, the Vector DBC editor. It offers the most extensive functionality available, including quick "consistency checks" of DBC files
- **Kvaser Database Editor:** Kvaser provides a free and easy-to-use CAN DBC editor, which includes the ability to append DBC files and visualize the signal construction
- **canmatrix:** This open source Python DBC module lets you load DBC files, edit them and export them to other formats. It is used in e.g. asammdf and our Python API

CAN data processing tools

Most CAN data processing tools support DBC files - including below:

- **asammdf GUI**: The asammdf GUI lets you load raw MDF4 data and DBC convert it to human-readable form - as well as create quick plots, analyses and exports
- **Python API**: Our Python API lets you automate the DBC conversion of your data e.g. for large-scale data analysis or as part of telematics dashboard integrations
- **Other tools**: Our MDF4 converters let you quickly convert your CANedge MF4 data to e.g. Vector ASC, PEAK TRC and more

How to create a DBC file?

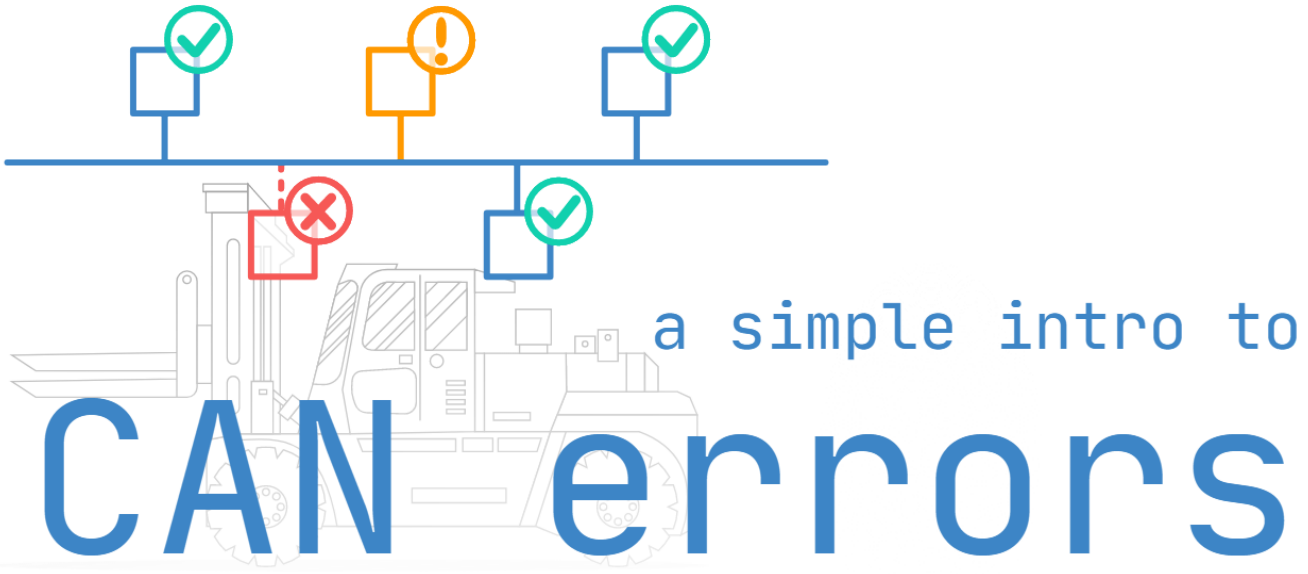
Getting started: If you need to construct a new DBC file, we recommend using one of the DBC editors above. For beginners, we recommend the Kvaser DBC editor. When creating a new DBC file, you can typically select a DBC template (e.g. a CAN FD DBC, J1939 DBC, NMEA 2000 DBC etc). Next, start by adding a single message and a single signal to the DBC and save it.

To ensure your DBC looks OK, we recommend to open the DBC in a text editor. This way you can verify that the basic DBC syntax looks as you'd expect - and you can use this version as a benchmark for comparison. It's a good idea to maintain git revisioning on any changes to the DBC from here.

Test your DBC: As a second step, we recommend to test the DBC file using a CAN bus decoder software tool. For example, if you're using a CANedge CAN bus data logger to record raw CAN data from your application, you can use the free CAN decoder software, asammdf, to load your raw data and your DBC file. This way you can quickly extract the signal you added in the DBC - and verify via a visual plot that the construction looks OK.

Expand your DBC: Next, you can add any remaining CAN messages and signals, as well as comments/descriptions, value tables etc. We recommend to do regular checks as before to ensure the construction is OK.

Check consistency: Finally, we recommend doing a full consistency check both in Vector's DBC editor, CANDB++, and Kvaser's DBC editor. In CANDB++ select 'File/Consistency Check' and keep an eye out for critical errors (though note that your DBC may be sufficiently valid for most tools, even if some issues are reported). In the Kvaser database editor, you can select each message to quickly spot signals with invalid fields (they'll be highlighted in yellow). Once you are done, we always recommend doing a visual analysis of your scaled CAN data to ensure that you do not have e.g. endianness, scale factor or offset errors.



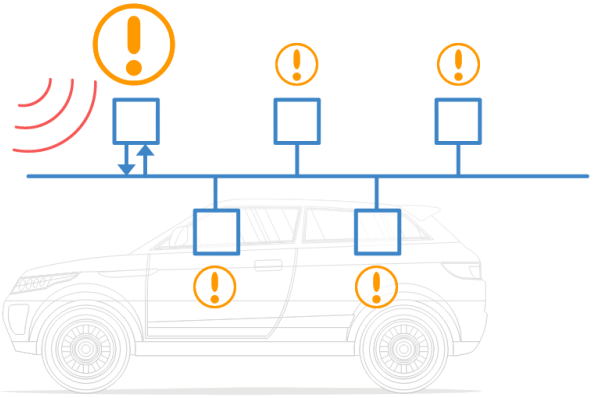
CAN Bus Errors Explained - A Simple Intro

Need a practical intro to CAN bus errors?

In this tutorial you'll learn about the basics of CAN error handling, the 5 CAN bus error types, the CAN error frame and CAN node error states.

To get practical, we'll also generate & record CAN errors in 6 experiments.

What are CAN bus errors?



As explained in our simple intro to CAN bus, the Controller Area Network is today the de facto standard across automotives and industrial automation systems.

A core benefit is the robustness of CAN, making it ideal for safety critical applications. Here, it is worth noting:

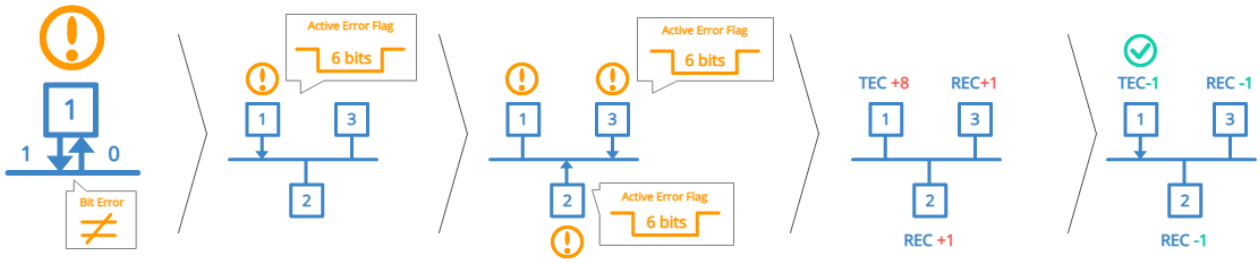
Error handling is vital to the robustness of CAN.

CAN bus errors can occur for several reasons - faulty cables, noise, incorrect termination, malfunctioning CAN nodes etc. Identifying, classifying and resolving such CAN errors is key to ensuring the continued performance of the overall CAN system.

In particular, error handling identifies and rejects erroneous messages, enabling a sender to re-transmit the message. Further, the process helps identify and disconnect CAN nodes that consistently transmit erroneous messages.

How does CAN error handling work?

Error handling is a built-in part of the CAN standard and every CAN controller. In other words, every CAN node handles fault identification and confinement identically. Below we've made a simple illustrative example:



Example step-by-step

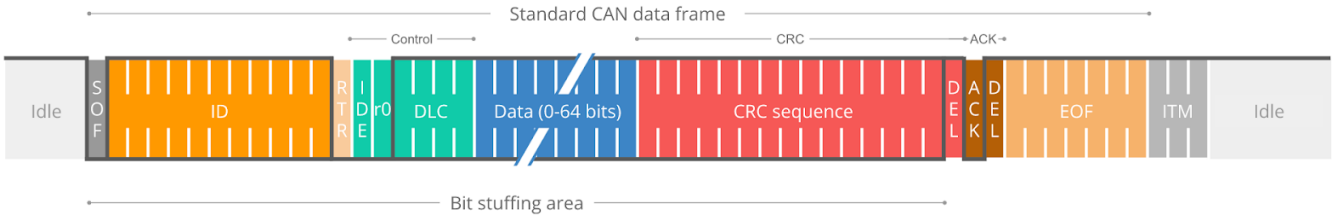
1. CAN node 1 transmits a message onto the CAN bus - and reads every bit it sends
2. In doing so, it discovers that one bit that was sent dominant was read recessive
3. This is a 'Bit Error' and node 1 raises an Active Error Flag to inform other nodes
4. In practice, this means that node 1 sends a sequence of 6 dominant bits onto the bus
5. In turn, the 6 dominant bits are seen as a 'Bit Stuffing Error' by other nodes
6. In response, nodes 2 and 3 simultaneously raise an Active Error Flag
7. This sequence of raised error flags comprise part of a 'CAN error frame'
8. CAN node 1, the transmitter, increases its 'Transmit Error Counter' (TEC) by 8
9. CAN nodes 2 and 3 increase their 'Receive Error Counter' (REC) by 1
10. CAN node 1 automatically re-transmits the message - and now succeeds
11. As a result, node 1 reduces its TEC by 1 and nodes 2 and 3 reduce their REC by 1

The example references a number of concepts that we will detail shortly: Error frames, error types, counters and states.

The CAN bus error frame

In the illustrative example, the CAN nodes 'raise Active Error Flags', thus creating an 'error frame' in response to detecting a CAN error.

To understand how this works, let us first look at a "normal" CAN frame (without errors):

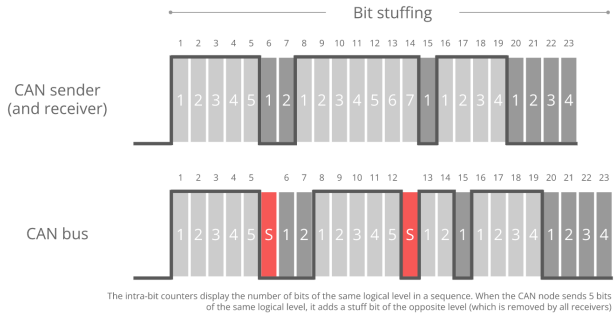


CAN bus bit stuffing

Notice that we highlighted 'bit stuffing' across the CAN frame.

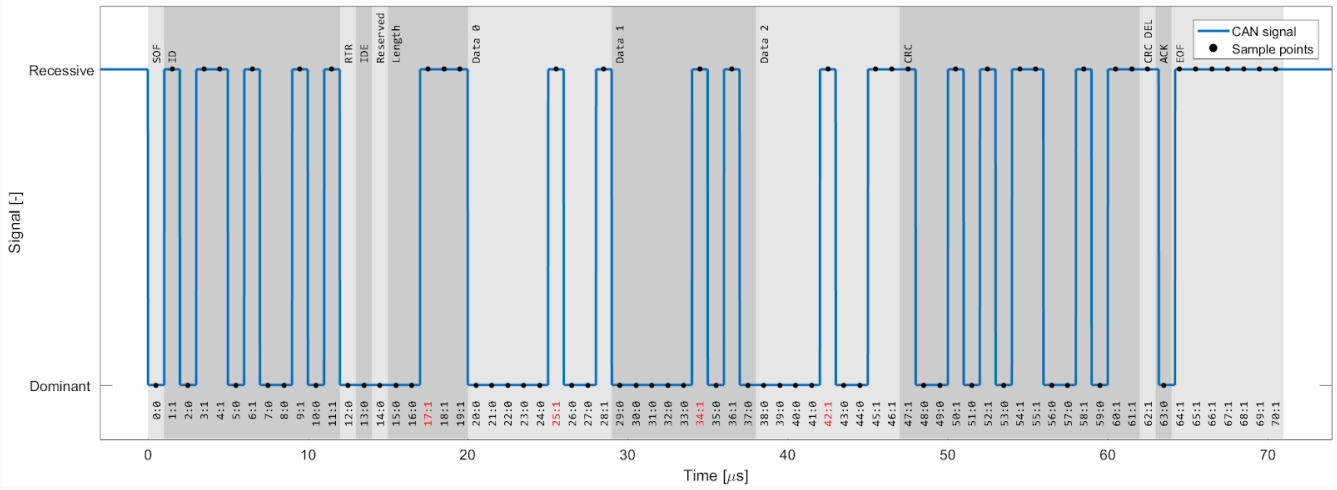
Bit stuffing is a subtle, but vital part of the CAN standard. Basically it states that whenever a CAN node sends five bits of the same logic level (dominant or recessive), it must send one bit of the opposite level. This extra bit is automatically removed by the receiving CAN nodes. This process helps ensure continuous synchronisation of the network.

As per the previous example, when CAN node 1 detects an error during the transmission of a CAN message, it immediately transmits a sequence of 6 bits of the same logic level - also referred to as raising an Active Error Flag.



Oscilloscope example

If we measure the transmission of a CAN frame via an oscilloscope and digitize the result, we can also see the stuff bits in practice (see the red timestamp marks):



Active Error Flags

As we just learned, such a sequence is a violation of the bit stuffing rule - aka a 'Bit Stuffing Error'. Further, this error is visible to all CAN nodes on the network (in contrast to the 'Bit Error' that resulted in this error flag being raised). Thus, the raising of error flags can be seen as a way of "globalizing" the discovery of an error, ensuring that every CAN node is informed. Note that the other CAN nodes will see the Active Error Flag as a Bit Stuffing Error. In response they also raise an Active Error Flag.

As we'll explain shortly, it is important to distinguish between the error flags. In particular, the first error flag (from the 'discovering' node) is often referred to as a 'primary' Active Error Flag, while the error flags of subsequent 'reacting' nodes are referred to as the 'secondary' Active Error Flag(s).

3 CAN error frame examples

Let's look at three example scenarios:

Example 1: 6 bits of error flags

Here, all CAN nodes simultaneously discover that an error exists in a CAN message and raise their error flags at the same time.

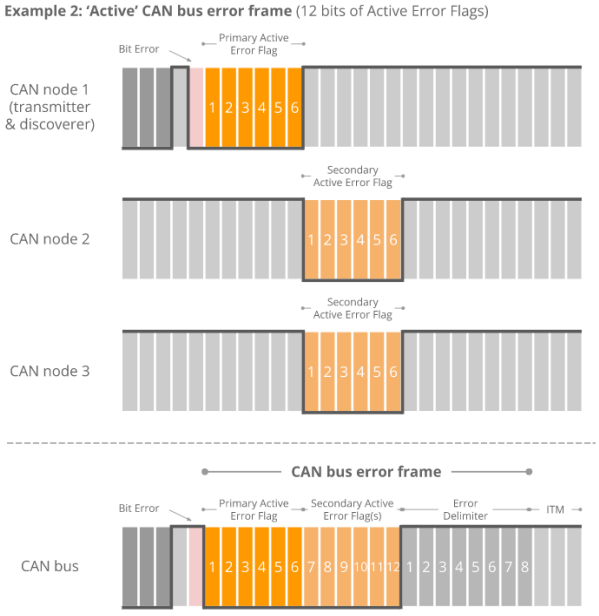
The result is that the error flags all overlap and the total sequence of dominant bits lasts for 6 bits in total. All CAN nodes will in this case consider themselves the 'discovering' CAN nodes.

This type of simultaneous discovery is less common in practice. However, it could e.g. happen as a result of Form Errors (such as a CRC delimiter being dominant instead of recessive), or if a CAN transmitter experiences a bit error during the writing of a CRC field.

Example 2: 12 bits of error flags

Here, CAN node 1 transmits a dominant bit, but reads it as recessive - meaning that it discovers a Bit Error. It immediately transmits a sequence of 6 dominant bits.

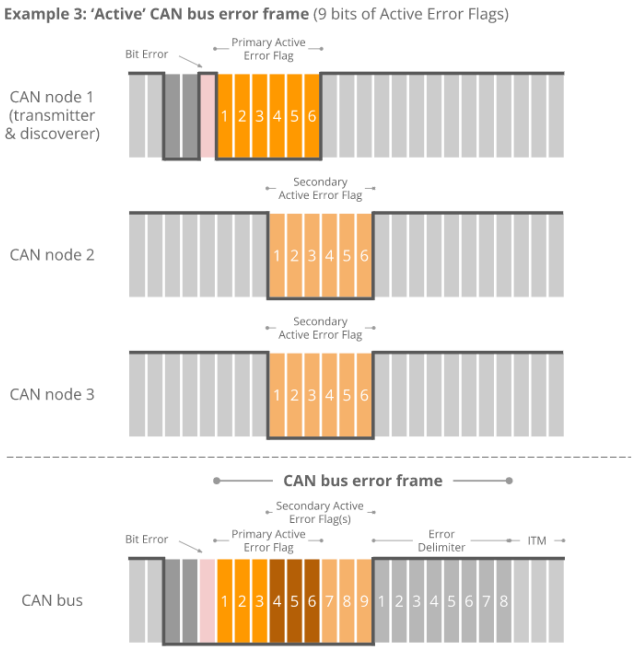
The other nodes only discover the Bit Stuffing Error after the full 6 bits have been read, after which they simultaneously raise their error flags, resulting in a subsequent sequence of 6 dominant bits - i.e. 12 in total.



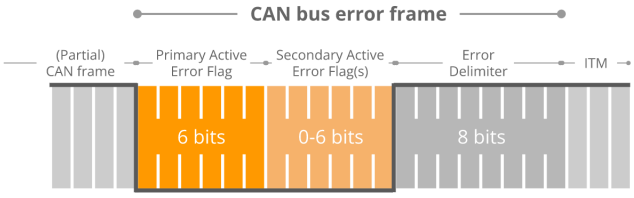
Example 3: 9 bits of error flags

Here, CAN node 1 has already transmitted a sequence of 3 dominant bits when it discovers a Bit Error and begins sending 6 dominant bits.

Once halfway through the primary Active Error Flag, nodes 2 and 3 recognize the Bit Stuffing Error (due to the 3 initial dominant bits being followed by another 3 dominant bits) and they begin raising their error flags. The result is that the sequence of dominant bits from error flags becomes 9 bit long.



The above logic of raising error flags is reflected in what we call an 'active' CAN error frame.



Note in particular how the secondary error flags raised by various nodes overlap each other - and how the primary and secondary flags may overlap as well. The result is that the dominant bit sequence from raised error flags may be 6 to 12 bits long.

This sequence is always terminated by a sequence of 8 recessive bits, marking the end of the error frame.

In practice, the active error frame may "begin" at different places in the erroneous CAN frame, depending on when the error is discovered. The result, however, will be the same: All nodes discard the erroneous CAN frame and the transmitting node can attempt to re-transmit the failed message.

Passive Error Flags

If a CAN node has moved from its default 'active' state to a 'passive' state (more on this shortly), it will only be able to raise so-called 'Passive Error Flags'. A Passive Error Flag is a sequence of 6 recessive bits as seen below.

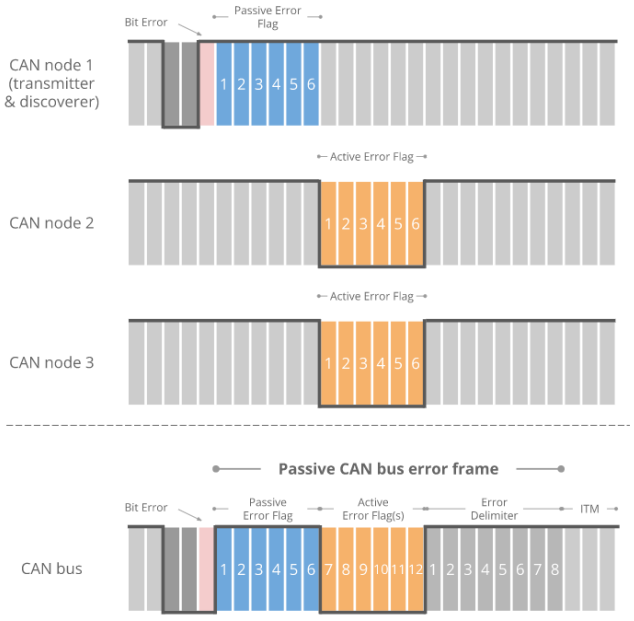
In this case it's relevant to distinguish between a Passive Error Flag raised by a transmitting node and a receiving node.

Example 4: Transmitter is Error Passive

As shown in the illustration (Example 4), if a transmitter (such as CAN node 1 in our example) raises a Passive Error Flag (e.g. in response to a Bit Error), this will correspond to a consecutive sequence of 6 recessive bits.

This is in turn detected as a Bit Stuffing Error by all CAN nodes. Assuming the other CAN nodes are still in their Error Active state, they will raise Active Error Flags of 6 dominant bits. In other words, a passive transmitter can still "communicate" that a CAN frame is erroneous.

Example 4: 'Passive' CAN bus error frame (transmitter in Error Passive mode)

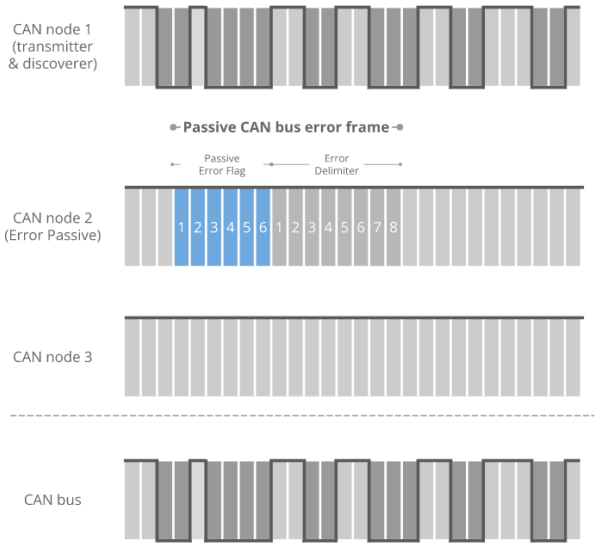


Example 5: Receiver is Error Passive

In contrast, if a receiver raises a Passive Error Flag this is in practice "invisible" to all other CAN nodes on the bus (as any dominant bits win over the sequence of recessive bits) - see also Example 5.

Effectively, this means that an Error Passive receiver no longer has the ability to destroy frames transmitted by other CAN nodes.

Example 5: 'Passive' CAN bus error frame (receiver in Error Passive mode)



CAN error types

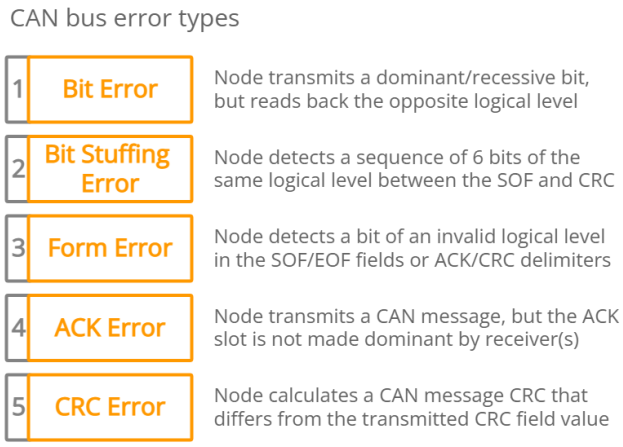
Next, let us look at what errors may cause CAN nodes to raise error flags.

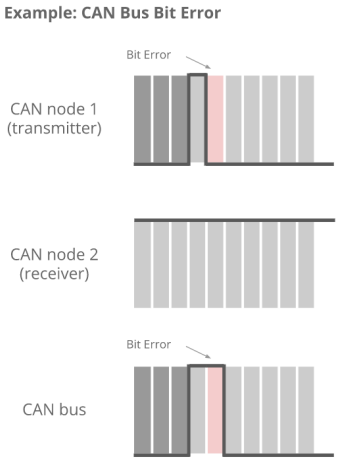
The CAN bus protocol specifies 5 CAN error types:

1. Bit Error [Transmitter]
2. Bit Stuffing Error [Receiver]
3. Form Error [Receiver]
4. ACK Error (Acknowledgement) [Transmitter]
5. CRC Error (Cyclic Redundancy Check) [Receiver]

We've already looked at Bit Errors and Bit Stuffing Errors briefly, both of which are evaluated at the bit level. The remaining three CAN error types are evaluated at the message level.

Below we detail each error type:

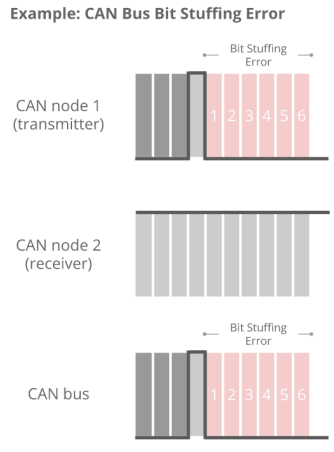




#1 Bit Error

Every CAN node on the CAN bus will monitor the signal level at any given time - which means that a transmitting CAN node also "reads back" every bit it transmits. If the transmitter reads a different data bit level vs. what it transmitted, the transmitter detects this as a Bit Error.

If a bit mismatch occurs during the arbitration process (i.e. when sending the CAN ID), it is not interpreted as a Bit Error. Similarly, a mismatch in the acknowledgement slot (ACK field) does not cause a Bit Error as the ACK field specifically requires a recessive bit from the transmitter to be overwritten by a dominant bit from a receiver.

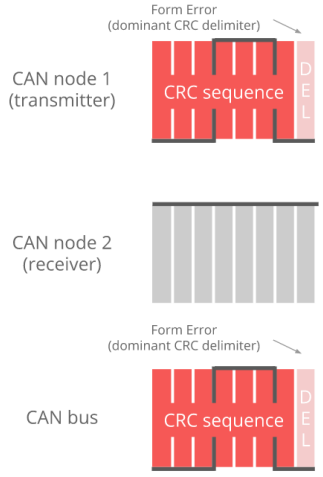


#2 Bit Stuffing Error

As explained, bit stuffing is part of the CAN standard. It dictates that after every 5 consecutive bits of the same logical level, the 6th bit must be a complement. This is required to ensure the on-going synchronization of the network by providing rising edges. Further, it ensures that a stream of bits are not mis-interpreted as an error frame or as the interframe space (7 bit recessive sequence) that marks the end of a message. All CAN nodes automatically remove the extra bits.

If a sequence of 6 bits of the same logical level is observed on the bus within a CAN message (between the SOF and CRC field), the receiver detects this as a Bit Stuffing Error aka Stuff Error.

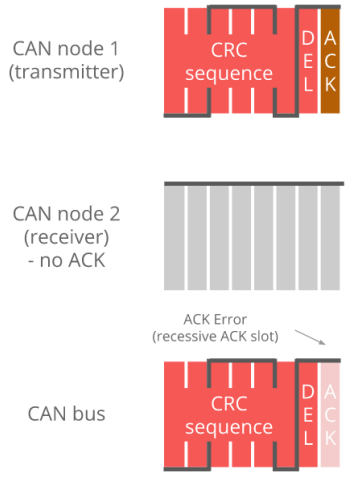
Example: CAN Bus Form Error



#3 Form Error

This message-level check utilises the fact that certain fields/bits in the CAN message must always be of a certain logical level. Specifically the 1-bit SOF must be dominant, while the entire 8-bit EOF field must be recessive. Further, the ACK and CRC delimiters must be recessive. If a receiver finds that any of these are bits are of an invalid logical level, the receiver detects this as a Form Error.

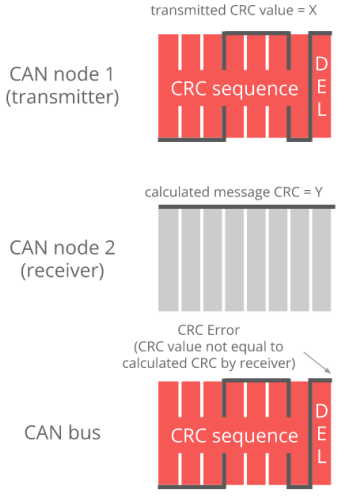
Example: CAN Bus ACK Error



#4 ACK Error (Acknowledgement)

When a transmitter sends a CAN message, it will contain the ACK field (Acknowledgement), in which the transmitter will transmit a recessive bit. All listening CAN nodes are expected to send a dominant bit in this field to verify the reception of the message (regardless of whether the nodes are interested in the message or not). If the transmitter does not read a dominant bit in the ACK slot, the transmitter detects this as an ACK Error.

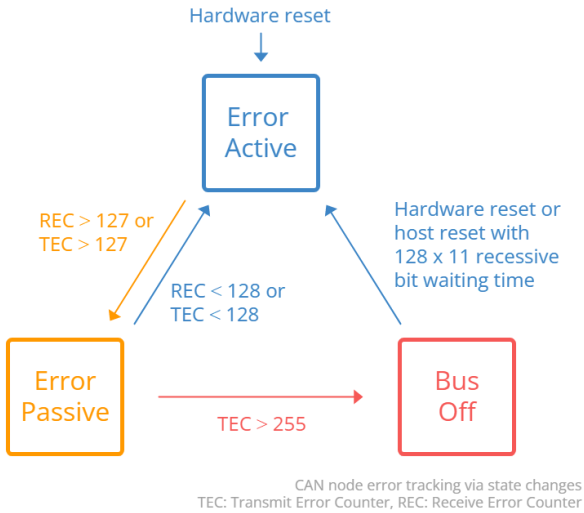
Example: CAN Bus CRC Error



#5 CRC Error (Cyclic Redundancy Check)

Every CAN message contains a Cyclic Redundancy Checksum field of 15 bits. Here, the transmitter has calculated the CRC value and added it to the message. Every receiving node will also calculate the CRC on their own. If the receiver's CRC calculation does not match the transmitter's CRC, the receiver detects this as a CRC Error.

CAN node states & error counters



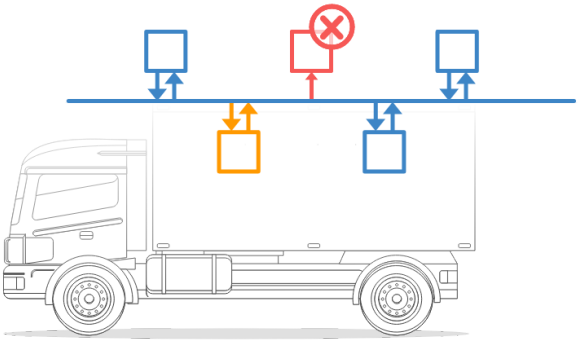
CAN node error tracking via state changes
TEC: Transmit Error Counter, REC: Receive Error Counter

As evident, CAN error handling helps destroy erroneous messages - and enables CAN nodes to retry the transmission of erroneous messages.

This ensures that short-lived local disturbances (e.g. from noise) will not result in invalid/lost data. Instead, the transmitter attempts to re-send the message. If it wins arbitration (and there are no errors), the message is successfully sent.

However, what if errors are due to a systematic malfunction in a transmitting node? This could trigger an endless loop of sending/destroying the same message - jamming the CAN bus.

This is where CAN node states and error counters come in.



In short, the purpose of CAN error tracking is to confine errors by gracefully reducing the privileges of problematic CAN nodes.

Specifically, let's look at the three possible states:

1. **Error Active**: This is the default state of every CAN node, in which it is able to transmit data and raise 'Active Error Flags' when detecting errors
2. **Error Passive**: In this state, the CAN node is still able to transmit data, but it now raises 'Passive Error Flags' when detecting errors. Further, the CAN node now has to wait for an extra 8 bits (aka Suspend Transmission Time) in addition to the 3 bit intermission time before it can resume data transmission (to allow other CAN nodes to take control of the bus)
3. **Bus Off**: In this state, the CAN node disconnects itself from the CAN bus and can no longer transmit data or raise error flags

Every CAN controller keeps track of its own state and acts accordingly. CAN nodes shift state depending on the value of their error counters. Specifically, every CAN node keeps track on a Transmit Error Counter (TEC) and Receive Error Counter (REC):

- A CAN node enters the Error Passive state if the REC or TEC exceed 127
- A CAN node enters the Bus Off state if the TEC exceeds 255

How do the error counters change?

Before we get into the logic of how error counters are increased/reduced, let us revisit the CAN error frame as well as the primary/secondary error flags.

As evident from the CAN error frame illustration, a CAN node that observes a dominant bit after its own sequence of 6 dominant bits will know that it raised a primary error flag. In this case, we can call this CAN node the 'discoverer' of the error.

At first, it might sound positive to have a CAN node that repeatedly discovers errors and reacts promptly by raising an error flag before other nodes. However, in practice, the discoverer is typically also the culprit causing errors - and hence it is punished more severely as per the overview.

CAN bus error counter logic

TEC +8	Transmitter raises primary error flag
REC +8	Receiver raises primary error flag
REC +1	Receiver raises secondary error flag
REC -1	Receiver successfully receives message
TEC -1	Transmitter successfully sends message

TEC: Transmit Error Counter, REC: Receive Error Counter

Details on TEC/REC counters

There are some additions/exceptions to the above rules, see e.g. this overview.

Most are pretty straight-forward based on our previous illustrative example. For example, it seems clear that CAN node 1 would increase the TEC by 8 as it discovers the Bit Error and raises an error flag. The other nodes in this case increase their REC by 1.

This has the intuitive consequence that the transmitting node will quickly reach the Error Passive and eventually Bus Off states if it continuously produces faulty CAN messages - whereas the receiving nodes do not change state.

The case where a receiver raises the primary error flag may seem counter-intuitive. However, this could for example be the case if a receiver CAN node is malfunctioning in a way that causes it to incorrectly detect errors in valid CAN messages. In such a case, the receiver would raise the primary error flag, effectively causing an error. Alternatively, it can happen in cases where all CAN nodes simultaneously raise error flags.

CAN/LIN data & error logger

The CANedge1 lets you easily record data from 2 x CAN/LIN buses to an 8-32 GB SD card - incl. support for logging CAN/LIN errors. Simply connect it to e.g. a car or truck to start logging - and decode the data via free software/APIs. Further, the CANedge2 (WiFi) and CANedge3 (3G/4G) let you auto-transfer data to your own server - and update devices over-the-air.

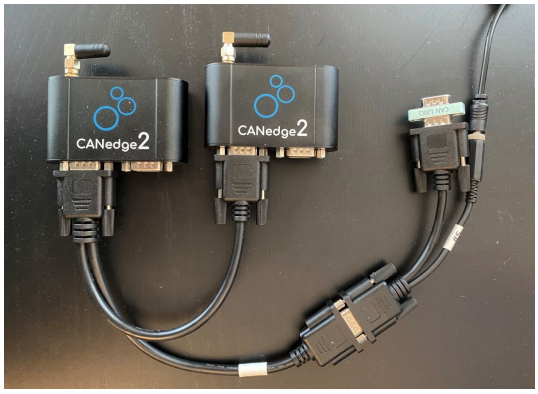
Examples: Generating & logging error frames

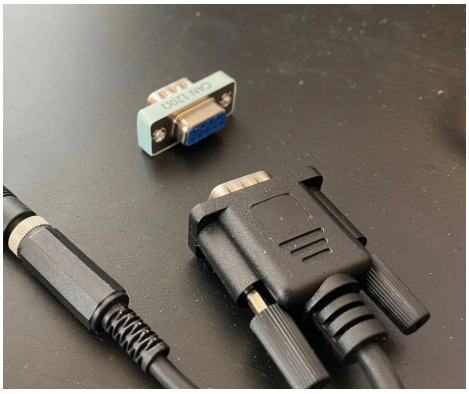
We have now covered the theoretical basics of CAN errors and CAN error handling. Next, let us look at generating and logging errors in practice. For this we will use a couple of CANedge devices - and for some tests a PCAN-USB device.

Test #1: No CAN bus errors

As a benchmark, we start with a test involving no CAN bus errors. Here, a CANedge2 'transmitter' sends data to another CANedge2 'receiver' - and both log CAN bus errors.

By loading the MF4 log file in the asammdf GUI we verify that no CAN errors occurred during this test, which is to be expected.





Test #2: Removing the CAN bus terminal resistor

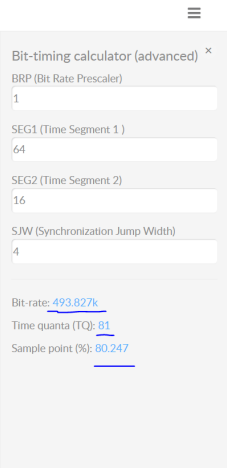
In this test, we remove the CAN termination in the middle of a log session. This effectively corresponds to immediately setting the bit level to dominant. As a result, the CANedge2 transmitter immediately starts logging **Bit Errors** (which occur when it attempts to transmit a recessive bit, but reads a dominant bit). The CANedge2 Receiver logs **Bit Stuffing Errors** as it detects 6 consecutive dominant bits. These errors are recorded until the termination is added again.

[see the online article for screenshots of the errors in asammdf]

How relevant is this in practice?

Lack of termination is rarely a practical issue if you're recording data from a vehicle, machine etc. However, it's a common issue when working with 'test bench' setups. Here, the lack of termination may cause confusion as it can be difficult to distinguish from an inactive CAN bus. If in doubt, enabling error frame logging on the CANedge can be useful in troubleshooting.

Test #3: Setting an incorrect baud rate



In this test we configure the CANedge receiver node to have a baud rate of 493.827K vs. the baud rate of the transmitter of 500K. This is a fairly extreme difference and results in **ACK Errors** for the transmitter and **Bit Stuffing Errors** for the receiver.

In more realistic scenarios, smaller differences in the baud rate configuration of various nodes may cause intermittent error frames and thus message loss.

[see the online article for screenshots of the errors in asammdf]

How relevant is this in practice?

This example is rather extreme. However, in practice we have sometimes seen CAN buses that use standard bit rates (250K, 500K, ...), but with specific bit timing settings that differ from the ones that are typically recommended (and hence used by the CANedge). This will not lead to a complete shut-down of the communication, but rather periodic frame loss of a few percentages. To resolve this, you can construct an 'advanced bit rate' in the CANedge configuration, essentially setting up the bit-timing to better match the CAN bus you're logging from.

Test #4: Removing the acknowledging CAN node

In this test, we use three CANedge units configured as follows:

- CANedge1: Configured to acknowledge data
- CANedge2 A: Configured in 'silent mode' (no acknowledgement)
- CANedge2 B: Configured to transmit a CAN frame every 500 ms

In the default setup, data is transmitted by the CANedge2 B onto the CAN bus and recorded with no errors. However, if we remove the CANedge1 from the bus there are no longer any CAN nodes to acknowledge the frames sent by the transmitter.

As a result, the transmitter detects **ACK Errors**. In response, it increases its Transmit Error Counter and raises Active Error Flags onto the CAN bus. These are in turn recorded by CANedge2 A (which silently monitors the bus) as **Form Errors**.

[see the online article for screenshots of the errors in asammdf]

The CANedge records Form Errors due to the fact that the transmitter raises them upon identifying the lack of a dominant bit in the ACK slot. As soon as a dominant bit is observed by the receiver in the subsequent EOF field (which should be recessive), a Form Error is detected.

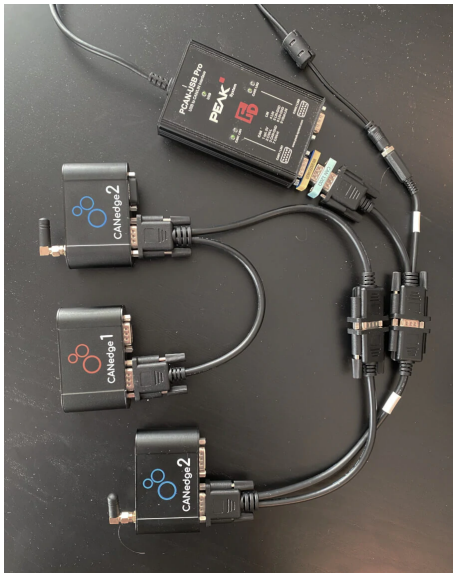
As evident, the transmitter broadcasts 16 Active Error Flags as its TEC is increased from 0 to 16 x 8 = 128. The transmitter has now exceeded the threshold of a TEC of 127 and enters Error Passive mode. As a result, the transmitter still experiences ACK Errors, but now only raises Passive Error

Flags (not visible to the receiver). At this point, the transmitter keeps attempting to transmit the same frame - and the receiver keeps recording this retransmission sequence.

How relevant is this in practice?

This type of error is one we often encounter in our support tickets. Specifically, users may be trying to use our CAN loggers to record data from a single CAN node (such as a sensor-to-CAN module like our CANmod). If they decide to enable 'silent mode' on the CANedge in such an installation, no CAN nodes will acknowledge the single CAN node broadcasting data - and the result will either be empty log files, or log files filled with retransmissions of the same CAN frame (typically at very high frequency).

Test #5: CAN frame collisions (no retransmission)



When setting up a CAN bus, it is key to avoid overlapping CAN IDs. Failing to do so can result in frame collisions as two CAN nodes may both believe they've won the arbitration - and hence start transmitting their frames at the same time.

To simulate this, we use the same setup as in test #4. In addition, we connect a PCAN-USB device as a secondary transmitter.

The CANedge2 transmitter is now configured to output a single CAN frame every 10 ms with CAN ID 1 and a payload of eight 0xFF bytes. Further, we configure the CANedge2 to disable retransmission of frames that were disrupted by errors. The PCAN-USB outputs an identical CAN frame every 2 ms with the 1st byte of the payload changed to 0xFE. The PCAN device has retransmissions enabled.

This setup quickly creates a frame collision, resulting in the CANedge and PCAN transmitters detecting a **Bit Error**. In response to this, both raise an Active Error Flag, which is detected as a **Bit Stuffing Error** by the CANedge receiver. The PCAN device immediately attempts a retransmission and succeeds, while the CANedge waits with further transmission until the next message is to be sent.

[see the online article for screenshots of the errors in asammdf]

How relevant is this in practice?

This type of error should of course never happen in e.g. a car, since the design and test processes will ensure that all CAN nodes communicate via globally unique CAN identifiers. However, this problem can easily occur if you install a 3rd party device (e.g. a sensor-to-CAN module) to inject data into an existing CAN bus. If you do not ensure the global uniqueness of the CAN IDs of external CAN nodes, you may cause frame collisions and hence errors on the CAN bus. This is particularly important if your external CAN node broadcasts data with high priority CAN IDs as you may then affect safety critical CAN nodes.

Test #6: CAN frame collisions (incl. retransmission)

In this test, we use the same setup as before, but we now enable retransmissions on the CANedge2 transmitter.

In this case, the frame collision results in a sequence of subsequent frame collisions as both the CANedge2 and the PCAN-USB device attempt to re-transmit their disrupted messages.

Due to the resulting Bit Errors, both raise a total of 16 Active Error Flags, which are detected as Bit Stuffing Errors by the silent CANedge2 receiver. Both transmitters then enter Error Passive mode and stop raising Active Error Flags, meaning none of them can destroy CAN frames on the bus. As a result, one of the transmitters will succeed in transmitting a full message, thus ending the retransmission frenzy - and enabling both devices to resume transmission. However, this only lasts for a few seconds before another collision occurs.

How relevant is this in practice?

The collision handling is a good example of how effective the CAN error handling is at 'shutting down' potentially problematic sequences and enabling CAN nodes to resume communication. If a frame collision occurs, it is likely that both CAN nodes will be set up to attempt retransmission, which would cause a jam if not for the error handling and confinement.

LIN bus errors

Similar to CAN bus errors, the LIN protocol also specifies a set of four error types, which we outline briefly below. The CANedge supports both CAN/LIN error frame logging.

#1 LIN Checksum Error

As for the CAN CRC Error, this error type implies that a LIN node has calculated a different checksum vs. the one embedded in the LIN bus frame by the transmitter. If you're using the CANedge as a LIN Subscriber, this error may indicate that you've configured the device 'frame table' with incorrect identifiers for some of the LIN frames on the bus.

This can in turn be used to 'reverse engineer' the correct lengths and IDs of proprietary LIN frames via a step-by-step procedure. See the CANedge Docs for details.

#2 LIN Receive Error

These occur if a specific part of the LIN message does not match the expected value, or if there is a mismatch between what is transmitted vs. read on the LIN bus.

#3 LIN Synchronization Error

This error indicates an invalid synchronization field in the start of the LIN frame. It can also indicate a large deviation between the configured bit rate for a LIN node vs. the bit rate detected from the synchronization field.

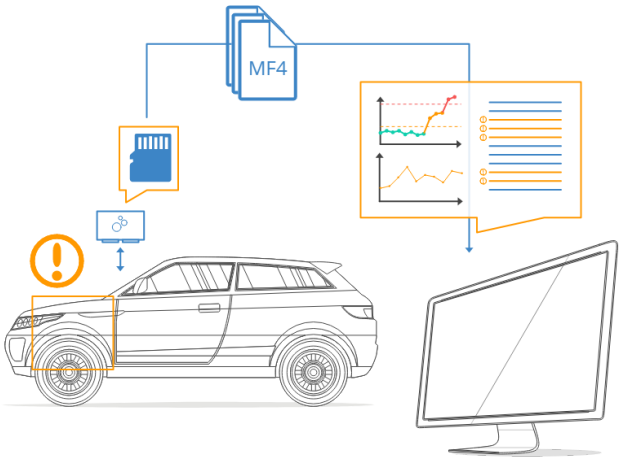
#4 LIN Transmission Error

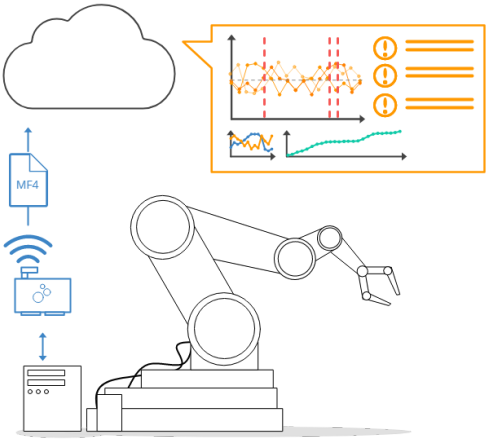
Transmission errors can occur for LIN identifiers registered as SUBSCRIBER messages. If there are no nodes responding to a SUBSCRIBER message, a transmission error is logged.

Example use cases for CAN error frame logging

CAN bus diagnostics in OEM vehicles

An automotive OEM may have the need to record CAN error frames in the field during late stage prototype testing. By deploying a CANedge, the OEM engineering team will both be able to troubleshoot issues based on the actual CAN signals (speed, RPM, temperatures) - as well as issues related with the lower layer CAN communication in their prototype systems. This is particularly vital if the issues of interest are intermittent and e.g. only happen once or twice per month. In such scenarios, CAN bus interfaces are not well suited - and it becomes increasingly relevant to have a cost-effective device to enable scalable deployments for faster troubleshooting.





Remotely troubleshooting CAN errors in machinery

An OEM or aftermarket user may need to capture rare CAN error events in their machines. To do so, they deploy a CANedge2 to record the CAN data and related error frames - and automatically upload the data via WiFi to their own cloud server. Here, errors are automatically identified and an alert is sent to the engineering team to immediately allow for diagnosing and resolving the issue.

FAQ

Does the CANedge support all CAN/LIN error types?

Yes, the CANedge is able to record all CAN/LIN error types. It does, however, not currently record its own error counter status as this is deemed less relevant from a logging perspective.

Will a CANedge raise error flags?

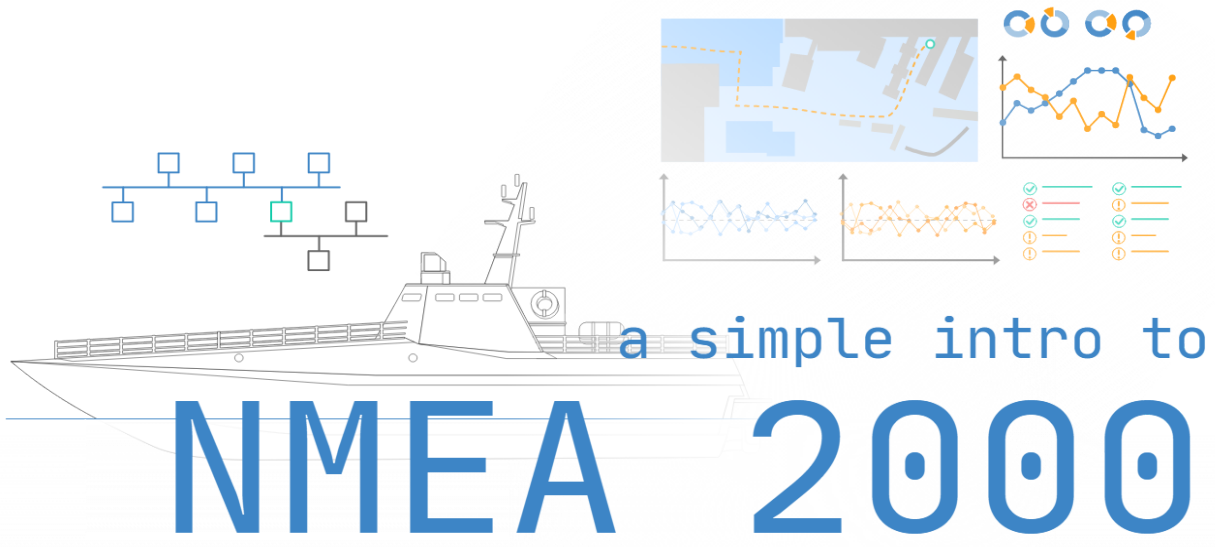
The CANedge is only able to raise error flags onto the CAN bus if it is configured in its 'normal' mode, in which it is also able to transmit messages. If in 'restricted' mode it can listen to CAN frames and acknowledge CAN frames - but not raise Active Error Flags onto the bus. In 'monitoring' mode (aka 'silent mode') it can listen to the CAN bus traffic, but not acknowledge messages nor raise Active Error Flags. The CANedge will always record internal CAN/LIN error frames.

What information can be recorded regarding a CAN/LIN error?

If a CAN frame is erroneous, resulting in an error frame, the CANedge generally only records the error type - without any data related to the erroneous frame (beyond the timestamp). One exception to this rule is for acknowledgement errors, where the CANedge will still record unacknowledged CAN frames (incl. from retransmission attempts).

Is error handling a cybersecurity risk?

Some researchers have pointed out the risk that 'bad actors' could utilize the CAN bus error handling functionality to enforce remote 'bus off' events for safety-critical ECUs. This is a good example of why CAN bus data loggers & interfaces like the CANedge2 with remote over-the-air data transfer and updates need to be highly secure (see also our intro to CAN cybersecurity). For a nice overview of a remote bus off attack, see this intro by Adrian Colyer.



NMEA 2000 Explained - A Simple Intro

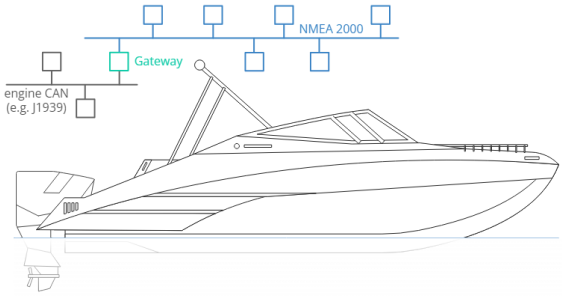
Need a simple, practical intro to NMEA 2000?

In this guide we introduce the N2K protocol (IEC 61162-3) used in maritime vessels such as boats, yachts, tugs and other ships. In particular, we'll cover the NMEA 2000 standards, key concepts, links to other protocols (NMEA 0183, CAN/J1939), topology and connectors. We also cover NMEA 2000 PGNs and Fast Packets.

To ensure this stays practical, we also explain how to record and decode NMEA 2000 data - with practical use cases and examples. Learn more below!

What is NMEA 2000?

NMEA 2000 (IEC 61162-3) is a communication standard used in the maritime industry for connecting e.g. engines, instruments and sensors on boats. It is based on the Controller Area Network (CAN) and allows data to be sent/received between devices over a single network 'backbone' cable.



The higher layer functionality of NMEA 2000 (aka N2K) is based on SAE J1939 and ISOBUS (ISO 11783), commonly used in heavy-duty vehicles and agriculture/forestry machinery, respectively.

NMEA 2000 succeeds the older NMEA 0183 and is today used in most modern maritime vessels like boats, ships, yachts and related equipment/sensors.

Multiple boat networks

Importantly, a maritime vessel may use multiple networks.

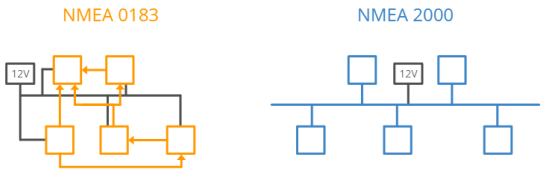
For example, a twin engine speed boat may comprise an NMEA 2000 network connecting both outboard engines to the helm accessories (control head, keys, displays). This network will typically be constructed by the boat OEM. The engines themselves may internally communicate via e.g. the J1939 protocol with a gateway/filter module feeding engine related data to the NMEA 2000 network.

In addition, various sensors measuring e.g. GPS position, wind, water depth etc. may be added to same NMEA 2000 network. However, cases also exist where such sensors are retrofitted by the boat operator onto a separate NMEA 2000 network.

NMEA 2000 vs NMEA 0183

NMEA 0183 (IEC 61162-1) is the predecessor to NMEA 2000 (IEC 61162-3). The NMEA organization published the NMEA 0183 protocol with the purpose of standardizing the communication between different electronic equipment in maritime vessels.

The protocol replaced NMEA 0180 and NMEA 0182. The NMEA 0183 protocol remains in active use today, although NMEA 2000 has become the industry standard.



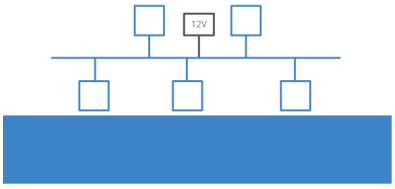
Details on NMEA 0183

The NMEA 0183 protocol uses the electrical standard RS232/RS422 for communication instead of CAN bus. In this setup, one 'talker' (e.g. a GPS module) can communicate with multiple 'listener' nodes (e.g. an autopilot and a chartplotter). However, it is not possible to have multiple 'talkers' on the same network (in contrast to CAN). As a result, a separate NMEA 0183 network is required for every 'talker' node - which quickly becomes complex when the networks are interconnected.

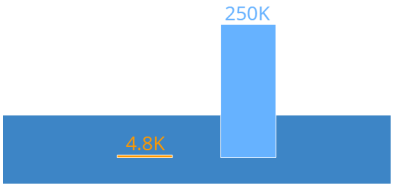
Data is communicated in an ASCII format as standardized 'sentences' defined in the application layer. The data is transmitted as printable ASCII characters in the range 0x20 to 0x7E and reflect 'physical values' (e.g. degrees), rather than 'raw data' as in NMEA 2000. For a detailed overview of the message/sentence structure, see this table.

Key benefits of NMEA 2000 vs. NMEA 0183

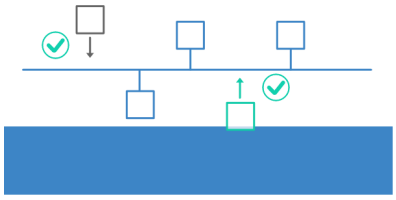
Overall, NMEA 2000 is a far more modern and flexible communication protocol compared to NMEA 0183, and is therefore today widely used in marine electronic systems. Below we outline the key advantages:



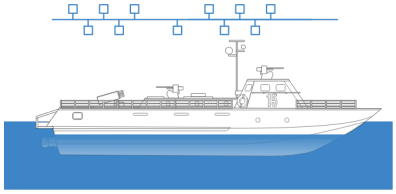
Simpler & lower cost network
NMEA 2000 allows all nodes to communicate with each other (multiple talkers) and the power supply is directly integrated in the backbone, reducing wiring costs and simplifying the network vs. NMEA 0183. The standardized T-connectors also simplify integration



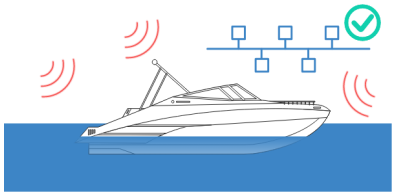
Faster data transmission
NMEA 2000 enables a higher data transmission rate than NMEA 0183 (250K vs. 4.8K), meaning it can transmit more data in a given amount of time. This is useful when real-time data is required, such as in navigation or engine monitoring applications



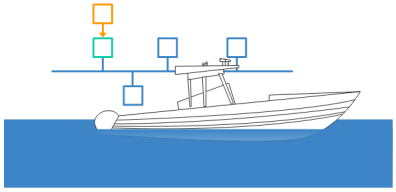
Better device compatibility
NMEA 2000 is designed to allow different devices from different manufacturers to be connected to the same network and to exchange data in a standardized way. This makes it easier to mix and match devices on the same network, and reduces the risk of compatibility issues



Larger network size
NMEA 2000 networks can be larger than NMEA 0183 networks, as they are designed to support more devices (up to 50 nodes) and transmit data over longer distances. This can be useful in larger vessels or systems where a greater number of devices need to be connected



More robust communication
NMEA 2000 uses the Controller Area Network (CAN) bus, which is a digital communication protocol that is less susceptible to noise and interference than the serial communication protocol used by NMEA 0183. This makes NMEA 2000 networks more reliable and resilient



Backwards compatibility
NMEA 2000 enables integration of NMEA 0183 devices via low cost gateways, enabling the integration of existing electronics into new networks. The reverse is, however, not true - i.e. NMEA 2000 nodes cannot be added to NMEA 0183 networks

NMEA history and certification

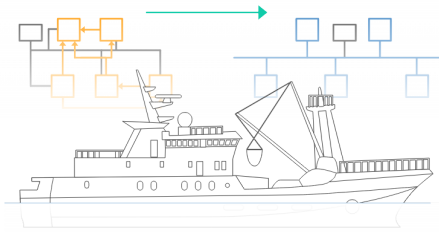
The National Marine Electronics Association (NMEA) is a non-profit organization founded in 1957. It represents manufacturers, distributors, and dealers in the maritime industry and helps ensure device compatibility across brands/manufacturers through the promotion of data communication standardization, education and certification.

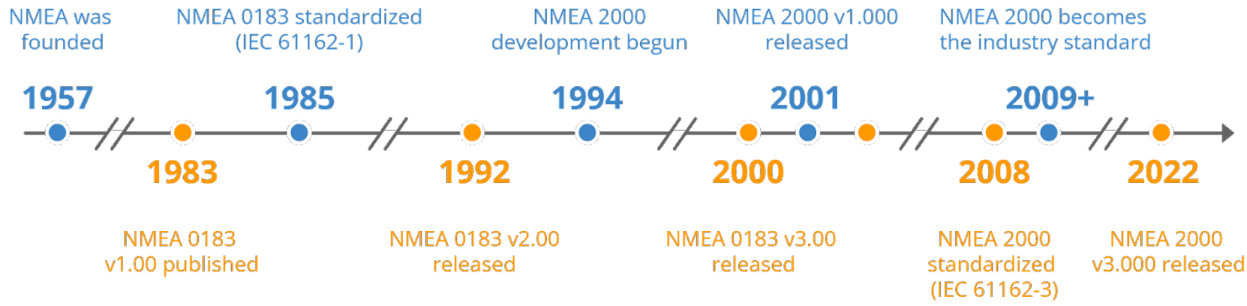
NMEA and AEF

To some extent, the role of NMEA can be compared to that of the Agricultural Industry Electronics Foundation (AEF) in the agriculture/forestry industry. Both NMEA 2000 and ISOBUS share the need for strict protocol conformance requirements in order to deliver on the promise of plug & play integration of products across manufacturers. Here, the organisations play a key role in facilitating a standardised process for product certification - as well as in providing a database for identifying conforming manufacturers/products.

NMEA history

- 1957: National Marine Electronics Association founded
- 1983: NMEA 0183 v1.00 published by NMEA
- 1985: NMEA 0183 standardized as IEC 61162-1
- 1992: NMEA 0183 v2.00 released
- 1994: N2K development begun by Standards Committee
- 2000: NMEA 0183 v3.00 released
- 2001: NMEA 2000 v1.000 released
- 2008: NMEA 2000 standardized as IEC 61162-3
- 2008: NMEA 0183 v4.00 released
- ~2009: NMEA 2000 becomes the industry standard
- 2022: NMEA 2000 v3.000 released



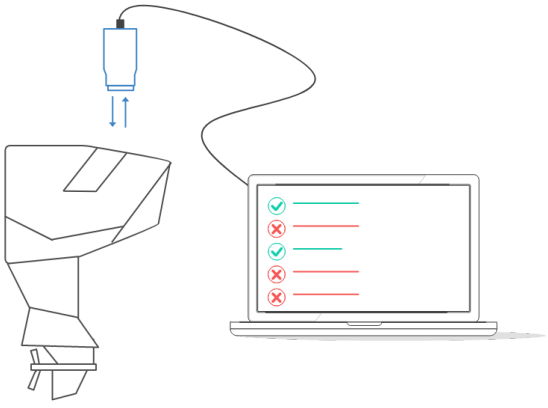


NMEA 2000 conformance certification

To ensure interoperability of N2K nodes, NMEA facilitates a certification process through which a product can become 'NMEA 2000 Certified' and add a certification logo to the product. In practice, the certification is done via a self-certification process using mandated test equipment (the NMEA 2000 Certification Tool, i.e. a CAN interface/software) and validation by NMEA. Certification does not imply that a specific product must produce certain data. Rather, it ensures that a product is compatible with other NMEA 2000 certified products - enabling them to coexist on the same network without causing interference.

NMEA certification classes and levels

The NMEA 2000 product certification can be granted as both a Class 1 and Class 2, with a Level A and B in each. Class 1 devices rely on a single NMEA 2000 interface for communication, while Class 2 devices provide two interfaces for the purpose of redundancy. For each class, Levels A and B allow for distinguishing between sophisticated devices (with full communication support incl. e.g. implementation of ISO TP communication) and more simplistic devices with a smaller scope of required communication functionality.



NMEA 2000 approved vs. certified

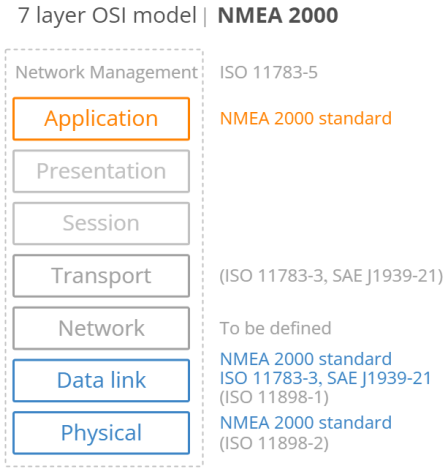
Some third-party products may use alternative classifications from the 'NMEA 2000 Certified':

1. NMEA 2000 Approved: These products are able to implement NMEA 2000 and have passed the physical hardware tests of the N2K certification program
2. NMEA 2000 Certified for use with [Approved Product Name]: These products may be used in combination with a specific NMEA 2000 Approved product, together creating a fully compatible product
3. NMEA 2000 Certified for use with [Certified Gateway Product Name]: These products may be used in combination with an NMEA 2000 Certified gateway product.

Finally, another classification is 'NMEA 2000 Approved Diagnostic Tool - Not for permanent connection to a NMEA 2000 backplane' - intended for tools that may e.g. disrupt safety-critical functionality in the NMEA 2000 network if used incorrectly.

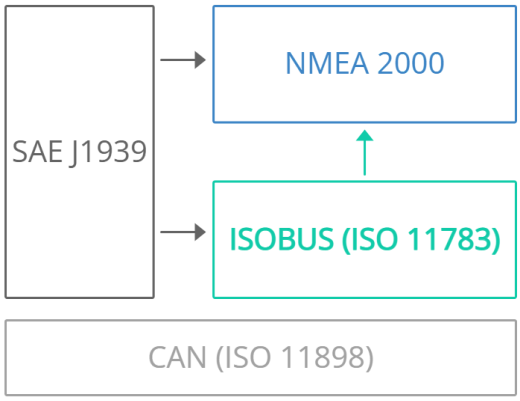
NMEA 2000 OSI model & standards

Below we briefly outline the link between each OSI model layer and the NMEA 2000 standard (as well as referenced standards).



- Application: This layer is fully defined by the NMEA 2000 standard and defines approved PGNs/signals incl. a provision for manufacturer-specific proprietary messages. NMEA Appendix A defines PGNs, while Appendix B provides a full list of the standardized NMEA 2000 PGNs and signals
- Transport: In NMEA 2000, the transport layer is responsible for establishing and maintaining connections between devices, as well as fragmenting and reassembling messages as needed. This can be done via the transport protocol used in ISOBUS/J1939 or via NMEA Fast Packets (see also the Data Link layer)
- Network: This layer is to be further defined in the future in the NMEA 2000 standard
- Data Link: This layer is responsible for transmitting data across the communication channel and ensuring that the data is delivered correctly. The NMEA 2000 standard uses the CAN protocol at the data link layer, which includes error detection and correction mechanisms to ensure reliable transmission. Note that this layer references ISO 11783-3 (ISOBUS), SAE J1939-21 and 11898-1 (CAN) The NMEA 2000 standard defines additional requirements/functionality, including the NMEA Fast Packet for multiframe communication
- Physical: This layer is fully defined by the NMEA 2000 standard, including signal voltages, cables and connectors. Note also the inclusion of 11898-2 (CAN) to reflect the role of CAN bus as the basis for NMEA 2000. This also specifies the standard baud rate of 250K for NMEA 2000 and details on cables and connectors

Further, the 'Network Management' is defined by ISO 11783-5 (ISOBUS). Note in particular that all NMEA 2000 compliant devices must support dynamic address claiming



NMEA 2000 vs. J1939 vs. ISOBUS

As evident, NMEA 2000 is closely related to J1939 and ISOBUS. Here we briefly clarify the link between these protocols. First, let's do a very quick history recap:

In 1994, the Society of Automotive Engineers (SAE) published the J1939 protocol for use in heavy-duty vehicles. Next, ISOBUS was derived from J1939 for agriculture/forestry machinery, formally standardized in 2007 (ISO 11783).

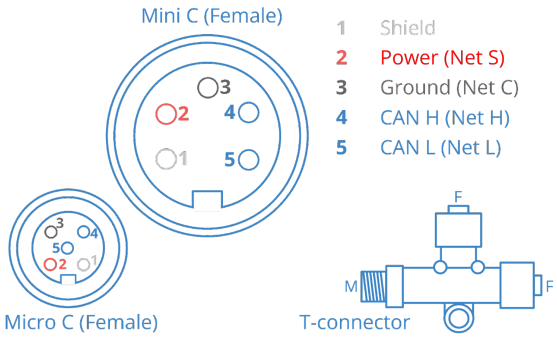
NMEA 2000 was released in 2001 and references parts of SAE J1939 and ISO 11783 almost interchangeably. ISO 11783-3 is referenced for the data link layer, but in turn references J1939-21. Similarly, NMEA 2000 references ISO 11783-5 for the network management, which is virtually identical to J1939-81.In other words, NMEA 2000 is based on ISO 11783 standards that are in turn based on J1939 standards.

Practical implications

In practical applications, NMEA 2000 is often seen in combination with J1939 or ISOBUS. For example, some boat engines use J1939 internally and deploy a gateway/filter to parse certain information onto standardized NMEA 2000 PGNs. In some cases, NMEA 2000 and J1939 messages even co-exist on the same CAN bus. This is possible as none of the NMEA 2000 PGNs overlap with J1939 PGNs. Similarly, NMEA 2000 messages are often broadcast onto an ISOBUS network since many GPS modules are used in both maritime and agricultural use cases. See our ISOBUS intro for details.

NMEA 2000 connectors & network topology

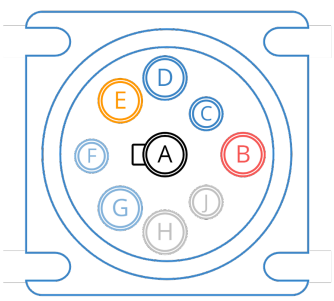
Below we describe the most common connectors encountered in maritime vessels:



#1 T connector (M12 5-pin)

The T-connector (aka T-piece) is used in constructing the NMEA 2000 CAN bus 'backbone' in a linear fashion and for connecting equipment. It is a three-way connector consisting of two female connectors and one male connector. The connectors are standardised DeviceNet 5-pin A-coded M12 connectors. The physical connector and pinout differ between mini type (for backbones) and micro type (for small backbones and for drop cables). The micro M12 connector is also used in industrial applications like CANopen and enable easy connection of CAN based equipment - e.g. CAN bus data loggers.

m12 adapter cable



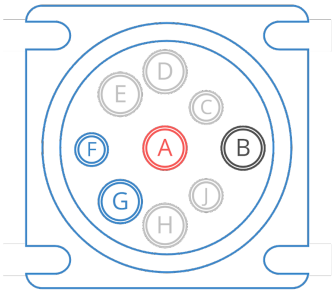
A Ground
B Battery power
C CAN 1 H
D CAN 1 L
E CAN shield
F J1708 (+) / CAN 2 H
G J1708 (-) / CAN 2 L
H OEM specific
J OEM specific

Type 1 (black): CAN 1 = 250K
Type 2 (green): CAN 1 = 500K

#2 Engine connector (J1939 9-pin)

Many maritime engines (e.g. from Cummins) deploy the J1939 protocol for their internal communication. Some of this information may be broadcast onto the NMEA 2000 network directly or through a gateway that converts the PGN information to the corresponding NMEA 2000 encoding. Many engines come with separate diagnostic connectors that allow direct access to the raw J1939 data - and the most common connector for these would be the J1939 deutsch 9-pin connector.

j1939 adapter cable

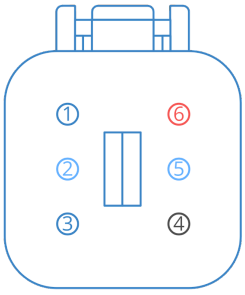


A Battery power
B Ground
C Not specified
D Not specified
E Not specified
F CAN L
G CAN H
H Not specified
J Not specified

#3 Engine connector (CAT 9-pin)

Some maritime engines (e.g. from Caterpillar) use a deutsch 9-pin CAT adapter. This looks almost identical to the J1939 9-pin connector, but uses a different pinout.

cat adapter cable



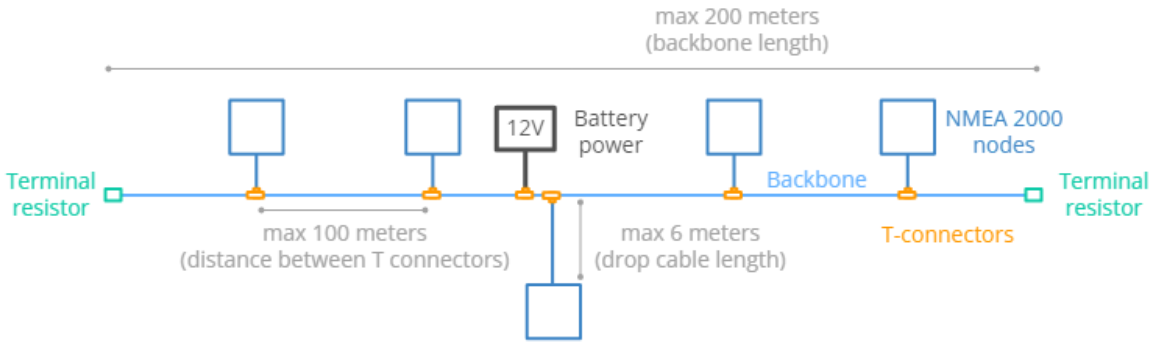
1 CAN 1 L
2 CAN 2 L
3 CAN 1 H
4 Ground
5 CAN 2 H
6 Battery power

#4 Engine connector (Deutsch DT06-6P)

In some maritime engines (e.g. from Volvo Penta), alternative diagnostic connectors are used such as the DT06-6P (male 6-pin) and DT06-6S (female 6-pin). You'll often be able to convert from these connectors to other popular connectors (like OBD2) via off-the-shelf adapters.

NMEA 2000 physical layer requirements

The NMEA 2000 standard also specifies requirements in terms of the physical layer as shown below.



Details on the NMEA 2000 topology

- The standard bit rate is 250 kb/s
- The CAN bus should be constructed as a linear 'backbone'
- The distance between two T-connectors must not exceed 100m
- The maximum length of the network is 200m
- The network supports up to 50 connected devices
- The max 'drop cable' length (between a T-connector and device) is 6m
- The max cumulative drop cable length is 78m
- Cables must adhere to DeviceNet (ODVA) standards
- The backbone must include a grounded power supply in the 9-16 V DC range (e.g. from the vessel 12 V DC power supply or from an isolated 15 V DC supply)
- Three cable types: Mini/thick (for backbones requiring 4-8 amps per leg), mid (for backbones) and micro/thin (for drop cables and small backbones)

NMEA 2000 Fast Packet

A key part of the NMEA 2000 standard is the 'Fast Packet'.

As per SAE J1939-21 and ISO 11783-3, the NMEA 2000 standard supports multi-packet communication of up to 1785 bytes via the ISO 15765-2 (ISO TP) standard.

However, NMEA 2000 often requires frame payloads that exceed 8 bytes, but far less than 1785 bytes. Thus, the Fast Packet was introduced as a more efficient transport protocol for medium-size payloads up to 223 bytes without any transfer protocol delays. See the table overview for a comparison.

NMEA 2000 - Single Frame vs. ISO TP vs. Fast Packets

Single Frame	ISO TP (ISO 11783 Multi Packet)	NMEA 2000 Fast Packet
Up to 8 bytes	Up to 1785 bytes	Up to 223 bytes
Destination defined by PGN	Send any PGN to specific device	Destination defined by PGN
No handshaking	Optional handshaking	No handshaking
No transfer protocol delay	Transfer protocol delays	No transfer protocol delay
PGN is part of CAN ID	PGN is part of data payload	PGN is part of CAN ID

Frame identification

In the NMEA 2000 Fast Packet methodology, every frame retains the original PGN identifier. This means that the multi-frame message can be uniquely identified at the ID level, in contrast to the ISO TP implementation in J1939, ISOBUS and **UDS**, in which message identification requires extraction of information from the 1st CAN frame payload data.

Fast Packet trace example

Below we show an NMEA 2000 Fast Packet example. As evident, all the transmitted frames contain a 'sequence counter' in the 1st nibble and a 'frame counter' in the 2nd nibble of the 1st byte. Further, the first frame transmitted contains the total payload length in the 2nd byte. To enable decoding, the frame needs to be reassembled as shown.

Timestamp (Epoch)	CAN ID	PGN (HEX)	PGN (DEC)	DataBytes
1620692704.079450	11F80F00	1F80F	129039	001B127CEAD5123D
1620692704.080000	11F80F00	1F80F	129039	0131F3D0ACF2231A
1620692704.080600	11F80F00	1F80F	129039	0203FFFF00000000
1620692704.081200	11F80F00	1F80F	129039	0320FFFF0070FEFF
...				
1620692705.079500	11F80F00	1F80F	129039	201B127CEAD5123D
1620692705.080050	11F80F00	1F80F	129039	2131F3D0ACF2231A
1620692705.080650	11F80F00	1F80F	129039	2207FFFF00000000
1620692705.081200	11F80F00	1F80F	129039	2320FFFF0070FEFF
1620692704.079450	11F80F00	1F80F	129039	127CEAD5123D ... 20FFFF0070FEFF
1620692705.079500	11F80F00	1F80F	129039	127CEAD5123D ... 20FFFF0070FEFF

legend
NMEA 2000 PGN
(0x1F80F = AIS Class B Position Report)
#data bytes (0x1B=27)
sequence counter
frame counter
payload data (27 bytes)

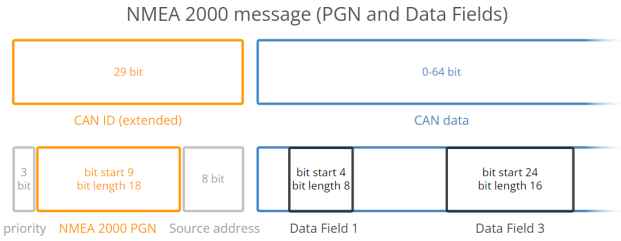
The CANedge Python API enables automatic reassembly of Fast Packets into single with the full data payload (stripped of the sequence counters and payload length). This in turn enables easy DBC decoding of the data using the NMEA 2000 DBC file. See our api-examples repository for script examples.

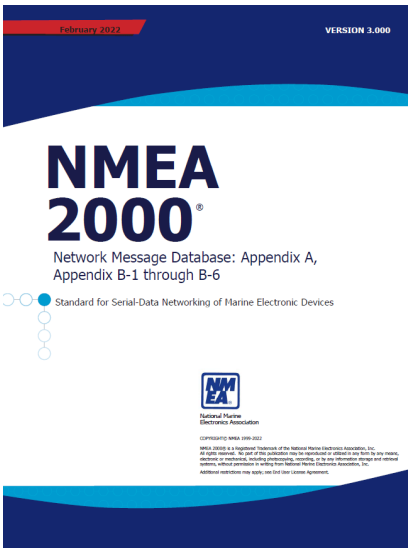
N2K PGN & data fields

NMEA 2000 uses the concept of Parameter Group Numbers (PGN) from the J1939 protocol. This means that messages are identified based on 18-bit PGNs, which can be extracted from the full 29-bit CAN identifiers.

Signals aka parameters will be 'packaged' in the data payload of each message. Note that in contrast to J1939, signals are not referred to as Suspect Parameter Numbers (SPN), but instead as Data Fields (DF) or simply parameters.

We do not cover the PGN structure here as it is already covered in our SAE J1939 intro. However, we will provide a few practical comments on NMEA 2000 PGNs and parameters.



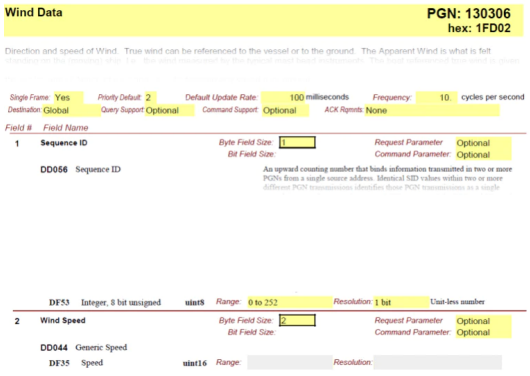


NMEA 2000 PGN database

While NMEA 2000 was originally based on J1939, the standard J1939 PGNs (from J1939-71) do not sufficiently cover the maritime requirements of the NMEA standard.

To solve this, the NMEA 2000 application layer defines a number of PGNs unique to NMEA 2000, which are described in the NMEA 2000 standard, Appendix B. The appendix lists a number of PGNs and the data fields contained within each of them. Each data field is described through a 'Data Dictionary'. This can be compared to how J1939-71 describes each PGN and their underlying SPNs.

As evident from the PGN extract, the Appendix B PDF contains information for decoding parameters like Wind Speed. However, extracting this manually would be time-consuming as the PDF contains 170+ PGNs and 1500+ parameters. Instead, we recommend using the [NMEA 2000 DBC file], which includes the PGN and parameter information in a structured way, ready for use in decoding raw NMEA 2000 data.



Regarding proprietary NMEA 2000 PGNs

The NMEA 2000 standard allows manufacturers to create their own proprietary PGNs, though with three conditions:

1. The proprietary PGN contains device specific information (e.g. a unique calibration)
2. The data is being used for test purposes
3. A suitable PGN is not available in the current NMEA 2000 PGN database

In general, manufacturers are encouraged to leverage the standardized PGNs (or migrate to these when they become available). Further, if proprietary PGNs are used in a device, NMEA encourages manufacturers to publish the decoding information.

Examples of NMEA 2000 signals in Appendix B

The NMEA 2000 appendix B contains 1500+ parameters aka Data Fields, encoded in NMEA 2000 specific PGNs. Below are some examples of the maritime-oriented signals included:

- Alerts
- GNSS position
- Vessel speed
- Heading
- Rudder angle

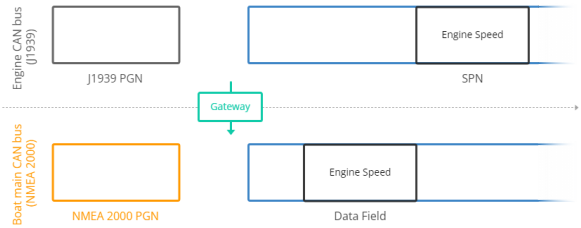
- Rate of turn
- Yaw, pitch, roll
- Engine speed
- Temperatures
- Fuel pressure
- Fuel rate
- Engine load
- Transmission
- Battery status
- Water depth
- Wind speed
- Air temperature
- ... and more

NMEA vs J1939 PGNs [+DBC files]

In maritime vessels, J1939 is often used 'within the engines' (e.g. outboard engines) to communicate various engine-related sensor data like engine speed, temperatures, etc.

In such cases, a 'gateway' will typically convert a subset of the J1939 PGN information into NMEA 2000 PGNs, thus allowing e.g. displays to visualize the engine information.

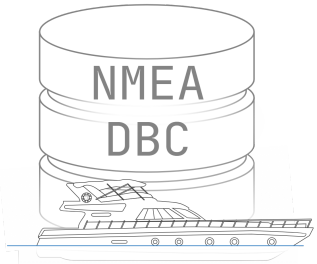
For example, the J1939 PGN 0xF004 (61444) contains Engine Speed - while it is encoded in the N2K PGN 0x1F200 (127488).



Original J1939 PGN transmission

As such, if you're recording data only from the NMEA 2000 network, you'll typically get the 'pure' NMEA 2000 PGN information plus the 'filtered' J1939 PGN information (encoded as NMEA 2000 PGNs). In some cases, however, the 'original' J1939 PGNs will be directly output onto the NMEA 2000 network. This is possible because the protocols are 'harmonized', i.e. they use the same bit rate of 250K and none of the CAN IDs overlap.

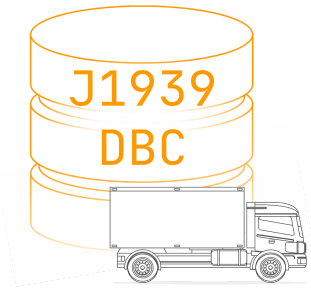
In short, to maximize the vessel data you can decode, you should consider the following DBC files:



NMEA 2000 data (vessel, sensors)

If you connect to the NMEA 2000 network (e.g. via an M12 connector), you will generally record the broadcast NMEA 2000 data. To decode this, you can use the NMEA 2000 DBC, which will allow you to extract information on the standardized PGNs from the network. This may include 'filtered' engine information as well.

learn more



J1939 data (engines)

In some cases, the NMEA 2000 network may also contain the original J1939 PGNs from the engine. Alternatively, you may be able to access the original J1939 information by connecting directly to the engine via a separate diagnostic connector (typically a deutsch 9-pin J1939 connector). In either case, you can use the J1939 DBC file to decode the standardized J1939 PGNs.

learn more

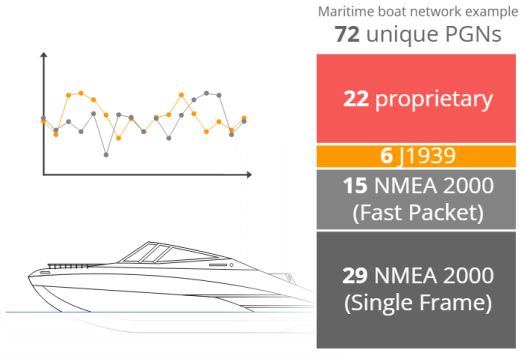
OEM data (proprietary)

There will almost always be some NMEA 2000 PGNs that are proprietary. These are in principle only known to the OEM, but since the NMEA organization encourages OEMs to publish their encoding rules, it is likely that you will find the relevant information in technical manuals or on-request. If so, you can add them to your existing DBC file by modifying it via a DBC editor (see our DBC intro for details).

Example: Decoding data from a boat

To showcase the breakdown of PGNs you'll often see in maritime CAN data, consider a log file from a speed boat. In this case, a CANedge1 has been used to record data from the NMEA 2000 backbone, which also includes (at least some of) the original J1939 data from the outboard engine. The log file contains **161 unique CAN IDs**, corresponding to **72 unique PGNs.**

As evident, the majority of the PGNs are decoded by the NMEA 2000 DBC (as Single Frames or Fast Packets). Further, the J1939 DBC is able to decode some of the J1939 PGNs, while the rest are OEM specific. The proprietary PGNs may originate from the J1939 engine network or some of the NMEA 2000 network equipment.



Note regarding J1939 PGNs

Note here that the connection of the CANedge1 was made on the NMEA 2000 network - not directly onto the J1939 network (i.e. the engine's diagnostic connector). As a result, it is not known in this case whether more J1939 PGNs would be available via a direct connection to the engine. Further, in most NMEA 2000 networks, the engine J1939 PGNs are not directly broadcast onto the NMEA 2000 network, hence you'd normally need to record these separately. This is often done with the CANedge, connecting the 1st port to the NMEA 2000 network and the 2nd port to the J1939 engine diagnostic connector.

Logging NMEA 2000 maritime data

To log maritime CAN bus data from an NMEA 2000 network or J1939 engine, we recommend to check out our maritime telematics intro. Below we will briefly recap the most relevant steps to consider:

#1 How to choose a CAN logger

If you need to log N2K/J1939 data to an SD card we recommend the CANedge1. The device can be left in the vessel to record data for weeks or months. To collect the data, simply extract the SD. Alternatively, the CANedge2 (WiFi) and CANedge3 (3G/4G) let you auto-upload data to your own server. The device logs data to the SD card and uploads it when the vessel is near-shore with connectivity.

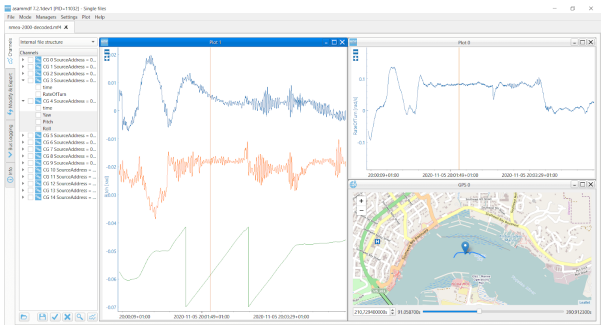
#2 How to connect to the N2K network

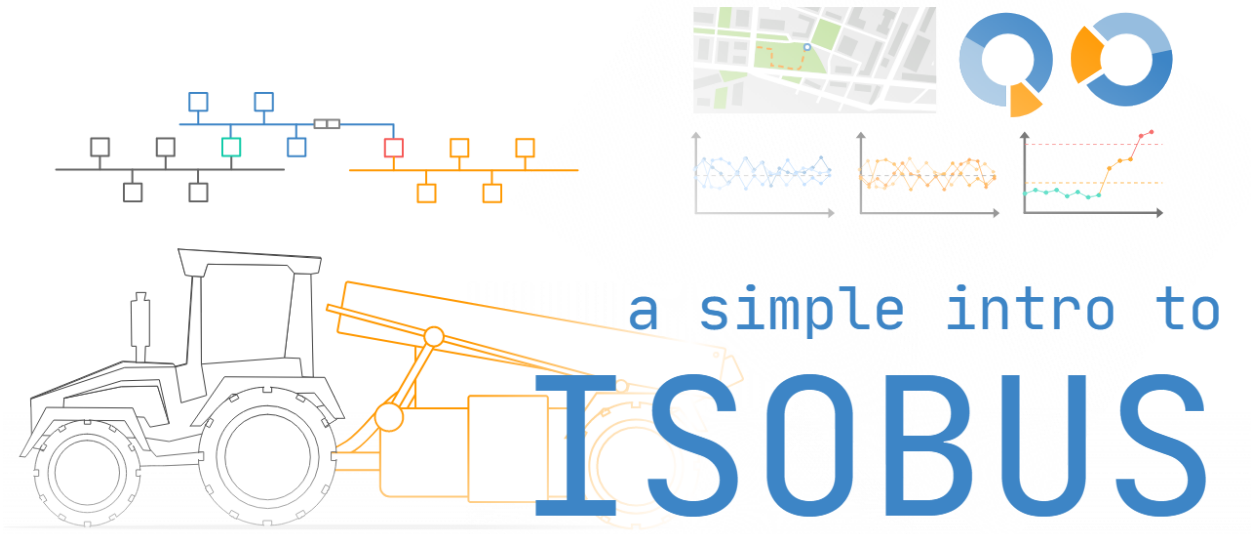
Next, you'll need to identify how to connect to the CAN bus. The simplest way to connect to the NMEA 2000 network is to use a DB9-M12 adapter to connect via a T-connector. This gives you access to all the raw NMEA 2000 data from that network. Optionally, you can use the 2nd CAN channel of the CANedge to connect to another NMEA 2000 network (if one exists) or directly to the J1939 engine network. In the latter case, you can use one of the engine diagnostic connectors listed previously.

#3 How to process the CAN data

To process the raw NMEA 2000 and/or J1939 vessel data, you need a CAN software tool. In this, you'll load the log files and related DBC files to convert the raw data to physical values.

The CANedge software tools enable different forms of analysis. For example, you can use the MF4 converters to transform the raw log files to popular file formats (e.g. ASC, TRC, CSV), for use in tools like Vector CANalyzer, Warwick X-Analyser 3 or PEAK PCAN-Explorer. Or, you can use the asammdf GUI to DBC decode the data and create visual plots incl. GPS maps of your vessel routes. Further, you can deploy automated data processing via the Python API if you need to perform statistical analysis, automated reporting or database integration. The API can also be used to integrate the data with Grafana dashboards.





ISOBUS (ISO 11783) Explained - A Simple Intro

Need a simple, practical intro to ISOBUS (ISO 11783)?

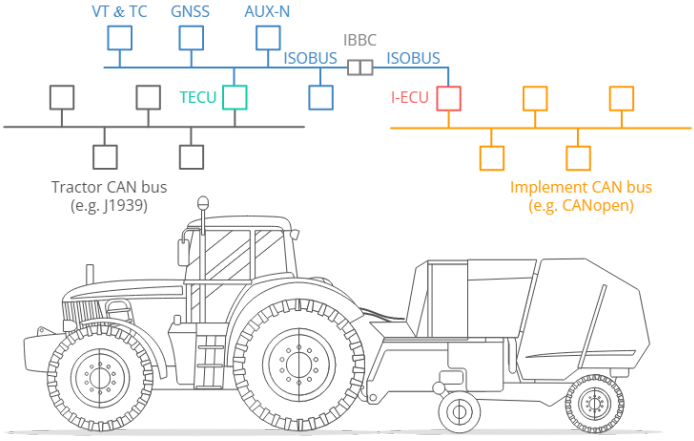
In this guide we introduce the ISOBUS protocol used in agricultural vehicles (like tractors) and implements (like sprayers, seeders). In particular, we'll cover the ISO 11783 standards, history, the link to J1939 and connectors.

To ensure this stays practical, we also explain how to record and decode ISOBUS data - with practical use cases and dashboard playground examples.

Learn more below!

What is ISOBUS?

ISOBUS (ISO 11783) is a standardised communication protocol used in agriculture and forestry machinery. The main purpose is to enable plug & play integration between vehicles (e.g. tractors) and implements (e.g. sprayers) across manufacturers.



The protocol is based on the Controller Area Network (CAN bus). Specifically, it has been derived from the SAE J1939 protocol used in most heavy-duty vehicles (including most tractors).

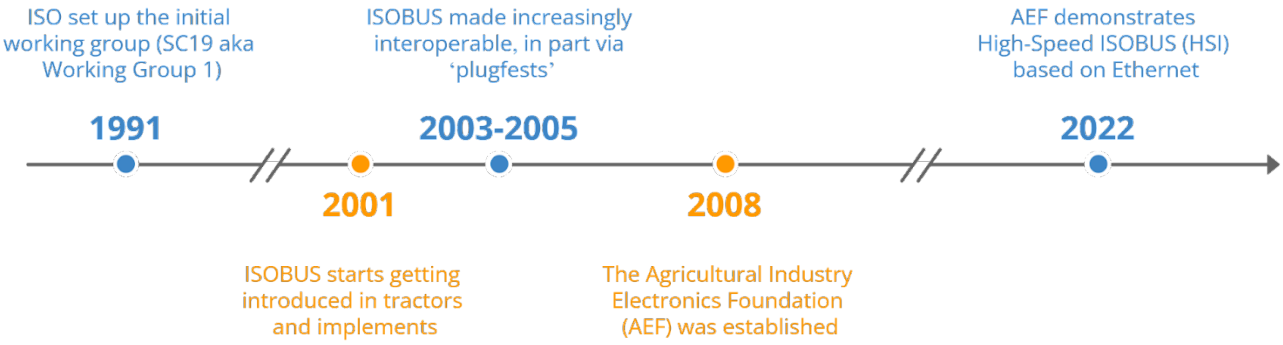
Today, ISOBUS is implemented in most modern tractors and implements (like balers, sprayers, fertilizers, seed drills) and enables standardised communication between them.

For example, a tractor may use J1939 internally. The Tractor ECU (TECU) then serves as a gateway to the tractor's ISOBUS network. Similarly, a sprayer may use e.g. J1939/CANopen internally, while an ISOBUS gateway facilitates the communication with the tractor. Assuming ISOBUS is properly implemented in both tractor and implement, the integration is seamless.

With certified ISOBUS solutions, end users avoid cluttering cabins with implement-specific terminals - and instead control any implement from a single universal terminal.

ISOBUS history and AEF

- 1991: ISO set up the initial working group (SC19 aka Working Group 1)
- 2001: ISOBUS started getting introduced in tractors and implements
- 2003-2005: ISOBUS made increasingly interoperable, in part via 'plugfests'
- 2008: The Agricultural Industry Electronics Foundation (AEF) was established
- 2022: AEF demonstrates High-Speed ISOBUS (HSI) based on Ethernet



AEF and ISOBUS certification

When ISOBUS solutions initially started rolling out in the early nineties, most were early-stage solutions. From 2001, ISOBUS experienced an adoption 'breakthrough' with tens of thousands of new ISOBUS tractors/implements sold globally.

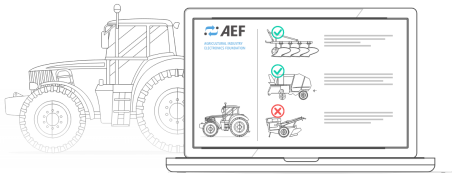
However, in practice, the standard was complex and interpreted differently by manufacturers - causing incompatibility issues in the field (and a general loss of confidence in the ISOBUS promise). In response, AEF was established in 2008.

AEF

AGRICULTURAL INDUSTRY ELECTRONICS FOUNDATION

AEF members

AEF originally comprised seven agriculture equipment manufacturers (John Deere, CNH, Claas, AGCO, Kverneland Group, Grimme, Pottinger) and two associations (Association of Equipment Manufacturers, AEM, and German Engineering Federation, VDMA). Today, FederUnacoma serves as the 3rd core member association, while there are now 8 manufacturer core members including KUHN. In total, more than 200 companies/institutions are now members of AEF.



AEF has a set of core objectives

- Define guidelines for implementations of ISOBUS
- Coordinate improvements to ISOBUS incl. certification tests
- Coordinate collaboration in the agriculture industry
- Organize certification support, training and consulting

Further, AEF has developed a number of tools and products:

- AEF Conformance Test: Used by AEF accredited test labs to certify ISOBUS components
- AEF Database: Used by dealers/users globally to assess compatibility and conformance of ISOBUS components

Practical relevance

The conformance testing and related database are vital in ensuring a consistent, standardized implementation of ISOBUS - allowing end users to lookup components across brands and manufacturers to ensure compatibility prior to purchase. This ensures that the original promise of a 'plug & play' cross-brand experience with ISOBUS vehicles and implements can be realized in practice.

ISOBUS OSI model & standards

Next, we will briefly consider the 7 layer OSI model for ISOBUS including the link to the 14 parts of ISO 11783 standard.

Note how we also reference CAN bus (ISO 11898) for the physical layer and data link layers, to reflect the role CAN plays as the basis for ISOBUS.

While not shown in the illustration, SAE J1939 also plays a role as reference for some of the layers/standard parts - for example, J1939-71 provides the definition for most powertrain messages as described in ISO 11783-8.

Below we briefly recap the 14 parts of the ISO 11783 standard.

7 layer OSI model	**ISOBUS (ISO 11783)**
Application	ISO 11783-7/8/9/10/12/13/14
Presentation	ISO 11783-6/7/11
Session	ISO 11783-6/7/10/14
Transport	ISO 11783-3/5
Network	ISO 11783-4/5
Data link	ISO 11783-3 (ISO 11898-1)
Physical	ISO 11783-2 (ISO 11898-2)

ISO 11783-1: General standard for mobile data communication

This serves as an overview of the entire ISO 11783 standard collection, providing the basis for a standardized serial communication network between forestry/agriculture tractors and implements. It describes the OSI model layers and references the other standards.

ISO 11783-2: Physical layer

This standard specifies the physical layer of the network. Since ISOBUS is based on CAN, this standard is closely linked to ISO 11898-2 describing the CAN bus physical layer. In addition, ISO 11783-2 describes ISOBUS specific connectors like the ISOBUS Breakaway Connector (IBBC).

ISO 11783-3: Data link layer

This standard describes the ISOBUS message formats. Specifically, it dictates the use of extended CAN IDs and the use of J1939-style Parameter Group Numbers (PGN). The standard also specifies a transport protocol for the communication of multiframe ISOBUS messages, i.e. data that exceeds the 8 bytes of Classical CAN frames.

ISO 11783-4: Network layer

Here, the use of 'network interconnect units' are described for connecting two separate networks with differing architectures. This gateway ensures both electrical and message isolation of each network.

ISO 11783-5: Network management

This defines the process for determining source addresses (SA) and resolving address conflicts. Source addresses can be preset or dynamically claimed by each controller during power up.

ISO 11783-6: Virtual terminal

This standard describes the virtual terminal as an ECU that provides the tractor operator with an interface for controlling the tractor and/or implement(s). The control should be exerted through the use of the standardized ISO 11783 messages. The VT is a core feature of ISOBUS.

ISO 11783-7: Implement messages application layer

This standard describes standardized application messages for the ISOBUS implement. This also serves as the basis for most of the messages in our ISOBUS DBC file. To some extent, this can be compared to J1939-71, which serves as the basis for most of the J1939 PGN and SPN decoding information. In the case of ISOBUS, messages described include e.g. wheel/ground based speed and direction, key switch state, aux valves, lights and more.

ISO 11783-8: Powertrain messages

This describes powertrain messages used in the tractor network implementation. Effectively, it references the SAE J1939-71 standard, stating that any overlapping messages should be implemented as per J1939-71.

ISO 11783-9: Tractor ECU

When an implement is connected to a tractor, it is critical to ensure consistent cross-communication. Here, the tractor ECU should allow the implement to request information from ECUs on the tractor network in accordance with ISO 11783-7, as well as respond to implement commands. This also describes details on safe mode operation.

ISO 11783-10: Task controller and management information system data interchange

A 'farm management computer' may be available on the implement network. This enables the operator (typically a farmer) to define tasks/actions for the implement to perform. The 'task controller' receives these tasks via an interface from the computer. The tasks are scheduled by the controller, which ensures execution in the implement through

control functions. The controller also receives status information from the implement ECUs, which is parsed back to the farm management computer. The standard defines the operation of the task controller, including the standardized messages used for the control functions.

ISO 11783-11: Mobile data element dictionary

This is closely linked to 11783-10, providing a data dictionary to define the data elements used in the task controller messages.

ISO 11783-12: Diagnostics services

This specifies messages used for diagnostics to identify faults and issues on the network. It can be compared to J1939-73

ISO 11783-13: File server

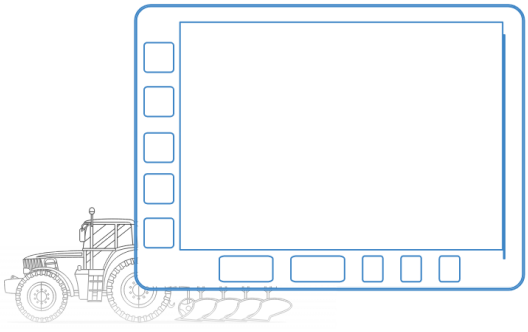
This defines an ECU that provides data storage on the network (e.g. for multimedia), as well as a set of commands for use by read/write data to the file server.

ISO 11783-14: Sequence control

This more recent standard (first issued in 2013) details how to perform sequence control within the ISO 11783 network. Here, an operator can record sequences of control functions for later on-demand replay.

ISOBUS Functionalities

AEF refers to a number of ISO 11783 based 'products' as ISOBUS Functionalities. Effectively, these can be sold to end users as a separate modules on the ISOBUS. In this section we briefly detail each of the AEF ISOBUS Functionalities:



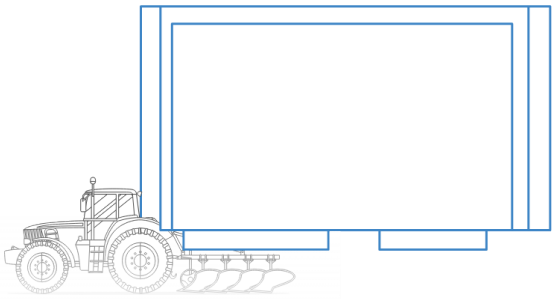
#1 Virtual Terminal (VT) / Universal Terminal (UT)

This refers to the Virtual Terminal specified in ISO 11783-6. The VT is sometimes referred to as an Universal Terminal (UT) because of the capability it provides: To allow the end user to operate different implements (even across brands/manufacturers) via a single terminal. We'll refer to it as the VT throughout.

The VT is a core part of what makes ISOBUS visually attractive: Instead of a messy tractor cabin containing a multitude of implement-specific displays, everything can now be handled through a single display - across all ISOBUS compatible implements.

In practice, the VT is initialized through a sequence of CAN messages being sent between the VT and the implement. Here, the VT informs the implement of

its specifications, allowing the implement to send its user interface to the VT for graphical visualization. Once initialized, the operator can perform various implement control tasks through the VT.

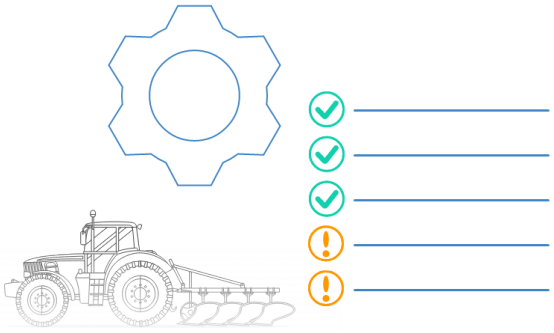


#2 Tractor ECU (TECU)

The Tractor ECU (TECU) serves as a gateway between the tractor CAN bus (typically J1939) and the ISOBUS. The TECU provides selective info to the implement like speed, RPM etc. from the tractor network via the ISOBUS.

In more recent versions of Tractor ECUs, the implement can take over full control of the tractor - incl. steering, speed, braking, transmission etc. This enables increasingly autonomous solutions where tractor and implement operate seamlessly as a single system.

Note also that the implement often has a similar ECU, sometimes referred to as the Implement ECU (I-ECU). This serves as a gateway between the implement's internal CAN bus and the ISOBUS.

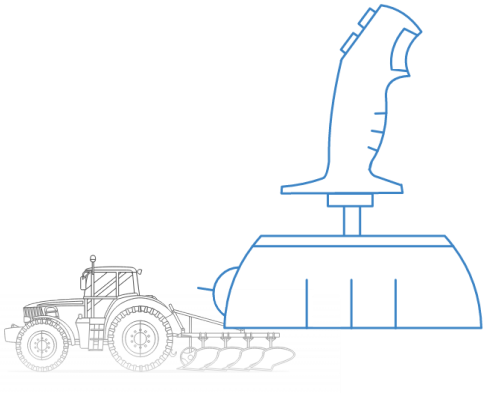


#3 Task Controller (TC)

The Task Controller is a piece of software that records and provides information on operations and assists in planning. It takes inputs from the operator via an interface and serves to schedule tasks for the implement, executing them through control functions. In particular, it allows for automated and thus more fine-grained control over the implement - key in e.g. precision farming.

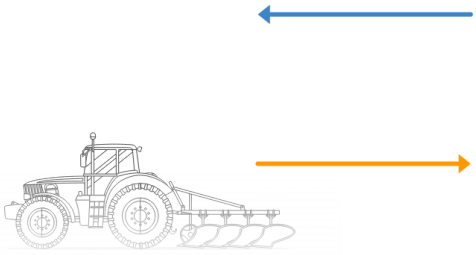
The Task Controller consists of three sub protocols:

1. Basic (TC-BAS): This is used for tracking data totals (e.g. how much product has been applied), with info provided by the implement. This makes it easy to e.g. extract documentat ion of the work done via the operator interface
2. Section (TC-SEC): This enables automated switching of sections in e.g. a sprayer based on for example GPS position, allowing for more precise use of implement functionality
3. Geo (TC-GEO): This is used for GPS-based task automation, such as executing geo-based seeding control. For example, this is useful if certain areas in a field need customized handling



#4 Auxiliary Control (AUX)

This ISOBUS Functionality often reflects a joystick or similar control element, which enables more fine-grained control over implement functionality. Note that both a new/old version exists which are not cross-compatible (AUX-N/AUX-O).

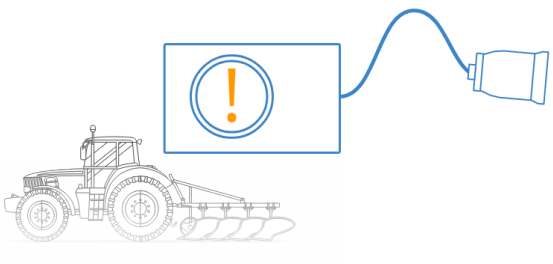


#5 Tractor Implement Management (TIM)

As mentioned in the TECU section, there is increasingly a demand for bi-directional control between tractor and implement, meaning in particular that the implement should eventually be able to control the tractor in some use cases. This would improve productivity and reduce operator fatigue.

#6 Logging of device values (LOG)

This is focused on logging data from the tractor/implement, independently of the task being executed. To some extent, it can be compared to retrofitting a CAN data logger to record all messages being communicated on the CAN buses. However, in the case of LOG, the aim is to allow for the data to be exported in the ISO XML format (similar to what is supported for Task Controller data).



#7 ISOBUS Shortcut Button (ISB)

This enables deactivation of selective implement functions. This can be relevant in use cases where multiple implements are being controlled from a single tractor Virtual Terminal. Here, one implement might be in the foreground, while the user needs to quickly deactivate functionality in another implement.

ISOBUS vs SAE J1939

A common question is how ISO 11783 (ISOBUS) relates to the SAE J1939 protocol.

In short, ISOBUS was originally derived from J1939 and a core goal of ISO 11783 is to remain compatible with J1939 through close alignment with SAE. In a sense, one can say that ISOBUS is today harmonized with J1939. This is useful as many tractor networks will be based on SAE J1939, meaning that Tractor ECUs (TECU) often need to serve as gateways between ISOBUS and J1939.

J1939 vs ISOBUS (ISO 11783)

Topic	J1939	ISOBUS
Application layer	PGNs and SPNs	Virtual Terminal, Task Controller, File Server ...
Address claiming	Optional	Mandatory
Fast packet TP	N/A	Used for GNSS (NMEA 2000)
Extended TP	N/A	Used for e.g. VT, File Server
Physical layer/connector	See J1939-1X	See ISO 11783-2 (different)
Conformance test	Optional	Mandatory in practice via AEF

Further, various J1939 standards serve as direct references for the ISOBUS standards. For example, J1939-71 defines powertrain messages and serves as a reference for ISO 11783-8. Both ISOBUS and J1939 also rely on CAN bus (CAN 2.0B) as the lower layer for communication and both leverage the concept of 18 bit PGNs to identify messages.

However, J1939 is mainly used in 'closed systems' and therefore ISOBUS has been extended beyond J1939 to serve the need for plug & play implements. It is therefore important to note that ISOBUS is not equal to J1939. We outline a number of key differences in the table. For details see below:

ISO 11783 vs. J1939 - details

In particular, note that the ISOBUS application layer includes a number of additional components, such as the Virtual Terminal and Task Controller.

ISOBUS also enables alternative transport protocol methods like Fast Packets e.g. for GNSS data (also known from NMEA 2000) and Extended Transport Protocol (e.g. for sending data to the VT or File Server). ISOBUS also introduces a number of Data Dictionary Identifiers, which are similar to SPNs - though they are generally transmitted with 1 SPN per message, rather than multiple. Due to the need for implement interoperability, dynamic address claiming is mandatory in ISOBUS, whereas it's often skipped in J1939 (due to the closed system logic). Further, ISOBUS introduces a number of alternative physical connectors vs. J1939 as we'll detail shortly.

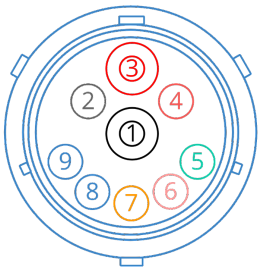
The ISOBUS connectors

ISO 11783-2 describes a number of physical connectors used in tractors and implements.

1 TBC power
2 CAN H
3 TBC return path
4 CAN L

#1 Bus extension connector

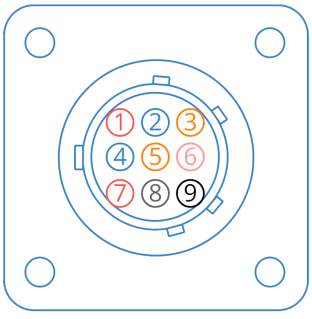
This connector enables the extension of the ISOBUS signal lines from the implement within the tractor. This is e.g. used for adding additional devices such as Virtual Terminals. The connector will be located in the tractor cabin on the right side of the operator's seat.



1 Ground (chassis)
2 ECU ground
3 Power
4 ECU power
5 TBC connect/remove
6 TBC power
7 TBC return path
8 CAN H
9 CAN L

#2 Implement Bus Breakaway Connector

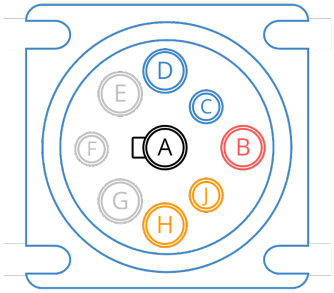
The IBBC enables the connection of implements to the tractor. The connector is located at the rear of the tractor. In some cases an optional IBBC may also be installed on the front of the tractor for use with implements that are front mounted. The IBBC must have a cap for covering it against dust and the weather when no implement is connected.



1 ECU power
2 CAN 1 L (input)
3 CAN 2 L (output)
4 CAN 1 H (input)
5 CAN 2 H (output)
6 TBC power
7 ECU power
8 TBC ground
9 ECU ground

#3 In-cab connector (LBS)

The tractor cabin may include an optional in-cab connector for establishing a connection to the ISO 11783 network. This is sometimes referred to as LBS.



A Ground
B Battery power
C CAN 1 H (tractor)
D CAN 1 L (tractor)
E Not specified
F Not specified
G Not specified
H CAN 2 H (implement)
J CAN 2 L (implement)

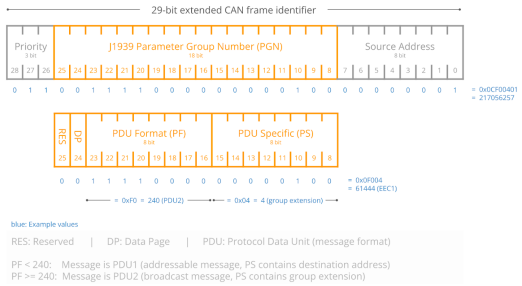
#4 Diagnostic connector

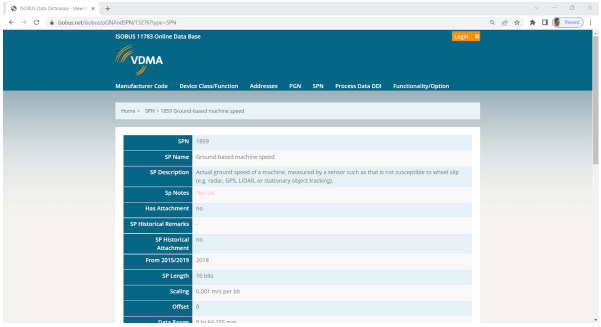
The diagnostic connector is located in the tractor cabin in a location that is easily accessible. As we'll detail shortly, the diagnostic connector can be a good option for connecting external data logging devices for diagnostics - in particular if you need access to both the J1939 and ISOBUS networks in parallel.

ISOBUS PGN and SPN [+ DBC]

ISO 11783 uses the concept of Parameter Group Numbers (PGN) and Suspect Parameter Numbers (SPN) from the SAE J1939 protocol. This means that messages are identified based on 18-bit PGNs, which can be extracted from the full 29-bit CAN identifiers. In turn, signals or parameters are referred to as SPNs and will be 'packaged' in the data payload of each message.

We will not cover the PGN structure in detail as it is already covered in our SAE J1939 intro. However, we will provide a few practical comments on ISOBUS PGNs and SPNs.





ISOBUS PGN and SPN database

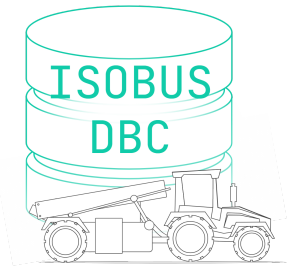
The ISO 11783-7 standard specifies PGN and SPN details unique to ISOBUS. The decoding details are publicly available from the VDMA ISOBUS Data Dictionary. Signals include:

- Front/rear hitch pitch/roll/yaw angle
- Machine selected speed
- PTO information - rear shaft speed
- Engine speed, wheel based speed
- Tractor lights

The public database includes decoding rules for ~800 signals, of which ~650 can be directly transformed to a format useful in e.g. telematics and data logging. As explained below, we provide a DBC version of these ~650 ISO 11783 message decoding rules.
Note that the ISOBUS network may also contain OEM proprietary data that cannot be decoded using the public information.

ISOBUS vs. J1939 vs. NMEA 2000 [+ DBC files]

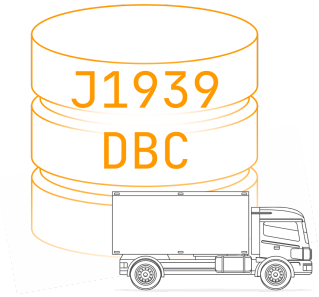
If you record data from a tractor/implement, multiple protocols (and hence DBC files) may be relevant.



ISOBUS data (implement/tractor)

Naturally, the ISOBUS network will contain data related to the communication between tractor and implement. This information will often be decodable using the ISO 11783-7 decoding rules mentioned above. To enable quick decoding of the standard ISOBUS data, you can use our ISOBUS DBC file.

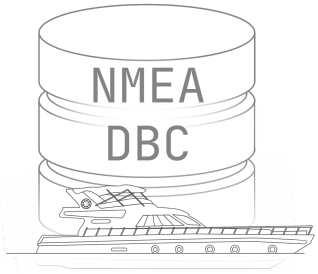
learn more



J1939 data (tractor)

If you record data directly from the tractor's CAN bus, it will most likely be J1939 data. Similarly, if you log data from the ISOBUS network, some of the data from the Tractor ECU will typically be J1939. In either case, the J1939 data requires the use of decoding rules from the J1939 Digital Annex. For this purpose, you can use our J1939 DBC file.

learn more



NMEA 2000 data (GNSS)

Often, the ISOBUS network contains NMEA 2000 encoded GNSS information. The NMEA 2000 data may be broadcast as a combination of single-frames and NMEA 2000 Fast Packet TP frames. To decode the data, you can use our NMEA 2000 DBC file.

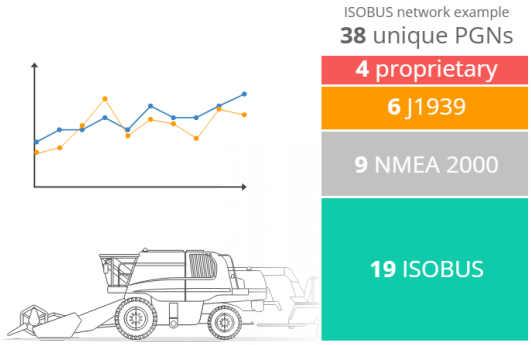
learn more

OEM proprietary data (tractor/implement)

Of course, there will almost always be some messages that are proprietary regardless of the protocol. These are only known to the OEM (Original Equipment Manufacturer) and cannot be decoded using the standard DBC files. Sometimes these messages follow one of the above standards, though in other cases they may use another standard entirely (e.g. CANopen).

Example: Decoding data from an ISOBUS network

To exemplify the division that you'll often see in agriculture CAN bus data, consider a log file from a Fendt tractor with an implement from another manufacturer. In this case, a CANedge2 has been used to record data from the ISOBUS network. Through this, 337 unique CAN IDs have been recorded, corresponding to 38 unique PGNs.



The PGNs can be split by protocol as follows:

- ISOBUS DBC: 19 PGNs (~50%)
- NMEA 2000 DBC: 9 PGNs (~24%)
- J1939 DBC: 6 PGNs (~16%)
- Proprietary: 4 PGNs (~10%)

To get this ISOBUS sample data, get our J1939 data pack.

> **Note on J1939 PGNs**
>
> Note here that the connection of the CANedge2 was made on the ISOBUS network - not directly onto the J1939 network. As a result, there are very few decodable J1939 PGNs, since they only reflect the J1939 PGNs repeated by the Tractor ECU (TECU) onto the ISOBUS. Had the CANedge2 been connected to the J1939 network directly, there would be a significantly higher amount of CAN IDs overall - including many more J1939 PGNs.

Logging tractor/implement data

In this section we outline how you can record and process data from tractors/implements.

#1 How to choose a CAN bus data logger

If you need to record your data offline, you can use the standalone CANedge1 to log 2 x CAN to an SD card. The device can be left in the tractor to record data for weeks or months. To collect the data, simply extract the SD card.

Alternatively, you can use the CANedge2 (WiFi) or CANedge3 (3G/4G) to automatically upload data to your own server (self hosted or cloud). This enables e.g. large-scale data collection, automated reporting or predictive maintenance.

#2 How to connect to the CAN bus

Once you've selected a CAN bus data logger, you'll need to identify how to interface with the CAN networks. We generally recommend to use the diagnostic connector if possible. As outlined previously, this enables access to both the J1939 network of the tractor - and the ISOBUS network.

You can use our J1939-DB9/DB9 (H + J) adapter cable, which perfectly matches the diagnostic connector pinout and enables you to log both the J1939 and ISOBUS networks separately into the same log file on the CANedge.

Further details

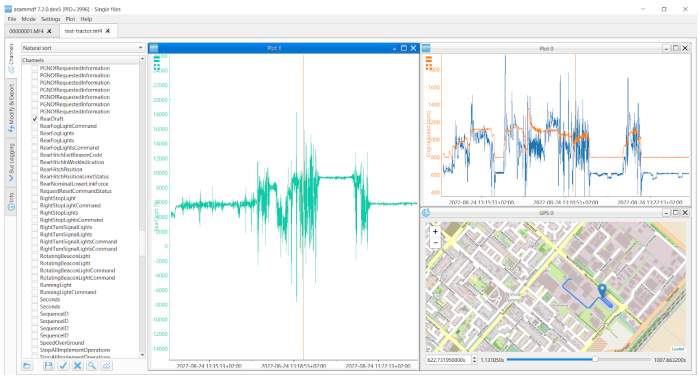
Alternatively, you can also use our regular J1939-DB9 adapter if you only need to access the J1939 network. Often many of the messages will be repeated across networks, so for many use cases this may suffice.

Note that if you aim to connect directly to an implement diagnostic connector (rather than through the tractor cabin), the CAN 1 pins (C + D) will typically give access to the implement's internal CAN bus - while H + J will still give access to the ISOBUS network.

#3 How to process the CAN data

To process the raw tractor/implement data, you'll need a suitable CAN software tool. Specifically, you'll load the log files and related DBC files to extract the physical values.

The CANedge software tools enable various forms of analysis. For example, you can use the MF4 converters to quickly convert the log files to popular file formats, for use in software like Vector CANalyzer or PEAK PCAN-Explorer. Or, you can use the asammdf GUI to DBC decode the data and create visual plots.



You can also perform advanced data processing via the Python API - ideal if you need to perform statistical analysis, automated reporting or database integration.

The Python API can also be used to integrate the data with dashboards - more below.

Example: Tractor dashboard with ISOBUS, J1939 and NMEA 2000 data

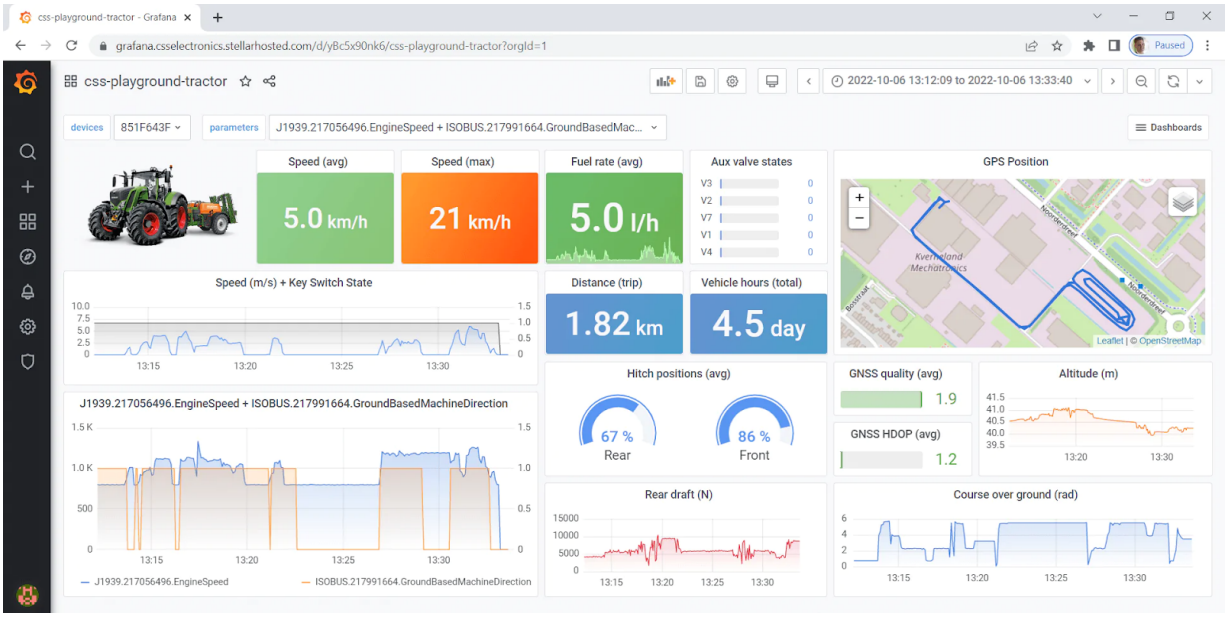
To illustrate the role of ISOBUS, J1939 and NMEA 2000 in tractor/implement data logging, check out our Grafana dashboard playground. Here, each signal is prefixed with the protocol from which it originates.

See also our telematics dashboard intro to learn more about Grafana dashboards.

Details on the recorded data

The log file was recorded using a CANedge2 connected to the implement diagnostic connector. Here, the CANedge2 specifically recorded data from the ISOBUS network. In other words, the log file does not include the tractor's internal J1939 data, except for the signals that are repeated onto the ISOBUS network.

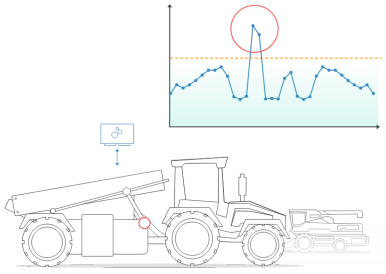
To decode the data, we used both the J1939 DBC, ISOBUS DBC and NMEA 2000 DBC. We also leveraged the ISO TP functionality of the CANedge Python API to extract some of the NMEA 2000 Fast Packet data. Note that the proprietary OEM specific data is not included in the playground as this is not part of the standardized DBC files.



[open playground](#) [get sample data](#)

Data logging use case examples

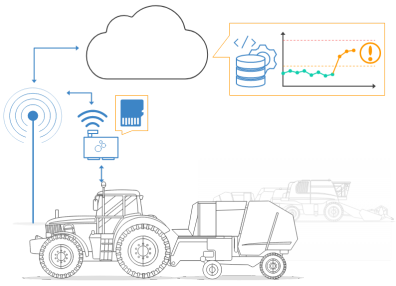
Here we outline a number of classic data logging use cases within agriculture vehicles.



Tractor black box data logger

Need to capture intermittent issues - or handle warranty disputes?

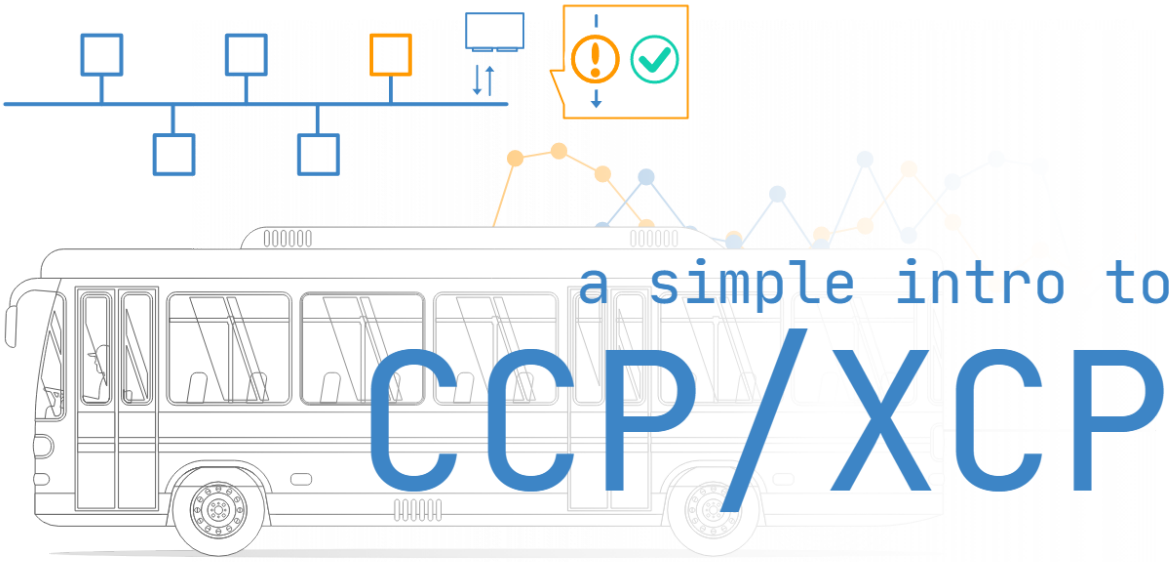
By deploying a CAN bus data logger permanently in your tractors, you will gain access to a rolling window of insight into the full scope of your CAN bus networks. This makes it an ideal solution for capturing rare issues to speed up diagnostics - or to handle cases of warranty disputes between the OEM and end user.



Monitoring operational agriculture data remotely

Need to remotely collect operational data for post analysis?

As an operator or systems integrator, you may need to collect data across a fleet of agriculture vehicles in the field. Here, the CANedge2 can be deployed to enable logging of the raw CAN data. The data can be uploaded when the vehicles return to a stationary WiFi network - or you can deploy each CANedge2 with a 3G/4G hotspot for continuous data collection from the field. Using the CANedge Python API, it's easy to perform big data processing, advanced statistical analyses and reports.



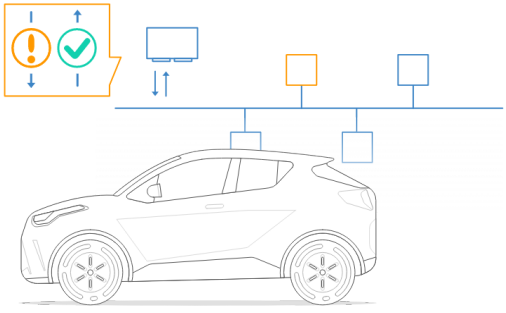
CCP / XCP on CAN Explained - A Simple Intro

Need a simple intro to CCP/XCP on CAN bus?

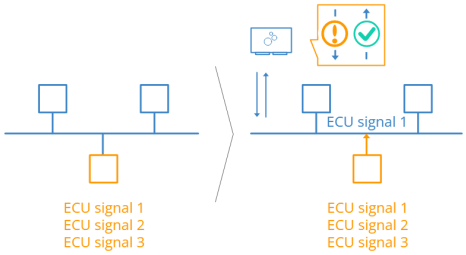
In this practical tutorial, we introduce the basics of the CAN Calibration Protocol (CCP) and the Universal Measurement and Calibration Protocol (XCP) on CAN. In particular, we'll focus on the CCP/XCP frame structures, trace examples and A2L files. We also cover practical ECU data logging via polling/DAQ - and how to decode the data.

What is CCP/XCP?

The CAN Calibration Protocol (CCP) is an interface that enables read/write access to an Electronic Control Unit (ECU). It enables calibration, data measurement, flashing and more. The Universal Measurement and Calibration Protocol (XCP) is the successor to CCP with various improvements - including support for more transport layers such as Ethernet, FlexRay and SxL.



The CCP/XCP protocols have extensive overlaps, but also important differences. To avoid confusion, we will first focus on covering the CCP protocol - and subsequently go through XCP on CAN with explicit clarification on important differences.



To understand the motivation for CCP/XCP, let's revisit our simple intro to CAN bus. As explained here, CAN enables communication of data between different ECUs in a vehicle/machine. Inputs and outputs of every ECU will be broadcast on the CAN bus. However, the inner workings of an ECU is a black box. Here, CCP/XCP provides direct access to the inner workings of an ECU. This lets you request high-frequency parameter data that may otherwise only be known to the ECU. Further, it also lets you modify the ECU algorithms and variables, making it easy to test and calibrate ECUs. Importantly, CCP/XCP enables these interfaces in a standardized way across ECU manufacturers.

History of CCP/XCP

The CAN Calibration Protocol (CCP) was originally developed by a calibration systems company, Helmut Kleinknecht. Within a few years, it was improved by a working group, ASAP (Arbeitskreis zur Standardisierung von Applikationssystemen) that included Audi, BMW, VW and others. Later ASAP was renamed to ASAM (Association for Standardization of Automation and Measuring Systems).

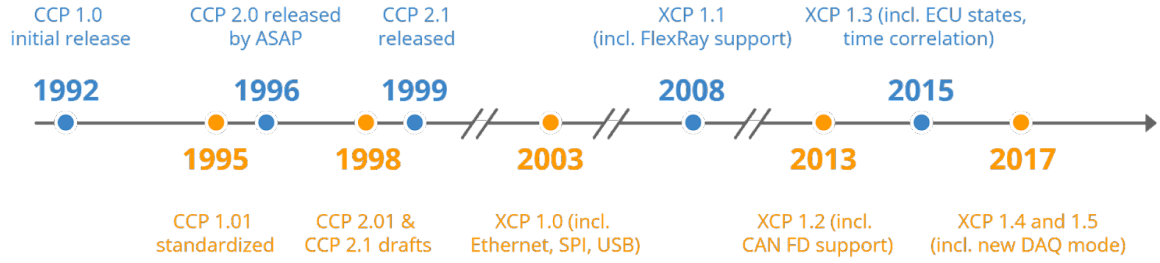
CCP
CAN Calibration Protocol

XCP
Universal Measurement and Calibration Protocol

Below are the key milestones:

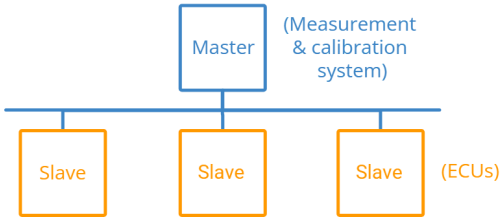
- 1992: CCP 1.0 initial release by Helmut Kleinknecht
- 1995: CCP 1.01 standardized by ASAP
- 1996: CCP 2.0 released by ASAP
- 1998: Drafts of CCP 2.01 and CCP 2.1 were prepared
- 1999: CCP 2.1 was released in February
- 2003: XCP 1.0 incl. support for CAN, Ethernet, SPI, USB
- 2008: XCP 1.1 incl. support for FlexRay
- 2013: XCP 1.2 incl. ECU description file updates + CAN FD
- 2015: XCP 1.3 incl. ECU states, bypass handling, time correlation
- 2017: XCP 1.4 incl. Improvements and new DAQ mode
- 2017: XCP 1.5 incl. software debugging without a debug adapter

Today, XCP (aka ASAM MCD-1 XCP) is the successor to CCP (aka ASAM MCD-1 CCP). However, in practice many ECUs still use CCP, which makes it relevant to understand both protocols and key differences.



Master-slave architecture

The CCP/XCP protocol is based on a single-master/multi-slave concept. An external measurement & calibration tool (e.g. a PC/device/data logger) serves as the master and is able to read/write from one or more ECUs aka slaves.

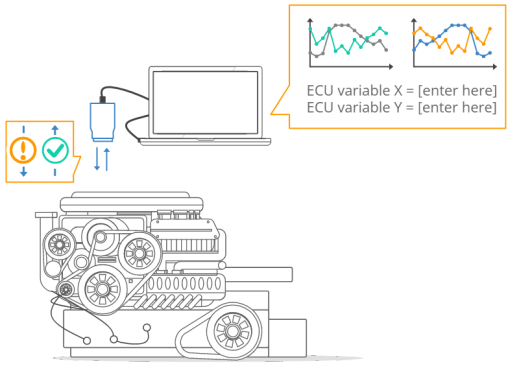


The interface between the master and slave is called ASAP1 or ASAM MCD-1. The CCP/XCP standard also describes the ASAP2 or ASAM MCD-2 MC interface between the master and an ECU description file. In practice, this database describes all relevant information about an ECU in a standardized file format called A2L (ECU Description Files) - which we will cover shortly.

CCP/XCP use cases

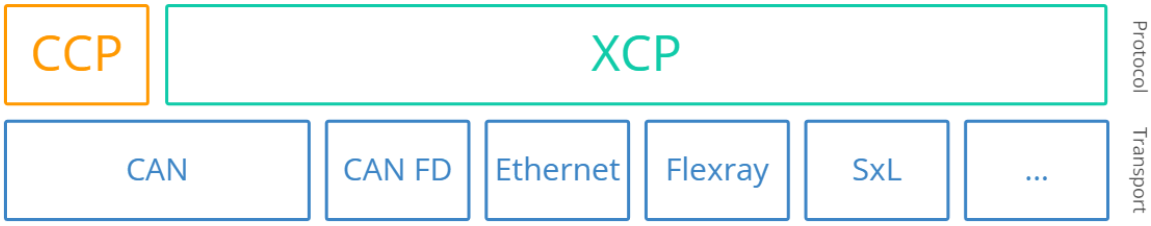
The CCP/XCP protocol enables multiple use cases:

- Plug & play ECU measurement and calibration
- Recording of ECU data at microsecond resolution
- Access to data internal to the ECU (not broadcast on CAN)
- Measurement via polling or based on events (time, triggers)
- Real-time calibration/adjustment of ECU algorithm variables
- Flash programming of ECUs
- Optional authentication for secure access
- Mainly for OEM development - rarely used post production



Major changes in XCP vs. CCP

- XCP adds support for CAN FD, Ethernet, FlexRay, SxL and more
- Less interpretation - more consistent implementations
- New 'stimulation' (STIM) mode for bypassing ECU calculations
- Predefined/dynamic DAQ lists for efficient communication
- Support for synchronous use of different DAQ modes
- ECU auto detection (master can poll slaves for information)
- More precise data acquisition by measuring ECU timestamps
- More data throughput via new optional commands

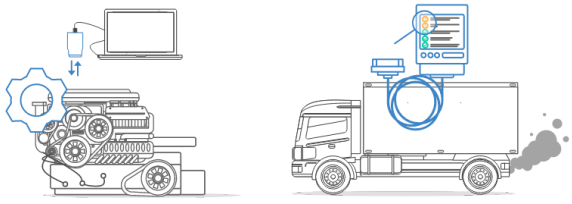


CCP/XCP vs. UDS

Before we deep-dive further on CCP/XCP, it can be useful to understand the role of these communication protocols vs. a slightly similar protocol, Unified Diagnostic Services (UDS).

As evident from the illustration, CCP/XCP is designed specifically for pre-launch measurement and calibration by the OEMs. Typically, the CCP/XCP access to ECUs is disabled once vehicles are ready for launch. In contrast, UDS is typically not available in early stage prototype development and only later added - also being available for communication post launch.

UDS focuses on diagnostics, whereas the diagnostics capabilities of CCP/XCP are light-weight. UDS may also be used by e.g. field technicians and in OEM specific scan tools.



CCP/XCP

- Mainly used by OEMs pre-launch
- Measurement & calibration
- Read/write by ECU address
- Efficient & advanced DAQ
- OEM specific/proprietary
- A2L files are core

UDS

- Also used in field
- Diagnostics/field service
- Read/write by identifier
- Diagnostic trouble codes
- OEM specific/proprietary
- Basis for e.g. WWH-OBD

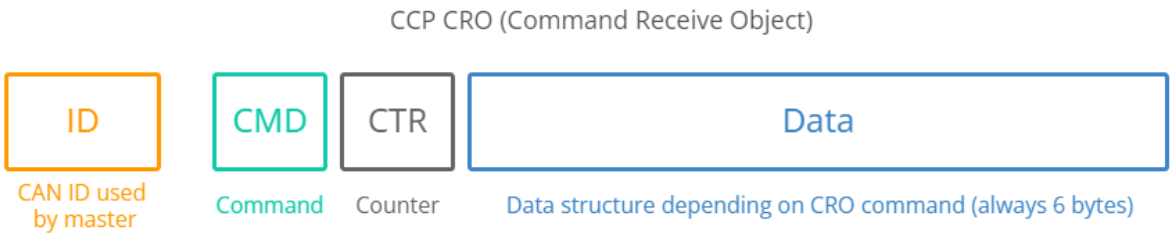
The two protocols do share similarities, though - e.g. in their support for polling/cyclic data acquisition, ECU flashing and read/write access by address (CCP/XCP) or ID (UDS).

CCP message types

To understand how CCP communication works we will first review the CCP message types. Overall, CCP communication is done through a request/response logic - similar to what is used in OBD2, UDS etc. This communication consists of two types of messages: The Command Receive Object (CRO) and the Data Transmission Object (DTO).

CRO - Command Receive Object

The Command Receive Object (CRO) is a CAN frame sent by the master to an ECU. The 1st byte of the data payload is the command byte (CMD). This controls what command is being issued by the master to the ECU as per the table overview.



To track the commands issued, the 2nd byte is a counter (CTR). It is indented by +1 for every command sent by the master and is mirrored in the response from the slave ECU. The remaining 6 bytes depend on the command.

Overview of CRO CMD values

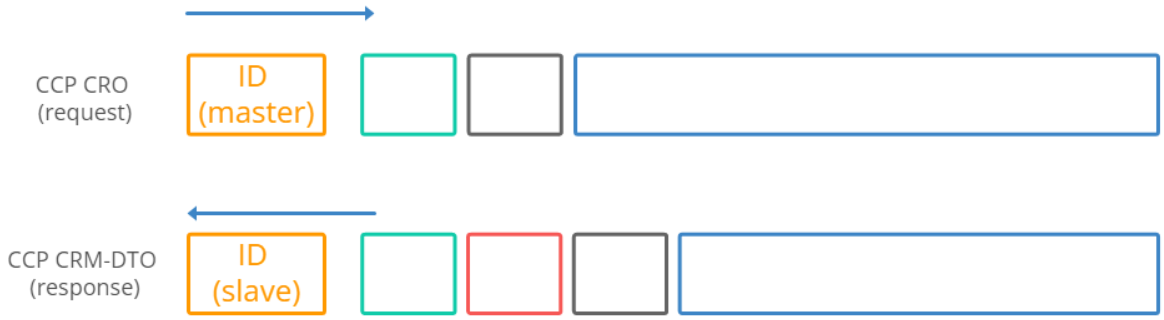
For a list of CRO commands used in CCP, see below table.

CCP CRO (Command Receive Object) - CMD (Command) table

CMD code	Command	Type	Optional?
0x01	CONNECT	Initialization	
0x1B	GET_CCP_VERSION	Initialization	
0x17	EXCHANGE_ID	Initialization	
0x12	GET_SEED	Authorization	Optional
0x13	UNLOCK	Authorization	Optional
0x02	SET_MTA		
0x03	DNLOAD		
0x23	DNLOAD_6		Optional
0x04	UPLOAD		
0x0F	SHORT_UP	Polling measurement	Optional
0x11	SELECT_CAL_PAGE		Optional
0x14	GET_DAQ_SIZE	DAQ measurement	
0x15	SET_DAQ_PTR	DAQ measurement	
0x16	WRITE_DAQ	DAQ measurement	
0x06	START_STOP	DAQ measurement	
0x07	DISCONNECT	DAQ measurement	
0x08	START_STOP_ALL	DAQ measurement	Optional
0x0C	SET_S_STATUS		Optional
0x0D	GET_S_STATUS		Optional
0x0E	BUILD_CHECKSUM		Optional
0x10	CLEAR_MEMORY		Optional
0x18	PROGRAM		Optional
0x22	PROGRAM_6		Optional
0x19	MOVE		Optional
0x05	TEST		Optional
0x09	GET_ACTIVE_CAL_PAGE		Optional
0x20	DIAG_SERVICE		Optional
0x21	ACTION_SERVICE		Optional

CCP CAN frame identifiers

In CCP, two CAN identifiers are used: One for messages sent by the master (e.g. 0x701) and one for messages sent by the slave (e.g. 0x702). These identifiers will be specified as part of the A2L (aka ECU Description File), meaning that the master will acquire this information prior to initiating the connection.



Typically low priority IDs are used for CCP to avoid disturbing safety critical info on the bus.

Importantly, this means that the master does not target specific ECUs through various CAN IDs - but rather through a connection sequence as shown below.

Example: CCP CRO CONNECT message

Below is an example of a CRO used for initializing communication with a specific ECU:

Trace example: CCP initialization (CONNECT CMD)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.01	701	01 01 39 01 AA AA AA AA	master	CRO (CONNECT CMD)

Trace details

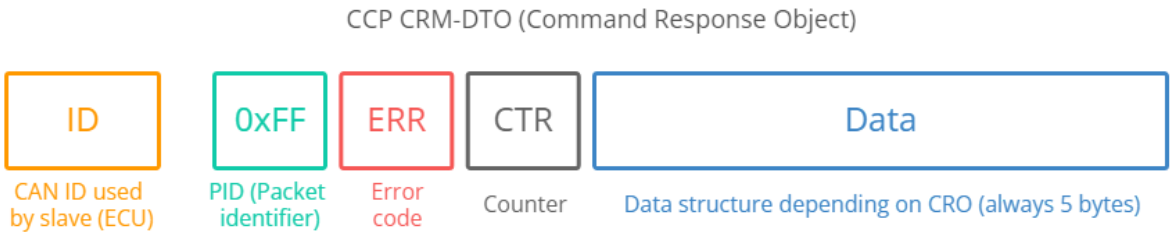
Here, the CAN ID reflects the ID used by the CCP master for communication. The 1st byte is 0x01, corresponding to the CONNECT command as evident from the previous table. The 2nd byte is the CTR value. The 3rd and 4th bytes are specific to the CONNECT command and correspond to the target ECU's station address in INTEL byte order. In other words, the above CRO is used by the master to establish a connection with ECU 0x0139. All subsequent communication will in this case be with this specific ECU until the master terminates the connection or connects to another ECU.

DTO - Data Transmission Object

The Data Transmission Object (DTO) is a CAN frame sent by the ECU to the master. Three types of DTOs exist as outlined below:

#1 CRM-DTO: Command Response Message

The CRM-DTO is sent by the ECU in response to a CRO from the master. Here, the CAN ID reflects the CAN ID used by the ECU (e.g. 0x702).



The data payload for the CRM can be broken down as follows:

- The 1st byte is the Packet Identifier (PID). For the CRM-DTO, the PID always equals 0xFF
- The 2nd byte is the error code (ERR), which can be used to e.g. inform the master of an invalid request.
- The 3rd byte is the counter (CTR), which will match the CTR value from the master's CRO
- The structure of the remaining 5 bytes depend on the original request made in the CRO

Below is the previous connection example trace, including a positive response CRM.

Trace example: CCP initialization (CONNECT CMD + ECU response)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.01	701	01 01 39 01 AA AA AA AA	master	CRO (CONNECT CMD)
1.04	702	FF 00 01 AA AA AA AA AA	slave	CRM (OK)

Trace details

The trace shows a connection to the ECU with station address 0x0139, which responds to the initialization command sent by the master. As evident, the 1st byte is 0xFF (as the message is a CRM). The 2nd byte shows that no errors occurred, while the 3rd byte matches the CTR of the CRO. With this sequence, the connection between the master and ECU 0x0139 is established.

Overview of CRM-DTO/EV-DTO ERR values

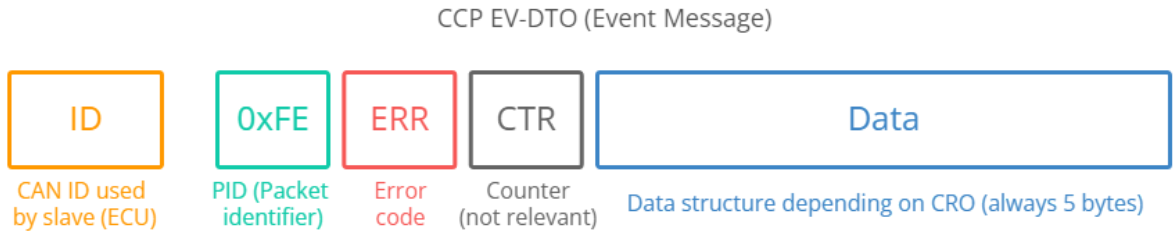
In the table below we show a set of CRM-DTO/EV-DTO error code values. Note that error codes can be sent in both CRM-DTOs (if they happen in direct response to a CRO from the master) or in EV-DTOs (if they happen asynchronously to the master's CRO commands).

CCP CRM-DTO/EV-DTO ERR code table

ERR code	Description	Error category	State transition
0x00	Acknowledge (no error)		
0x01	DAQ processor overload	C0	
0x10	Command processor busy	C1	None (wait)
0x11	DAQ processor busy	C1	None (wait)
0x12	Internal timeout	C1	None (wait)
0x18	Key request	C1	NONE
0x19	Session status request	C1	NONE
0x20	Cold start request	C2	COLD START
0x21	Cal. data init. request	C2	Cal. data init
0x22	DAQ list init. request	C2	DAQ list init
0x23	Code update request	C2	(COLD START)
0x30	Unknown command	C3	(FAULT)
0x31	Command syntax	C3	FAULT
0x32	Parameter(s) out of range	C3	FAULT
0x33	Access denied	C3	FAULT
0x34	Overload	C3	FAULT
0x35	Access locked	C3	FAULT
0x36	Resource/func. not available	C3	FAULT

#2 EV-DTO: Event Message

The EV-DTO is sent by the ECU in response to an internal event causing a status change in the ECU. This can be used to inform the master of errors that occurred since the last CRO.

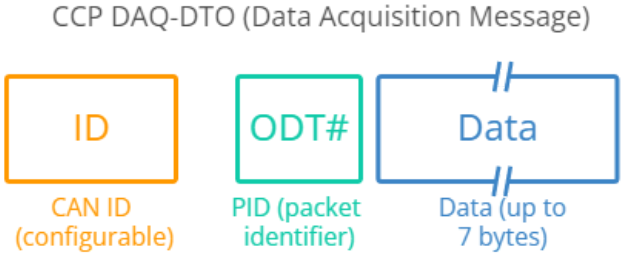


The structure of the EV-DTO is identical to the CRM-DTO from before, with the EV PID always equal to 0xFE. Note that the CTR does not have relevance for EV-DTOs.

#3 DAQ-DTO: Data Acquisition Message

The DAQ-DTO is used by the ECU to automatically send data to the master in response to a specific event (e.g. a cyclic counter, a button or similar).

In DAQ communication, the master starts out by 'configuring' the ECU for a specific measurement sequence. Once initiated, the ECU will output DAQ-DTOs without further CROs from the master.



In the DAQ-DTO the PID is a reference to an 'Object Descriptor Table' (ODT), while up to 7 bytes carry data related to the ODT. More on this shortly.

Below is an example of a DAQ-DTO message sent by an ECU to a master.

Trace example: CCP DAQ-DTO frame with 5 byte signal payload

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
12.55	712	25 A1 42 34 AF 22	slave	DAQ-DTO

Trace explained

The trace shows the DAQ-DTO communication from a slave following an initial configuration sequence. The data relates to ODT list #37 (0x25) with a payload of 5 bytes. Note how we do not pad the unused bytes (as this is not required for DAQ-DTOs). Note also how a new CAN ID is introduced for this DAQ response (more on this later).

General note on CCP messages

CRO, CRM-DTO and EV-DTO messages must have a payload length of 8. Any unused bytes are padded with arbitrary values (we use 0xAA in this intro). DAQ-DTO messages may use the actual payload size.

To recap: A master tool uses CRO messages to send various commands to a slave ECU. The slave ECU can in turn use DTO messages to respond to the master. Next we'll show how data measurement can be done via these messages.

How to record ECU data via CCP

Let's imagine that you're an engineer working at an automotive OEM. You've been tasked to extract the value of a specific parameter from an ECU over an extended period.

How do you do it? In the following sections we outline two methods: **Polling** and **DAQ**.

Data acquisition via CCP polling

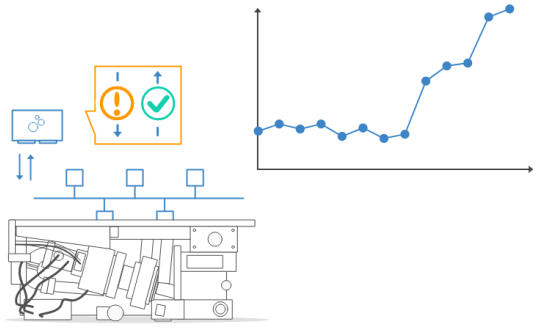
The simplest solution would be to use a CCP measurement concept called polling.
Below we illustrate how such a communication flow may look:

Trace example: CCP polling via SHORT_UP command

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.51	701	0F 02 04 00 12 34 56 78	master	CRO (SHORT_UP CMD)
1.53	702	FF 00 02 F1 2A 71 2F AA	slave	CRM (OK + DATA)

Here, the master sends a request to the slave for 0x04 bytes of data stored at the 'source address' 0x12345678. The 4th byte is 0x00, which reflects the 'address extension' for this ECU source address, which can e.g. reflect a specific memory segment.

The slave responds with the 4 bytes of data, 0xF12A712F, for this particular source address. This above flow can be used to e.g. extract real-time parameter data, such as an RPM signal. It's somewhat similar to how you can request RPM via an OBD2 request/response flow in most cars.



Effectively, 'polling' is simply a sequence of CRO/CRM-DTOs with the master using a command called SHORT_UP (short upload).

Pros & cons of CCP polling

Polling is simple to set up - simply establish an ECU connection and send the correct request message every X ms. For many practical use cases this can be a perfectly viable solution. However, polling has two significant downsides.

#1 Polling is inefficient

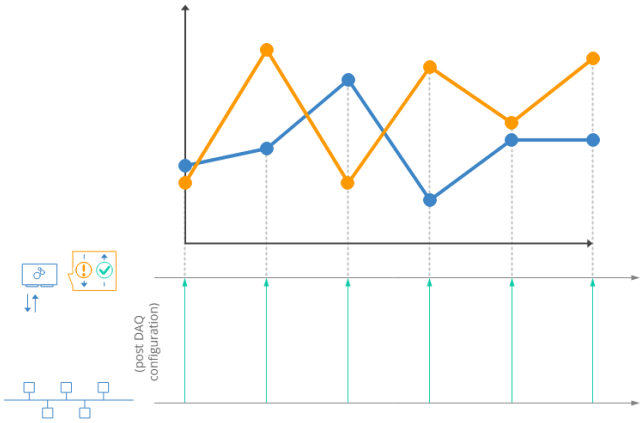
Polling requires a request message for every response. If you e.g. need to measure a 2-byte signal at 100 Hz, that increases the busload by 200 frames/second. The inefficiency is both due to the need for requesting every single response - but also the fact that response messages are forced to be 8 bytes long, with 3 bytes spent on overhead.

#2 Polling is asynchronous

When polling for multiple signals, the request/responses need to be sent sequentially with a small delay between each request. As a result, the signal observations are not time synchronized. This makes data analysis and post processing more difficult and less precise.

Data acquisition via CCP DAQ

CCP offers an alternative data measurement technique called DAQ (Synchronous Data Acquisition). DAQ is a bit more complex to initiate, but solves both the downsides of polling.



In simple terms, DAQ works as follows:

The master specifies what measurements to record from the slave, as well as what event should trigger the communication of data. For example, the master could request that signals A and B are broadcast every 10 ms, while signal C may be broadcast every 100 ms. The master can even configure slaves to broadcast signals based on events such as a button push or angle change. Once the DAQ configuration is complete, the master 'starts' the sequence and the slave now autonomously broadcasts the requested signals using DAQ-DTO messages.

This eliminates the request messages from the master. In addition, the signal data is packaged more efficiently in the DAQ-DTOs vs. the CRM-DTOs, reducing the busload. Further, with DAQ it is possible to package related signals in the same DAQ-DTO frames - ensuring that the requested signals are measured in a time synchronous manner - improving the quality of the data for analysis.

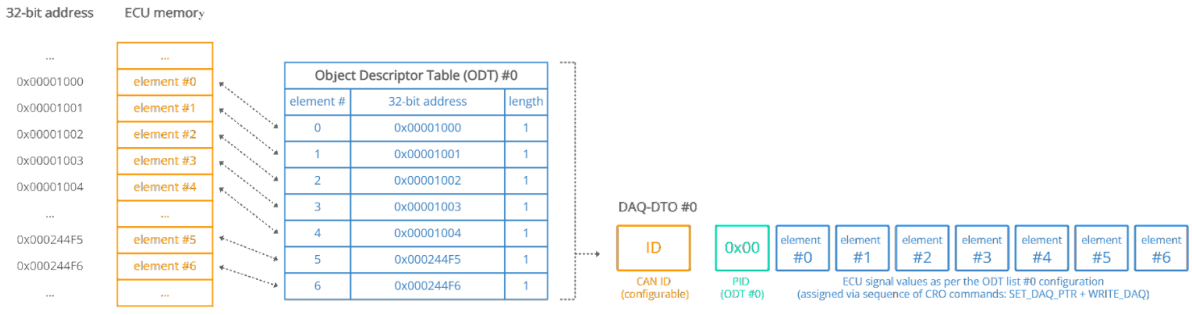
How to configure a DAQ sequence

In order to use the DAQ mode for data measurement, the master has to first configure the ECU accordingly. In practice, this process is typically handled by sophisticated GUI tools (more on this later). However, to fully understand what is going on we'll review what happens 'under the hood'.

Two concepts are key in DAQ: **Object Descriptor Tables (ODT)** and **DAQ lists**.

Object Descriptor Table

The Object Descriptor Table (ODT) is a list of references to data elements from the ECU memory. If we take the 'short upload' example used in CCP polling, the master sends a request specifying three elements: The data length (in bytes), address extension and source address. With this info, the ECU finds the data and sends it to the master. The concept of an ODT is similar: An ODT is simply a list of element references. Each entry in the ODT reflects an ECU source address - and optionally also an address extension and data length. In other words, an ODT entry contains the same info that we used to poll a signal.



ODT example & details

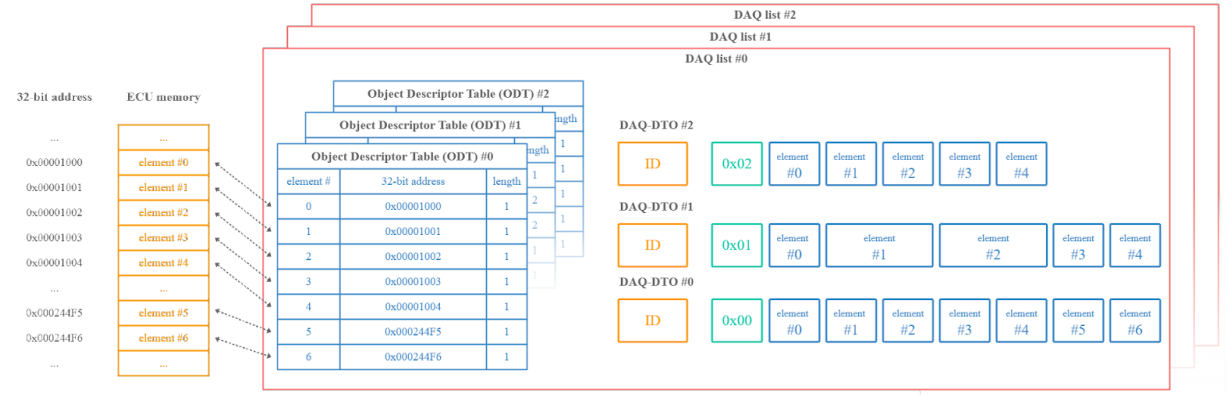
Let's take an example:
We wish to record 7 signals, each of them with a length of 1 byte. To do so, we define a new ODT #0 (PID 0x00) with 7 element entries. Element 1 refers to signal 1 with a specific ECU source address. Element 2 refers to signal 2 with another source address etc.

When we use the DAQ mode for data measurement, we can now refer to ODT #0 in order to get the ECU to provide us with time synced data on all 7 signals - within a single DAQ-DTO CAN frame. In other words: The ODT we defined describes the structure of a DAQ-DTO message, meaning that the ECU will now know how to package the 7 signal bytes in a single CAN frame - and the master knows how to extract the 7 signal bytes from that CAN frame.
Notice the impact on the busload: If we were to poll all 7 signals, it would require 14 CRO/CRM-DTO CAN frames per cycle - now it only requires 1 DAQ-DTO CAN frame.

Further, we can easily plot these 7 signals together as they share the same CAN frame timestamp - making it much easier to perform analysis. For the same reason, ODT lists are typically defined so as to group related signals together. In practice, a master will often define multiple ODT lists and assign PIDs for each of them in the range of 0x00 to 0xFD. Each ODT then defines the structure of a separate DAQ-DTO.

DAQ lists

When working with multiple ODT lists, it's useful to group them together. These groups of ODT lists are referred to as DAQ lists. For example DAQ list #0 may contain ODT #0, ODT #1 and ODT #2.



A DAQ list is characterized by the way in which the data is sampled:

- DAQ list #0 may be set up so that all DAQ-DTOs within it are sampled every 10 ms
- DAQ list #1 might be configured so that DAQ-DTOs in it are sampled when a button is pressed
- ...

In other words, a separate DAQ list is required for each unique sampling logic requested for the data measurement. We will describe the DAQ and ODT initialization shortly, but first consider below example trace for an already initiated DAQ sequence:

Trace example: CCP DAQ messages (multiple DAQ and ODT lists)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
2.44	712	00 AF 32 4A 00 00 00 F1	slave	DAQ-DTO (DAQ #0|ODT #0)
2.44	712	01 22 01 00 A0 BB 09 F1	slave	DAQ-DTO (DAQ #1|ODT #1)
2.44	712	02 01 00 FF FF FF 1A 20	slave	DAQ-DTO (DAQ #1|ODT #2)
2.45	712	00 AF 32 4B 00 0A 00 F1	slave	DAQ-DTO (DAQ #0|ODT #0)
2.46	712	00 AF 32 4B 01 0A 00 F1	slave	DAQ-DTO (DAQ #0|ODT #0)
2.47	712	00 AF 33 4A 01 0B 00 F1	slave	DAQ-DTO (DAQ #0|ODT #0)
...	...	...	...	...
2.54	712	00 AF 33 4C 01 0B 00 F1	slave	DAQ-DTO (DAQ #0|ODT #0)
2.54	712	01 23 01 11 A1 BC 08 F4	slave	DAQ-DTO (DAQ #1|ODT #1)
2.54	712	02 02 01 FE FE FF 1B 20	slave	DAQ-DTO (DAQ #1|ODT #2)
2.55	712	00 AF 34 4C 02 0C 00 F1	slave	DAQ-DTO (DAQ #0|ODT #0)
2.56	712	00 AF 34 4D 02 0C 00 F1	slave	DAQ-DTO (DAQ #0|ODT #0)
2.57	712	00 AF 34 4E 02 0A 00 F1	slave	DAQ-DTO (DAQ #0|ODT #0)
...	...	...	...	...
2.64	712	00 AF 31 4D 01 0A 00 F1	slave	DAQ-DTO (DAQ #0|ODT #0)
2.64	712	01 24 05 01 A0 BC 09 F6	slave	DAQ-DTO (DAQ #1|ODT #1)
2.64	712	02 01 00 FF FF FF 1C 21	slave	DAQ-DTO (DAQ #1|ODT #2)
2.65	712	00 AF 31 4D 01 0B 00 F1	slave	DAQ-DTO (DAQ #0|ODT #0)
...	...	...	...	...

Trace explained

In this example, the target ECU is broadcasting data across three ODT lists in total, as evident from the 1st bytes spanning from 0x00 to 0x02. While it's not explicitly clear from the trace, the three ODTs are split into two DAQ lists: One DAQ list contains ODT #0 with a sampling frequency of 10 ms - while DAQ list #1 contains the remaining ODT lists #1 to #2 with a sampling frequency of 100 ms. This is why the DAQ-DTO with PID 0x00 is observed more frequently in the trace vs. the other DAQ-DTOs.

How to initialize a DAQ sequence

We've now looked at both the ODT and DAQ lists - but how do we define these over CAN?

In simple terms, the configuration of DAQ lists is done through a (potentially long) sequence of CRO/CRM-DTO frames. Here, the master essentially specifies the entire DAQ/ODT/element structure element-by-element. To define a new signal (aka element), the master specifies the address, length and address extension. Next, the master informs informs the ECU how to package the element in DAQ-DTOs by linking the element to an ODT# and DAQ#.
To understand this in detail, see below trace example and subsequent explanation:

Trace example: CCP DAQ initialization and initial DAQ-DTOs

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.51	701	14 03 05 AA 00 00 07 12	master	CRO (GET_DAQ_SIZE)
1.55	702	FF 00 03 03 07 AA AA AA	slave	CRM (OK)
1.62	701	15 04 05 07 00 AA AA AA	master	CRO (SET_DAQ_PTR)
1.69	702	FF 00 04 AA AA AA AA AA	slave	CRM (OK)
1.71	701	16 05 01 00 00 00 10 00	master	CRO (WRITE_DAQ)
1.74	702	FF 00 05 AA AA AA AA AA	slave	CRM (OK)
1.79	701	15 06 05 07 01 AA AA AA	master	CRO (SET_DAQ_PTR)
1.82	702	FF 00 06 AA AA AA AA AA	slave	CRM (OK)
1.85	701	0F 07 01 00 00 00 10 01	master	CRO (WRITE_DAQ)
1.88	702	FF 00 07 AA AA AA AA AA	slave	CRM (OK)
...	...	...	...	... (SET_DAQ_PTR + WRITE_DAQ x 4)
2.24	701	15 16 05 07 06 AA AA AA	master	CRO (SET_DAQ_PTR)
2.27	702	FF 00 16 AA AA AA AA AA	slave	CRM (OK)
2.30	701	0F 17 01 00 00 00 10 06	master	CRO (WRITE_DAQ)
2.34	702	FF 00 17 AA AA AA AA AA	slave	CRM (OK)
2.37	701	06 18 01 05 07 03 00 10	master	CRO (START_STOP)
2.41	702	FF 00 18 AA AA AA AA AA	slave	CRM (OK)
2.44	712	07 AF 32 4A 00 00 00 F1	slave	DAQ-DTO
2.54	712	07 AB 22 4C 01 00 00 F1	slave	DAQ-DTO
2.64	712	07 AD 24 4B 02 0F 00 F2	slave	DAQ-DTO
2.74	712	07 AB 25 4C 03 0E 00 F1	slave	DAQ-DTO
...	...	...	...	...

Trace explained

So we have quite a bit going on here, but let's break it down:

We start by using the CRO command GET_DAQ_SIZE. This has two purposes: It informs us about the size of the specified DAQ list in terms of ODT lists - and it clears the current list. Essentially, this works like a 'reset' command for the DAQ list, in this case DAQ list #5.

As part of the GET_DAQ_SIZE command we also provide the 11-bit CAN ID 0x712 in the last 4 bytes. This informs the ECU that it should use CAN ID 0x712 to broadcast the subsequent DAQ-DTOs. This is subtle, but an important aspect: Essentially, the master is able to specify the CAN ID for every DAQ list, which in turn also enables the master to initiate DAQ measurement across multiple ECUs in parallel.

The ECU confirms the CRO from the master and informs us that the DAQ list #5 has 3 ODTs with the first PID being 0x07. Next, the master starts populating DAQ list #5 and ODT list #7 with the source address information for 7 x 1-byte signals. This is done through a simple loop consisting of 7 repetitions of two commands: SET_DAQ_PTR and WRITE_DAQ.

First, the master uses SET_DAQ_PTR to "select" DAQ list #5, ODT list #7 and element #0 of the ODT list. Next, the master uses WRITE_DAQ to write the contents of this element reference: A length of 1, an address extension of 0 and a source address of 0x00001000.

Basically, we have now told the ECU to "package" the value of our 1st signal (which is stored in the specified source address) into the 2nd byte of the DAQ-DTO that corresponds to DAQ list #5 and ODT list #7 (the 1st byte of the DAQ-DTO being the PID 0x07).

After this, we repeat the process for the remaining 6 signals until the ODT #7 has been completed.

In this simplistic case we only care about this particular DAQ and ODT list. Therefore, the final step is to start the DAQ measurement via the START_STOP command. Here, we specify that we wish to start DAQ list #5 and ODT list #7. As

part of this, we specify the timing parameters, namely that event channel 0x03 should be used with a prescaler of 10. The ECU specific event channel details are specified in the ECU description file.

As evident, the ECU will now start broadcasting data from DAQ list #5 and ODT list #7 at the specified frequency. The CAN frame payloads include the ODT PID 0x05 in the 1st byte and the 7 'element' signal values in the remaining payload.

Note also that you can alternatively use the START_STOP command to 'prepare' multiple DAQ lists and ODT lists for measurement and then use the START_STOP_ALL command to simultaneously start or stop all of them.

How to disconnect from an ECU

Once the data acquisition has completed, the master may disconnect from the ECU via the DISCONNECT command.

Let's look at an example trace:

Trace example: CCP disconnection (DISCONNECT CMD + ECU response)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
24.01	701	07 88 01 AA 02 07 AA AA	master	CRO (DISCONNECT)
24.04	702	FF 00 88 AA AA AA AA AA	slave	CRM (OK)

Trace explained

Here, the 3rd byte 0x01 means that we end the session entirely (in contrast to a temporary disconnection). This effectively resets the ECU including our previous configuration. To target the specific ECU of interest, we specify the station address (Intel format) of the ECU, i.e. 0x702. As evident, DAQ is more convoluted to set up - but enables efficient time synced communication of ECU data.

Decoding CCP signal data from ECUs

In CCP polling/DAQ, you are in practice recording raw CAN frames with specific signal encoding structures. In order to make sense of the recorded data, you need to decode it to human-readable form aka physical values. This is the same concept as we've explained in several other CAN-related intros, incl. our intro to CAN bus, J1939, OBD2 and DBC files. However, CCP decoding involves some extra complexities that we will explain below.

Decoding CCP polling data

Let's review an extended version of the CCP polling request/response from before:

Trace example: CCP polling via SHORT_UP command (two signals)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.51	701	0F 02 04 00 12 34 56 78	master	CRO (SHORT_UP CMD)
1.53	702	FF 00 02 F1 2A 71 2F AA	slave	CRM (OK + DATA)
1.55	701	0F 03 04 00 AB CD EF 00	master	CRO (SHORT_UP CMD)
1.57	702	FF 00 03 00 37 1A 81 AA	slave	CRM (OK + DATA)

This reflects a decoding challenge: The ECU sends different signals with the same CAN ID - with no way to identify them in the payload.

Detailed explanation

In the above example, the master is requesting two different 4-byte signal values from the ECU, namely from source addresses 0x12345678 and 0xABCDEF00. In both cases, the ECU responds with CAN ID 0x702.
In most CAN bus decoding scenarios, you would be able to simply lookup the response CAN ID and find the 'start bit' and 'bit length' of a given signal to extract the raw data from the payload.

However, that is not possible here as the ECU uses the same CAN ID across two different signals. In other words, the CAN ID is not sufficient to distinguish between the signal coming from 0x12345678 and the one coming from 0xABCDEF00.

To some extent, this is similar to how OBD2 responses from an ECU use the same CAN ID (typically 0x7E8) across different signals like speed and RPM. In the case of OBD2 we can, however, easily solve this because the OBD2 PID (Parameter Identifier) is included in the payload. Together with the CAN ID, this serves as a unique identifier within the response frame - enabling us to view the response as a case of multiplexing.

We cannot directly do this in CCP polling because the response payload does not include the ECU source address. In other words: In use cases with multiple CCP polling signals we need to combine information from the request & response message.

In simplistic terms, we can solve this by "repackaging" the payloads as follows:

Trace example: CCP polling - decoding via repackaging of payload

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.51	701	0F 02 04 00 12 34 56 78	master	CRO (SHORT_UP CMD)
1.53	702	FF 00 02 F1 2A 71 2F AA	slave	CRM (OK + DATA)
1.55	701	0F 03 04 00 AB CD EF 00	master	CRO (SHORT_UP CMD)
1.57	702	FF 00 03 00 37 1A 81 AA	slave	CRM (OK + DATA)
1.53	702	0F 02 04 00 12 34 56 78|FF 00 02 F1 2A 71 2F AA		CRO + CRM
1.57	702	0F 03 04 00 AB CD EF 00|FF 00 03 00 37 1A 81 AA		CRO + CRM

"multiplexor"

By doing so, we can process this using multiplexing logic - similar to how we handle OBD2 and UDS decoding. Here, the CAN ID to look up is 0x702 and the multiplexor is found in the 5th to 8th byte, while the signal value is found in the 12th to 15th byte. Two separate decoding rules can now be specified, with the relevant one dependent on the value of the 4-byte multiplexor.

In practice such a reconstruction of CAN frames can be done in e.g. a Python script, assuming the CAN trace includes both the request and response data. Once the CAN frames are reconstructed, it is possible to use DBC files for decoding the data via multiplexing. Alternatively, the data can be loaded into a software tool supporting CCP/XCP decoding directly.

Decoding CCP DAQ data

Let's briefly review a snippet of the post-initialization DAQ trace from before:

Trace example: CCP DAQ messages

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
2.44	712	07 AF 32 4A 00 00 00 F1	slave	DAQ-DTO
2.54	712	07 AB 22 4C 01 00 00 F1	slave	DAQ-DTO
2.64	712	07 AD 24 4B 02 0F 00 F2	slave	DAQ-DTO
2.74	712	07 AB 25 4C 03 0E 00 F1	slave	DAQ-DTO
...	...	...	...	...

As evident, the DAQ-DTO messages have a pre-specified CAN identifier (0x712 above) and a payload in which the 1st byte equals the ODT list identifier aka ODT PID (0x07 above). As a result, it is not necessary to "combine" the DAQ-DTO response frames to uniquely identify which frames contain which signals - this can directly be identified through a combination of the CAN ID and the ODT PID.

This fact makes the decoding of DAQ-DTO messages simpler than the polling messages. In fact, you can directly create a DBC file with multiplexing as long as you know the signal encoding in the remaining bytes of each ODT list.

Will the CCP DAQ decoding rules be known?

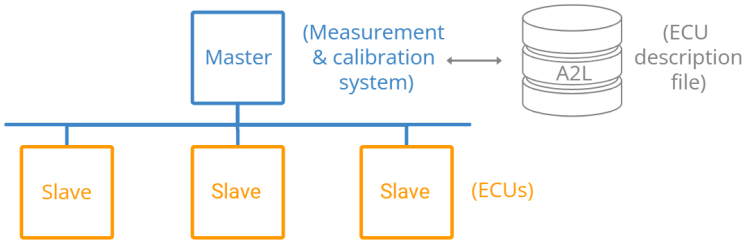
In most practical applications, it is reasonable to assume that you will know this signal encoding. This is because the master itself controls (via the initialization sequence) how to package each signal into the DAQ and ODT lists, as explained in the previous section.

For example, we know that the DAQ-DTOs with CAN ID 0x712 and ODT PID 0x07 contain the signal with source address 0x00001000 in the 2nd byte - because that is how we packaged it during the initialization. From the ECU description file we can then review how to interpret this 1-byte signal value. The description may state that this signal is a temperature measured in degC and it should be multiplied by a factor of 0.8 and offset by 20. If you're familiar with DBC files, you'll note that such information can easily be entered into a DBC, allowing for quick decoding of the trace data in most CAN software tools.
Naturally, if the master changes the initialization to package the data differently on another test run, the implication will be that the DBC file must be updated accordingly.

The above should make it clear that CCP polling/DAQ can be treated within the normal logic of CAN bus frame decoding - though both involve some tweaks. DBC files can be used to process the data - but this format is not the most common choice for working with CCP data. Below we explore another file format that is generally used, the A2L or ASAP2 format.

A2L - ECU Description Files

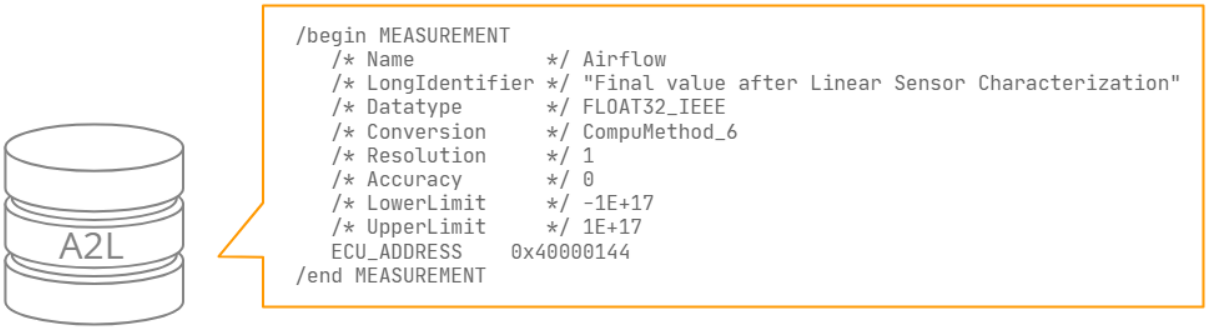
In the prior sections, we've occasionally referred to "ECU description files". An ECU description file contains everything a master tool needs to know to communicate with an ECU. From a data measurement perspective, this includes information on CAN identifiers, available signals, signal decoding rules, DAQ/ODT information etc.



In practice, the ASAP2 description format (*.A2L) is used to structure this information. The ASAP2 data definition was standardized by ASAM in the ASAM MCD-2 MC, with the first version 1.3.1 being released June 15 1999, i.e. the same year that the CCP 2.1 standard was released. The latest version of the standard, 1.7.0, was released in 2015.

You can view an example A2L file here.

In practice, the A2L file is used to "configure" the master device, enabling communication with the ECU. For example, in our previous examples we used the CAN IDs 0x701 and 0x702 for the CRO/DTO communication between the master and ECU. These IDs would be specified in the A2L file. The A2L file also describes how signals are stored in the ECU and how to decode them. See e.g. below snippet from the A2L example file:



A2L snippet explained

As before, we can compare this to the logic of signal encoding in other CAN protocols and in DBC files. The 'MEASUREMENT' tag is used to envelop the description of a measurement signal, in this case Airflow. It has a detailed description and a min/max defined by the lower/upper limit.

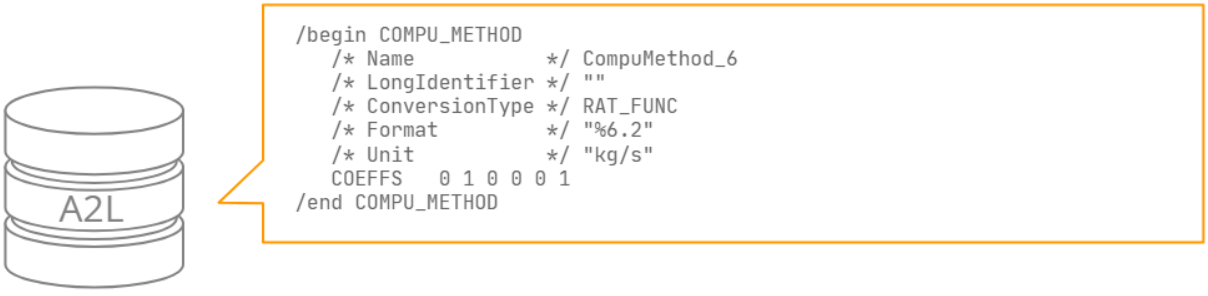
The signal also has a 4-byte source address, 0x40000144. If we recall the CCP polling outline, this source address could be used in a SHORT_UP command polling sequence to request the raw value for this particular ECU signal. Similarly, in the context of DAQ measurement, this source address would be used in the initialization process. In other words, an engineer can lookup the signal of interest (Airflow) in a A2L file or A2L GUI tool - and configure his master tool to request this.

Assuming we've now recorded a trace of raw values, we'd need to decode the information. As explained before, the first step would be to extract the raw bytes from the response payload - and the method for doing this will depend on whether we're using polling or DAQ measurement. Further, the byte ordering is not specified by the CCP protocol - but will be specified within the A2L file at a global or signal level.

Once we've extracted the signal bytes and converted them to decimal form, we need to know how to convert them.

In the DBC file context, this would always be done in the form of a 'linear' equation, i.e. y = bx + c, where b equals a scale factor and c equals an offset - both specified in the DBC file for each signal.

In contrast to DBC files, the A2L files allow for more complex conversion methods. As evident from the signal description, it refers to a conversion rule defined as "CompuMethod_6". We can look this up in another section of the A2L file:



A2L snippet explained

As outlined, this conversion rule uses a conversion type called RAT_FUNC. This is one of multiple supported conversion types and is defined as below:

y =(axx + bx + c)/(dxx + ex + f)

As evident, the rational function is a more sophisticated version of the linear function used in DBC files. In particular, it supports 5 coefficients, which need to be specified in the computation method. Above, they are set to 0 1 0 0 0 1, meaning the function simplifies to below:

y = x

In this simple case, the rational function simplifies to a linear function, similar to the one used in DBC files. A2L files in fact also support another conversion type, LINEAR, which is simply y = ax + b. In the case of the Airflow signal, this could have been used instead. Finally, the unit is also specified in the measurement details, meaning we can now plot the Airflow physical value as measured in kg/s. The Format field is %6.2, which should be read as %Length.Layout, with Length reflecting the overall length of the decoded signal and Layout reflecting the number of decimal places.

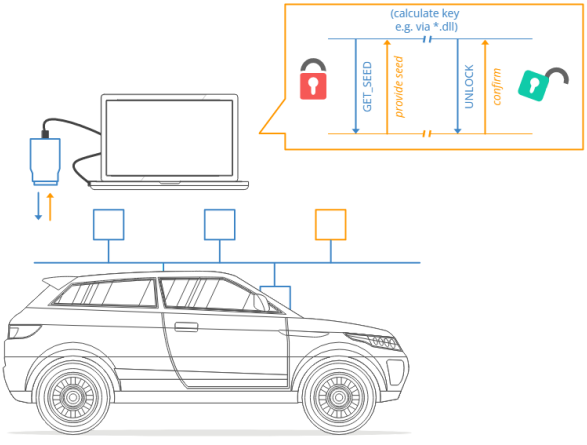
Practical use of A2L files

As is probably clear, one does not normally look through the A2L file via a text editor. Rather, the A2L file is simply loaded in a compatible tool (e.g. a PC GUI tool with a connected CAN interface, or a CAN data logger). This then allows the engineer to use the tool to perform the relevant measurement from the ECU without having to understand the A2L file structure and syntax.

This particular signal is simplistic enough to be described within the restrictive DBC file format - but A2L files clearly allow for more complex decoding rules. For more details on A2L file format, see the ASAM MCD-2 MC standard, as well as a practical overview.

Seed & key authorization

As explained previously, CCP communication is typically used by automotive OEM engineers to facilitate communication with ECUs in pre-production applications.



In some cases all it takes to communicate with an ECU is the A2L file and a suitable CAN tool. As evident, one can also simply extract a subset of the A2L file to configure a device e.g. for measurement of specific signals. However, without the A2L file, it's practically impossible to perform communication and hence many OEMs do not require any additional security measures beyond keeping the A2L file under lock.

However, some use cases warrant additional security. CCP supports this via a concept called 'seed & key' authorization. Here, the master requests a random seed from the ECU as part of the initialization of the ECU communication. The master receives the seed from the ECU and uses it as input for an internal security algorithm to calculate a key. The master then sends the key to the ECU via CAN - and if it matches the key calculated internally by the ECU, authorization is provided for further communication.
Below is an example trace for such a communication:

Trace example: CCP seed & key authorization sequence

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.51	701	12 05 02 AA AA AA AA AA	master	CRO (GET_SEED)
1.53	702	FF 00 05 01 14 15 16 17	slave	CRM (OK + SEED)
1.55	701	13 06 A3 12 FF B0 AA AA	master	CRO (UNLOCK + KEY)
1.57	702	FF 00 06 02 AA AA AA AA	slave	CRM (OK + RESOURCE MASK)

Trace explained

In this example, the master requests the seed via the GET_SEED command 0x12, specifying a 'resource mask' of 0x02 in the 3rd byte. The resource mask references the level of access requested - with 0x02 being DAQ access only (see the CCP 2.1 standard for details).

The ECU responds with a frame in which the 4th byte is 0x01, implying that DAQ access is protected (in contrast to 0x00 which would imply that authentication would not be required). The ECU provides the requested seed in the remaining 4 bytes. Based on the seed, the ECU calculates the key and provides it via the UNLOCK command, 0x13. The key is in this case 4 bytes long. The ECU responds positively including the requested resource mask 0x02, meaning that DAQ access is now unlocked.

Regarding security algorithm implementation

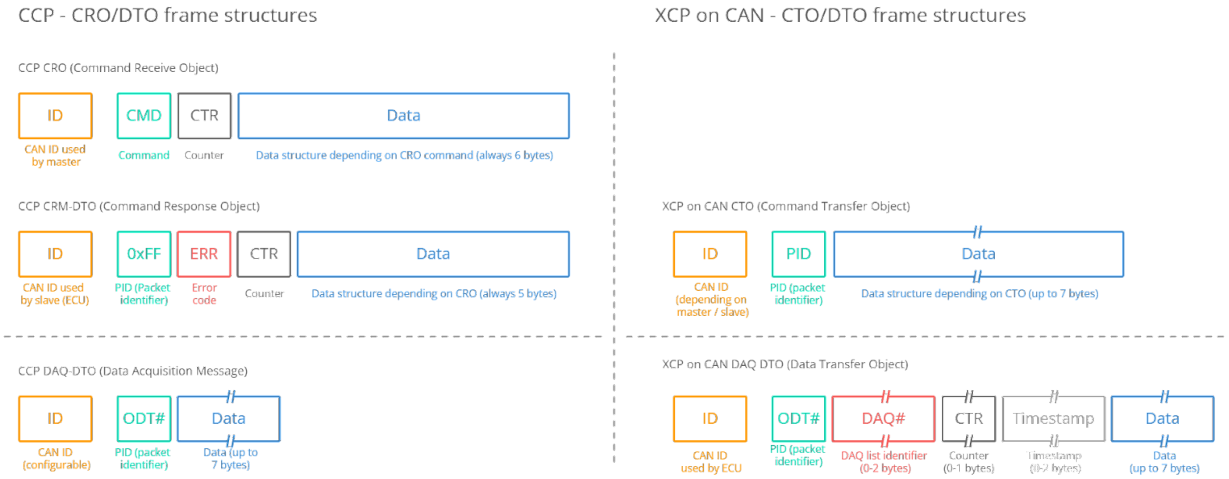
The most common implementation of the security algorithm is via a *.dll file, since the master device is frequently a Windows PC with a GUI tool and a CAN interface. The use of a *.dll file also enables standardization across CCP tools, while eliminating the need for the master tool to know the underlying algorithm used. It is, however, challenging to use *.dll files in standalone CAN bus data loggers and telematics devices (as they do not run Windows), hence some companies use alternative solutions - e.g. proprietary file formats better suited for embedded devices.

XCP on CAN - the basics

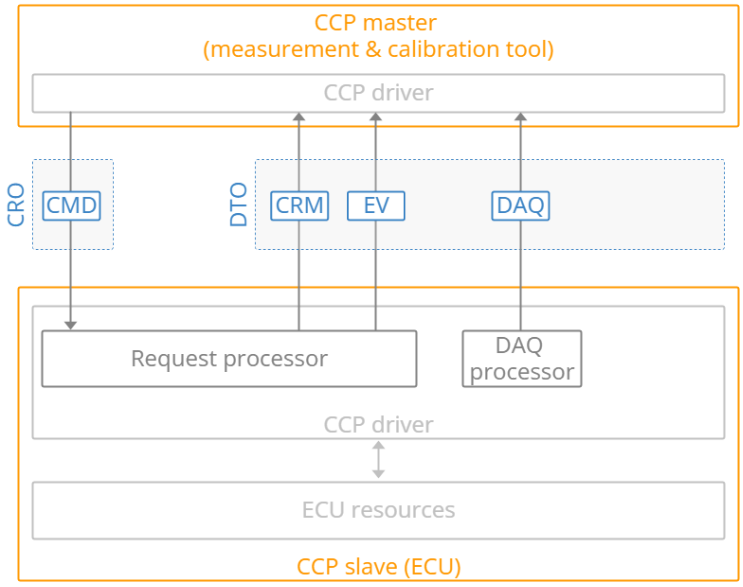
We have now gone through the basics of CCP. Next, we'll consider XCP on CAN with focus on frame structures.

The XCP packet

To understand the structural changes in XCP on CAN vs. CCP, consider below comparison of the CCP CRO/DTO and the XCP CTO/DTO. As evident, XCP packets include the **XCP CTO** (with a similar role as the CCP CRO and CRM-DTO) and the **XCP DTO** (with the XCP DAQ-DTO playing a similar role as the CCP DAQ-DTO).



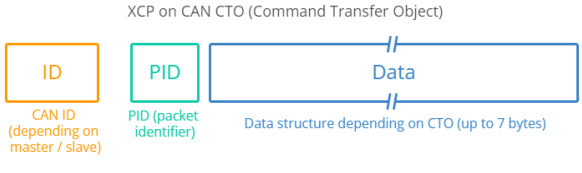
An alternative way to illustrate the role of the CCP vs. XCP messages is via below architecture comparison:



We will cover the XCP on CAN message types in more detail below.

XCP CTO - Command Transfer Object

In XCP on CAN, the CTO is a CAN frame for transferring control commands, incl. commands (CMD), command responses (RES), errors (ERR), events (EV) and service requests (SERV). In contrast to the CRO in CCP, the CTO is used by both the master and ECU. Note also that a CMD packet from the master must be answered with a RES or ERR packet from the ECU, while the other packet types are sent asynchronously.

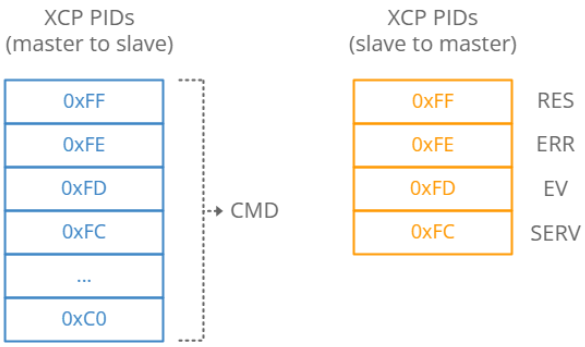


If we look at the CTO payload, the 1st byte reflects a Packet Identifier (PID). The remaining 7 bytes in an XCP on CAN CTO payload consists of data, specific to the type of CTO packet. If we compare this structure to the CRO of CCP, it's similar, except for the exclusion of the command counter byte in XCP.

CAN bus identifiers in XCP

XCP communication requires at least two CAN bus identifiers: One for the master (e.g. 0x551) and one for the ECU (e.g. 0x552). If the master needs to communicate via XCP with more than one ECU, an additional set of identifiers will be required.

The actual values of the CMD byte across XCP and CCP differ as well. For example, CCP uses 0x01 for the CONNECT command, while XCP uses 0xFF. See also the table overview for the assignment of CTO PIDs depending on whether messages are sent from the master or slave.



See also below comparison of CCP vs. XCP CMD codes:

Select CCP vs. XCP CMD values

The below table compares a subset of the command codes between CCP and XCP:

CCP vs. XCP command code comparison (examples)

CCP CMD	XCP CMD	Command
0x01	0xFF	CONNECT
0x12	0xF8	GET_SEED
0x13	0xF7	UNLOCK
0x02	0xF6	SET_MTA
0x04	0xF5	UPLOAD
0x0F	0xF4	SHORT_UP
0x15	0xE2	SET_DAQ_PTR
0x16	0xE1	WRITE_DAQ
0x07	0xFE	DISCONNECT
0x0E	0xF3	BUILD_CHECKSUM

Example: XCP CTO CONNECT message

As in CCP, let's look at how a CONNECT sequence may look in XCP on CAN:

Trace example: XCP on CAN initialization (CONNECT CMD + ECU response)

Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.01	701	FF 00	master	CTO (CONNECT)
1.04	702	FF 04 80 08 08 00 01 01	slave	CTO (OK + INFO)

Trace explained

Note that the example does not use the 0xAA padding, as this is optional in XCP on CAN.

In the trace, the master sends a CONNECT CMD (0xFF) to the ECU with the communication mode set to normal (0x00). Note that, in contrast to CCP, the master does not need to specify a station address in the payload of the CONNECT frame. This is because the CAN identifiers already uniquely identify which ECU the master is communicating with.

The ECU responds with a CAN frame in which the 1st byte equals the positive response PID (RES). The 2nd byte is the resource availability, similar to the seed & key CCP communication with 0x04 e.g. meaning that DAQ is available. The 3rd byte relates to the 'communication mode' (here 'optional'), while the 4th byte is the maximum CTO size (here 8 bytes). The 5th and 6th byte equal the maximum DTO size (here 8 bytes), while the 7th and 8th bytes equal the XCP protocol layer version and transport layer version respectively (both 1 in this case).

Example: XCP Polling

With a connection established, the master can e.g. initiate polling. As in CCP, this can be done via a SHORT_UPLOAD command:

Trace example: XCP on CAN polling via SHORT_UPLOAD command

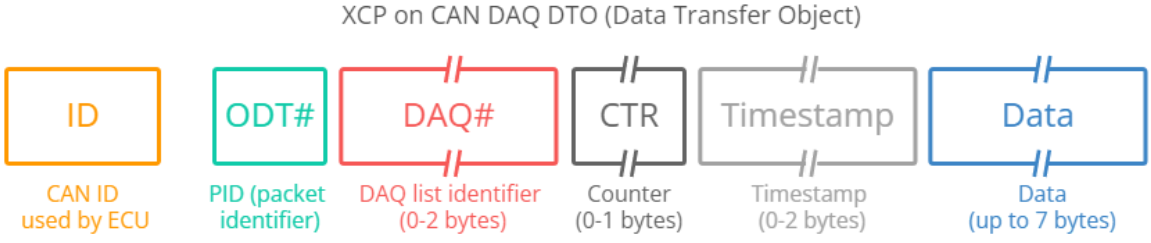
Time	CAN ID (HEX)	DataBytes (HEX)	Sender	Frame type
1.51	551	F4 02 00 00 78 56 34 12	master	CTO (SHORT_UPLOAD)
1.53	552	FF F1 2A	slave	CTO (OK + DATA)

Trace explained

Here, the master sends a SHORT_UPLOAD CMD (PID 0xF4) for 2 bytes of data. The 3rd byte is reserved, while the 4th byte of the CTO is the address extension (in this case 0). The remaining 4 bytes equal the source address 0x12345678 (INTEL byte order). In response to the command CTO, the ECU sends a response CTO with the value of the 2 bytes of data.

XCP DTO: Data Transfer Object

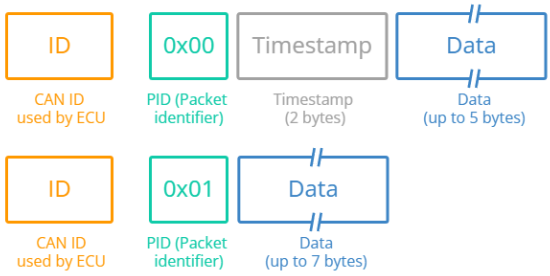
The XCP DTO is used for sending synchronous data. In particular, the DTO is used in DAQ measurement (similar to the role of the DAQ-DTO in CCP). In XCP, it also enables the transfer of 'stimulation' (STIM) data that can be used in bypassing the normal algorithm within an ECU. Stimulation and bypassing are topics that we will not cover here.



We will also not go into detail on the XCP DAQ measurement as it's similar in concept to CCP DAQ.

The XCP DTO timestamp

One important thing to note about XCP DAQ measurement, however, is that it enables the ECU to write the measurement time in the XCP DTO packets. The implication is that the master is able to correctly sync ECU data split across multiple frames by utilising the ECU measurement timestamp, rather than the master's own internal timestamp. The XCP DTO timestamp is optional and is implemented as an incrementing counter, with the incrementation logic specified in the A2L file. If a timestamp is to be included for a specific DAQ list, it will be written into the 1st ODT list (but not subsequent ones if more ODT lists exist within the same DAQ list).



DAQ/ODT identification

It is also worth noting that multiple methods exist for packing the DAQ list # and ODT list # information in the DAQ DTOs. The simplest case is 'absolute ODT numbers', where the ODT list # is unique across all DAQ lists. Here, the PID (i.e. ODT #) is sufficient for globally identifying an ODT list. Another method exists, however, where relative ODT list numbers are used across DAQ lists. Thus, you could e.g. have two ODT lists with PID = 0x00, meaning that they cannot be uniquely identified as the CAN ID is identical across the two DTOs. Here, a 1 or 2 byte DAQ list identifier can be added in the 2nd to 3rd bytes of the CAN frame payload. With this, one can again uniquely identify a specific ODT list. The A2L file and ECU will provide information on what method is used.

Optional XCP DTO counter field

A 1-byte counter field may optionally be used in the communication of XCP DTOs. If a CTR field is used in XCP DTO DAQ packets, the slave will only insert the value of the CTR in the 1st frame of a DAQ list, i.e. in the 1st ODT list.

Links for further reading on CCP/XCP

CCP/XCP of course contain numerous more topics that we have not taken the time to discuss here. Deliberately, we've focused on the basic topics and concepts related to data logging. However, below you'll find useful links for further reading.

- Vector's XCP Book v1.5: A detailed overview of the XCP 1.5 protocol
- XCP (Wikipedia): The basics on the XCP protocol
- ASAM MCD-1 XCP overview: Also provides a detailed overview of the XCP protocol
- ASAM ASAM MCD-2 MC overview: Provides an overview of the ECU description file standard
- Seed & key details (Vector): A useful intro to the basics on authorization

Using the CANedge for CCP/XCP data acquisition

The CANedge is a series of low cost, compact 2 x CAN/LIN bus data loggers. The CANedge is commonly used by automotive OEMs - and for this reason a frequently asked question is whether the device supports CCP/XCP on CAN communication. The answer to this is 'yes, depending on

your use case'. For details, see below. If you wish to use the CANedge for CCP/XCP on CAN, contact us.

CANedge + CCP/XCP

As we've seen in the previous sections, all it takes to facilitate CCP/XCP polling and CCP/XCP DAQ measurement is to be able to transmit a specific sequence of CAN frames, based on information that can be identified from an A2L file.
As long as you know how to construct the request frames, you can use the CANedge transmit functionality to set up a list of single-shot or periodic custom CAN frames to e.g. achieve the CCP/XCP initialization and CCP/XCP polling/DAQ measurement. The recorded MF4 log files with measurement data can then be processed in e.g. Vector tools or via our free open source software solutions. In the former case, you could e.g. use our MF4 converters to get the data into Vector tools and use A2L files for decoding the data. In the latter case, you could process the data via our Python API (for polling) and/or via asammdf. These tools would currently require creating a DBC file with the decoding rules as explained previously.

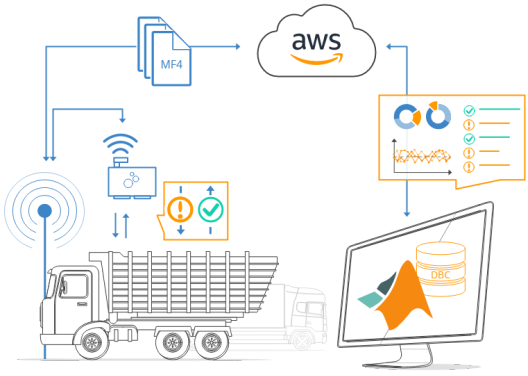
In other words, you can use the CANedge to measure signal data via CCP/XCP on CAN in standalone field deployments, which offers a low cost method for collecting this information at scale. In contrast, standalone CAN bus data loggers are not ideally suited for CCP/XCP calibration, except for very specific use cases such as remote calibration of variables via over-the-air updates.

CANedge seed & key authentication

The CANedge does not currently support seed & key authentication, so OEMs intending to use it for CCP communication would need to temporarily disable the authentication or to unlock it using a separate device. We are, however, potentially going to add support for this later on. If you have use cases that involve this functionality, feel free to contact us for further discussion.

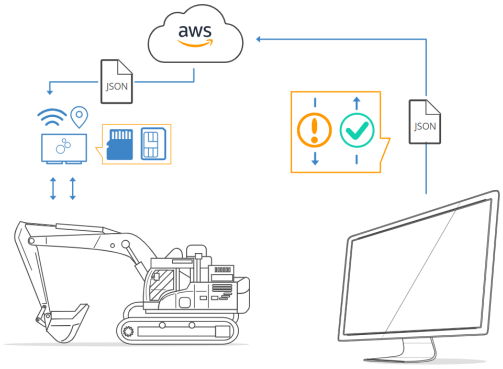
CCP/XCP data logging - applications

In this section we provide brief examples of how CCP/XCP data logging may be useful in practice.



CCP/XCP telematics for prototype vehicle fleet

As an OEM, you may need to collect e.g. CAN, CAN FD or LIN bus data from prototype vehicles in the field. In addition, you may need to collect specific data internal to an ECU, which can only be measured via CCP or XCP on CAN. Here, a CAN logger like the CANedge can be configured with a custom 'transmit list' consisting of single shot and/or periodic CAN frames. When deployed in the vehicle, the CANedge will then automatically transmit the pre-defined frames, allowing it to e.g. perform CCP/XCP polling of ECU data - or initialize DAQ measurement. The collected CCP/XCP data can be combined with the general CAN data in MF4 log files - and sent via WiFi/3G/4G to the OEM's own cloud server.



Remote CCP/XCP calibration via OTA updates

As a more sophisticated use case, an OEM could utilize a CANedge2 to perform over-the-air updates to an ECU. For example, a CANedge2 may be deployed in a prototype vehicle in which an OEM engineer would like to re-calibrate certain variables in the ECU. To achieve this, the engineer constructs the relevant sequence of single-shot CAN frames in a new configuration file for the CANedge. Next, the engineer deploys the new

configuration file on the CANedge2 S3 server, thus triggering an OTA update - allowing the CANedge2 to transmit the sequence into the vehicle remotely.

Thank you for reading our guide - we hope you found it useful!

CSS Electronics | DK36711949 | Soeren Frichs Vej 38K, 8230 Aabyhoej, Denmark

www.csselectronics.com | contact@csselectronics.com | +45 91 25 25 63 | LinkedIn

Products | Software | Guides | Case studies | 5 min CANedge intro

CSS Electronics

Danish developer & manufacturer of CAN hardware

DK **DANISH DESIGN:** Devices are designed from scratch by our Danish engineers

ISO **US PRODUCTION:** PCBs are assembled by our ISO 9001 US partner. EOL finalized in DK

100+ **GLOBAL:** We ship globally and our products are used across 100+ countries

<24H **SUPPORT:** We are known for extremely fast, high quality technical support

5K+ **REACH:** Devices are used by 5,000+ companies and trusted by top tier OEM engineers

CO2 **CO2 NEUTRAL:** We offset 100% of our CO2 emissions incl. production & shipping

CAN bus data loggers - simple-to-use. pro specs. interoperable.

Your data, your way. Easily log CAN data from your vehicle / machine - and extract it via SD, WiFi or 3G/4G. Process the interoperable data in your favorite tools - or via 100% free open source software/APIs.

Perfect for vehicle telematics, OEM development, diagnostics and more.

We offer two series of CAN bus data loggers: The CLX000 and CANedge. Further, we offer the CANmod sensor-to-CAN modules.

CLX000 - low cost CAN logger & interface

The CLX000 lets you log CAN data to an SD card in CSV format - and process it via the free software tools. Further, you can stream data in real-time via USB on your PC for e.g. reverse engineering and diagnostics purposes.

The CLX000 is ideal if you need a basic CAN logger, while also being able to stream data in real-time via USB.

CANedge - pro specs 2x CAN/LIN logger

The CANedge is our 2nd gen data logger. It combines ease-of-use with cutting-edge specs and powerful configurability. The time stamped data can be processed in your favorite tools - or via 100% free open source software/APIs.

The CANedge1 is recommended for pure data logging use cases.

The CANedge2/CANedge3 is ideal if you need to collect data via WiFi/LTE - e.g. pushing data to your server from a fleet of assets.

CANmod - sensor-to-CAN modules

The CANmod is a series of 'sensor-to-CAN' modules that output e.g. GNSS/IMU or thermocouple data. The modules integrate with any CAN bus system - meaning you can use them standalone to inject CAN data into e.g. a vehicle/machine CAN bus.

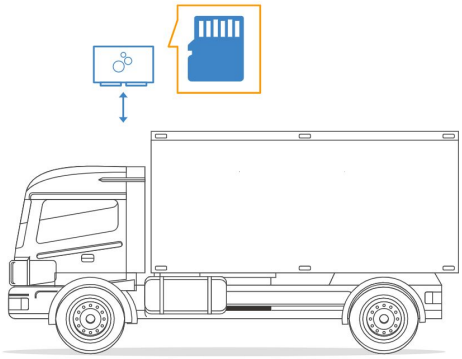
You can also use them as add-on modules for any CAN logger/interface - incl. In particular as add-ons for the CANedge.

Trusted by engineers at leading OEMs

VW · BOSCH · LIEBHERR · NISSAN · SIEMENS · JCB · TOYOTA · Continental · HYUNDAI · Cummins · CERN · JOHN DEERE

CSS Electronics ApS | +45 91 25 25 63 | contact@csselectronics.com | www.csselectronics.com

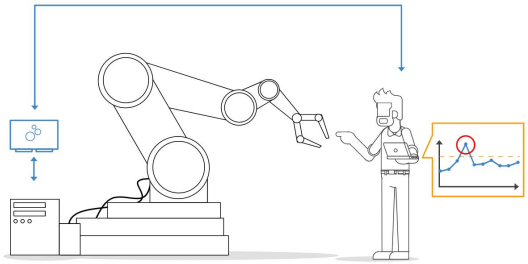
Example use cases



Vehicle 'black box'

Need to monitor data from your vehicle fleet?

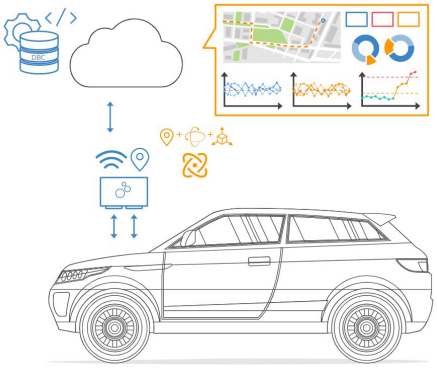
The CANedge1 is ideal for recording data over long periods from fleets - incl. cars (via OBD2) and heavy-duty vehicles (via J1939). With filters, compression & cyclic logging you can log years of data. Ideal for e.g. warranty/legal data storage or optimization.



Rare issue identification

Need to identify a rare, but critical issue?

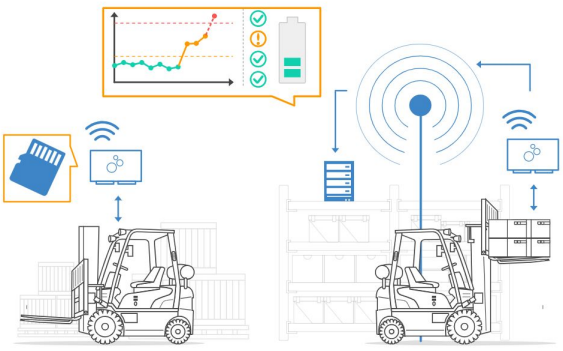
If you have e.g. production equipment that periodically exhibts an issue the CANedge1 is a great diagnostics tool. Simply connect it and leave it in. If the issue occurs, you can extract the raw data, DBC convert it and plot it for quick diagnostics.



Vehicle telematics

Need to monitor & optimize a vehicle fleet in near real-time?

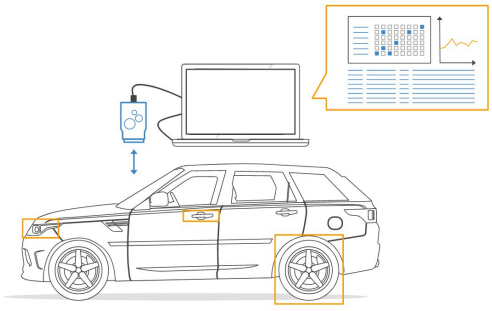
For 'on-road' upload of data, the CANedge3 can upload data via 3G/4G to e.g. your own AWS S3 cloud server. Alternatively, the CANedge2 can upload data when the vehicle periodically returns to e.g. a garage WiFi network. Ideal for OBD2 logging, truck fleet management, predictive maintenance and more.



Warehouse fleet management

Need to monitor a fleet of AGVs/forklifts in a warehouse?

For site managers/OEMs, the CANedge2 can collect data from AGV/forklift fleets e.g. for optimizing battery management. The device has strong WiFi range, can upload via multiple WiFi access points - and can store data on your local/dedicated/cloud server.



Reverse engineering proprietary data

Want to reverse engineer proprietary data parameters?

You may need to reverse engineer proprietary CAN data as part of e,g, creating dashboards or custom applications. The CLX000 lets you use both data logging and USB streaming as part of decoding exercises. Further, our free Wireshark plugin provides useful tools for reverse engineering CAN data.

CLX000

Low cost CAN logger & interface

PLUG & PLAY: Log data out-the-box. Standalone. Power via CAN connector

STREAM: Send/receive CAN data in real-time to PC via USB in SavvyCAN (DBC support)

STANDALONE: Log CAN data to 8-32 GB SD card (no PC needed). RTC (CL2000)

FREE SOFTWARE: All software is 100% free. DBC convert data via simple Windows GUI tool

COMPACT: Only 7 x 4 x 2 CM. 50G. 3 LEDs. Mini USB port for SD extraction + streaming

LOW COST: Lowest cost CAN logger & interface on the market

The plug & play CLX000 is a simple-to-use CAN logger. The device logs CAN data to an 8-32 GB SD card or streams it via USB to a PC.

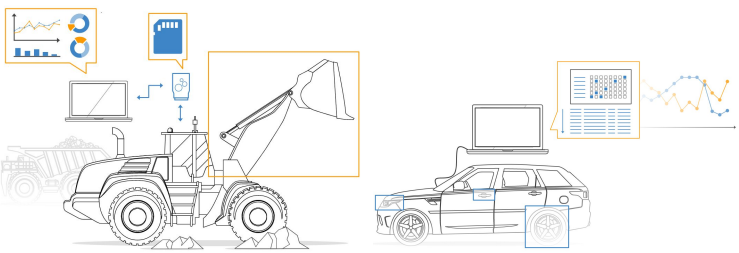
For the CL2000, data is date & time stamped via a real-time clock with battery backup.

The CLX000 is used in blackbox logging, reverse engineering or simple telematics use cases. For logging or WiFi/LTE use cases, see also the '2nd generation' CANedge series.

Compact CAN logger + interface

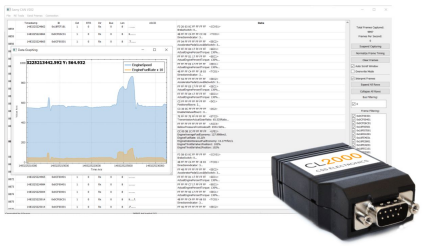
The CLX000 lets you both log raw CAN data to an 8-32 GB SD card and/or stream it via USB in real-time:

- Power via DB9 connector (<1W power consumption)
- Auto-detect bit rate - no configuration required
- Log raw CAN data in simple CSV format
- Easily extract data from the SD card via USB
- Configure device via simple CONFIG.INI file or editor
- Silent mode, filters, transmit lists, cyclic logging
- Timestamp data via real-time clock (CL2000)
- Stream data in real-time via USB in SavvyCAN
- Transmit custom data with real-time control
- DBC convert logged/streamed data via free software



Send/receive data in real-time via USB

With the free SavvyCAN software, it's easy to send/receive CAN data via USB. View raw traces, DBC decoded signals or create visual plots. The software also adds powerful reverse engineering tools.



Technical specs

GENERAL

Safety	CE, FCC, IC certified
Warranty	1-year warranty
Support	Free, fast & high quality support
Origin	Denmark
Software	100% free
Documentation	Online/PDF documentation

CAN BUS

Channels	1 x CAN (Classical)
Protocols	J1939, OBD2, CANopen, NMEA2000, ...

DATA LOGGING

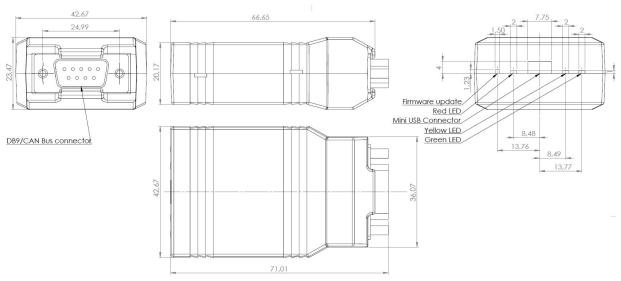
SD CARD	8-32 GB SD card (extract via USB)
Real-Time Clock	1 ms resolution (CL2000)
File format	CSV-style plain text format
Safety	100% power safe
Configuration	Simple configuration options

USB STREAMING

Stream	Send/receive raw/decoded CAN data

MECHANICAL/SUPPLY

Connectors	1 x DB9 (adapter cables available)
Input supply	+7V to +32V DC via DB9
Consumption	<1W
Dimensions	66.7 x 42.7 x 23.5 mm (L x W x H)
Weight	50 G
LEDs	3 LEDs (PWR, DATA, MEM)
Temperature	-20 degC to +65 degC
IP rating	IP40



Trusted by engineers at leading OEMs

CANedge1

2xCAN/LIN Data Logger (SD + RTC)

PLUG & PLAY: Log data out-the-box. Standalone. Power via CAN connector

CONFIGURABLE: Filters. Transmit lists. Triggers. Cyclic logging. Silent mode

PRO SPECS: Extractable 8-32 GB SD. 2xCAN/LIN. CAN FD. Zero data loss. 50 µs RTC

FAST & SECURE: Industrial SD card. Data encryption for GDPR/CCPA. Compression

COMPACT: Only 8 x 5 x 2 CM. 100G. Alu enclosure. 4 LEDs. Configure CH2 5V power out

INTEROPERABLE: Convert MDF4 to e.g. CSV, ASC, TRC. Free open source GUI/API

The plug & play 2xCAN/LIN logger records timestamped CAN data (Classical/CAN FD) to the extractable 8 GB industrial SD card.

It's easy-to-use: Simply power the device via your CAN connector to start logging raw data. Extract the data and process it using 100% free open source MDF4 software/API tools - or convert it to your favorite log file format (Vector ASC, PEAK TRC, CSV, ...).

The CANedge1 is ideal for logging of CAN/LIN systems over long periods - e.g. for OEM R&D, diagnostics or legal purposes.

New: The CANedge1 is now available with **optional GPS/IMU**.

Pro specs CAN logger - at half the cost

The CANedge1 combines innovative design, cutting-edge components - and incredibly low costs:

- Dual high speed CAN/LIN (incl. CAN FD) channels
- Extractable 8-32 GB industrial SD card (months of data)
- Binary MDF4 log file format (extensive tool support)
- Advanced message filtering & transmit functionality
- Start/stop logging triggers based on CAN ID & databytes
- Silent mode, bit rate auto-detection, cyclic logging
- CAN/LIN error frame logging
- Data compression & encryption (e.g. for GDPR, CCPA)
- Fast boot time. Safely disconnect during use

Open source software/API - naturally

All software/APIs for the CANedge is 100% free and open source.

Data is stored in the popular MDF4 standard to enable interoperability across CAN tools and custom systems.

Convert: Simple MDF4 converters let you convert data to e.g. CSV, ASC (Vector), TRC (PEAK) - for use in your favorite tools.

Process: The asammdf GUI lets you process your data incl. DBC conversion (J1939, OBD2, ...) and graphical plots (Windows/Linux).

Automate: Easy-to-use Python APIs let you automate processing of large amounts of data (incl. quickstart library on github).

Visualize: Easily visualize data in customizable dashboards

Technical specs

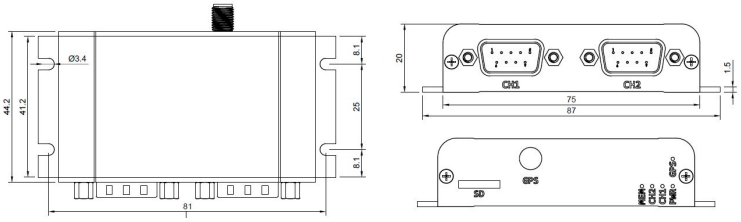
GENERAL	
Safety	CE, FCC, IC certified
Voltage tests	Transients ISO 7637-2:2011 by TÜV SÜD
Warranty	1-year warranty
Support	Free, fast & high quality support
Origin	Denmark
Software	100% free & open source
Documentation	Online/PDF documentation

CAN BUS/LIN BUS	
Channels	2 x CAN/CAN FD + 2 x LIN (master/slave)
Protocols	J1939, OBD2, CANopen, NMEA2000, FD, ...
Bit-rate	Auto-detect/simple/advanced customization

DATA LOGGING	
SD CARD	8-32 GB extractable industrial micro SDHC
Real-Time Clock	50 us resolution (incl. battery backup)
File format	MDF4 (.MF4) - easily process/convert
Safety	100% power safe
Configuration	Advanced options (filters, prescalers, compression, error frame logging, data encryption, triggers & more)

GNSS/IMU (optional)	
GNSS/IMU	Add 40+ GNSS/IMU signals (see deep dive)

MECHANICAL/SUPPLY	
Connectors	2 x DB9 (adapter cables available)
Input supply	+7V to +32V DC via Channel 1 DB9
Consumption	<1W
Dimensions	75 x 47 x 20 mm excl. antenna/flanges
Weight	100 G
LEDs	4-5 LEDs (PWR, CH1, CH2, MEM, GPS)
Temperature	-25 degC to +75 degC
IP rating	IP40



Trusted by engineers at leading OEMs

BOSCH LIEBHERR NISSAN SIEMENS JCB TOYOTA Continental HYUNDAI CERN JOHN DEERE

CSS Electronics ApS | +45 91 25 25 63 | contact@csselectronics.com | www.csselectronics.com

CANedge2

2xCAN/LIN Data Logger (SD + RTC + WiFi)

PLUG & PLAY: Log data out-the-box. Standalone. Power via CAN connector

SECURE WIFI: Push data via WiFi to your own server. Enterprise-grade security

PRO SPECS: Extractable 8-32 GB SD. 2xCAN/LIN. CAN FD. Zero data loss. 50 µs RTC

MANAGE FLEET: Easily update config/FW over-the-air across fleet. Auto-sync RTC via WiFi

COMPACT: Only 8 x 5 x 2 CM. 100G. Alu enclosure. 5 LEDs. Configure CH2 5V power out

INTEROPERABLE: Convert MDF4 to e.g. CSV, ASC, TRC. Free open source GUI/API

The plug & play 2xCAN/LIN logger records timestamped CAN data (Classical/CAN FD) to the extractable 8 GB industrial SD card.

The small device connects via WiFi access points (e.g. WLAN or 3G/4G routers) to securely push data to your server. Further, the device can be updated over-the-air. The CANedge2 is ideal for telematics & fleet management - as well as R&D field tests, diagnostics and predictive maintenance.

Software/APIs are free & open source - with no fees or lock-in.

New: The CANedge2 is now available with **optional GPS/IMU**.

Pro specs CAN logger - at half the cost

The CANedge2 combines innovative design, cutting-edge components - and incredibly low costs:

- Dual high speed CAN/LIN (incl. CAN FD) channels
- Extractable 8-32 GB industrial SD card (months of data)
- Binary MDF4 log file format (extensive tool support)
- Advanced message filtering & transmit functionality
- Start/stop logging triggers based on CAN ID & databytes
- Silent mode, bit rate auto-detection, cyclic logging
- CAN/LIN error frame logging
- Data compression & encryption (e.g. for GDPR, CCPA)
- Fast boot time. Safely disconnect during use

Open source software/API - naturally

All software/APIs for the CANedge is 100% free and open source.

Data is stored in the popular MDF4 standard to enable interoperability across CAN tools and custom systems.

Convert: Simple MDF4 converters let you convert data to e.g. CSV, ASC (Vector), TRC (PEAK) - for use in your favorite tools.

Process: The asammdf GUI lets you process your data incl. DBC conversion (J1939, OBD2, ...) and graphical plots (Windows/Linux).

Automate: Easy-to-use Python APIs let you automate processing of large amounts of data (incl. quickstart library on github).

Visualize: Easily visualize data in customizable dashboards

Technical specs

GENERAL

Safety	CE, FCC, IC certified
Voltage tests	Transients ISO 7637-2:2011 by TÜV SÜD
Warranty	1-year warranty
Support	Free, fast & high quality support
Origin	Denmark
Software	100% free & open source
Documentation	Online/PDF documentation

CAN BUS/LIN BUS

Channels	2 x CAN/CAN FD + 2 x LIN (master/slave)
Protocols	J1939, OBD2, CANopen, NMEA2000, FD, ...
Bit-rate	Auto-detect/simple/advanced customization

DATA LOGGING

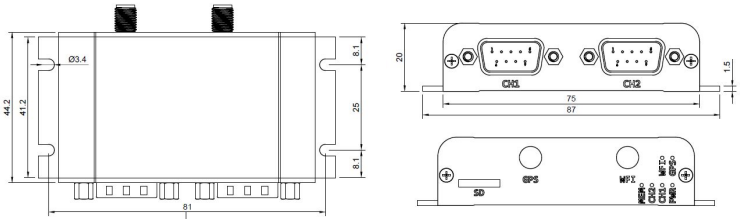
SD CARD	8-32 GB extractable industrial micro SDHC
Real-Time Clock	50 us resolution (incl. battery backup)
File format	MDF4 (.MF4) - easily process/convert
Safety	100% power safe
Configuration	Advanced options (filters, prescalers, compression, error frame logging, data encryption, triggers & more)

GNSS/IMU (optional)

GNSS/IMU	Add 40+ GNSS/IMU signals (see deep dive)

MECHANICAL/SUPPLY

Connectors	2 x DB9 (adapter cables available)
Input supply	+7V to +32V DC via Channel 1 DB9
Consumption	<1W
Dimensions	75 x 47 x 20 mm excl. antenna/flanges
Weight	100 G
LEDs	5-6 LEDs (PWR, CH1, CH2, MEM, WFI, GPS)
Temperature	-25 degC to +75 degC
IP rating	IP40



Trusted by engineers at leading OEMs

CANedge3

2xCAN/LIN Data Logger (SD + RTC + 3G/4G)

PLUG & PLAY: Log data out-the-box. Standalone. Power via CAN connector

SECURE 3G/4G: Push data via 3G/4G to your server. Update over-the-air. SIM. E2E security

PRO SPECS: Extractable 8-32 GB SD. 2xCAN/LIN. CAN FD. Zero data loss. 50 µs RTC

MANAGE FLEET: Easily update config/FW over-the-air across fleet. Auto-sync RTC via 3G/4G

COMPACT: Only 8 x 5 x 2 CM. 100G. Alu enclosure. 5 LEDs. Configure CH2 5V power out

INTEROPERABLE: Convert MDF4 to e.g. CSV, ASC, TRC. Free open source GUI/API

The plug & play 2xCAN/LIN logger records timestamped CAN data (Classical/CAN FD) to the extractable 8 GB industrial SD card.

The small device connects via 3G/4G (using your own SIM card) to securely push data to your server. Further, the device can be updated over-the-air. The CANedge3 is ideal for telematics & fleet management - as well as R&D field tests, diagnostics and predictive maintenance.

Software/APIs are free & open source - with no fees or lock-in.

New: The CANedge3 is available with **GPS/IMU**.

Pro specs CAN logger - at half the cost

The CANedge3 combines innovative design, cutting-edge components - and incredibly low costs:

- Dual high speed CAN/LIN (incl. CAN FD) channels
- Extractable 8-32 GB industrial SD card (months of data)
- Binary MDF4 log file format (extensive tool support)
- Advanced message filtering & transmit functionality
- Start/stop logging triggers based on CAN ID & databytes
- Silent mode, bit rate auto-detection, cyclic logging
- CAN/LIN error frame logging
- Data compression & encryption (e.g. for GDPR, CCPA)
- Fast boot time. Safely disconnect during use

Open source software/API - naturally

All software/APIs for the CANedge is 100% free and open source.

Data is stored in the popular MDF4 standard to enable interoperability across CAN tools and custom systems.

Convert: Simple MDF4 converters let you convert data to e.g. CSV, ASC (Vector), TRC (PEAK) - for use in your favorite tools.

Process: The asammdf GUI lets you process your data incl. DBC conversion (J1939, OBD2, ...) and graphical plots (Windows/Linux).

Automate: Easy-to-use Python APIs let you automate processing of large amounts of data (incl. quickstart library on github).

Visualize: Easily visualize data in customizable dashboards

Technical specs

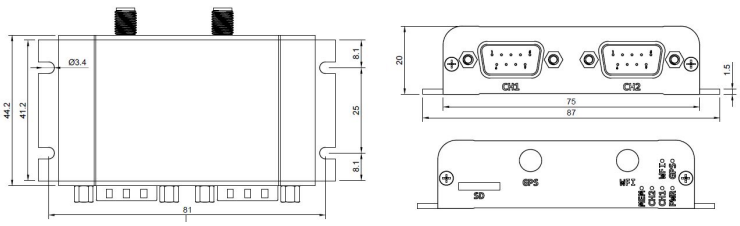
GENERAL	
Safety	CE, FCC, IC certified
Voltage tests	Transients ISO 7637-2:2011 by TÜV SÜD
Warranty	1-year warranty
Support	Free, fast & high quality support
Origin	Denmark
Software	100% free & open source
Documentation	Online/PDF documentation

CAN BUS/LIN BUS	
Channels	2 x CAN/CAN FD + 2 x LIN (master/slave)
Protocols	J1939, OBD2, CANopen, NMEA2000, FD, ...
Bit-rate	Auto-detect/simple/advanced customization

DATA LOGGING	
SD CARD	8-32 GB extractable industrial micro SDHC
Real-Time Clock	50 us resolution (incl. battery backup)
File format	MDF4 (.MF4) - easily process/convert
Safety	100% power safe
Configuration	Advanced options (filters, prescalers, compression, error frame logging, data encryption, triggers & more)

GNSS/IMU	
GNSS/IMU	Add 40+ GNSS/IMU signals (see deep dive)

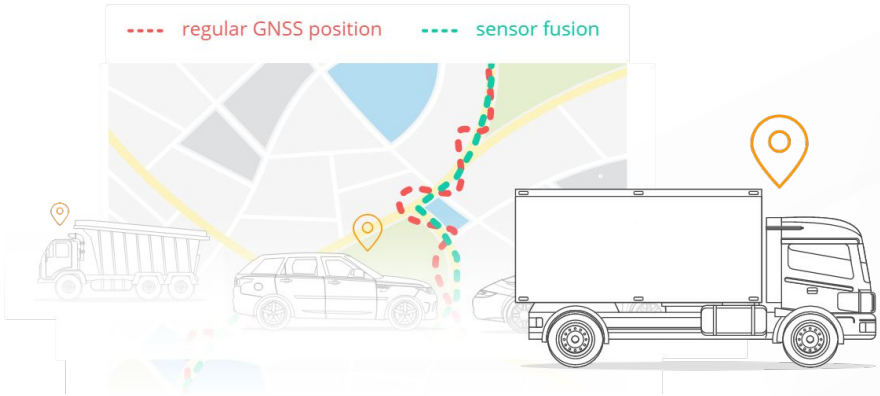
MECHANICAL/SUPPLY	
Connectors	2 x DB9 (adapter cables available)
Input supply	+7V to +32V DC via Channel 1 DB9
Consumption	<1W
Dimensions	75 x 47 x 20 mm excl. antenna/flanges
Weight	100 G
LEDs	6 LEDs (PWR, CH1, CH2, MEM, LTE, GPS)
Temperature	-25 degC to +75 degC
IP rating	IP40



Trusted by engineers at leading OEMs

BOSCH LIEBHERR NISSAN SIEMENS JCB TOYOTA Continental HYUNDAI CERN JOHN DEERE

CANedge - GNSS/IMU details



GPS & 3D IMU - enrich your data

The CANedge comes with optional cutting-edge GNSS/IMU functionality - perfect for use cases like vehicle telematics.

- Add 40+ GNSS/IMU signals to your 2 x CAN/LIN data
- Full GNSS support (GPS, Galileo, BeiDou, GLONASS)
- Built-in gyroscope (roll, pitch, yaw) + accelerometer (X, Y, Z)
- High precision via sensor fusion incl. in GNSS hostile areas
- Signals encoded as 'internal' CAN data (separate channel)
- Configurable message inclusion/frequency (up to 5 Hz)
- DBC file for easy decoding to human-readable form
- Optionally sync the CANedge RTC via precise GNSS time
- Flexible deployment via external GNSS antenna

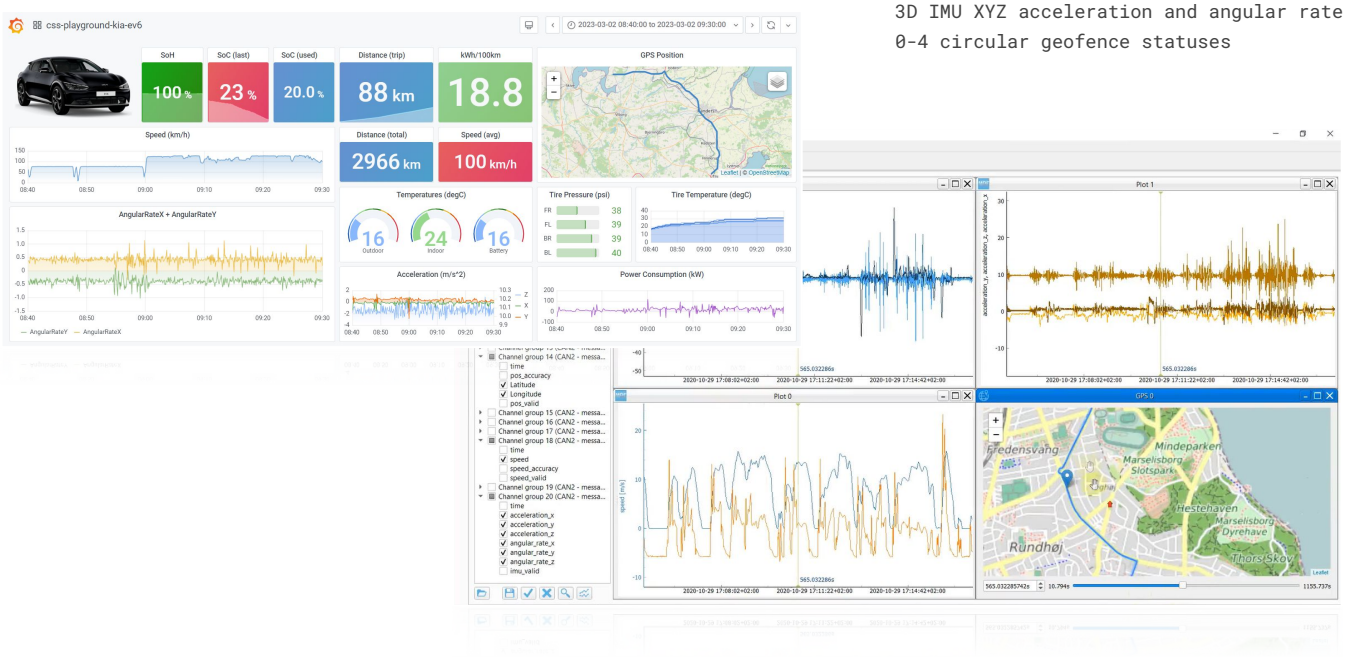
Technical specs (GPS & IMU)

GNSS & 3D IMU

Module	u-blox NEO-M9V sensor module with built-in gyroscope and accelerometer
GNSS	Combine GPS, Galileo, BeiDou, GLONASS
Sensor Fusion (UDR)	Up to 3x better accuracy incl. in GNSS hostile areas (tunnels, urban)
CAN encoding	GNSS/IMU data output on 'internal' pseudo CAN channel encoded as CAN frames [5 Hz]
Configurable	Filter/prescale internal GNSS/IMU data
Antenna	GPS antenna as option (required for GNSS)

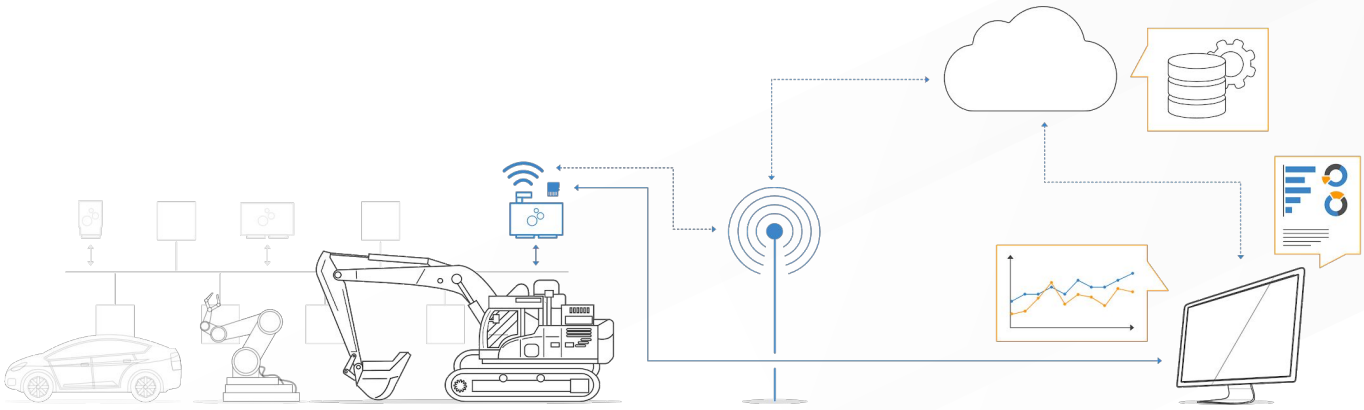
Signals

Position: Longitude and latitude
Time: High precision timestamp
Status: Fix type + satellite count
Speed: Travel speed in m/s and km/h
Altitude: Altitude in meters
Roll, pitch, heading [automotives only]
Distance traveled (since power on)
3D IMU XYZ acceleration and angular rate
0-4 circular geofence statuses



The CANedge units with GNSS/IMU are ideal for vehicle telematics, diagnostics and analysis - with easy visualization in e.g. Grafana or asammdf

CANedge2/CANedge3 details



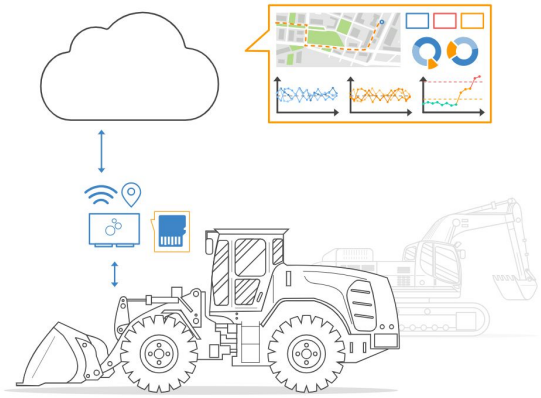
Reinventing telematics

Today, fleet telematics is vital across vehicles and machines.

However, end users face challenges: Expensive subscriptions. No data ownership. Security concerns. Vendor lock-in.

The CANedge2/CANedge3 provides a modern alternative:

- Upload data to your own local/dedicated/cloud server
- Add WiFi/SIM & server details to the config
- Log data to the SD. Auto-upload when connectivity
- Configure file splits to control upload frequency
- 100% secure: HTTPS, credential encryption and more
- Manage via over-the-air updates (FW & config)
- Zero fees. Zero lock-in. Interoperable. Open source tools



Technical specs (WiFi, 3G/4G, server)

WIFI or 3G/4G	
Transfer modes	Auto-push data to server from SD
Protocol	HTTP/HTTPS for fast, secure transfer
CANedge2 WiFi	Add 1-4 WiFi prioritized access points
CANedge3 LTE	Add your own micro SIM card
Over-The-Air	Configurable OTA firmware/config updates
WiFi Heartbeat	Device optionally sends periodic status
LAN Standard	IEEE 802.11 b / g / n (CANedge2)
Antenna	External (SMA)
Server Interface	S3 REST - Use with AWS, MinIO, Azure, ...

SECURITY	
HTTPS	Data + OTA updates optionally via TLS 1.2
WPA/WPA2	Supports WPA/WPA2
Credentials	Optionally encrypt WiFi/SIM/S3 passwords
Firmware	All firmware updates are digitally signed
User Access	Manage user access via S3 policies

Easily manage data/devices on your server

The device uploads data to an S3 server - which makes it easy to manage your server files via any S3-compatible tools or SDKs.

Further, the 100% optional CANcloud tool lets you manage your S3 server devices & data via your browser:

- Host yourself - or simply log into your server via our link
- Monitor device status across your fleet via dashboard
- Browse, download, share & delete uploaded log files
- Easily update config/firmware over-the-air
- Browser based (works on all OS & devices)
- 100% free and open source - easy to customize

The CANcloud status dashboard lets you monitor device data upload and their status (SD %left, firmware version, config status)

Trusted by engineers at leading OEMs

CANmod.gps

GPS-to-CAN with 3D Inertial Sensor & UDR

PLUG & PLAY: Standalone - no PC required. Integrate with any CAN bus to add GNSS/IMU data. DBC included

INERTIAL DATA: Built-in gyroscope (roll, pitch, yaw) and accelerometer (X, Y, Z). 100 Hz frequency

COMPACT: Only 7 x 2 x 5 CM. 70G. Alu enclosure. 4 LEDs. 5-26 V DC via DB9. USB for config/firmware/stream

SENSOR FUSION: High precision position and attitude data via sensor fusion of the GNSS/IMU

USE GLOBALLY: 1 Hz GNSS position. Hot start via battery backup. GPS, Galileo, BeiDou, GLONASS

CONFIGURABLE: Configure CAN IDs, bit rate, data frequency & geofences via JSON config and GUI

This standalone GPS-to-CAN module produces GNSS position and 3D inertial data (via a gyroscope & accelerometer) and outputs it via configurable CAN bus frames.

The module supports 'Untethered Dead Reckoning' - meaning that even if the GNSS signal is lost entirely, the module can deliver continuous positioning through IMU-based estimates (no external inputs required).

You can integrate the module with any CAN bus, e.g. vehicle networks or CAN hardware. As an example, you can use it as a plug & play add-on module for the CANedge.

Incl. antenna. Optional adapters (dropdown).

Easily add GNSS/IMU data to any CAN bus

The CANmod.gps makes it easy to add position and 3D inertial data to your CAN bus - e.g. for use by ECUs or CAN hardware.

- Compatible with any high speed CAN bus (2.0A, 2.0B)
- Fully configure CAN IDs, bit rate and message frequency
- Power at 5-26 V DC via standard DB9 adapter cables
- Optionally record data via any CAN interface/logger/...
- Ex: Use as add-on for the CANedge (power via 2nd port)
- DBC included for decoding to human-readable form
- Optionally stream sensor data via USB in real-time
- White label e.g. for inclusion in your production

Technical specs

GENERAL

Safety	CE, FCC, IC certified
Functionality	The device produces GNSS/IMU data and outputs it via CAN bus and/or USB
Warranty	1-year warranty
Support	Free, fast & high quality support
Origin	Denmark
Software	100% free & open source
Documentation	Online/PDF documentation

CAN BUS

Channels	1 x CAN/CAN FD
CAN IDs	Fully configurable (CAN 2.0A/2.0B)
Bit-rate	Fully configurable (up to 1 Mbit/s)

SENSOR GNSS/IMU

Module	NEO M8U (GNSS + gyro + accelerometer)
Sensor Fusion	Enhanced precision in GNSS hostile areas
Accuracy	Position: 2.5 m CEP Heading: 1 degree (50% at 30 m/s) Velocity: 0.05 m/s (50% at 30 m/s)
Battery Backup	Battery enables 'aided starts' (3s)

SIGNALS

CAN Signals	Position: Longitude & latitude [1 Hz] Time: Precise epoch timestamp [1 Hz] GNSS status and satellite count [1 Hz] Speed: Travel speed in m/s [1 Hz] Altitude: Altitude in meters [1 Hz] Attitude: Roll, pitch, heading [1 Hz] Distance since power on and total [1 Hz] Geofences: Status of geofences [1 Hz] IMU:Angular/acceleration rates [100 Hz]

MECHANICAL/SUPPLY

Connectors	1 x DB9 (adapter cables available)
Input supply	+5V to +26V DC via DB9 (pin 1 or 9)
Consumption	<1W
Dimensions	52.5 x 70.0 x 24.5 mm (L x W x H)
Weight	70 G
LEDs	4 external LEDs (PWR, CAN, MEM, SIG)
Temperature	-25 degC to +70 degC
IP rating	IP40
Antenna	u-blox ANN-MS-0-005 (magnetic base, 5m)

Trusted by engineers at leading OEMs

CANmod.temp

4 x Thermocouple-to-CAN [Temperature Sensor]

PLUG & PLAY: Standalone - no PC required. Integrate with any CAN to add temperature data. DBC included

PRECISE: High accuracy with cold-junction compensation & line frequency filtering. Fault detection

COMPACT: Only 7 x 2 x 5 CM. 80G. Alu enclosure. 3 LEDs. 5-26 V DC via DB9. USB for config/FW/stream

USE ANY TYPE: Configure each thermocouple channel type (B, E, J, K, N, R, S, T)

THERMOCOUPLES: Output data from 4 thermocouples at 5 Hz. High signal resolution of 0.01 degC

CONFIGURABLE: Configure CAN IDs, bit rate, message frequencies and more via JSON config and GUI

This thermocouple-to-CAN module produces temperature data from 4 thermocouple sensors and outputs it via CAN bus (standalone) or USB.

The module supports all thermocouple types (B, E, J, K, N, R, S, T) and can be easily daisy-chained.

You can integrate the module with any CAN bus system - e.g. to provide temperature data for your ECUs or existing CAN hardware.

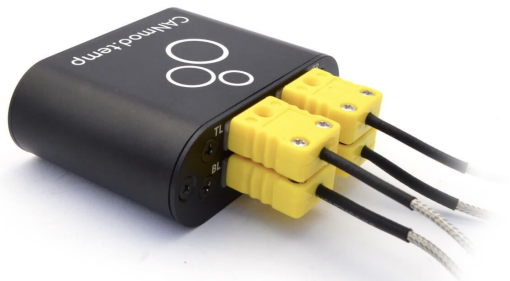
As an example, you can use the module as an add-on for the CANedge - ideal for e.g. engine dynos, EGT analysis or temperature telematics.

Probes not included.

Add temperature data to any CAN bus

Add temperature sensor data via 4 thermocouples to your CAN bus - e.g. for use by ECUs or CAN hardware.

- Compatible with any high speed CAN bus (2.0A, 2.0B)
- Fully configure CAN IDs, bit rate and message frequency
- Daisy-chain multiple modules for 8, 12, 16, ... channels
- Power at 5-26 V DC via standard DB9 adapter cables
- Optionally record data via any CAN interface/logger/...
- Ex: Use as add-on for the CANedge (power via 2nd port)
- DBC included for decoding to human-readable form
- Optionally stream sensor data via USB in real-time
- White label e.g. for inclusion in your production



Example: Log/stream sensor data

The CANmod.temp is often used as an 'add-on' for the CANedge. This setup lets you record e.g. vehicle data via Channel 1 and temperature data via Channel 2. The data can be easily DBC decoded via e.g. the asammdf GUI, Python or MATLAB.

You can also stream the sensor data in real-time via USB using SavvyCAN to view raw/decoded data (e.g. via plots) - ideal for validating your setup pre-deployment or for lab testing.

Technical specs

GENERAL	
Safety	CE, FCC, IC certified
Functionality	Supports 4 x thermocouples for outputting temperature data via CAN and/or USB
Warranty	1-year warranty
Support	Free, fast & high quality support
Origin	Denmark
Software	100% free & open source
Documentation	Online/PDF documentation

CAN BUS	
Channels	1 x CAN/CAN FD
CAN IDs	Fully configurable (CAN 2.0A/2.0B)
Bit-rate	Fully configurable (up to 1 Mbit/s)

SENSOR TEMPERATURE	
Module	Pro specs MAX31856MUD+
Channels	4 thermocouple sensors/channels
Temperature	Supports -210°C to +1800°C
Formatting	DegC via automated edge linearization
Accuracy	Lab-grade with cold junction compensation
Resolution	Up to 0.01 degC resolution
Other	Line frequency filtering, fault detection, overvoltage protection

SIGNALS	
CAN Signals	Probe temperatures in degC [5 Hz] Ambient temperature [5 Hz] Faults: Status of each sensor [5 Hz]

MECHANICAL/SUPPLY	
Connectors	1 x DB9 (adapter cables available)
Input supply	+5V to +26V DC via DB9 (pin 1 or 9)
Consumption	<1W
Dimensions	52.5 x 70.0 x 24.5 mm (L x W x H)
Weight	70 G
LEDs	3 external LEDs (PWR, CAN, MEM)
Temperature	-25 degC to +70 degC
IP rating	IP40

Trusted by engineers at leading OEMs

CANmod.input

Analog/Digital/Pulse to CAN Bus Converter

PLUG & PLAY: Standalone - no PC required. Integrate with any CAN bus to add input sensor data. DBC included

+DIGITAL: Digital input reading of each channel. 1 kHz. Configure low/high/hysteresis

COMPACT: 7 x 2 x 5 CM. 70G. 8 LEDs. 5-26 V DC via DB9. 3.3V excitation signals. USB for config/FW/stream

+PULSE: Pulse input reading of each channel. 16 kHz. Frequency or counter mode (up to 32 bit)

8 X ANALOG: 8 analog input channels (1 kHz, 10 bit). Configurable voltage ranges (0-0.625V to 0-10V)

CONFIGURABLE: Configure sensors, CAN IDs, bit rate, frequencies and more via JSON config and GUI

This sensor-to-CAN module produces analog, digital & pulse measurements from 8 input channels - and outputs the data via CAN bus. The module is 100% standalone, no PC required.

The compact device offers pro specs incl. high accuracy and high-frequency sampling - as well as configurable input ranges and digital thresholds.

The module integrates with any CAN bus to provide data for ECUs or CAN tools. For example, you can use the module as an add-on for the CANedge.

Optionally add the DB25-to-input adapter.

Easily add analog/digital/pulse data to any CAN bus system

Add analog/digital/pulse data via 8 input channels to your CAN bus - e.g. for use by ECUs or CAN hardware.

- Powerful parallel sampling of analog/digital/pulse signals
- Configure input range for optimal resolution/amplification
- Configure digital high/low levels incl. optional hysteresis
- Quickly connect sensors via DB25-input adapter cable
- Optionally output signals via CAN FD for fewer frames
- Daisy-chain multiple modules for 16, 24, 32, ... channels
- Dedicated excitation signal for input sensors (~3.3 V)
- Power device at 5-26 V DC via standard DB9 adapter cables
- Optionally record the data via any CAN interface/logger/...
- DBC included for easy decoding to human-readable form
- Optionally stream sensor data via USB in real-time

Example: Log/stream sensor data

The CANmod.input is often used as an 'add-on' for the CANedge. This setup lets you record e.g. vehicle data via Channel 1 and analog/digital/pulse data via Channel 2. The data can be easily DBC decoded via e.g. the asammdf GUI, Python or MATLAB.

You can also stream the sensor data in real-time via USB using SavvyCAN to view raw/decoded data (e.g. via plots) - ideal for validating your setup pre-deployment or for lab testing.

Technical specs

GENERAL

Safety	CE, FCC, IC certified
Functionality	Pproduces analog/digital/pulse data from 8 input sensors and outputs via CAN/USB
Warranty	1-year warranty
Support	Free, fast & high quality support
Origin	Denmark
Software	100% free & open source
Documentation	Online/PDF documentation

CAN BUS

Channels	1 x CAN/CAN FD
CAN IDs	Fully configurable (CAN 2.0A/2.0B)
Bit-rate	Fully configurable (up to 1 Mbit/s)

SENSOR TEMPERATURE

Channels	Supports 8 input channels
Types	Supports analog, digital, pulse sensors
Input Range	0-10V, 0-5V, 0-2.5V, 0-1.25V, 0-0.625V *Ranges are locked in channel pairs of two*
Resolution	10 bit
Thresholds	Digital high/low switch thresholds (0-100%) incl. dead-zone/hysteresis
Pulse Modes	Pulse inputs can be measured as frequencies or counters (up to 32 bit)

SIGNALS

CAN Signals	Analog output in millivolt (mV) [1000 Hz] Digital output as 'actual' (dead-zone, low, high) and 'low'/'high' [1000 Hz] Pulse output as a frequency/counter value (for reset/accumulate mode) [1000 Hz]

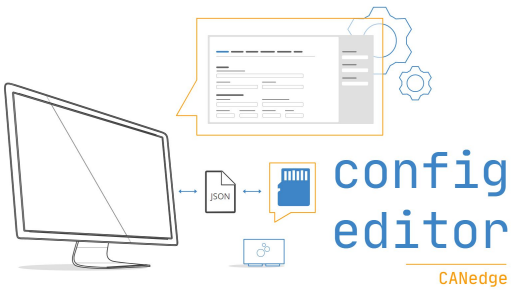
MECHANICAL/SUPPLY

Connectors	1 x DB9 (adapter cables available)
Input supply	+5V to +26V DC via DB9 (pin 1 or 9)
Consumpti	<1W
Dimensions	52.5 x 70.0 x 24.5 mm (L x W x H)
Weight	70 G
LEDs	8 external LEDs (PWR, CAN, MEM, CH1/CH2, CH3/CH4, CH5/CH6, CH7/CH8)
Temperature	-25 degC to +70 degC
IP rating	IP40

Trusted by engineers at leading OEMs

CANedge software/API tools

100% free and open source, naturally



Easily configure your device

The CANedge/CANmod JSON config can be modified via a GUI editor - either online via browser or offline (e.g. from the SD).

- GUI editor for user-friendly configuration
- Optionally edit your config directly in e.g. Notepad++
- Batch tool available for large-scale configuration OTA

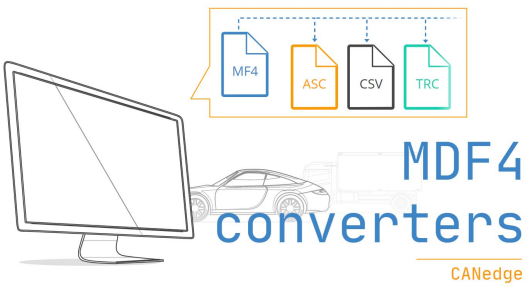
Learn more

Load data in your favorite tools

Simple MDF4 converters let you convert data to e.g. CSV, ASC (Vector), TRC (PEAK) - for use in your favorite tools.

- Drag & drop files/folders onto the converter to process
- Optionally use via the CLI or in scripts for automation
- Decompress/decrypt as part of conversion
- Works on both Windows/Linux

Learn more



DBC convert & plot your data via GUI/API

The free asammdf GUI/API lets you process your data:

- DBC convert data to physical values (incl. J1939, OBD2)
- Easily create advanced graphical plots
- Resample or concatenate your data
- GUI executable for Windows/Linux (no installation)
- Powerful Python API for big data automation

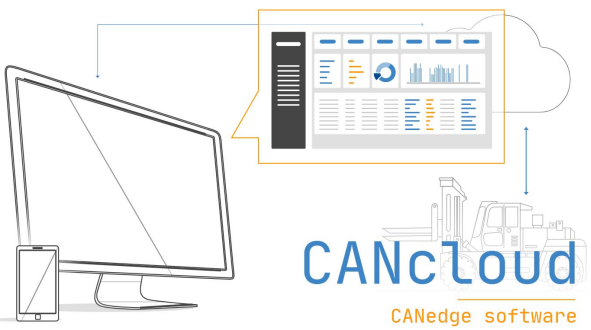
Learn more

Manage your server devices & data

CANcloud is a simple browser tool that lets you manage your S3 server devices & data from any PC/tablet with no installation.

- Host yourself - or simply log into your server here
- Monitor device status across your entire fleet
- Browse, download, share & delete uploaded log files
- Easily update config/firmware over-the-air
- Browser based (works on all OS & devices)

Learn more

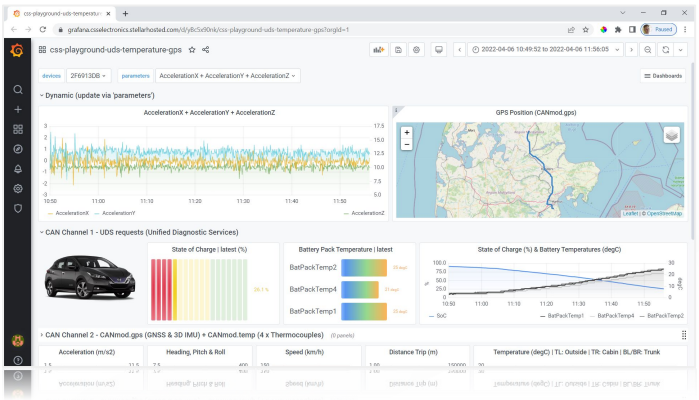


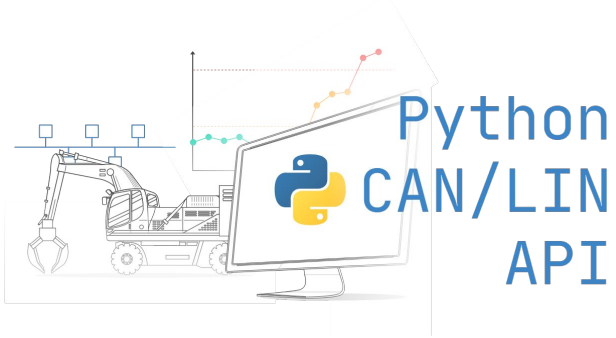
Trusted by engineers at leading OEMs

CSS Electronics ApS | +45 91 25 25 63 | contact@csselectronics.com | www.csselectronics.com

CANedge software/API tools

100% free and open source, naturally





Automate your data processing

Need to automate your CAN bus data processing via Python? The free Python API enables easy listing, loading and DBC decoding of your data - from local disk or your server.

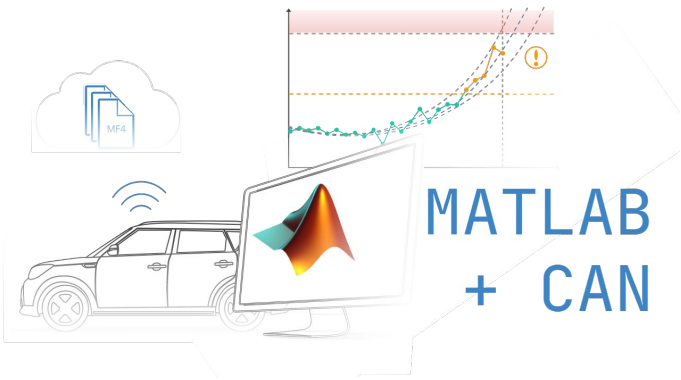
Learn more

Visualize your data in dashboards

With our plug & play dashboard integrations, you can quickly get your CAN/LIN data integrated with custom Grafana dashboards.

Perfect for presenting specific views e.g. for internal sharing, diagnostics - or as services towards clients.

Learn more



Easily load data in MATLAB

MF4 data from the CANedge can be natively loaded via MATLAB's Vehicle Network Toolbox - or converted to compatible MAT/CSV.

This makes it simple to continue using MATLAB for end users that are familiar with this tool from other projects.

Learn more

Trusted by engineers at leading OEMs

CSS Electronics ApS | +45 91 25 25 63 | contact@csselectronics.com | www.csselectronics.com