

# 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());  
        end  
    end  
endmodule
```

```

        $display("awaddr = %0d      awwsize = %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.

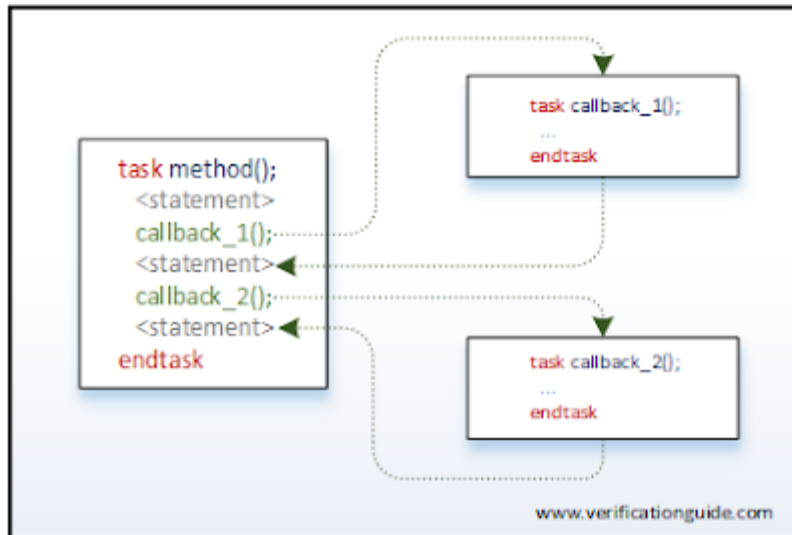
- Connection in AXI4:

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,

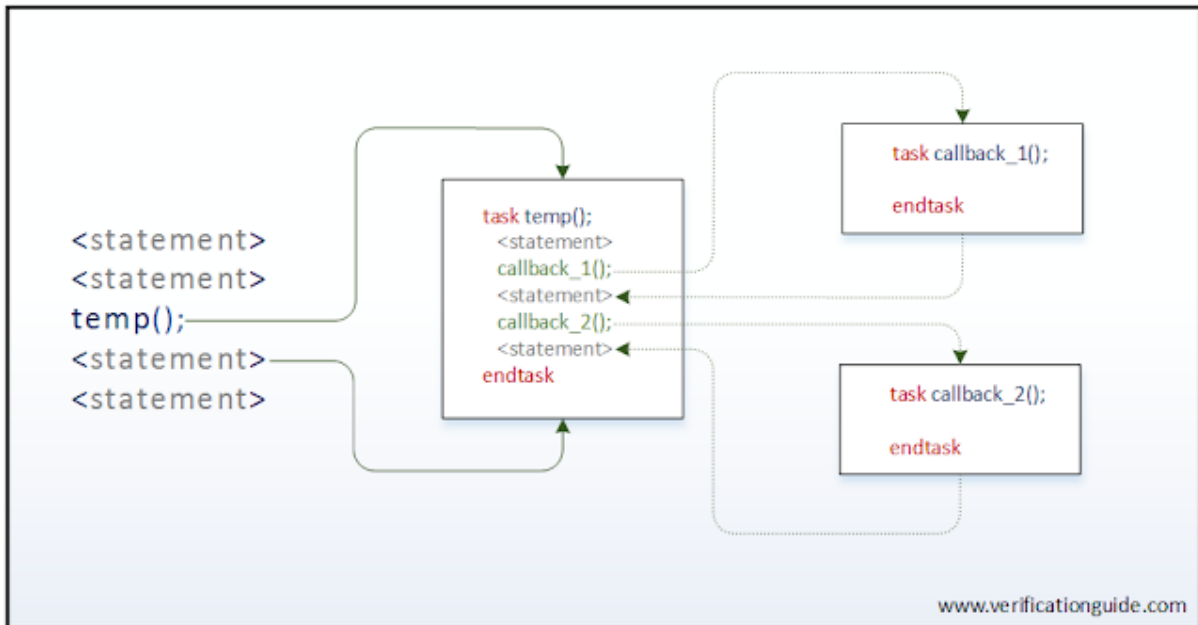
**Dummy methods are callback methods**

**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

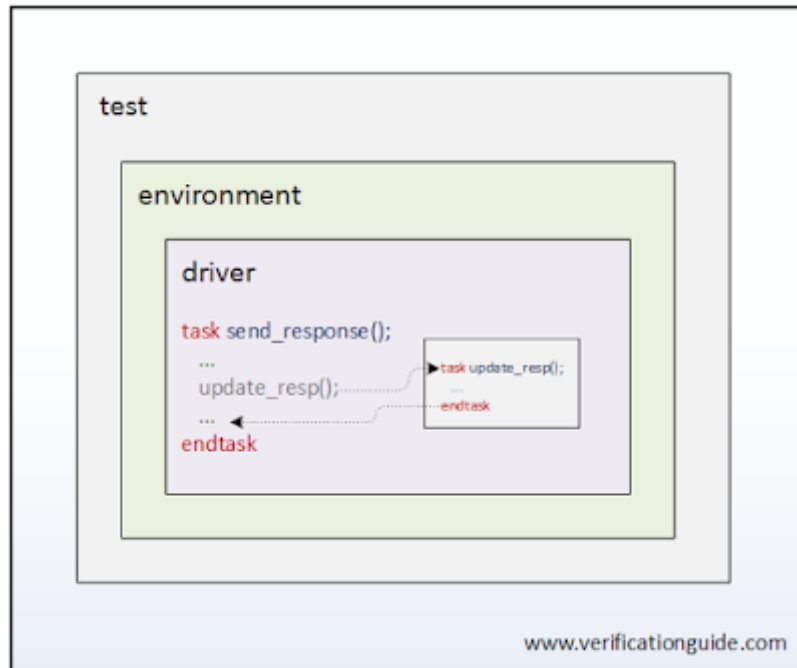
Callback example

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.**

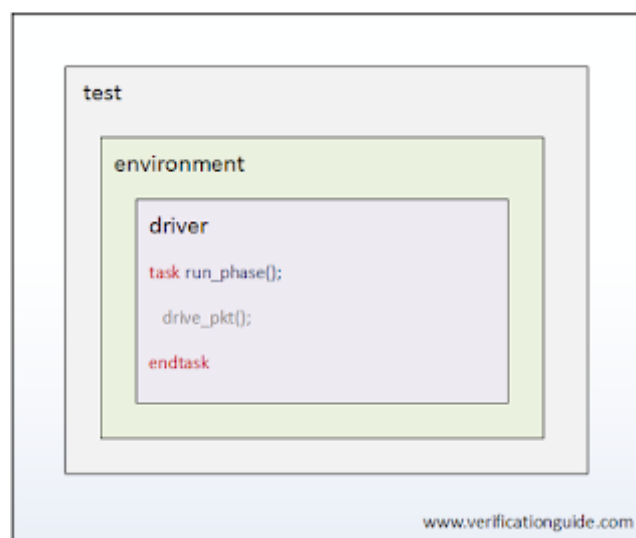
Only the required callback methods are explained here.

Callback Example

With a simple driver example will see to UVM Callback.

Below are components of example,

- Driver
- Environment
- Test

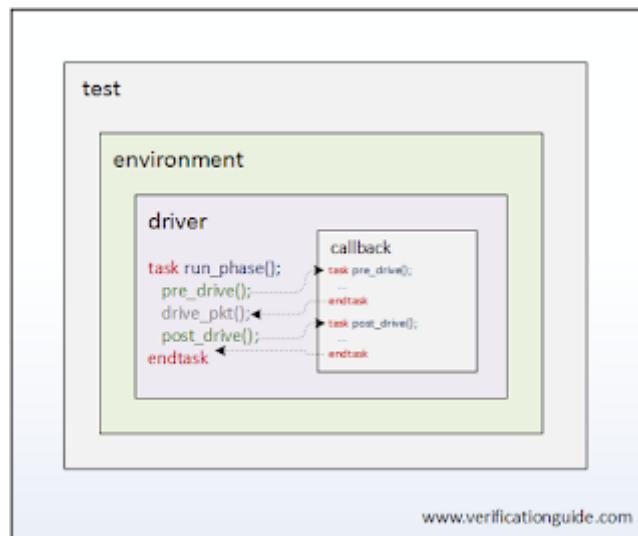


UVM testbench without callback

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
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**

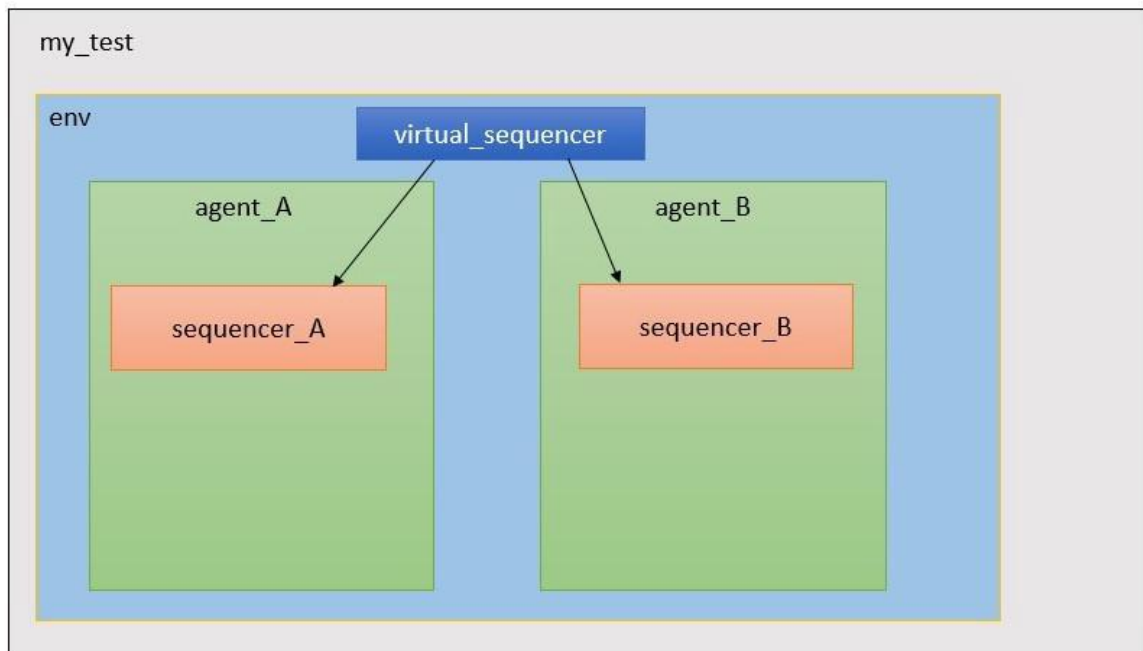
In SOC, there could be different modules that interact with different protocols. So, we need different drivers to drive corresponding interfaces. So we usually keep separate agents to handle the different protocols. So, we need to execute sequences on corresponding sequencers.

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.**



www.vlsiverify.com

### 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.driv\_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.driv\_B [core\_B\_driver] Driving  
from core B

### With virtual sequence and virtual sequencer using p\_sequencer 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.driv_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.driv_B [core_B_driver] Driving  
from core B
```

## With virtual sequence and virtual sequencer but without using p\_sequencer 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.driv_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

```

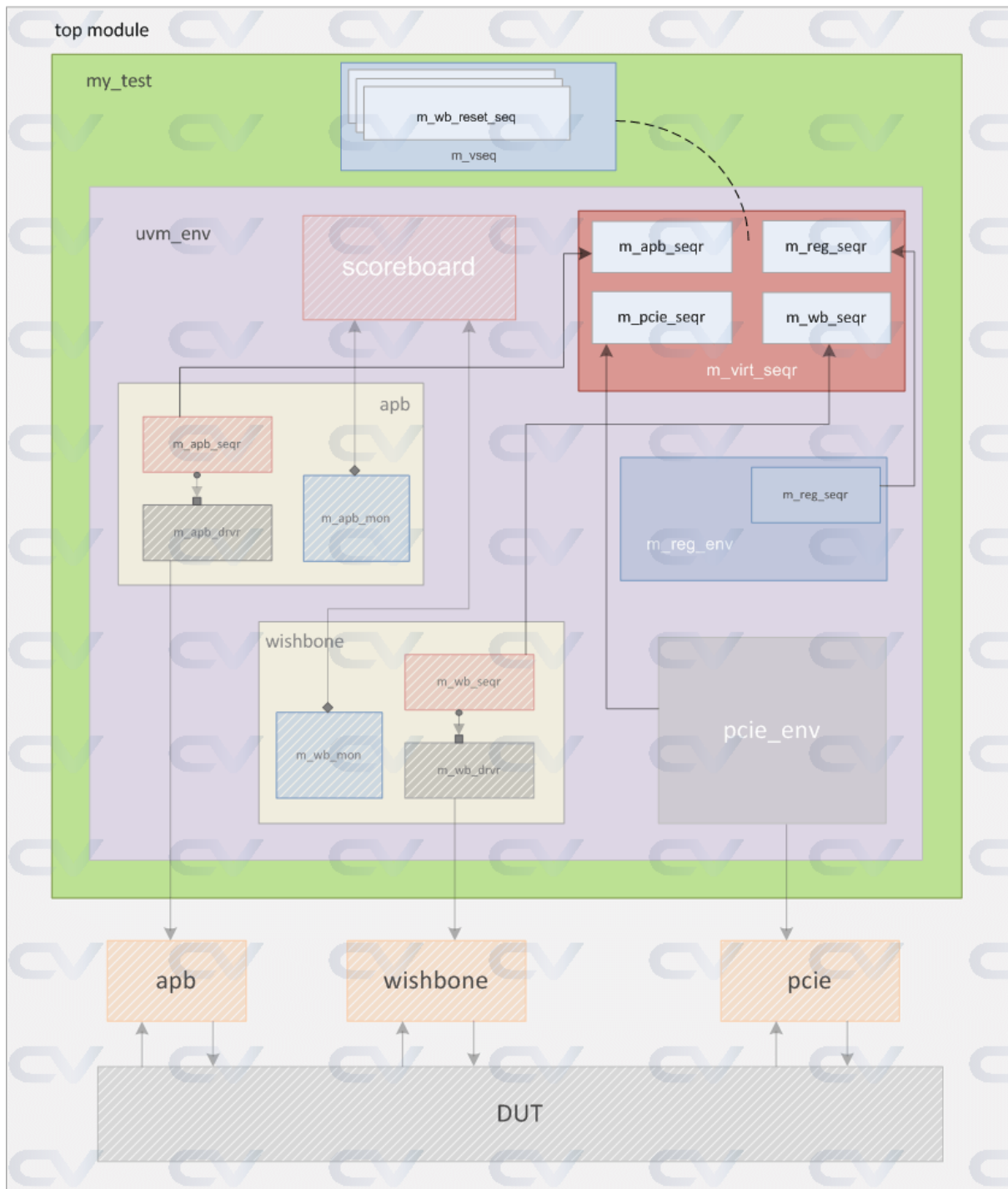
```

UVM_INFO driver.sv(55) @ 50: uvm_test_top.env_o.agt_B.driv_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 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);

```

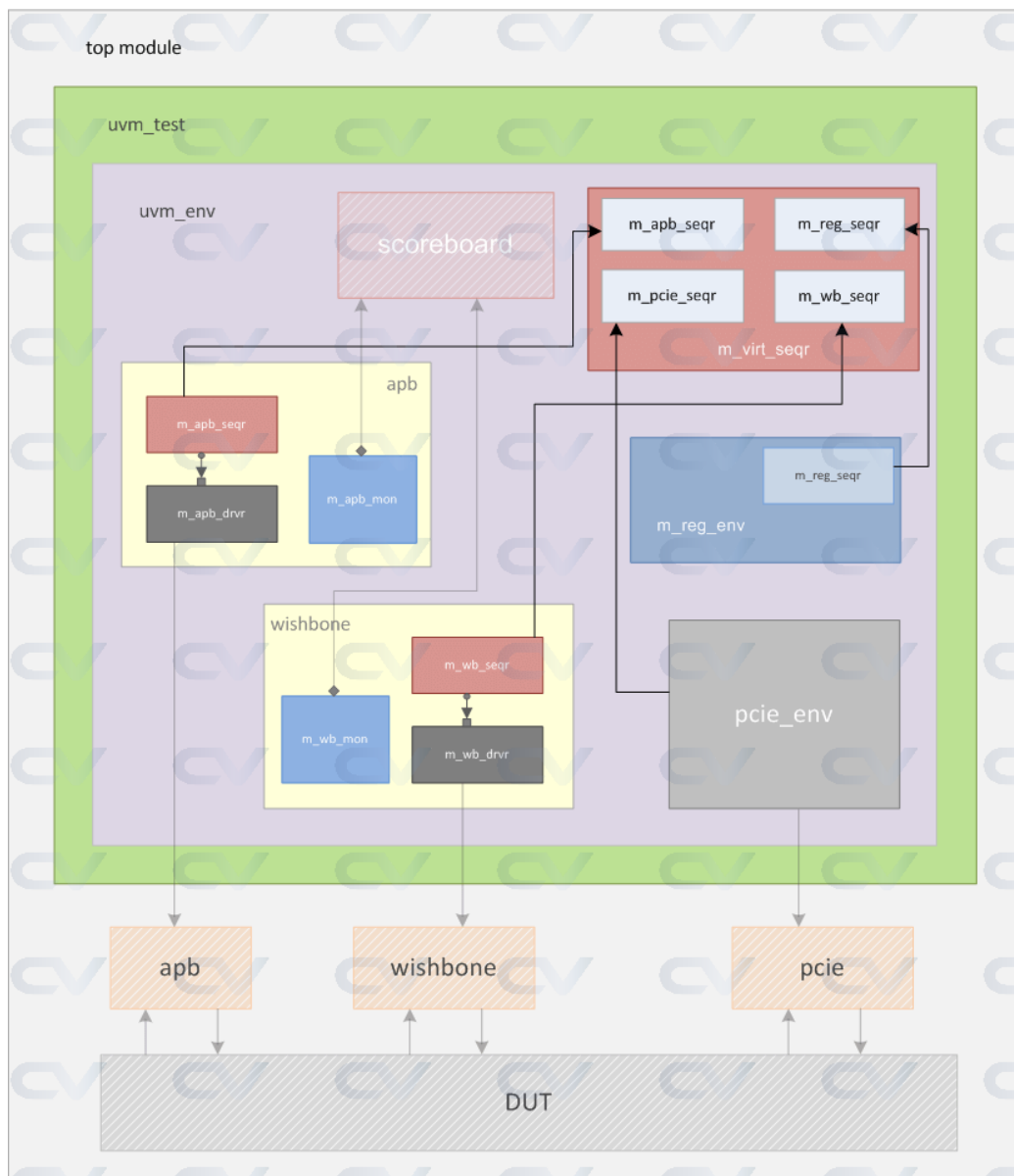
endtask

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 contains 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
endclass
```

```

...
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

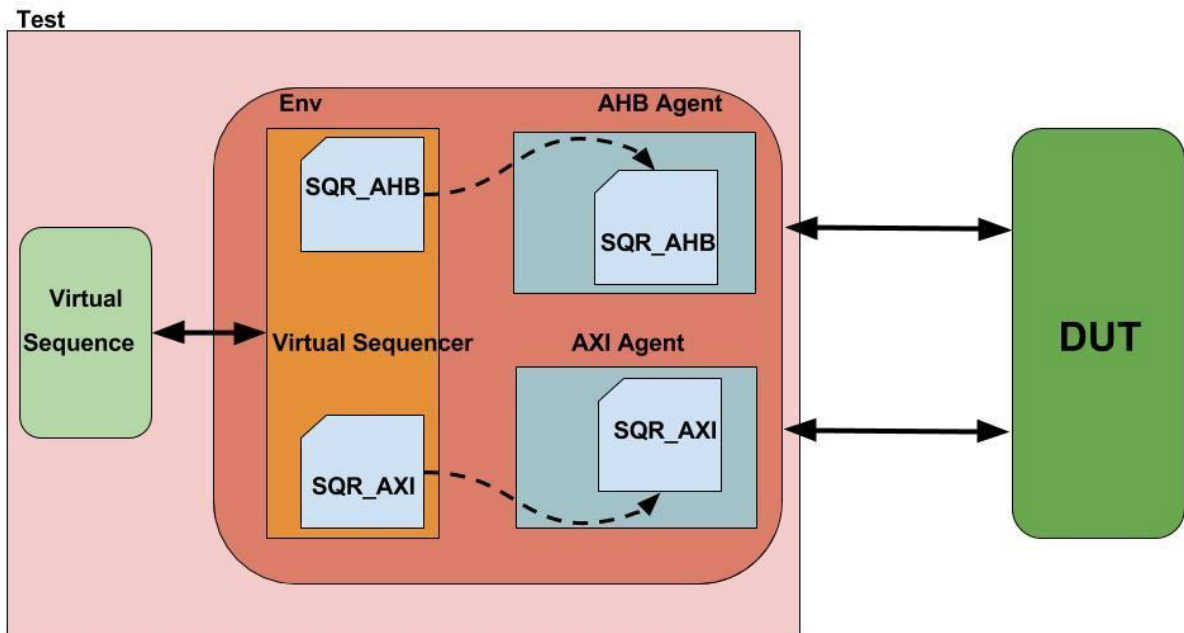
UVM\_INFO ./tb/spi\_agent.sv(74) @ 30000:  
uvm\_test\_top.m\_top\_env.m\_spi\_agent.m\_spi\_seqr@@m\_spi\_tx\_seq [tx\_SEQ] Starting  
spi\_tx\_seq

UVM\_INFO ./tb/my\_pkg.sv(75) @ 30000:

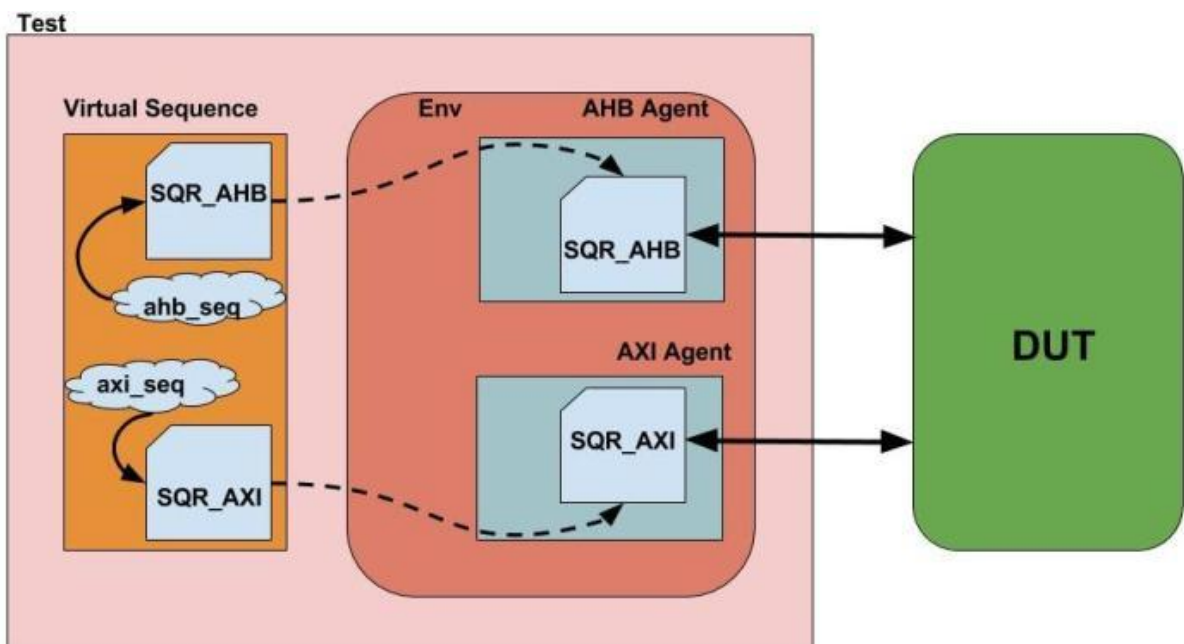
uvm\_test\_top.m\_top\_env.m\_virt\_seq@@m\_virt\_seq [VSEQ] End of virtual sequence

UVM\_INFO ./tb/my\_pkg.sv(108) @ 30000: uvm\_test\_top [SHUT] Shutting down test ...

--- UVM Report catcher Summary ---



Virtual Sequence and Virtual Sequencer



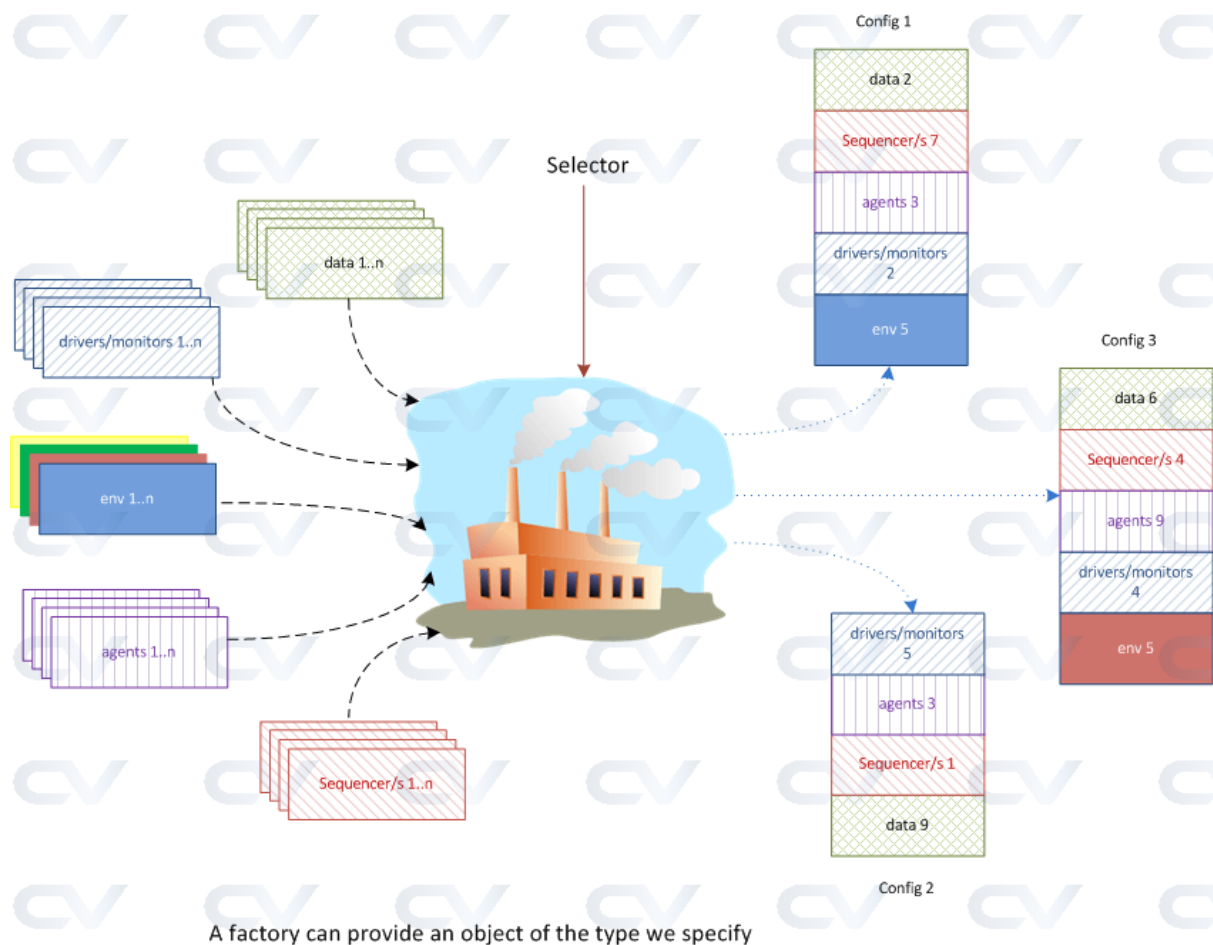
Virtual Sequence only

## 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*.



### 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

```

```

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 overridden. 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();
```

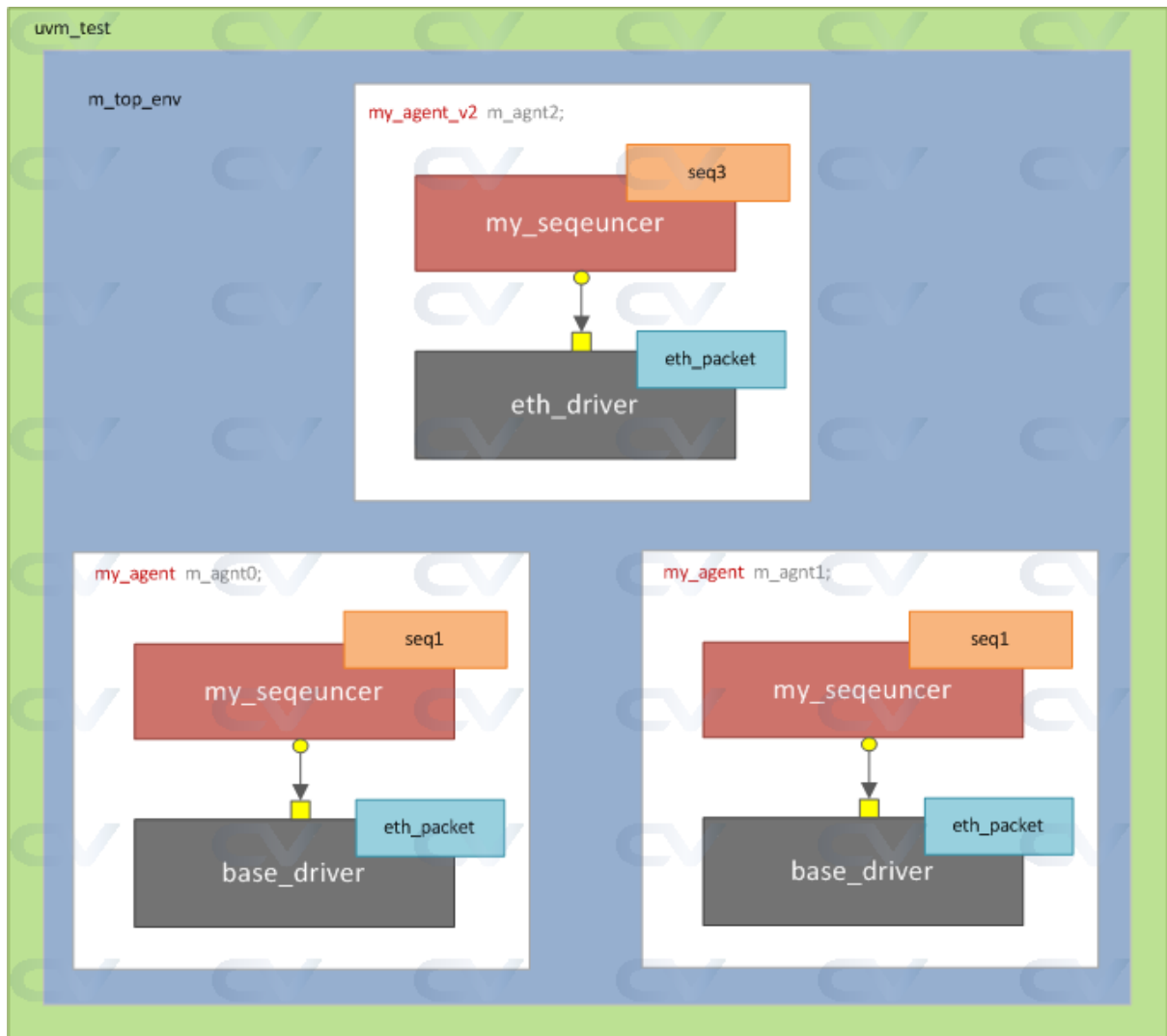
```
    ...
```

```
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\_sequeuncer

#### - Sequences

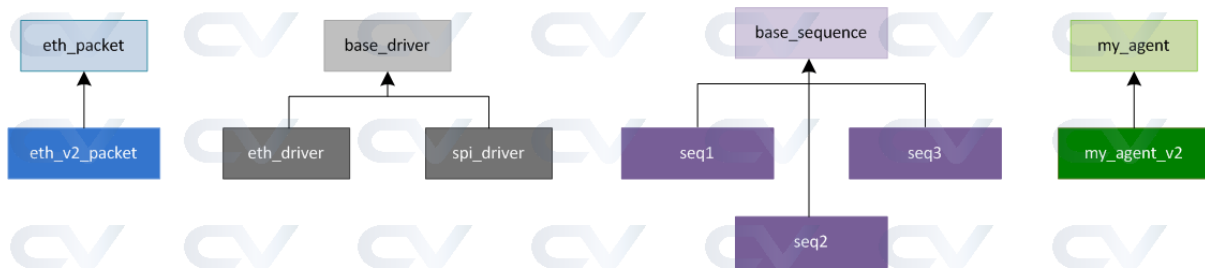
- base\_sequence (Base)

1. seq1 (child)
2. seq2 (child)
3. seq3 (child)

- Data

- eth\_packet (Base)
- 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);

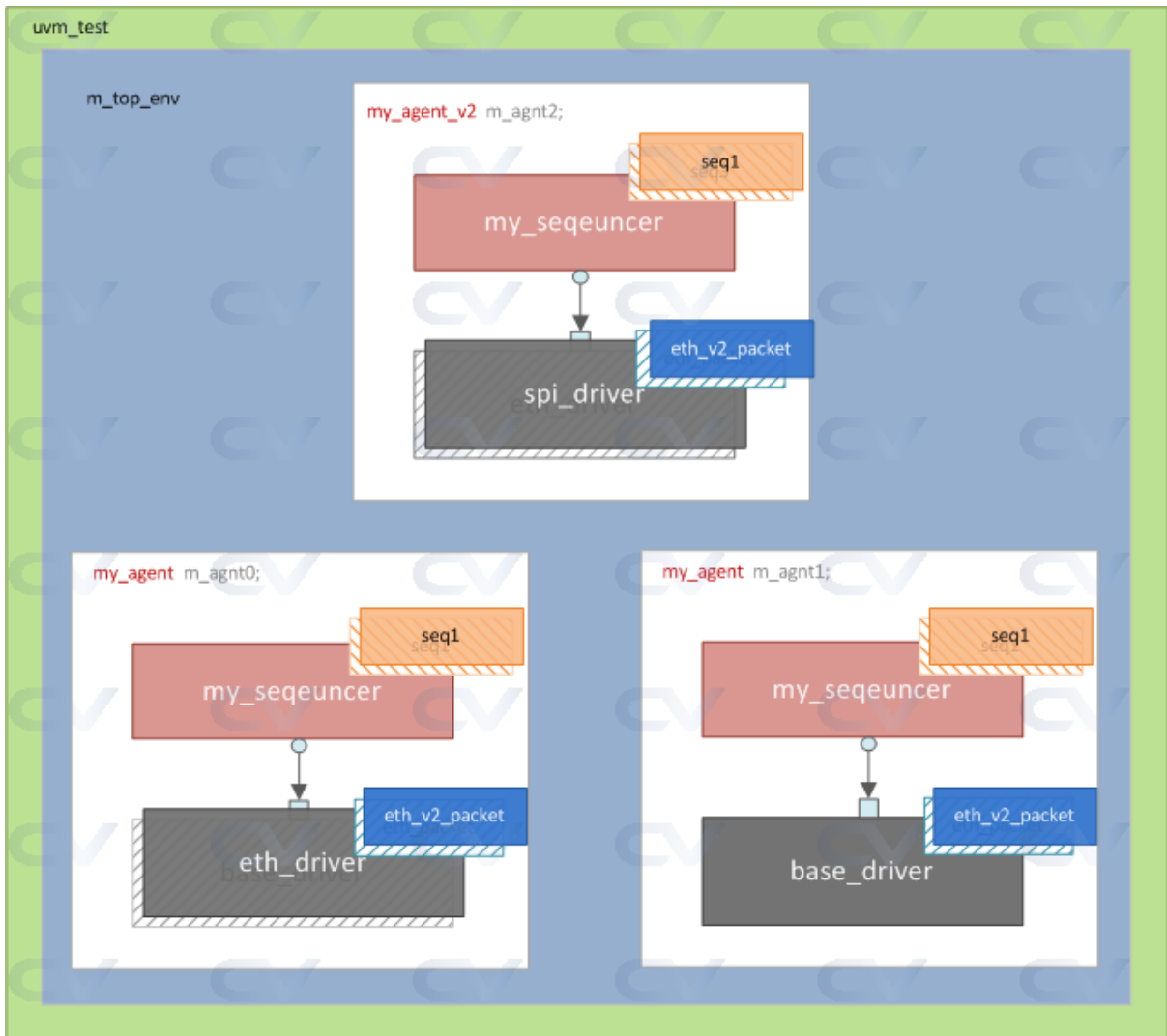
```

```
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);

`ifndef 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

`ifndef 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

`ifndef 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
```

```
    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] );	<i>index</i> 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_10 [string];
```

```
    initial begin
```

```

fruits_10 = { "apple" : 4,
             "orange" : 10,
             "plum" : 9,
             "guava" : 1 };

// size() : Print the number of items in the given dynamic array
$display ("fruits_10.size() = %0d", fruits_10.size());

// num() : Another function to print number of items in given array
$display ("fruits_10.num() = %0d", fruits_10.num());

// exists() : Check if a particular key exists in this dynamic array
if (fruits_10.exists ("orange"))
    $display ("Found %0d orange !", fruits_10["orange"]);

if (!fruits_10.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_10.first (f))

```

```
    $display ("fruits_I0.first [%s] = %0d", f, fruits_I0[f]);  
end
```

```
// last() : Get the last element in the array
```

```
begin  
    string f;  
    if (fruits_I0.last (f))  
        $display ("fruits_I0.last [%s] = %0d", f, fruits_I0[f]);  
end
```

```
// prev() : Get the previous element in the array
```

```
begin  
    string f = "orange";  
    if (fruits_I0.prev (f))  
        $display ("fruits_I0.prev [%s] = %0d", f, fruits_I0[f]);  
end
```

```
// next() : Get the next item in the array
```

```
begin  
    string f = "orange";  
    if (fruits_I0.next (f))  
        $display ("fruits_I0.next [%s] = %0d", f, fruits_I0[f]);  
end
```

```
end
```

```
endmodule
```

### **Simulation Log**

```
ncsim> run
```

```
fruits_I0.size() = 4
```

```
fruits_10.num() = 4
Found 10 orange !
Sorry, season for apricots is over ...
fruits_10.first [apple] = 4
fruits_10.last [plum] = 9
fruits_10.prev [guava] = 1
fruits_10.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:**

<b>Feature</b>	<b>RETRY</b>	<b>SPLIT</b>
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.

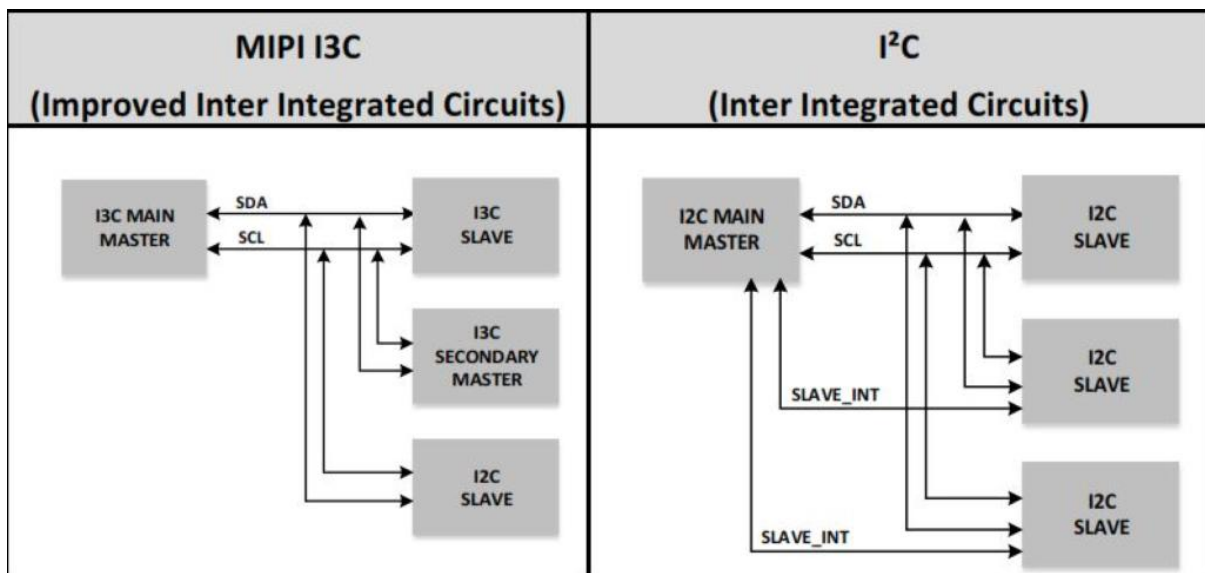
**I2C (Inter-Integrated Circuit):**

- 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.

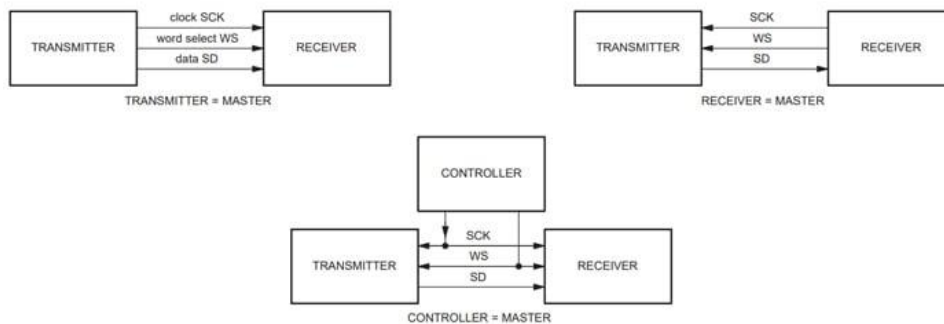
### I2C Data Rates:

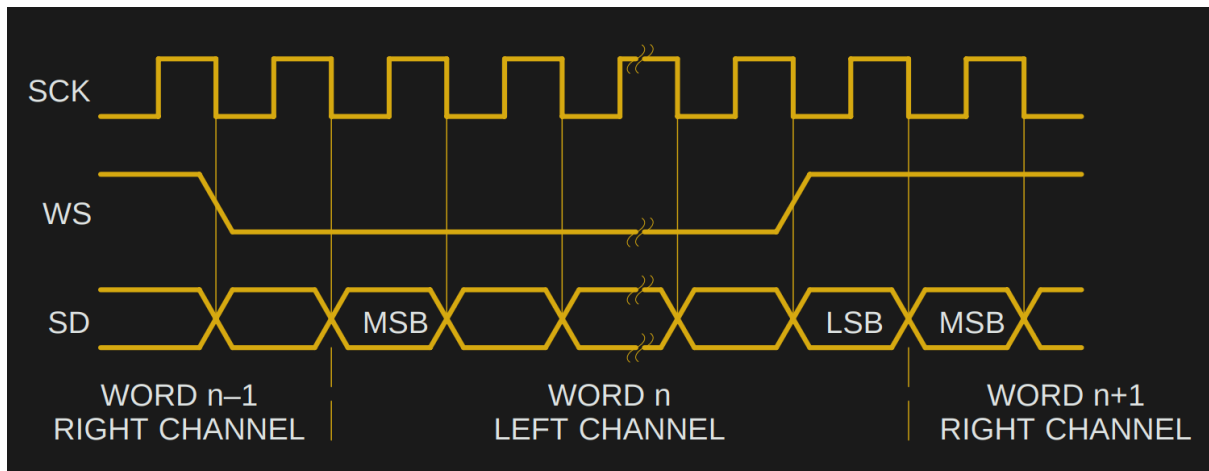
- **Standard Mode:** 100 kbit/s.
- **Fast Mode:** 400 kbit/s.
- **Fast Mode Plus:** 1 Mbit/s.
- **High-speed Mode:** 3.4 Mbit/s.

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.

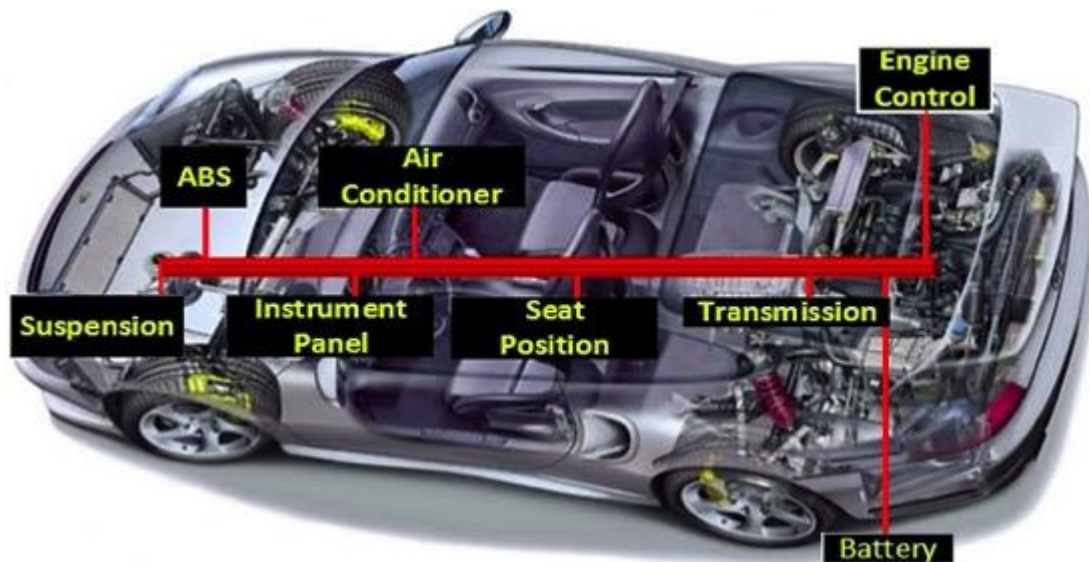
### **Supporting Information**

- [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.

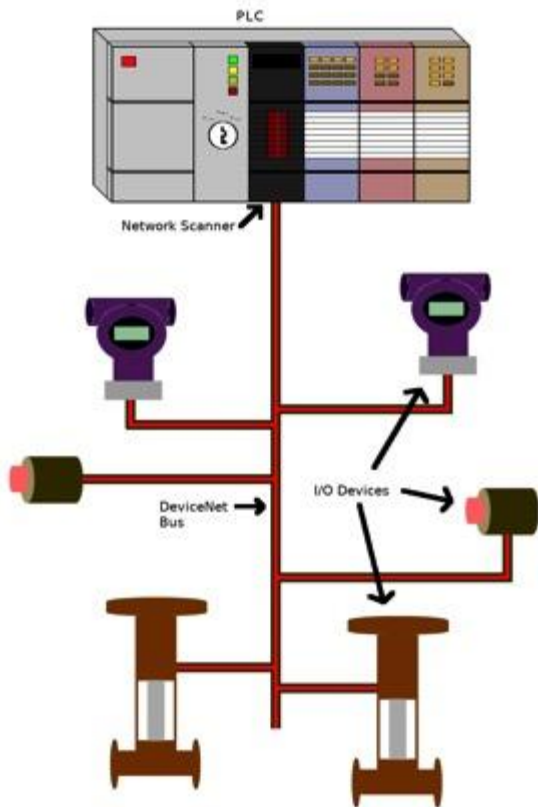
Originally invented by Bosch and later codified into the ISO11898-1 standard, CAN defines the data link and physical layer of the Open Systems Interconnection (OSI) model, providing a low-level networking solution for high-speed in-vehicle communications. In particular, CAN was developed to reduce cable 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.**

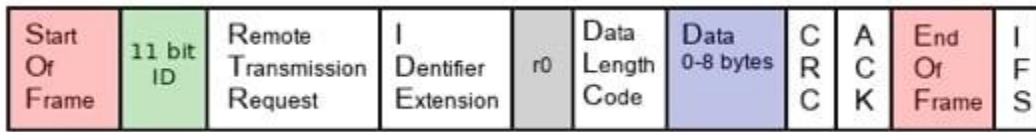
### CAN Message Frames

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**

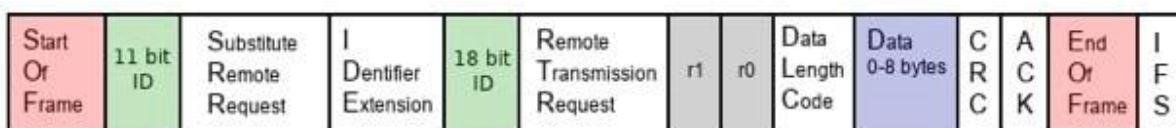
The first bit is the start of frame (SOF). This dominant bit represents the start of a CAN message. Next is the 11-bit identifier, which establishes the priority of the CAN message. The smaller the identifier, the higher the priority of the message.

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.

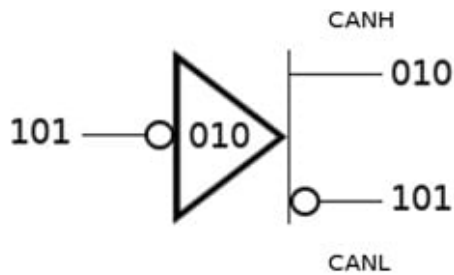
## **Bus Arbitration & Signaling**

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.

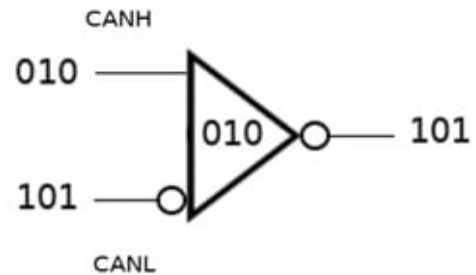
### **Inverted Logic**

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.

CAN Output Driver



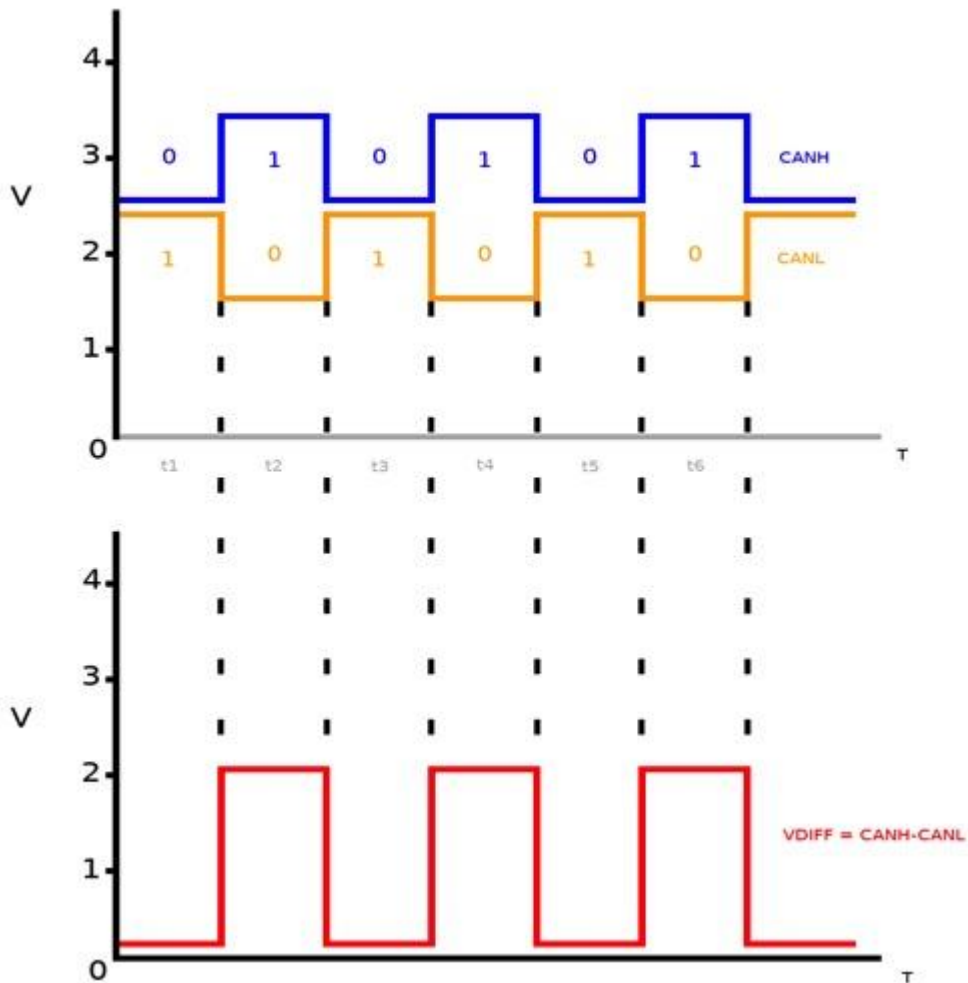
CAN Input Receiver



**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,  $t_1$ – $t_3$ , 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  $t_1$  in Figure 6. Notice that the CANH and CANL voltages are offset from one another. During time  $t_1$ ,  $CANH - CANL$  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 ( $V_{DIFF}$ ) 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.



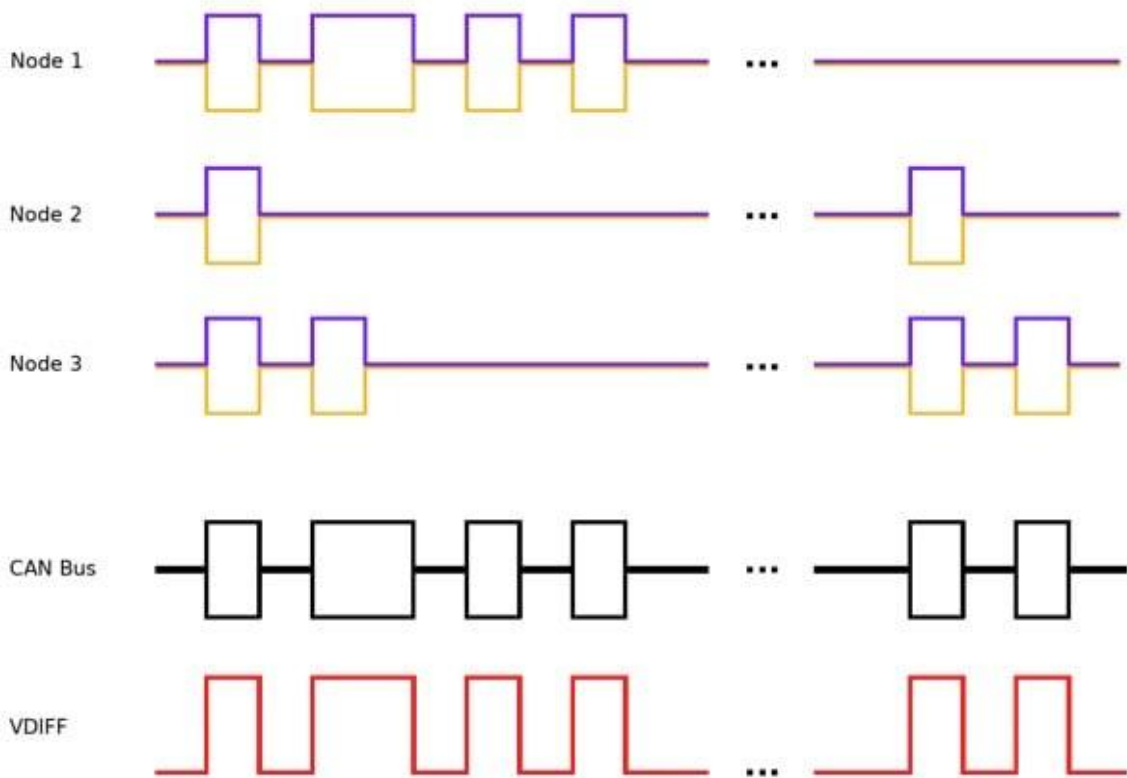
**Figure 6. CAN recessive and dominant states with differential voltage shown**

### 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 world objects where every object has its own property, characteristics defined by a blueprint. As this programming model is inspired by real-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 known 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.

Illustration 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.

Illustration 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??

Illustration 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**

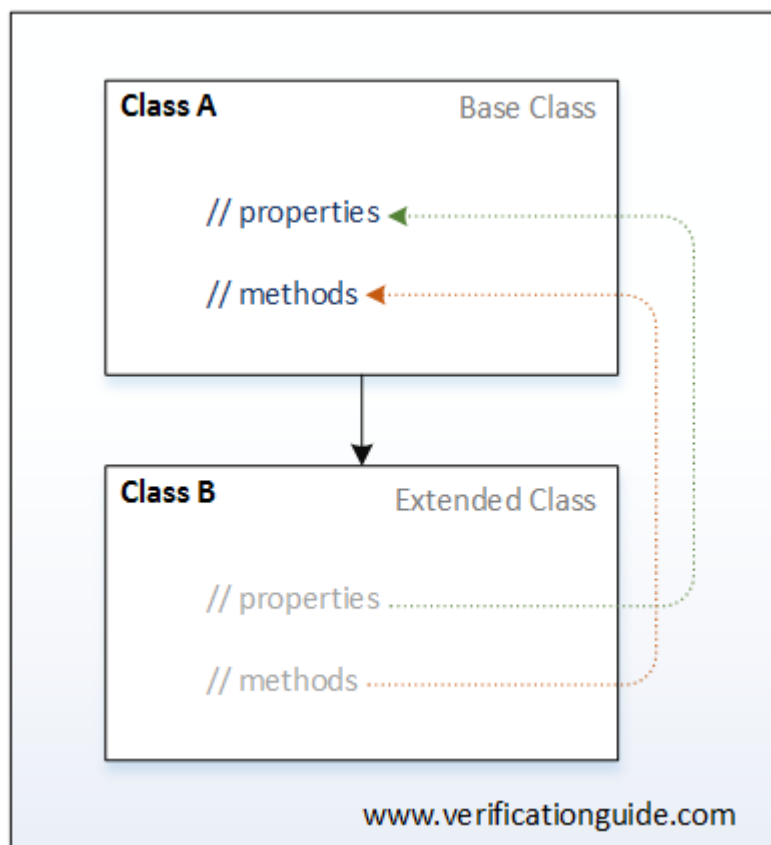
### **SystemVerilog Inheritance**

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 inheritance;
```

```
    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

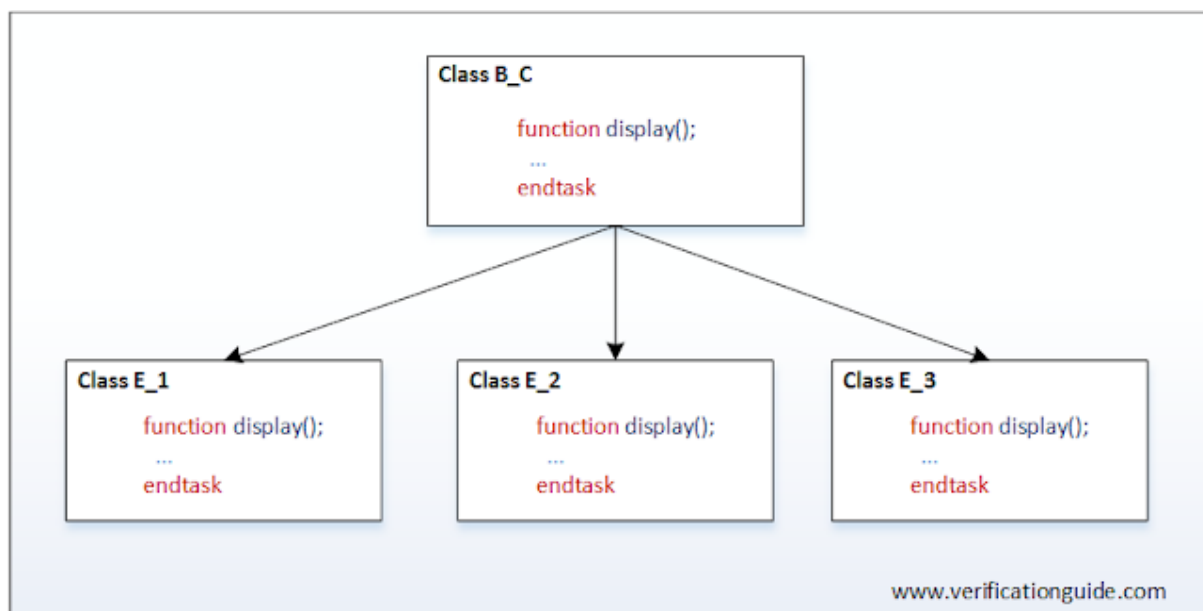
### Polymorphism in SystemVerilog

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 inheritance;
```

```

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 inheritance;
```

```
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 inheritance;
    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 inheritance;
```

```
    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 understand 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  $\geq$  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.verifianguild.com/modules.php?name=Forums&file=viewtopic&p=20045>)

```

property detect_glitch;
    time leading;          // declare the last edge occurrence 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.

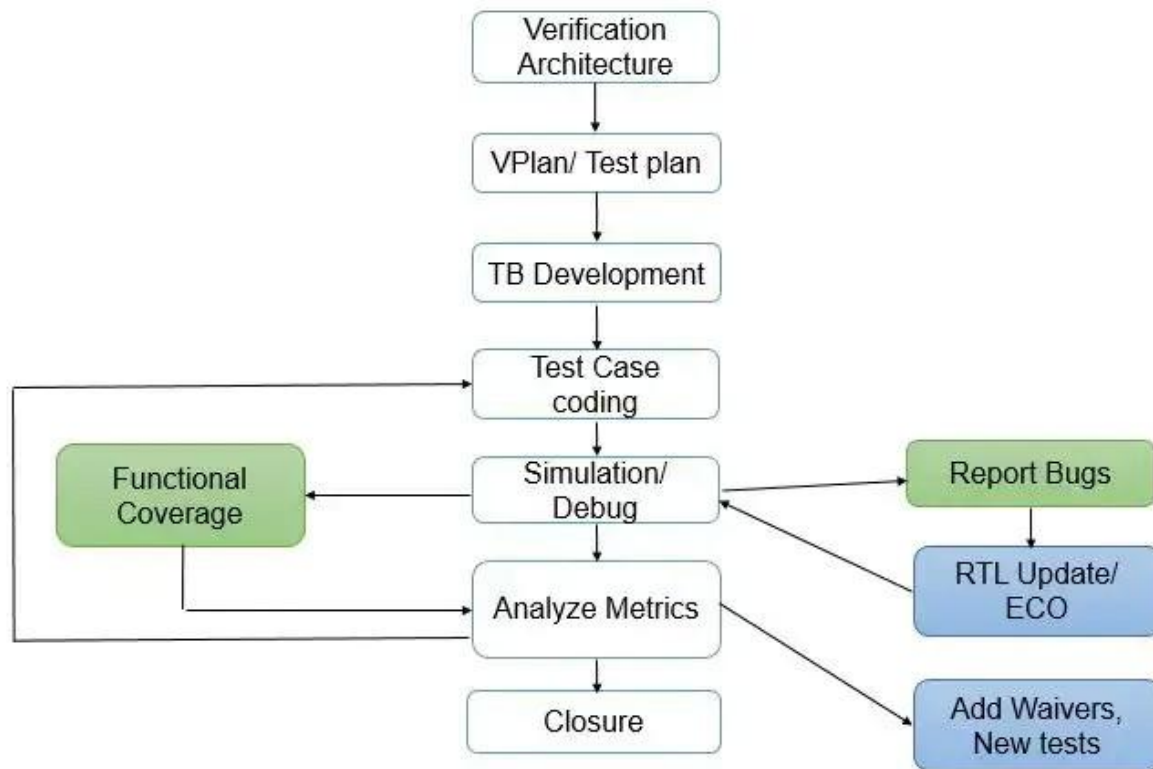
- **Formal Verification:**

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**

[www.vlsiverify.com](http://www.vlsiverify.com)

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 targeted 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.

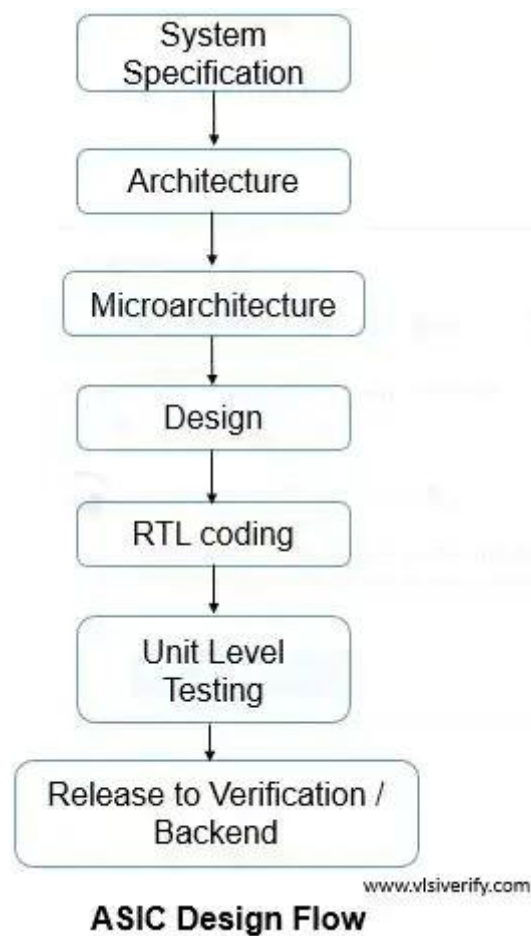
### **IP and SOC Verification**

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.

<b>Block/ IP Verification</b>	<b>System/ SoC Verification</b>
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



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.

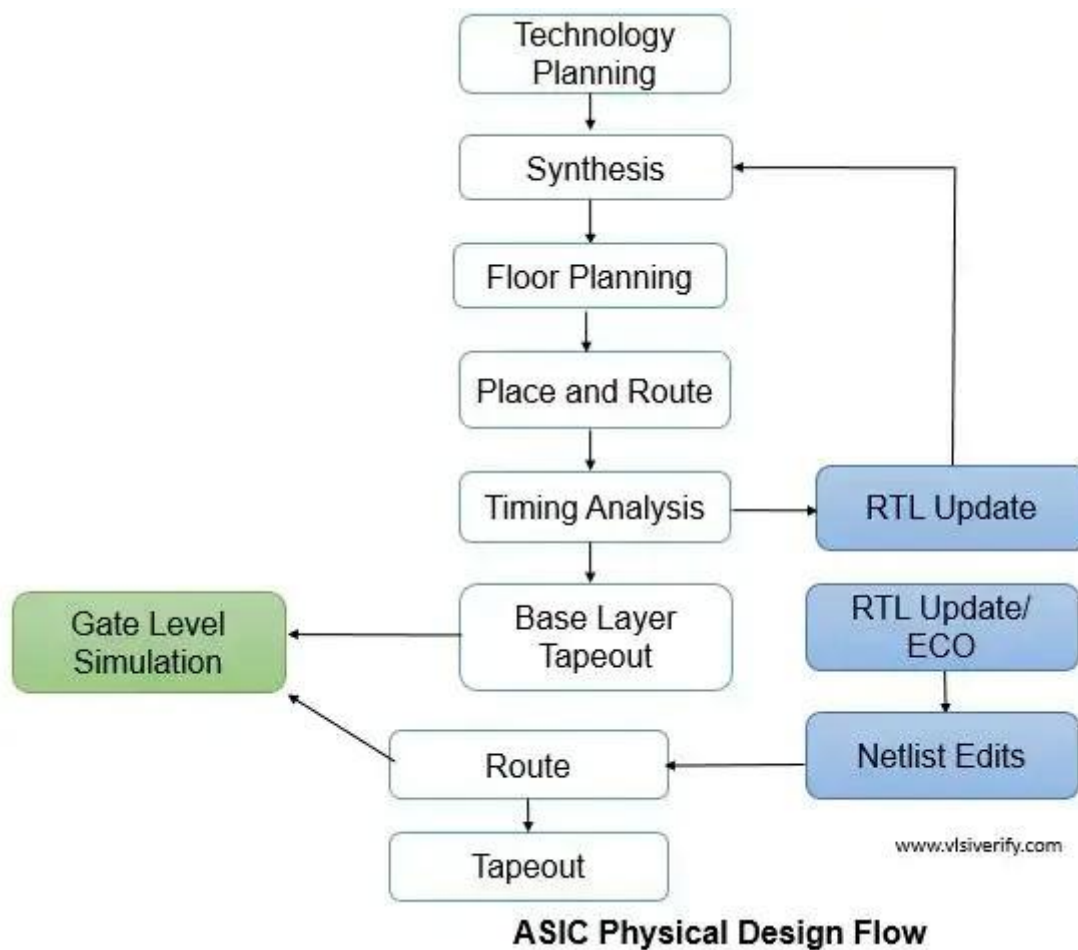
### RTL Coding

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



### 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.

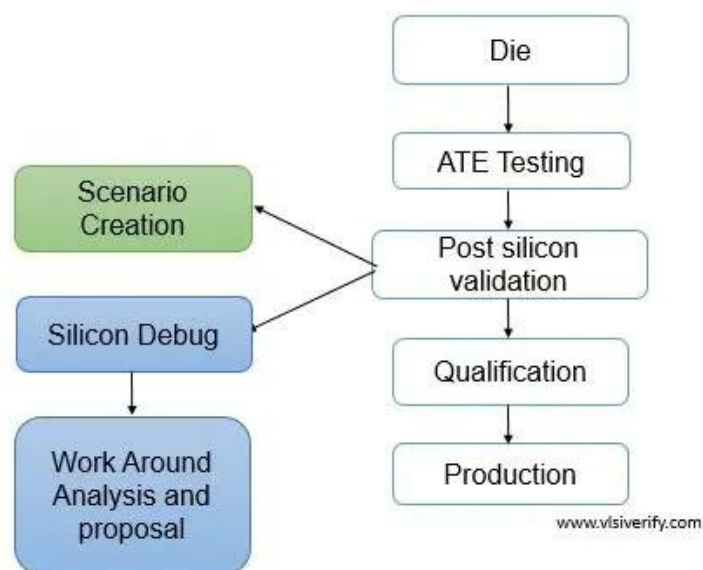
## Gate level Simulation (GLS)

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
```

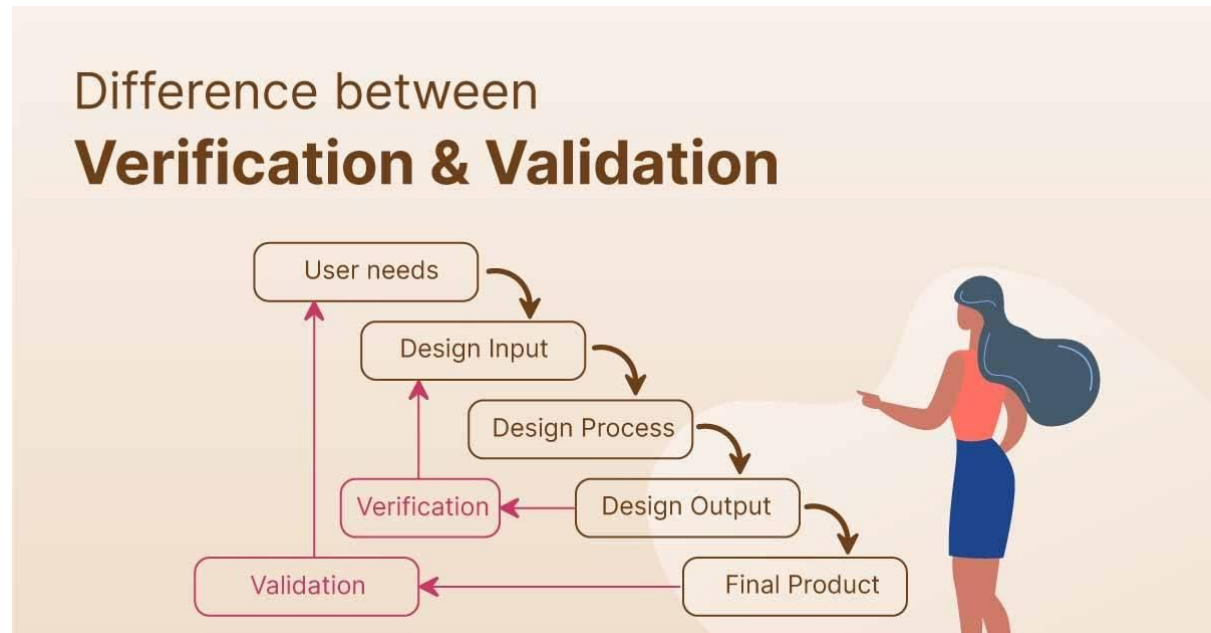
---

### 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:

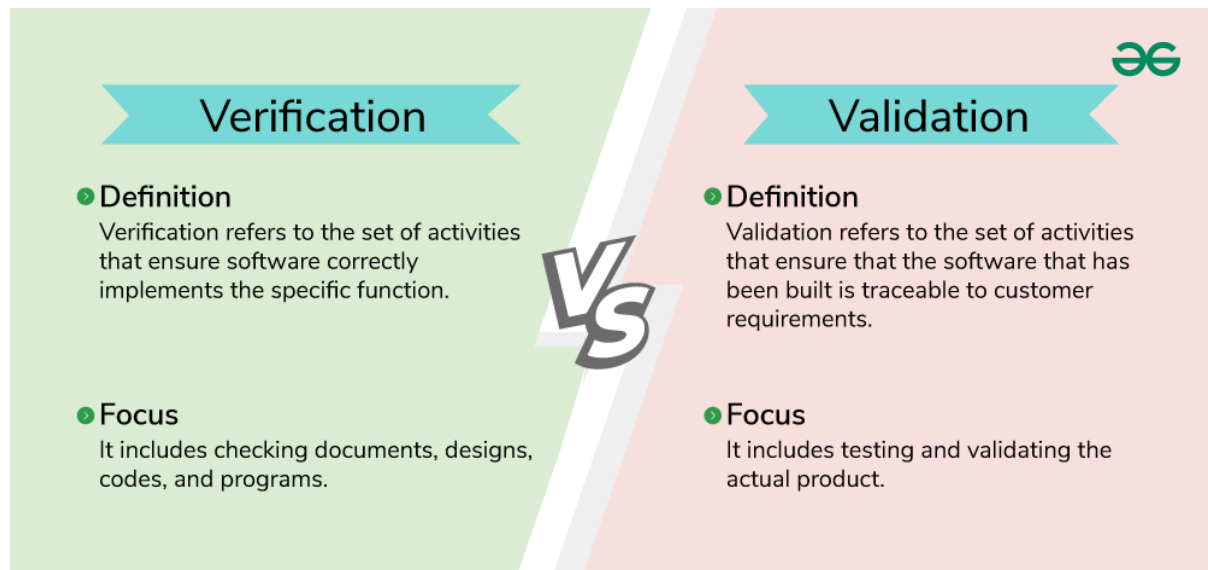
Feature	Verification	Validation
Focus	Internal aspects of the product	External aspects of the product
Purpose	Ensures the product is built correctly	Ensures the product meets user needs
Method	Static analysis, reviews, inspections	Dynamic testing, user acceptance testing
Timing	Throughout the development 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

	<b>Verification</b>	<b>Validation</b>
<b>Definition</b>	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.
<b>Focus</b>	It includes checking documents, designs, codes, and programs.	It includes testing and validating the actual product.
<b>Type of Testing</b>	Verification is the <a href="#">Static testing</a> .	Validation is <a href="#">Dynamic testing</a> .
<b>Execution</b>	It does <i>not</i> include the execution of the code.	It includes the execution of the code.
<b>Methods Used</b>	Methods used in verification are reviews, walkthroughs, inspections and desk-checking.	Methods used in validation are <a href="#">Black Box Testing</a> , <a href="#">White Box Testing</a> and <a href="#">Non-Functional testing</a> .
<b>Purpose</b>	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.
<b>Bug</b>	It can find the <a href="#">Bugs</a> in the early stage of the development.	It can only find the bugs that could not be found by the verification process.

	<b>Verification</b>	<b>Validation</b>
<b>Goal</b>	The goal of verification is application and software architecture and specification.	The goal of validation is an actual product.
<b>Responsibility</b>	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.
<b>Timing</b>	It comes before validation.	It comes after verification.
<b>Human or Computer</b>	It consists of checking of documents/files and is performed by human.	It consists of execution of program and is performed by computer.
<b>Lifecycle</b>	After a valid and complete specification the verification starts.	Validation begins as soon as project starts.
<b>Error Focus</b>	Verification is for prevention of errors.	Validation is for detection of errors.
<b>Another Terminology</b>	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.
<b>Performance</b>	Verification finds about 50 to 60% of the defects.	Validation finds about 20 to 30% of the defects.

	<b>Verification</b>	<b>Validation</b>
<b>Stability</b>	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:

- 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).
- 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.
- 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.
- 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.

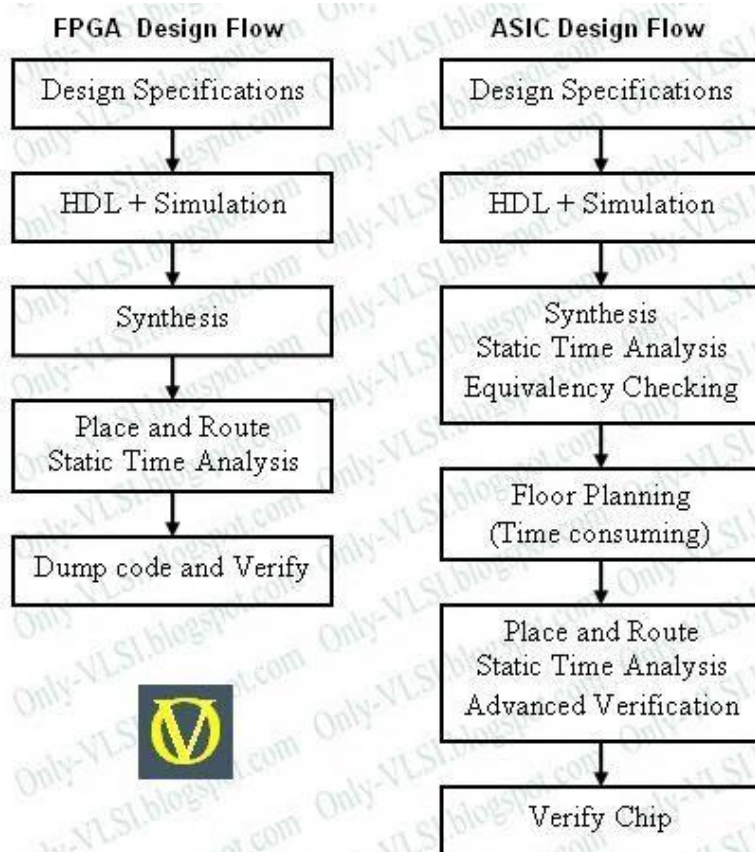
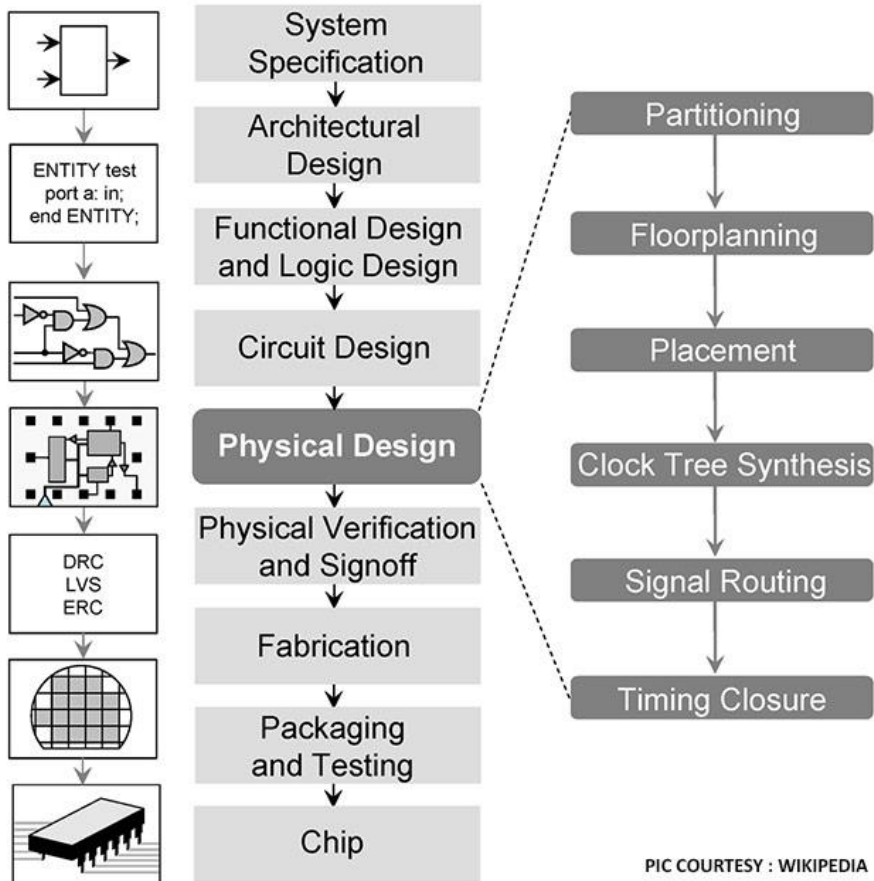
- **Content:**

Test cases, expected results, pass/fail criteria, and sometimes include debugging information.

Relationship:

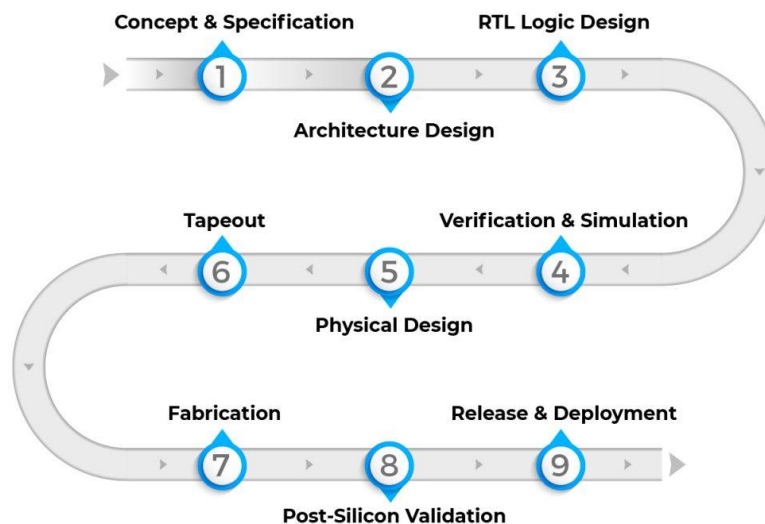
- Test plans are a crucial component of a verification plan.
- The verification plan provides the overall framework for verification, and test plans are used to execute the verification process.
- Test plans should align with and support the objectives outlined in the verification plan.

**Design flow of fpga and design flow of soc verification?**



## 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

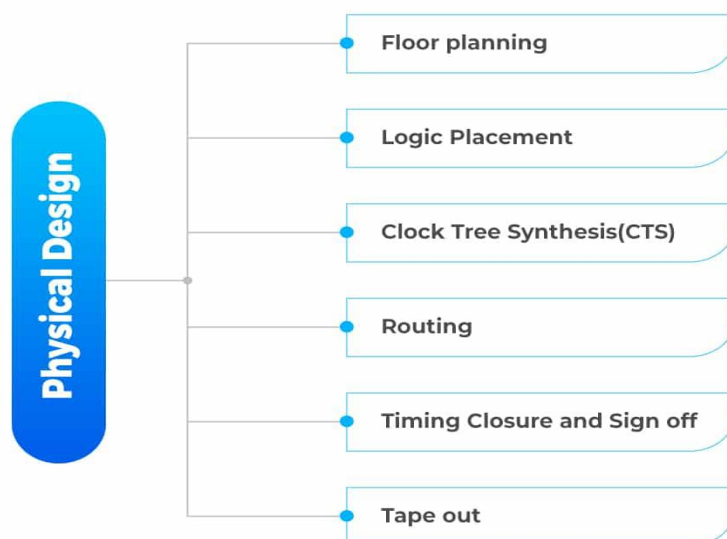
architecture. Hardware description languages (HDLs) like Verilog or VHDL are commonly used to represent the logic and behavior of the circuits. The RTL design specifies how the chip should function in terms of logical operations and data flow between memory blocks.

- **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**

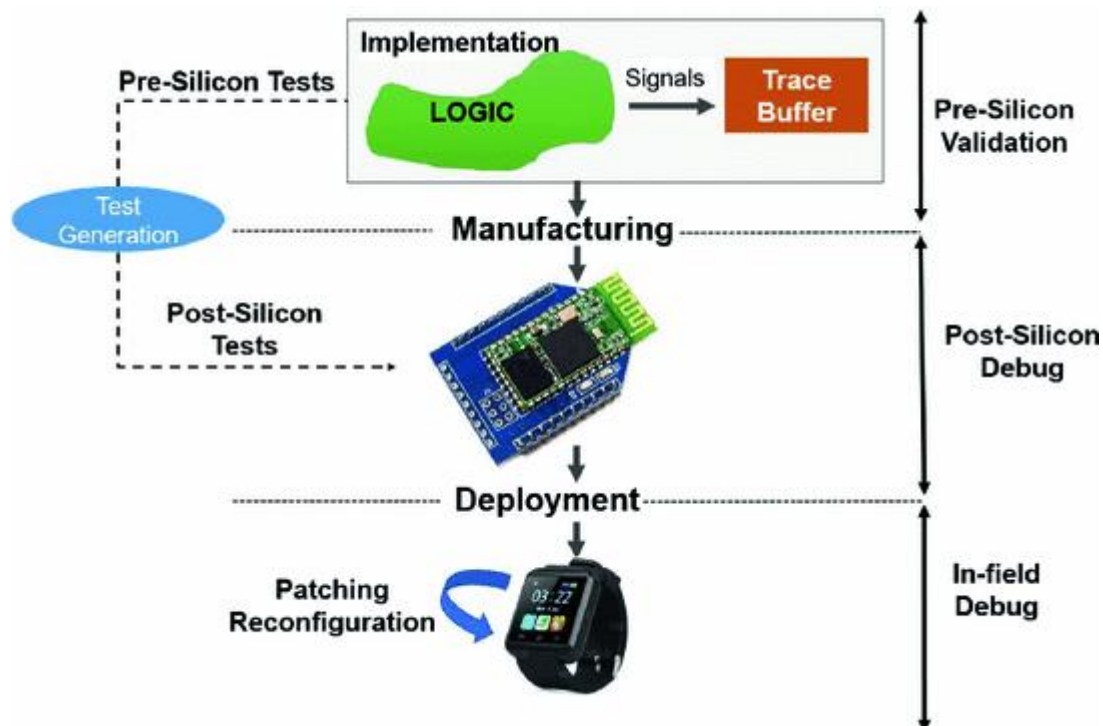
System-on-Chip (SoC), Field-Programmable Gate Array (FPGA), and Application-Specific Integrated Circuit (ASIC) development involve a wide range of tools that help with designing, verifying, and implementing complex integrated circuits. Hardware Description Language (HDL) Editors and Integrated Development Environments (IDEs) enable efficient code writing and simulation. Logic Synthesis, Place and Route, and Static Timing Analysis (STA) tools

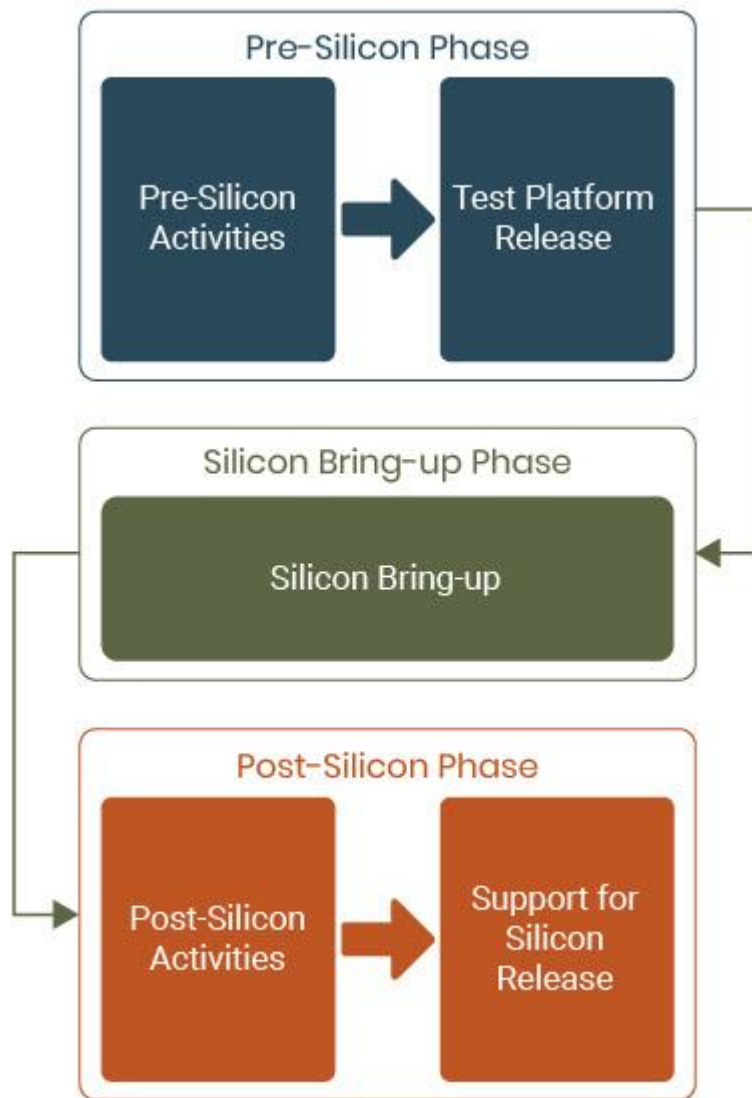
optimize the design process, ensuring optimal performance. Verification and Debugging tools facilitate rigorous testing and troubleshooting of a particular chip that is in the design phase. Additionally, High-Level Synthesis (HLS) and Emulation platforms streamline development, while Power Analysis and 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

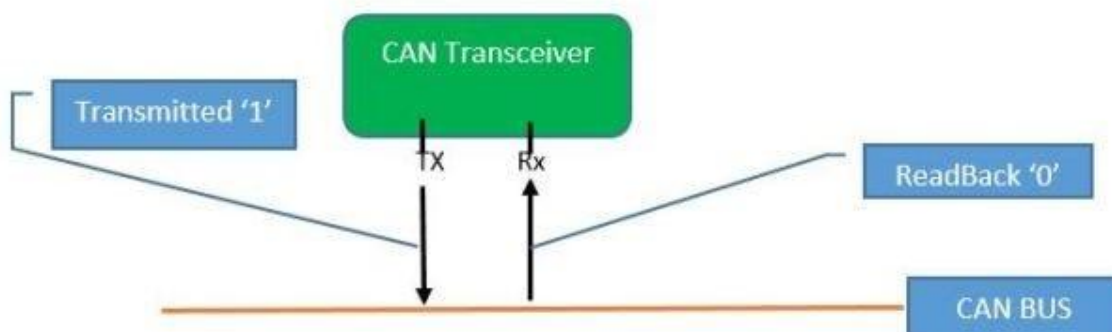
### Error Detection

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**.



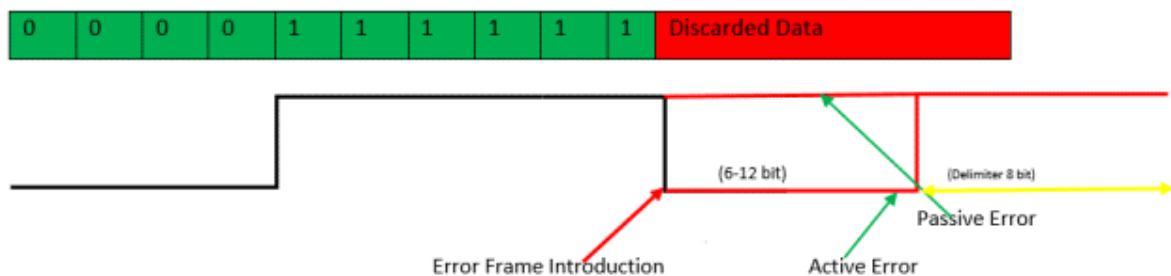
## Bit Error Situation

### 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 Delimiter, 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.



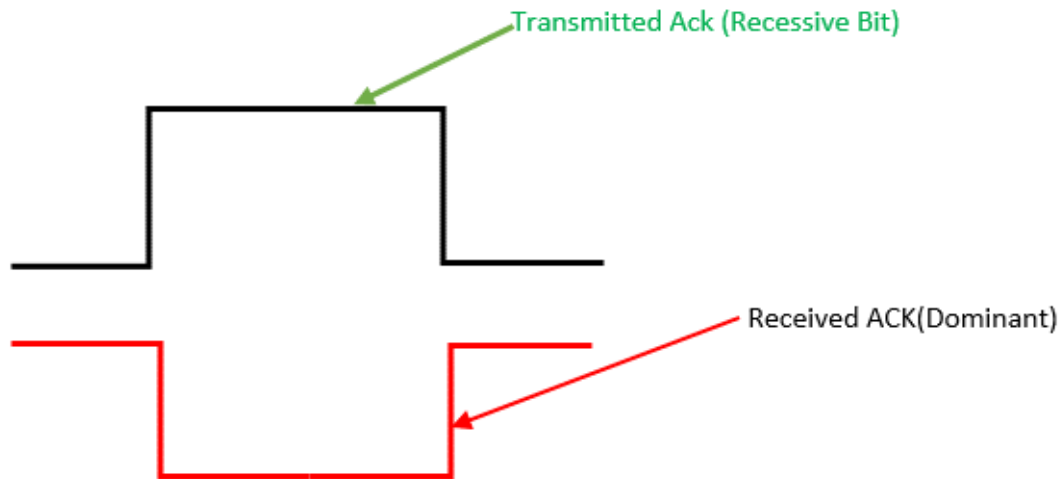
Stuff error Frame Introduction after 5 consecutive recessive bits

### 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

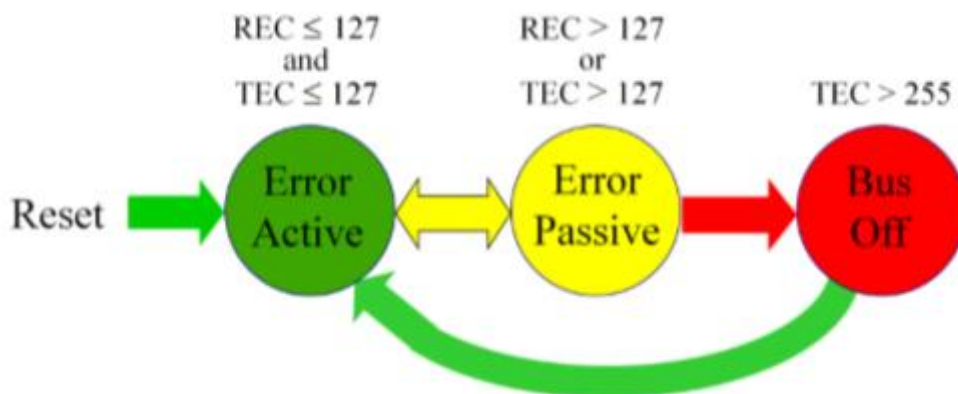
### Form Error(Introduce By Receiver)

As per CAN Frame Format, there are some fields of fixed length and format like CRC Delimiter, ACK Delimiter, EOF ,InterFrame Space and 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  $TEC \leq 127$  and  $REC \leq 127$ )
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  $TEC \leq 127$  and  $REC \leq 127$ . 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  $TEC \leq 127$  or  $REC \leq 127$ . 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’ ‘e’ ‘s’ ‘s’ ‘i’ ‘v’ 25856
hex	65	78 63 65 73 73 69 76 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

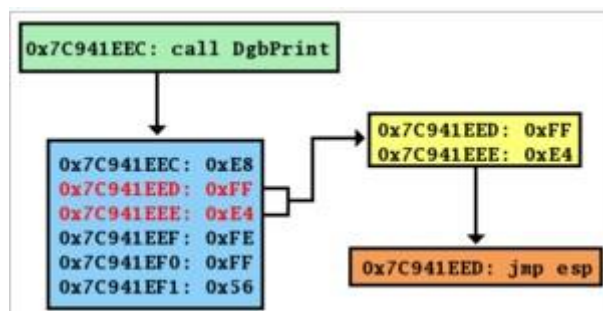
top of the buffer where the shellcode is located. This collection of no-ops is referred to as the “NOP-sled” because if the return address is overwritten with any address within the no-op region of the buffer it will “slide” down the no-ops until it is redirected to the actual malicious code by the jump 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 `strncpy` 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 writable 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.

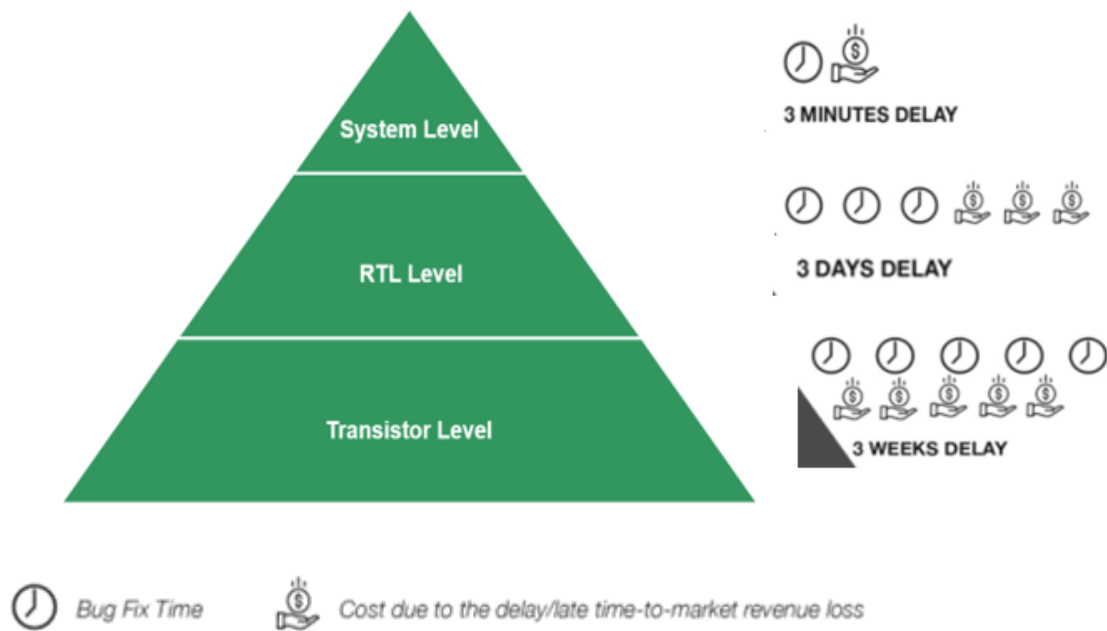


Fig 1

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.

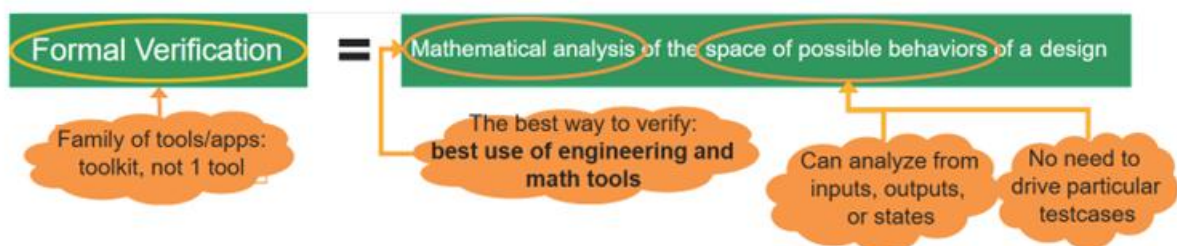


Fig 2

## Formal Verification Techniques:

The following figure illustrates the various formal verification techniques:

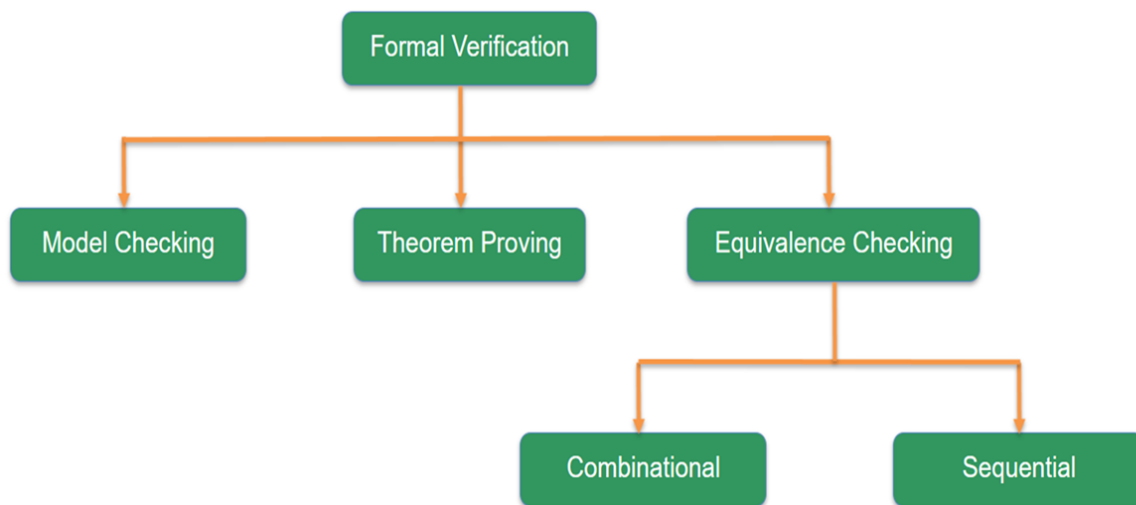


Fig 3

## 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.

## Property Checking

*A high level view*

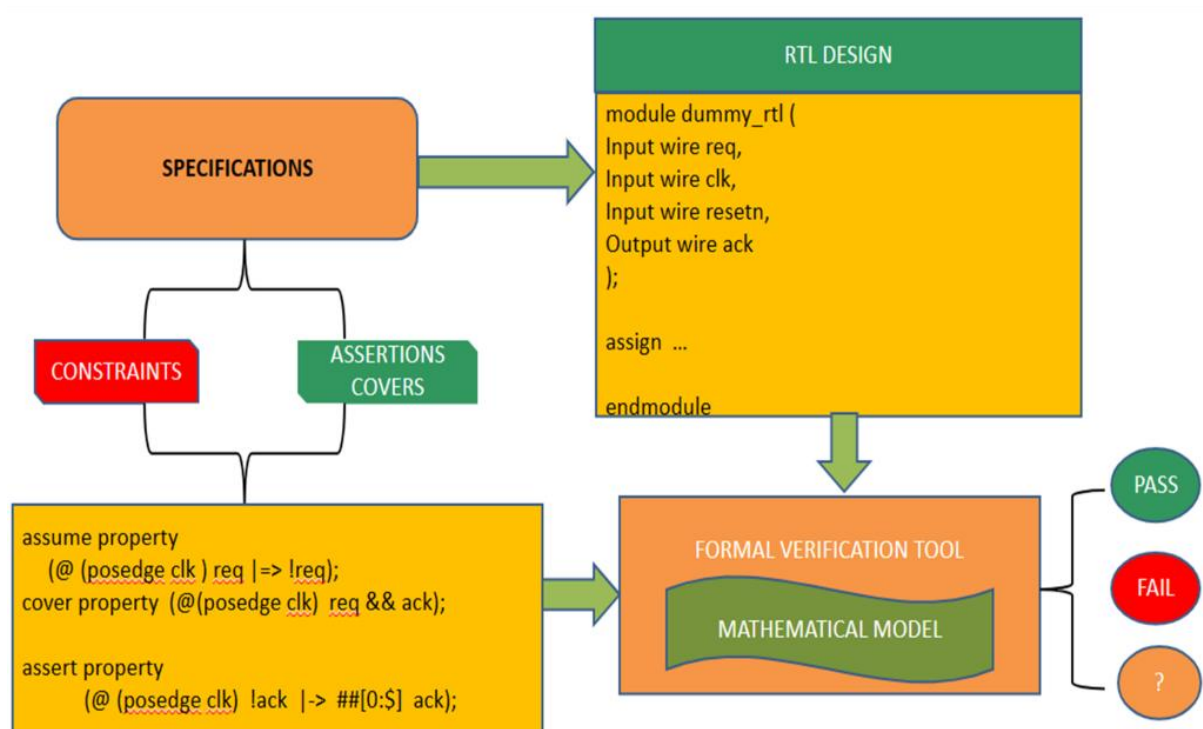


Fig 4: High level Architecture view for formal property verification

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.

Multiple Engines run for a property, When one finds a result, the others stop for that property

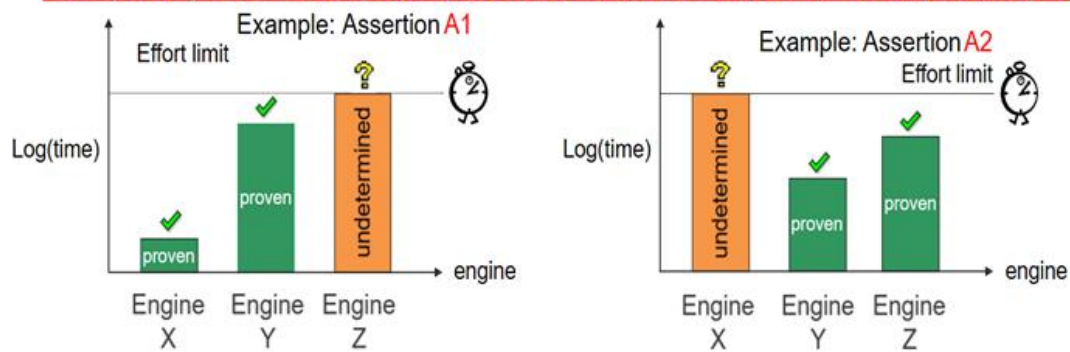


Fig 7: Graphical representation of various engine's performance

### 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 power 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)

 FPV	 SEC	 CONN	 CSR	 UNR
 CDC	 Superlint	 ABVIP	 DESIGN	 XPROP
			COV	

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

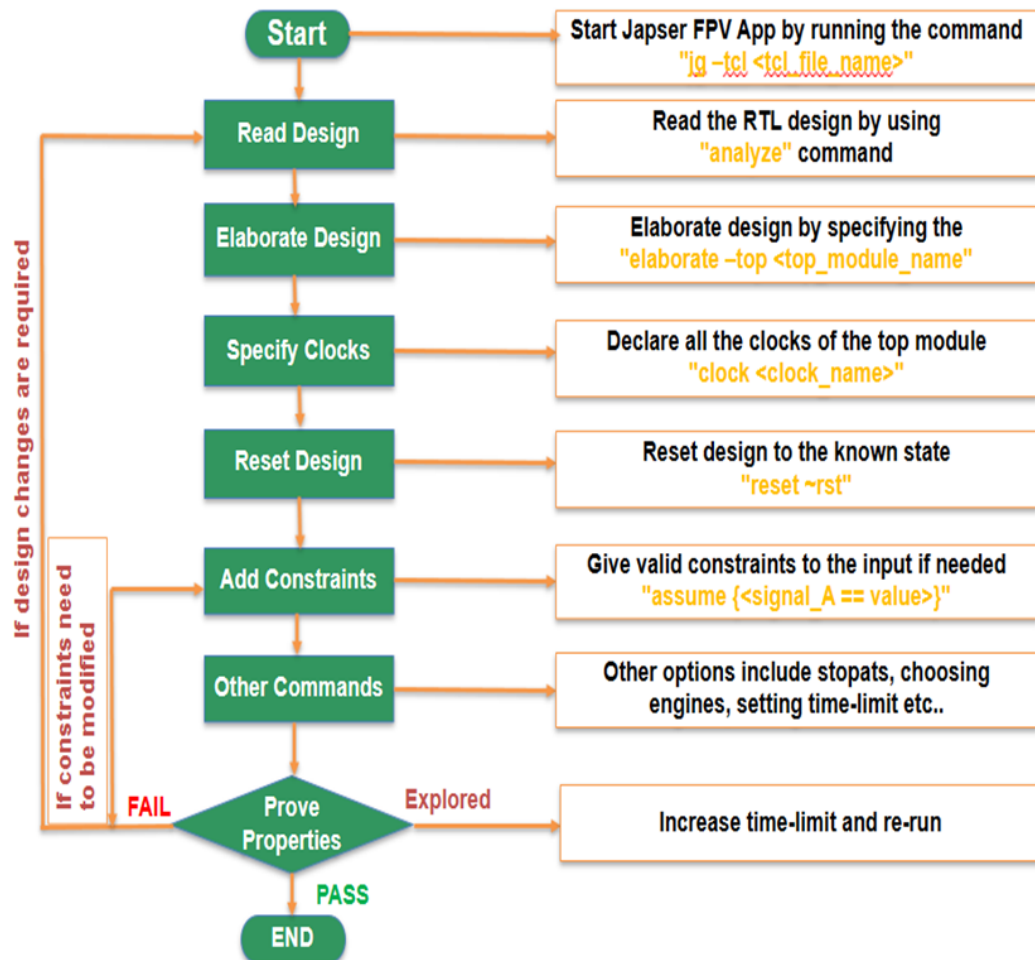


Fig 9: High Level Flowchart for formal property verification

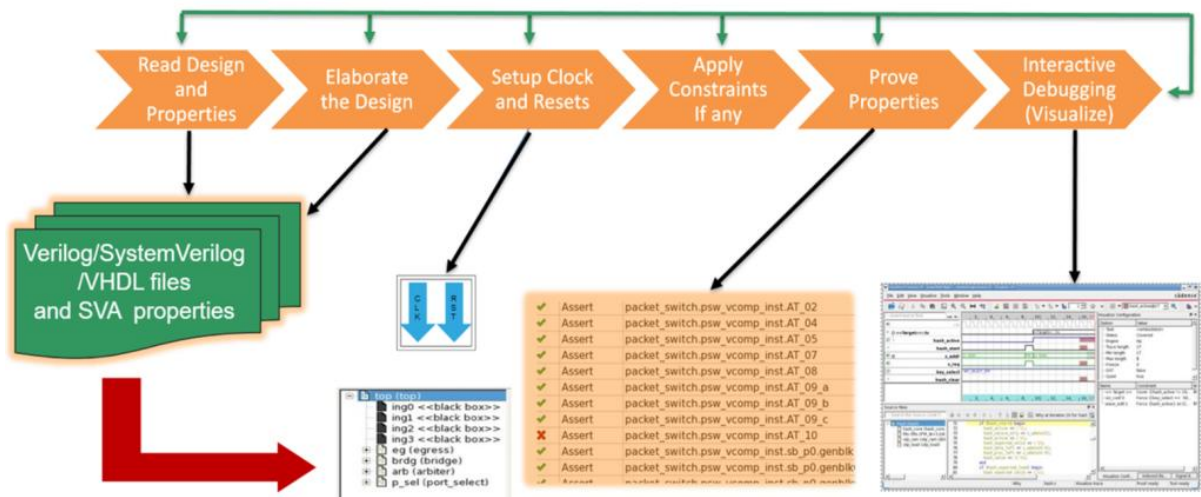


Fig 10

### Sample TCL File

```
A sample TCL file.
# Read design
analyze -sv09 \
+libext+.v+.sv+.svpt+.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/phenix_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:**

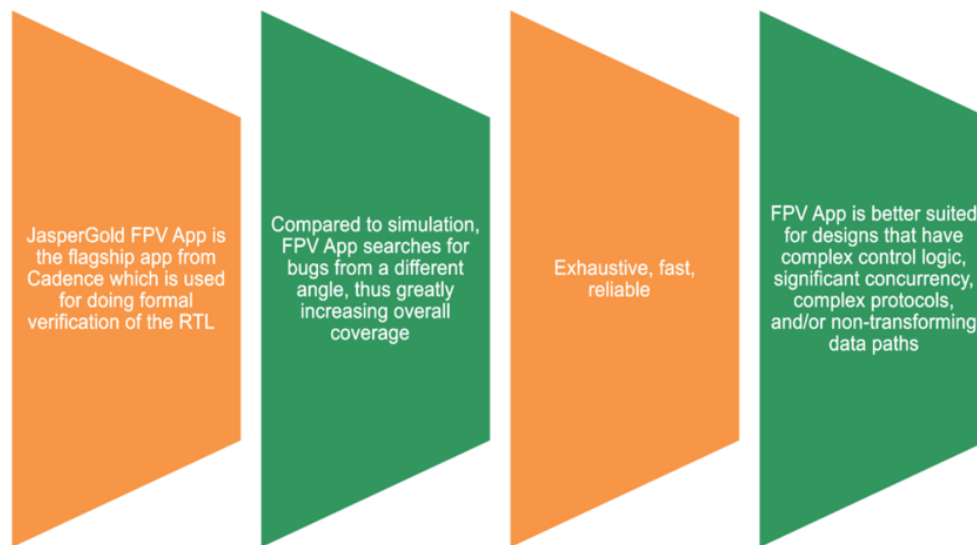


Fig 12

### 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:**

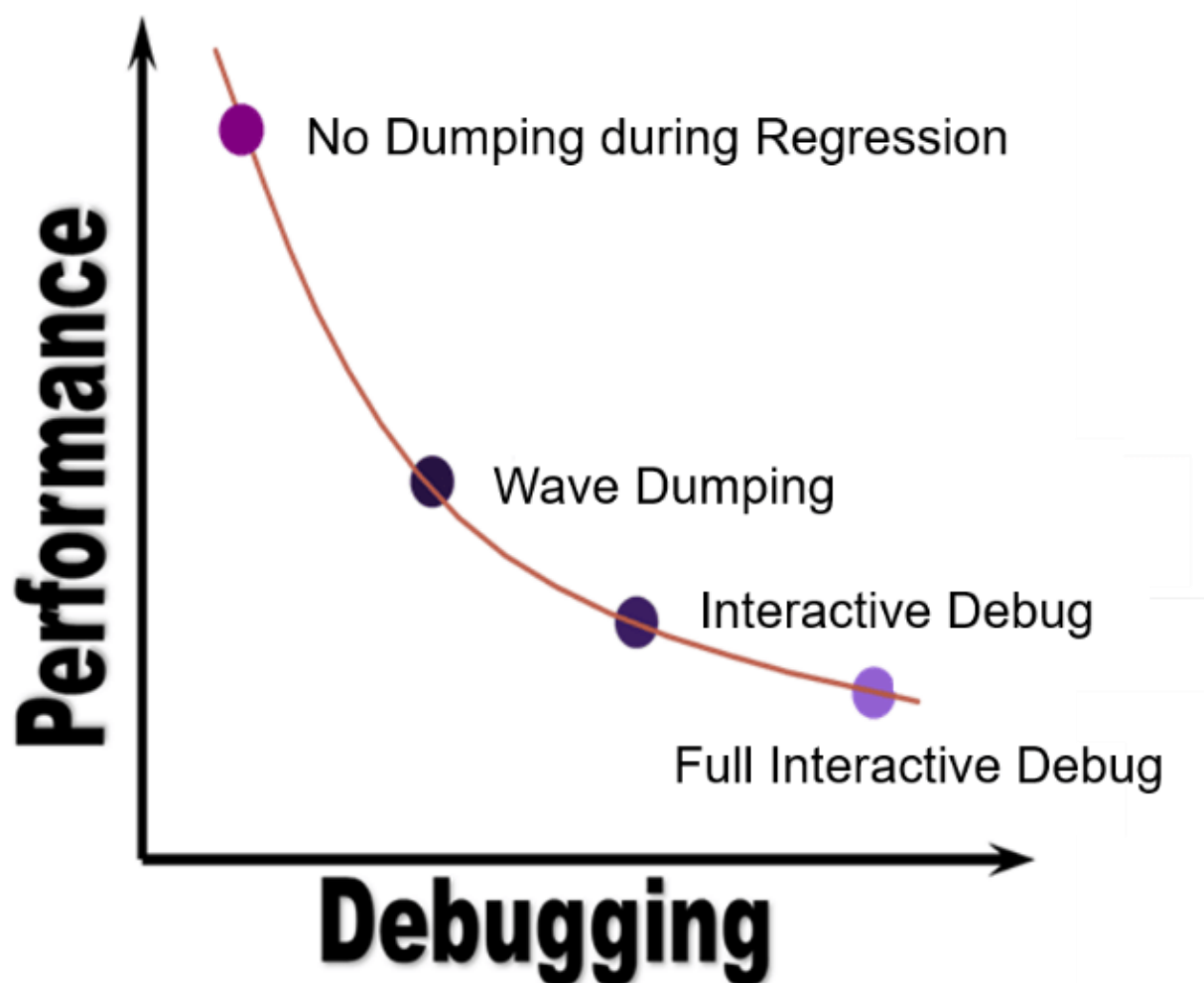
Waveform viewers offer various features, including signal probing, waveform analysis, assertion checking, and debugging.

- **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

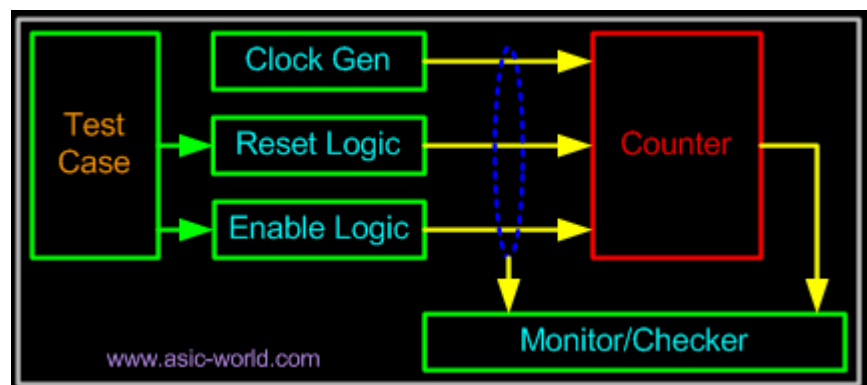
```

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 provides faster access.
- It acts as a 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**.

$$\text{Hit ratio} = (\text{Number of cache hits}) / (\text{Number of searches})$$

### 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 –

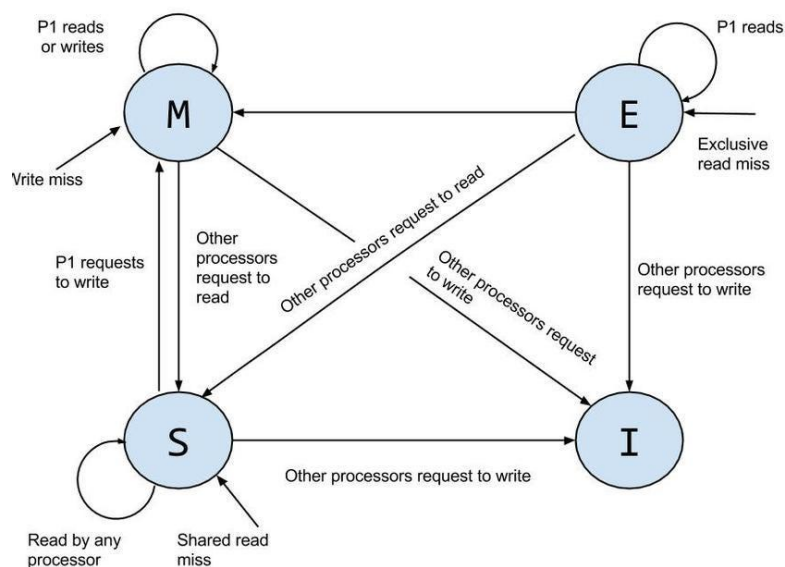
It is the most widely used cache coherence protocol. Every cache line is marked with one the following states:

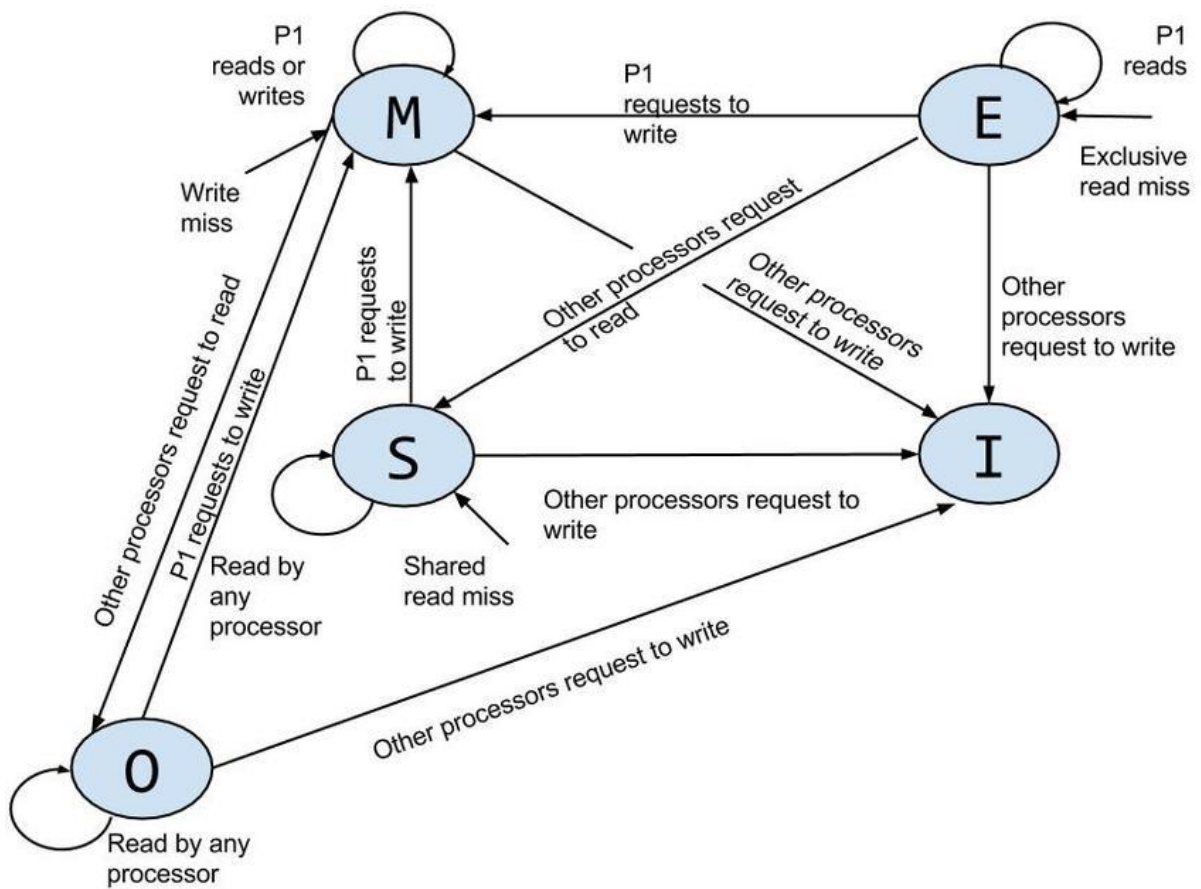
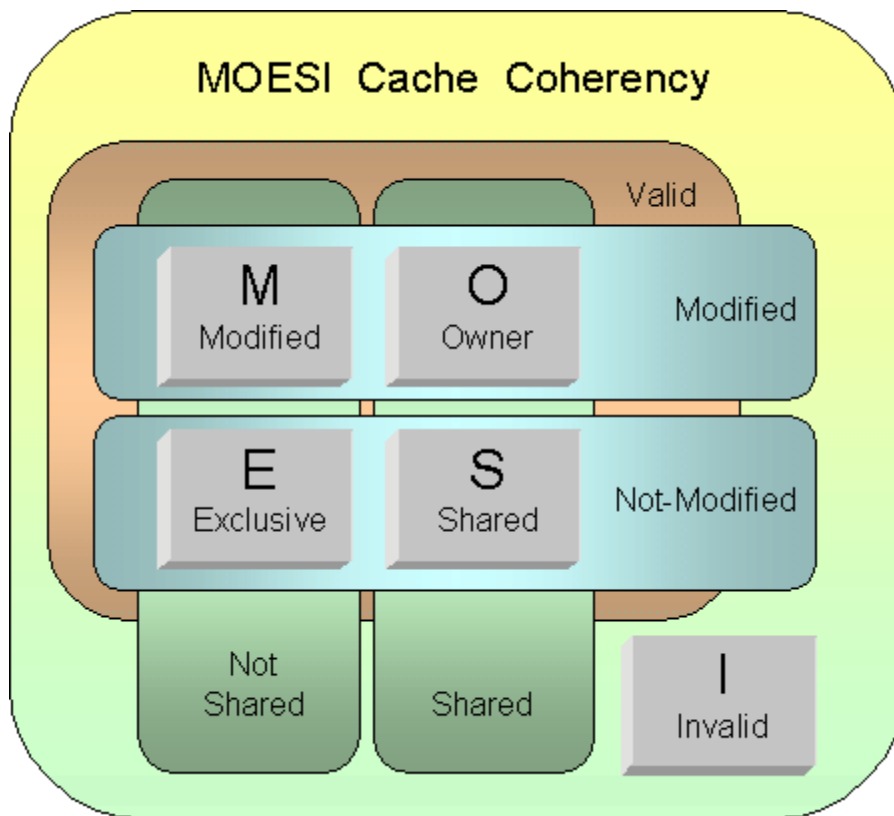
- **Modified –**  
This indicates that the cache line is present in current cache only and is dirty i.e its value is different from the main memory. The cache is required to write the data back to main memory in future, before permitting any other read of invalid main memory state.
- **Exclusive –**  
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.





## 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:  $\text{MIPS} = (\text{Clock Speed in Hz} * \text{Instructions per Cycle}) / 1,000,000$ . Alternatively, MIPS can be calculated as:  $\text{MIPS} = (\text{Number of Instructions} / \text{Execution Time}) * 10^6$ .

### **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 .

$X = 2 \text{ million} / 0.10 = 20 \text{ million}.$

No of MIPS =  $X / 1 \text{ million}$  (as the name tells about it)

No of MIPS =  $20 \text{ million} / 1 \text{ million} = 20.$

### **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) =  $\text{Clock Speed} / \text{CPI}$

$X = 400 \times 10^6 / 2$

$X = 200 \times 10^6$  instructions per second

For million instruction per second we divide by one million.

MIPS =  $200 \times 10^6 / 10^6 = 200$  MIPS.

### **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 Throughput?](#)
- [What is Latency?](#)
- [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, ensuring 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

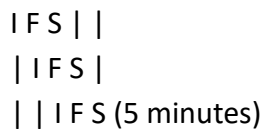
and 2nd stage accordingly. Therefore at the end of stage 3, we receive a new bottle for every minute. Hence, the average time taken to manufacture 1 bottle is:

Therefore, the average time intervals of manufacturing each bottle is:

**Without pipelining** =  $9/3$  minutes = 3m



**With pipelining** =  $5/3$  minutes = 1.67m



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:

$$\begin{aligned} ET_{\text{pipeline}} &= k + n - 1 \text{ cycles} \\ &= (k + n - 1) T_p \end{aligned}$$

In the same case, for a non-pipelined processor, the execution time of 'n' instructions will be:

$$ET_{\text{non-pipeline}} = n * k * T_p$$

So, speedup (S) of the pipelined processor over the non-pipelined processor, when 'n' tasks are executed on the same processor is:

$$S = \frac{\text{Performance of non-pipelined processor}}{\text{Performance of pipelined processor}}$$

As the performance of a processor is inversely proportional to the execution time, we have,

$$\begin{aligned} S &= \frac{ET_{\text{non-pipeline}}}{ET_{\text{pipeline}}} \\ \Rightarrow S &= \frac{[n * k * T_p]}{[(k + n - 1) * T_p]} \\ S &= \frac{[n * k]}{[k + n - 1]} \end{aligned}$$

When the number of tasks 'n' is significantly larger than k, that is,  $n \gg k$

$$\begin{aligned} S &= n * k / n \\ S &= k \end{aligned}$$

where 'k' are the number of stages in the pipeline. Also, **Efficiency** = Given speed up / Max speed up =  $S / S_{\text{max}}$  We know that  $S_{\text{max}} = k$  So, **Efficiency** =  $S / k$  **Throughput** = Number of instructions / Total time to complete the instructions So, **Throughput** =  $n / (k + n - 1) * T_p$

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 metrics as Throughput and latency.

### What is Throughput?

- 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 Latency?

- 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,  $\text{Latency} = \text{Execution time} / \text{Number of instruction executed}$ .
- It is 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:

### Example UASM Source File

```

N = 12           // loop index initial value
ADDC(r31, N, r1) // r1 = loop index
ADDC(r31, 1, r0) // r0 = accumulated product
loop: MUL(r0, r1, r0) // r0 = r0 * r1
      SUBC(r1, 1, r1) /* r1 = r1 - 1 */
      BNE(r1, loop, r31) // if r1 != 0, NextPC=loop

```

- **Comments** after //, ignored by assembler (also /\*...\*/)
- **Symbols** are symbolic representations of a constant value (they are NOT variables!)
- **Labels** are symbols for addresses
- **Macros** expand into sequences of bytes
  - Most frequently, macros are instructions
  - We can use them for other purposes

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.

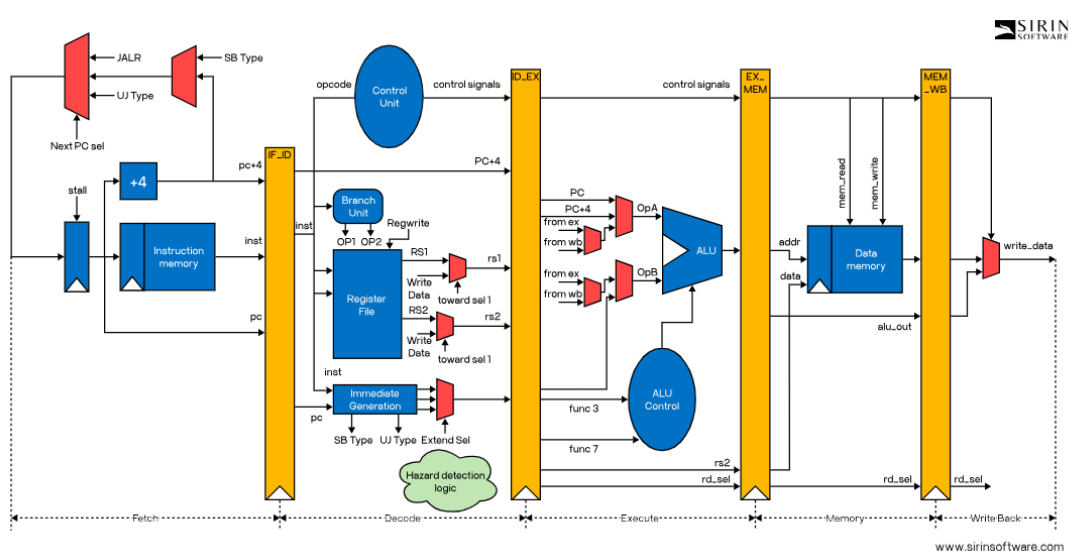
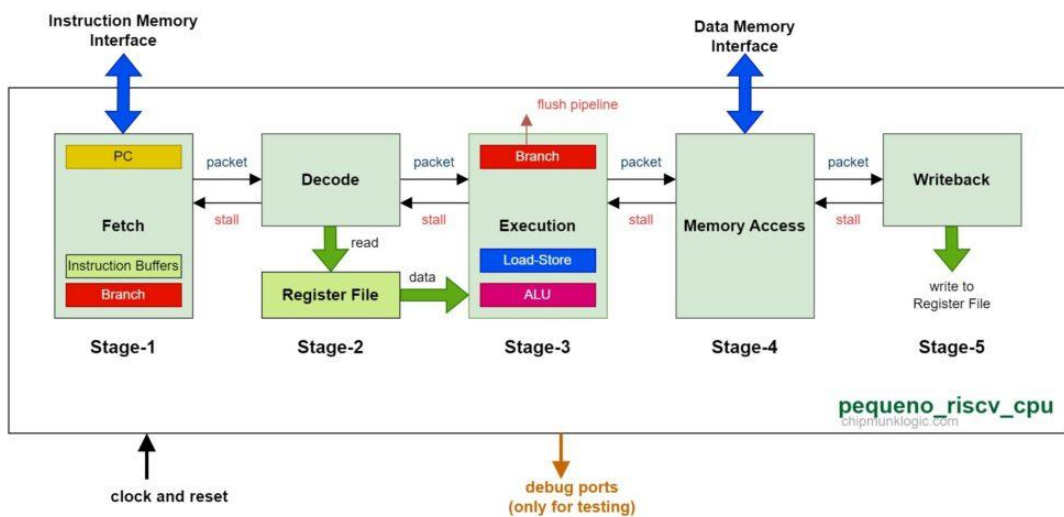
To the base instructions sets, they add optional extensions, e.g. for floating point, multiply and divide, etc., so like RV32IMAC is based on RV32I with the M, A, and C extensions.

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.

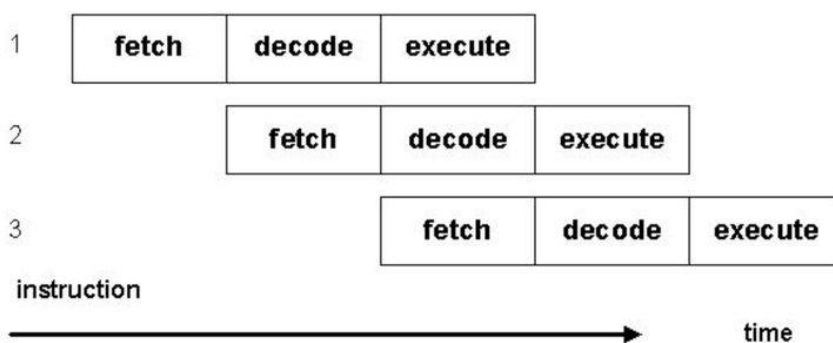
Otherwise one benefit of RV64 over RV32 is being able to address large address space than 4GB, which also applies to x86 vs. x64. Since we can run out of space for data with a 32-bit address space, having a larger address space and being able to do 64-bit computations in 64-bit registers is the gist 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

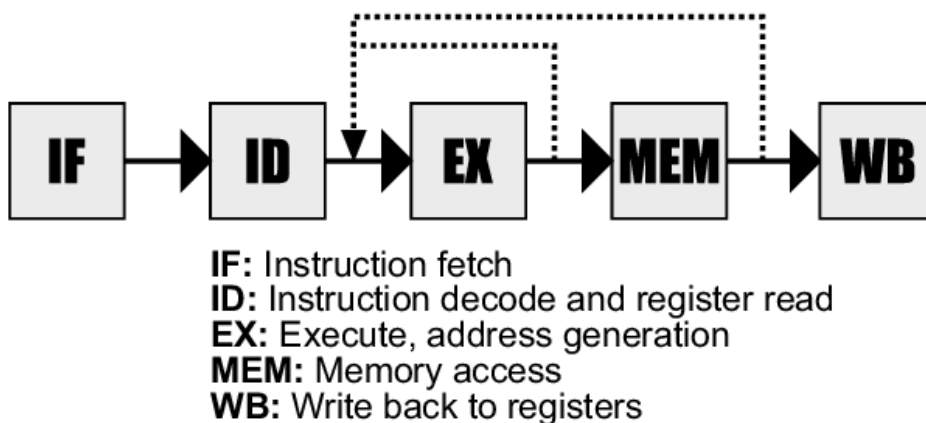
ARM processors up to the ARM7 employ a simple 3-stage pipeline with the following pipeline stages

1. **Fetch**
2. **Decode**
3. **Execute**



5

5-stage pipeline:



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 System Hardware Software & 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



## 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 Family Clear 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 Family Clear 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 an increased power usage making them less suitable for the mobile devices.
- More complex hardware design CISC processors require more complex circuitry to handle their varied 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.  <b>Variable-length encodings</b> of the instructions. <b>Example:</b> IA32 instruction size can range from 1 to 15 bytes.	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.  <b>Fixed-length encodings</b> of the instructions are used. <b>Example:</b> 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
<p>Arithmetic and logical operations can be applied to both memory and register operands.</p> <p>Implementation programs are hidden from machine-level programs. The ISA provides a clean abstraction between programs and how they get executed.</p>	<p>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.</p> <p>Implementation programs exposed to machine-level programs. Few RISC machines do not allow specific instruction sequences.</p>
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 <a href="#">RAM</a>

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

[ARM7](#)[ARM9](#)[ARM11](#)[ARM Cortex](#)[Key Differences](#)[ARM7 Core Architecture](#)[ARM9 Core Architecture](#)[ARM11 Core Architecture](#)[ARM Cortex-A Series](#)[Evolution 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
<b>Clock Speed</b>	<b>30-100 MHz</b>	<b>Up to 200 MHz</b>	<b>Up to 600 MHz</b>	<b>Up to 3 GHz+</b>
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.

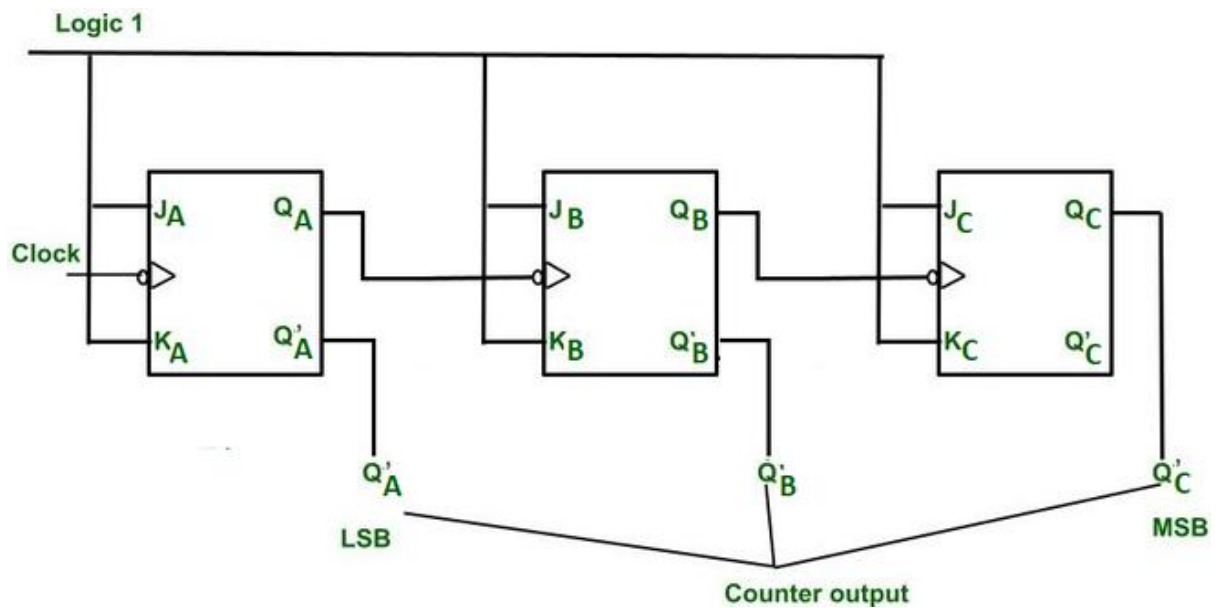
### 3 Bit Asynchronous Down Counter

- For the 3 bit counter, we require 3 flip flops and we can generate  $2^3 = 8$  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 fed as a clock to second FF and the output of the second FF is fed 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.

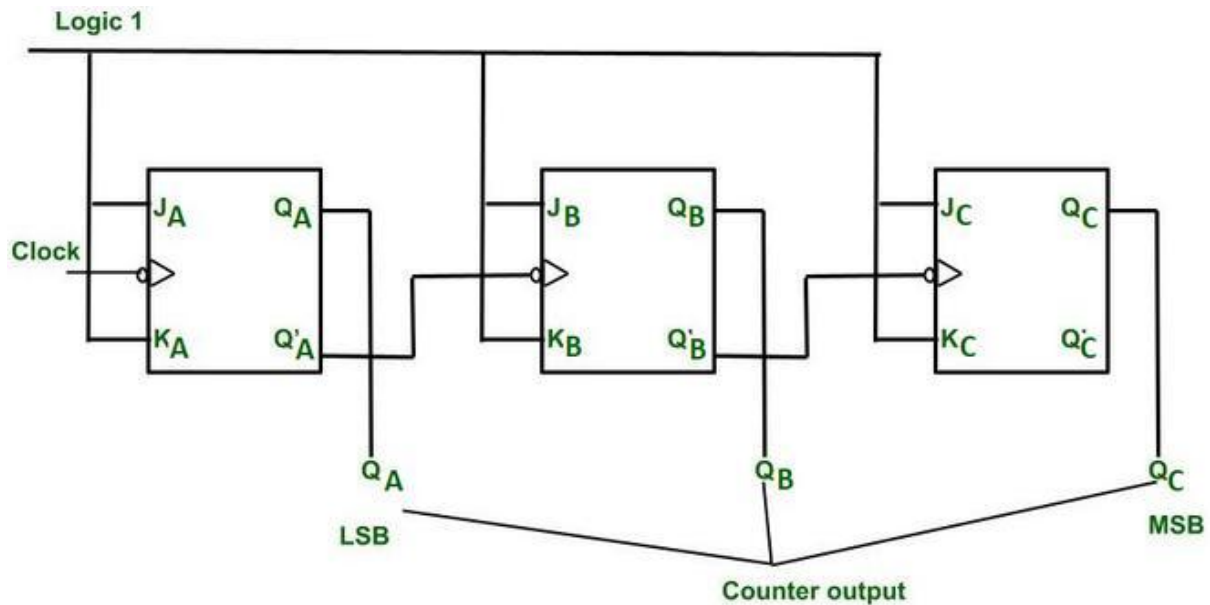


State table –

Clock	QC	QB	QA	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.

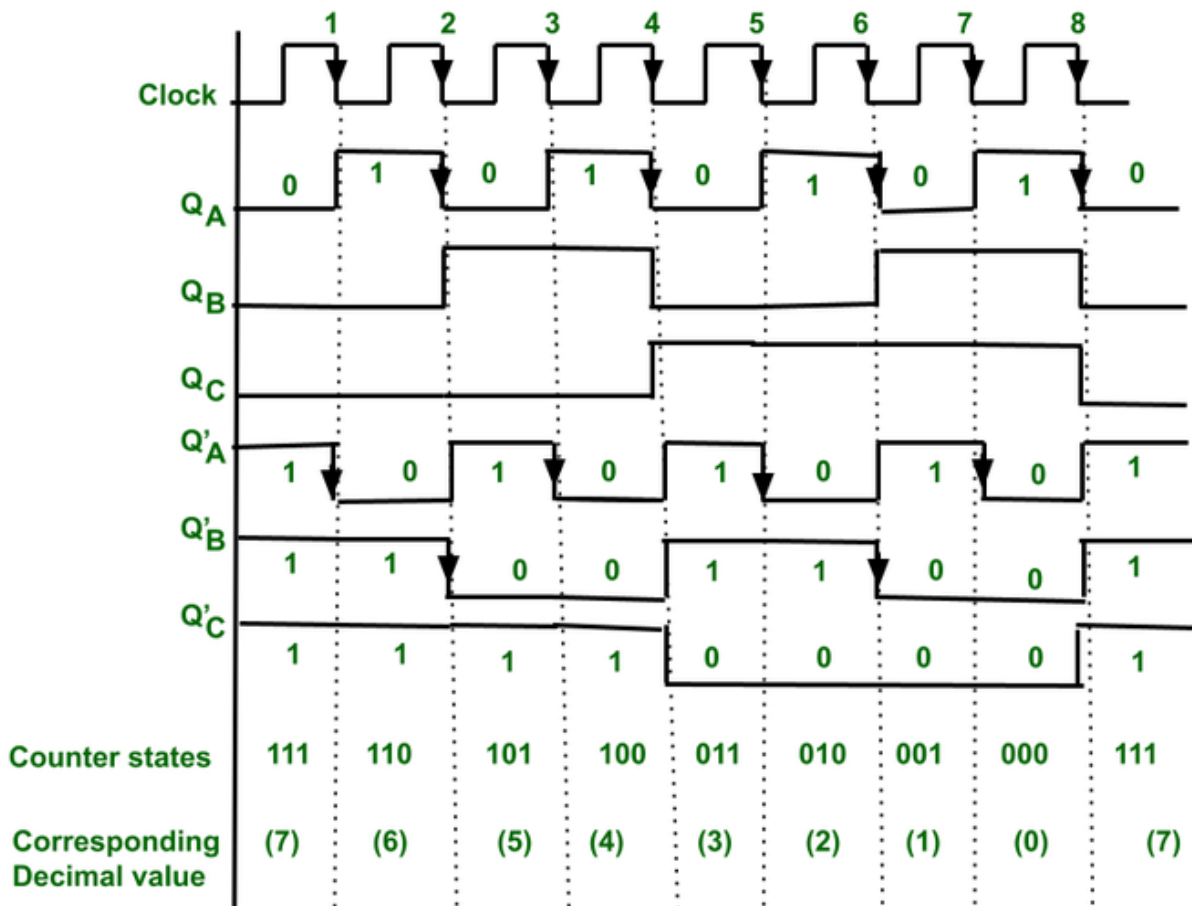
#### Explanation –

- 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  $Q_A = 0$   $Q_B = 0$  and  $Q_C = 0$ .

#### First Circuit

- The -ve edge clock pulse is provided to 1st counter. Therefore, the output state of the first counter (i.e.  $Q_A$ ) will be toggled at every falling edge of the clock pulse.
- As the  $Q_A$  is feed as a clock to the second FF, therefore the output state (i.e.  $Q_B$ ) will be toggled at every falling edge of  $Q_A$ .
- In the same manner, the  $Q_B$  acts as clock for the third FF, therefore the output state ( $Q_C$ ) of the third FF will be toggled for every falling edge of  $Q_B$ .
- 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)

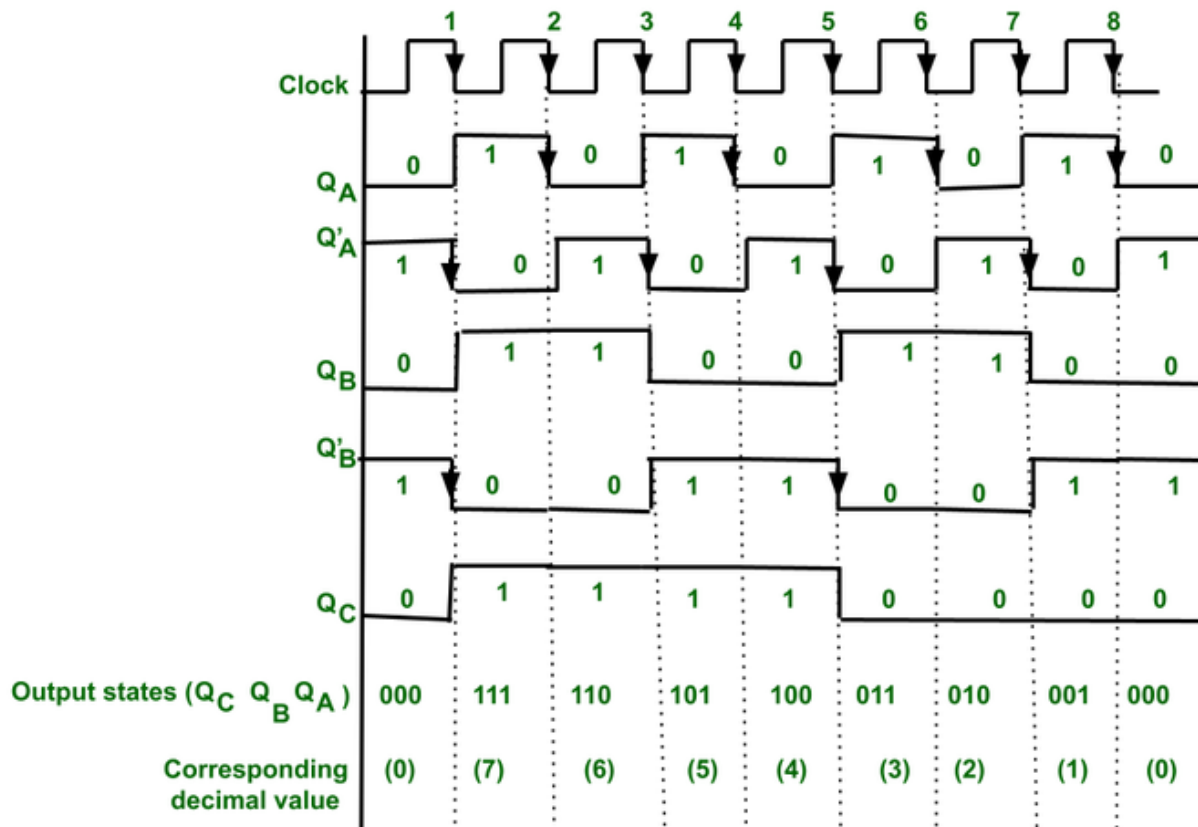
- After the 8th falling edge of the external clock pulse, the counter is reset to 000.



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-flop'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 situations, called *hazards*, that prevent the next instruction in the instruction stream from being executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining.

---

## 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

---

16

## 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**.

---

17

# 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	<b>MEM</b>	WB			
Instr 1		IF	ID	EX	MEM	WB		
Instr 2			IF	ID	EX	MEM	WB	
Instr 3				<b>IF</b>	ID	EX	MEM	WB

18

# 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	<b>MEM</b>	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Instr 3				<b>stall</b>	<b>IF</b>	ID	EX	MEM	WB

19

## Solution (2/2)

---

- Another solution is to use separate instruction and data memories.
- ARM used **Harvard** architecture, so we do not have this hazard

---

20

## 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>sub</sub>	EX	MEM	WB			
AND	R6,R1,R7			IF	ID <sub>and</sub>	EX	MEM	WB		
OR	R8,R1,R9				IF	ID <sub>or</sub>	EX	MEM	WB	
XOR	R10,R1,R11					IF	ID <sub>xor</sub>	EX	MEM	WB

---

21

# 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	<b>WB</b>		
SUB	R4,R5,R1		IF	ID <sub>sub</sub>	EX	MEM	WB	
AND	R6,R1,R7			IF	ID <sub>and</sub>	EX	MEM	WB

22

# 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	<b>WB</b>		
SUB	R4,R5,R1		IF	ID <sub>sub</sub>	EX	MEM	WB	
AND	R6,R1,R7			IF	ID <sub>and</sub>	EX	MEM	WB

23

# Forward Data

		Clock cycle number						
		1	2	3	4	5	6	7
ADD	R1,R2,R3	IF	ID	EX <sub>add</sub>	MEM <sub>add</sub>	WB		
SUB	R4,R5,R1		IF	ID	EX <sub>sub</sub>	MEM	WB	
AND	R6,R1,R7			IF	ID	EX <sub>and</sub>	MEM	WB

- The first forwarding is for value of **R1** from **EX<sub>add</sub>** to **EX<sub>sub</sub>**.  
The second forwarding is also for value of **R1** from **MEM<sub>add</sub>** to **EX<sub>and</sub>**.  
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.

25

# 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>sub</sub>	EX	MEM	WB	
AND	R6,R1,R7			stall	stall	IF	ID <sub>and</sub>	EX	MEM	WB

26

# 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
- 

27

# 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>sub</sub>	MEM	WB		
AND	R6,R1,R7			IF	ID	EX <sub>and</sub>	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.
- 

28

# 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>sub</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>sub</sub>**.

29

## 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

30

## 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
```

```
MULTDF8,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
      MULTDF8,F6,F0
      LD    F10,-8(R1)
      ADDD  F12,F10,F4
      MULTDF14,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

```
MULTDF4,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 (assuming F8 is unused):

```
MULTDF4,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,

```
MULTDF4,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.

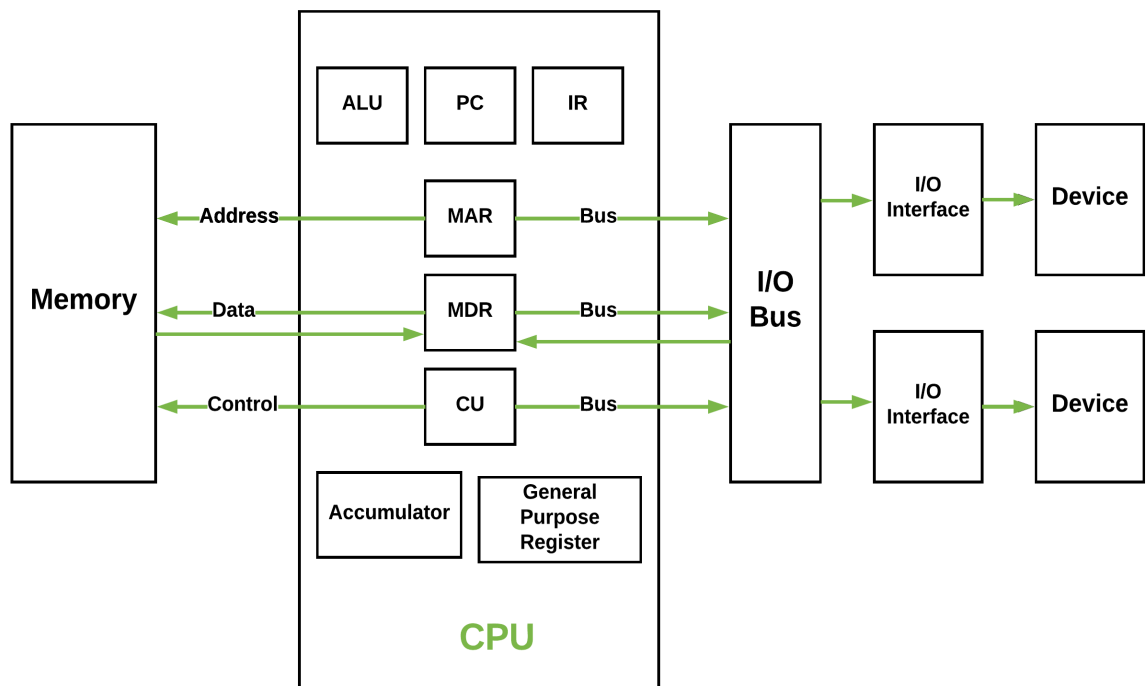


Figure – Basic CPU structure, illustrating ALU

- **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.

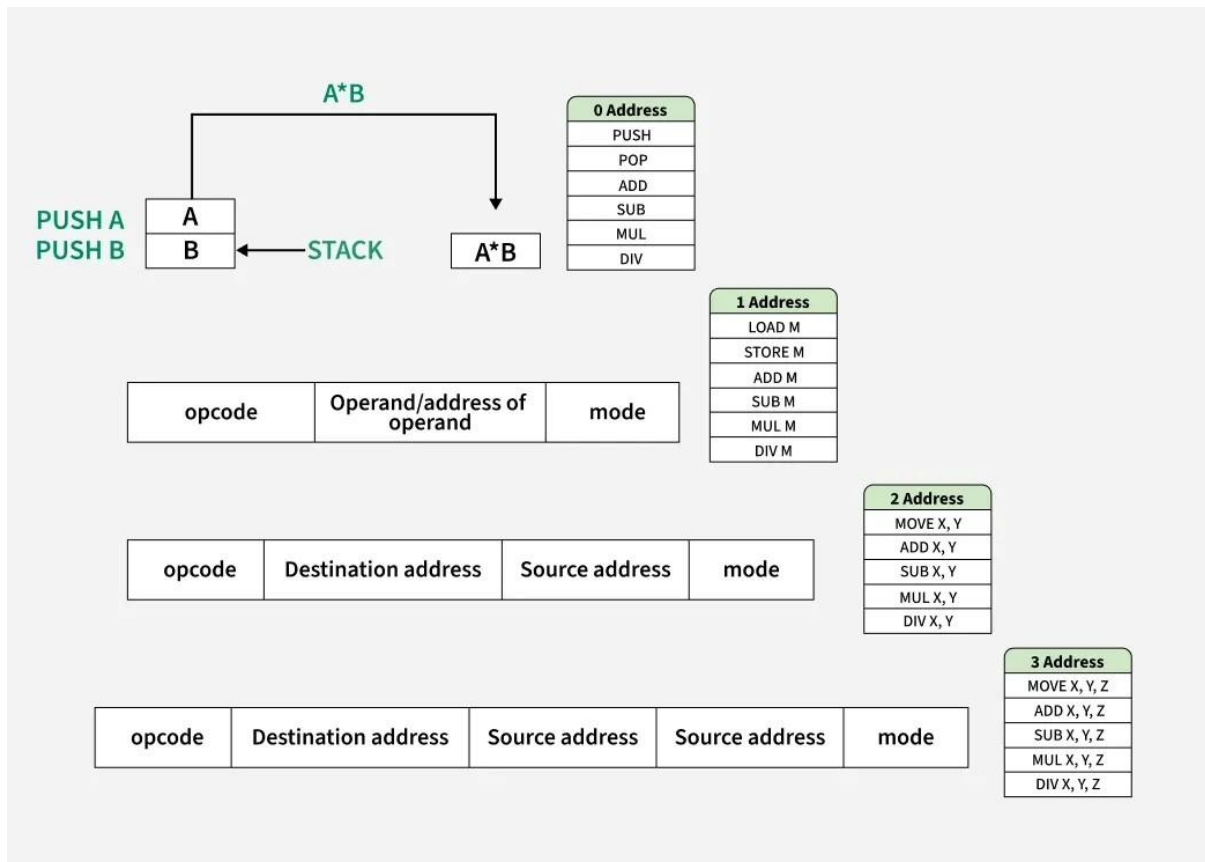
## Instruction Set and Addressing Modes

### 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

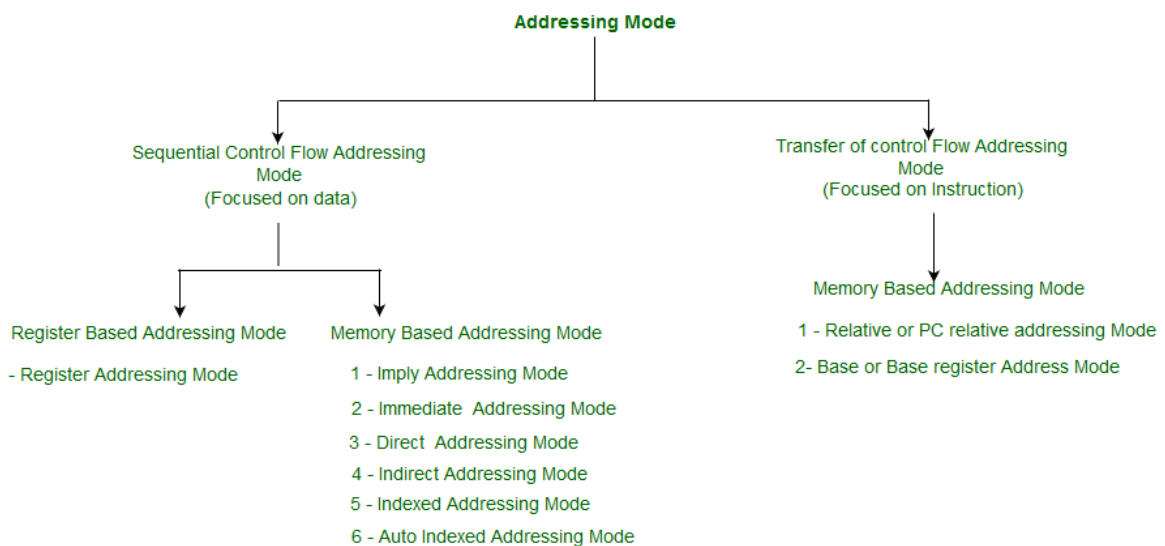
- **Data Transfer Instructions**
  - Move data between memory, registers, or I/O devices.
  - Example: LOAD, STORE, MOVE.
- **Arithmetic Instructions**
  - Perform arithmetic operations like addition, subtraction, multiplication, and division.
  - Example: ADD, SUB, MUL, DIV.

- **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
<b>Immediate Addressing</b>	The operand is directly given in the instruction.	ADD R1, 5 (Add 5 to R1)
<b>Register Addressing</b>	The operand is stored in a register.	ADD R1, R2 (Add R2 to R1)
<b>Direct Addressing</b>	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)
<b>Indirect Addressing</b>	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)
<b>Register Indirect</b>	Similar to indirect addressing, but specifically uses registers to hold the address of the operand.	LOAD R1, (R3) (Use R3 as pointer)
<b>Indexed Addressing</b>	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)
<b>Base Addressing</b>	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
<b>Relative Addressing</b>	The operand's address is determined by adding an offset to the current program counter (PC).	JUMP 200 (Jump to PC + 200)
<b>Implicit Addressing</b>	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 <a href="#">Hardwired control unit</a>	Uses both hardwired and <a href="#">microprogrammed control unit</a>
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 <a href="#">RAM</a> .	CISC required less RAM.
Here, Addressing modes are less.	Here, Addressing modes are more.

Read more about RISC vs CISC, [Here](#).

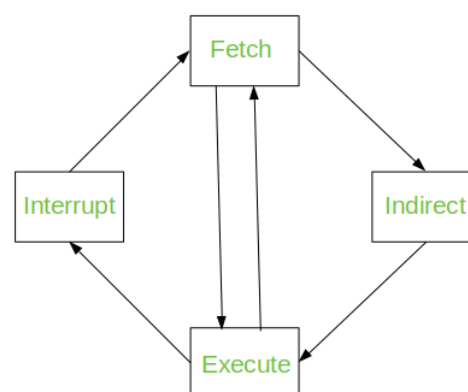
## Instruction Design and Format

### 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.

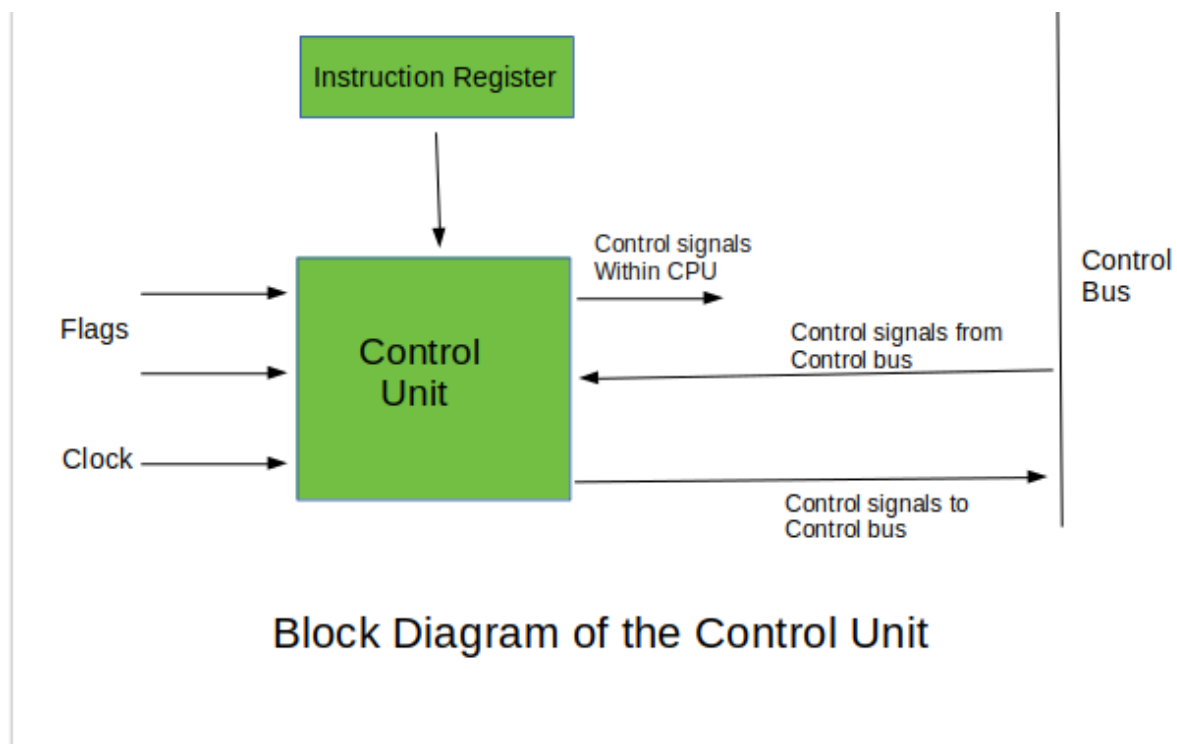
### Standard Instruction Cycle

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



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-  $\text{Log}_2(N)$  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:  $\text{MAR} \leftarrow (\text{PC})$

t2:  $\text{MBR} \leftarrow \text{Memory}$

$\text{PC} \leftarrow (\text{PC}) + 1$

t3:  $\text{IR} \leftarrow (\text{MBR})$

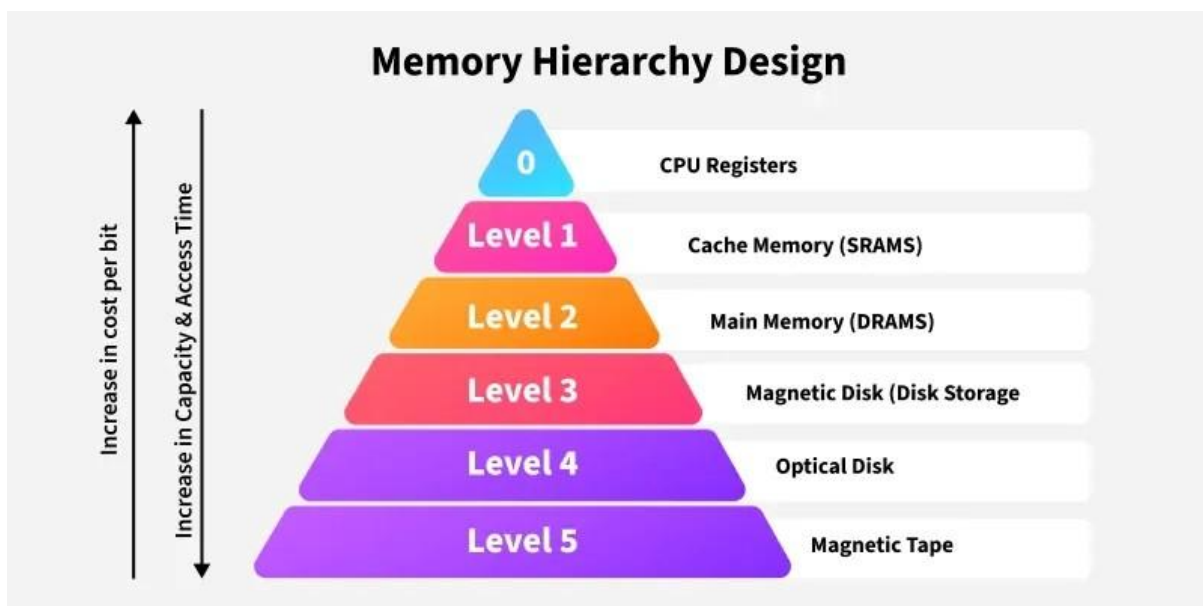
### Memory Organization

- 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.

<u>Byte Addressable Memory</u>	<u>Word Addressable Memory</u>
When the <i>data space in the cell = 8 bits</i> then the corresponding <i>address space</i> is called as <u>Byte Address</u> .	When the <i>data space in the cell = word length of CPU</i> then the corresponding <i>address space</i> is called as <u>Word Address</u> .

<u>Byte Addressable Memory</u>	<u>Word Addressable Memory</u>
Based on this data storage i.e. <i>Bytewise storage</i> , the memory chip configuration is named as <b>Byte Addressable Memory</b> .	Based on this data storage i.e. <i>Wordwise storage</i> , the memory chip configuration is named as <b>Word Addressable Memory</b> .

## Memory Hierarchy



**Simultaneous access memory organization:** If  $H_1$  and  $H_2$  are the Hit Ratios and  $T_1$  and  $T_2$  are the access time of L1 and L2 memory levels respectively then the

*Average Memory Access Time can be calculated as:*

$$T = (H_1 * T_1) + ((1 - H_1) * H_2 * T_2) \quad T = (H_1 * T_1) + ((1 - H_1) * H_2 * T_2)$$

**Hierarchical Access Memory Organization:** If  $H_1$  and  $H_2$  are the Hit Ratios and  $T_1$  and  $T_2$  are the access time of L1 and L2 memory levels respectively then

*Average Memory Access Time can be calculated as:*

$$T = (H_1 * T_1) + ((1 - H_1) * H_2 * (T_1 + T_2)) \quad T = (H_1 * T_1) + ((1 - H_1) * H_2 * (T_1 + T_2))$$

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.

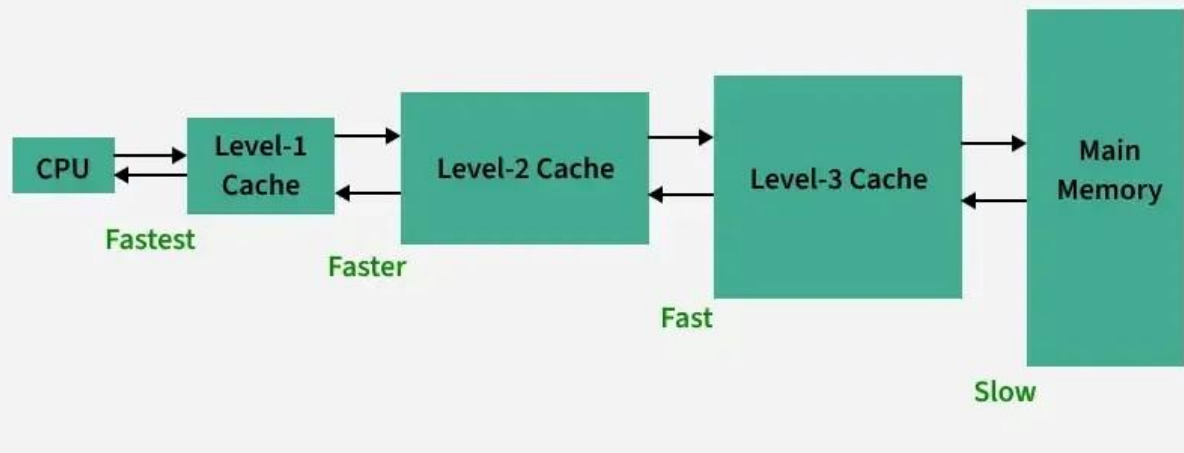
***Cache Set Number = Main Memory block number % Number of sets in cache***

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**

## Three Level Cache Organization



### 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.  $\text{Hit Ratio} = \frac{\text{Cache Hits}}{\text{Total Accesses}}$
  2. **Miss Ratio:** Percentage of memory accesses that result in a miss.  $\text{Miss Ratio} = 1 - \text{Hit Ratio}$
- Effective Memory Access Time (EMAT)**  
For 2-level cache:

$$EMAT = H_1 \times T_1 + (1 - H_1) \times [H_2 \times T_2 + (1 - H_2) \times T_M]$$

- $H_1, H_2$ : Hit ratios for L1 and L2 caches.
- $T_1, T_2$ : Access times for L1 and L2 caches.
- $T_M$ : 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:*

$$T_{read} = H \times T_{cache} + (1 - H) \times (T_{cache} + T_{memory\_block}) = T_{cache} + (1 - H) \times T_{memory\_block}$$

*For simultaneous access :*  $T_{read} = H \times T_{cache} + (1 - H) \times T_{memory\_block}$   
 $T_{write} = T_{memory\_block}$

**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:*

$$T_{read} = T_{write} = H \times T_{cache} + (1 - H) \times (T_{cache} + T_{memory\_block} + T_{write\_back}) = T_{cache} + (1 - H) \times (T_{memory\_block} + T_{write\_back}),$$

where  $T_{write\_back} = x \times T_{memory\_block}$ , where  $x$  is the fraction of

dirty blocks  $T_{read} = T_{write} = H \times T_{cache} + (1-H) \times (T_{cache} + T_{memory\_block} + T_{write\_back}) = T_{cache} + (1-H) \times (T_{memory\_block} + T_{write\_back})$ , where  $T_{write\_back} = x \times T_{memory\_block}$ , where  $x$  is the fraction of dirty blocks

For simultaneous access :

$T_{read} = T_{write} = H \times T_{cache} + (1-H) \times (T_{memory\_block} + T_{write\_back})$ , where  $T_{write\_back} = x \times T_{memory\_block}$ , where  $x$  is the fraction of dirty blocks  
 $T_{read} = T_{write} = H \times T_{cache} + (1-H) \times (T_{memory\_block} + T_{write\_back})$ , where  $T_{write\_back} = x \times T_{memory\_block}$ , where  $x$  is the fraction of dirty blocks

Read more about Cache Memory, [Here](#).

### Cache Miss

Type of Miss	Reason
<b>Compulsory Miss</b>	First-time access to data
<b>Conflict Miss</b>	Multiple blocks mapped to same cache line
<b>Capacity Miss</b>	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.

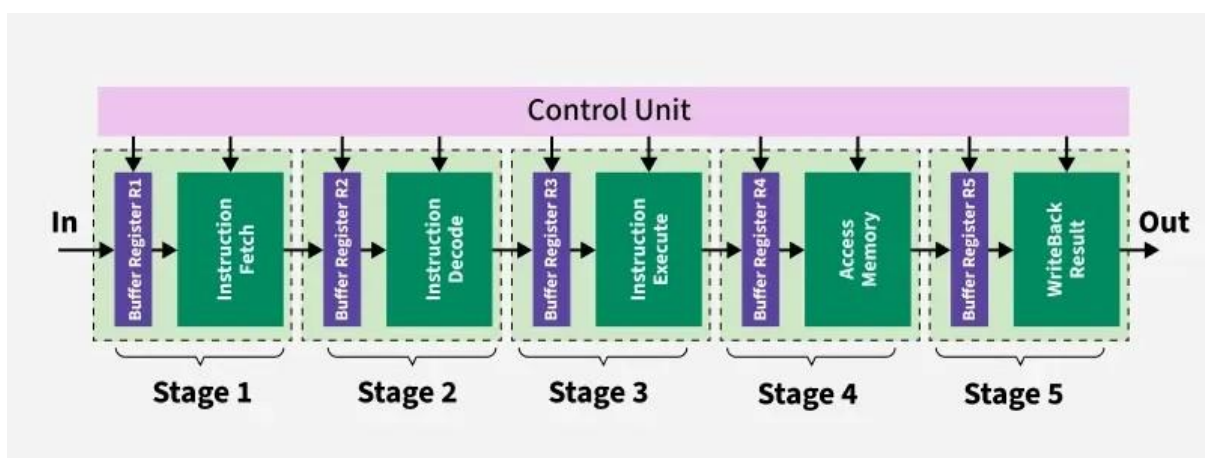
This process enables efficient and high-speed data transfer while freeing the CPU to perform other tasks.

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:

$$ET_{pipeline} = k + n - 1 \text{ cycles}$$

$$= (k + n - 1) T_p$$

In the same case, for a non-pipelined processor, execution time of 'n' instructions will be:

$$ET_{non-pipeline} = n * k * T_p$$

So, speedup (S) of the pipelined processor over non-pipelined processor, when 'n' tasks are executed on the same processor is:

$$S = \text{Performance of pipelined processor} / \text{Performance of Non-pipelined processor}$$

As the performance of a processor is inversely proportional to the execution time, we have:

$$S = ET_{non-pipeline} / ET_{pipeline}$$

$$\Rightarrow S = [n * k * T_p] / [(k + n - 1) * T_p]$$

$$S = [n * k] / [k + n - 1]$$

When the number of tasks 'n' are significantly larger than k, that is,  $n \gg k$

$$S = n * k / n$$

$$S = k$$

where 'k' are the number of stages in the pipeline. Also,

$$\text{Efficiency} = \text{Given speed up} / \text{Max speed up} = S / S_{max}$$

We know that,  $S_{max} = k$  So,

$$\text{Efficiency} = S / k$$

**Throughput** = Number of instructions / Total time to complete the instructions So,

$$\text{Throughput} = n / (k + n - 1) * T_p$$

Note: The cycles per instruction (CPI) value of an ideal pipelined processor is 1

### Performance of pipeline with stalls

$$\text{Speed Up (S)} = \text{CPI}_{non-pipeline} / (1 + \text{Number of stalls per instruction})$$

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. 0 and 1. So a normalised mantissa is one with only one 1 to the left of the decimal.

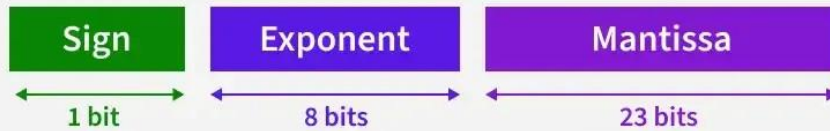
The IEEE 754 Standard is used to represent floating-point numbers in binary. It has two formats:

1. **Single Precision (32-bit)**
2. **Double Precision (64-bit)**

## IEEE 754 Floating Point Standard

S: Sign bit  
M: Mantissa  
E: Exponent

### Single Precision -32 bits



$$\text{Formula: } (-1)^S \times (1.M) \times 2^{(E-127)}$$

### Double Precision -64 bits



$$\text{Formula: } (-1)^S \times (1.M) \times 2^{(E-1023)}$$

*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**

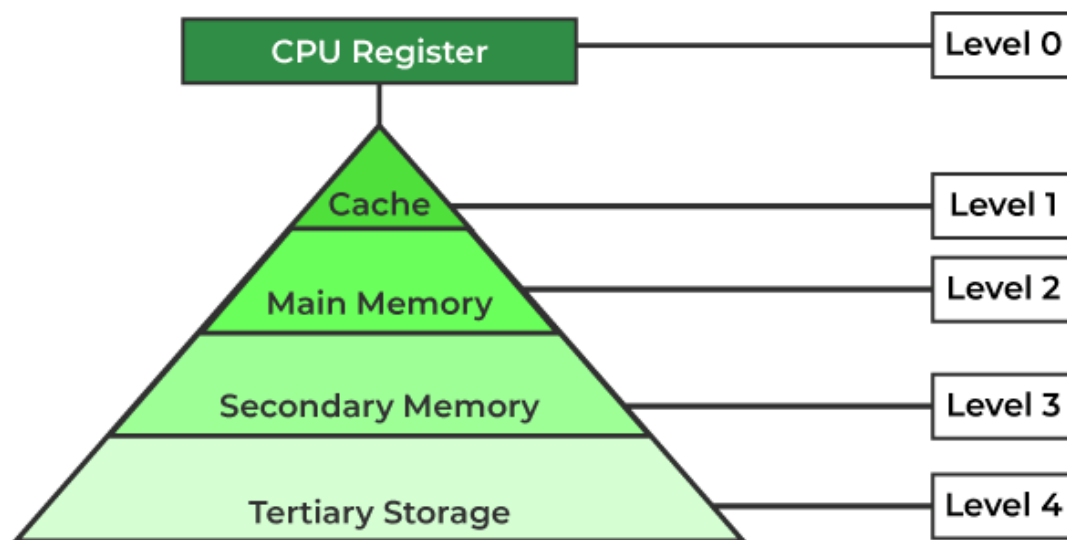
The term “execution” as used in computers refers to an action or task the computer actually undertakes as specified in the instruction that has been fetched and decoded. Once it has understood what is required of it, the computer carries out a certain operation. This could involve performing calculations, 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

### 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):** Slower 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 Architecture](#)[Computer Science](#)[Network](#)

---



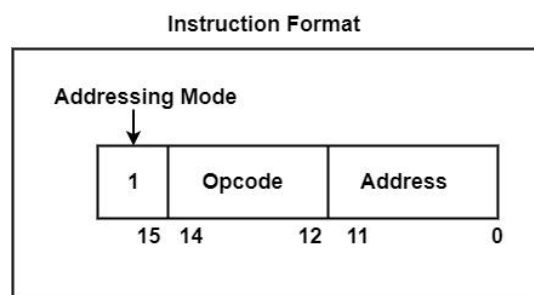
---

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  $2^n$  operations. These operations are implemented on information that is saved in processor registers or memory.

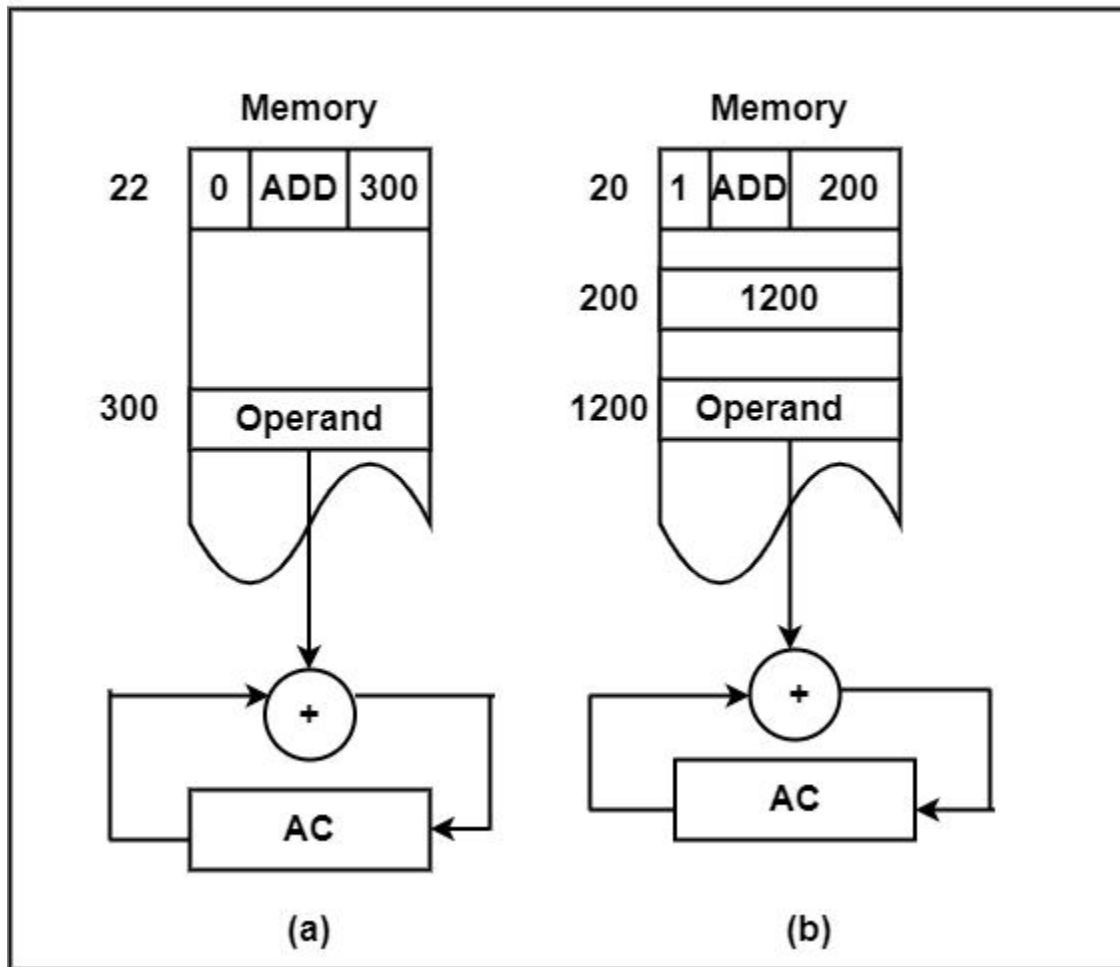
### Address

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.

## Direct and Indirect Address



### 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).

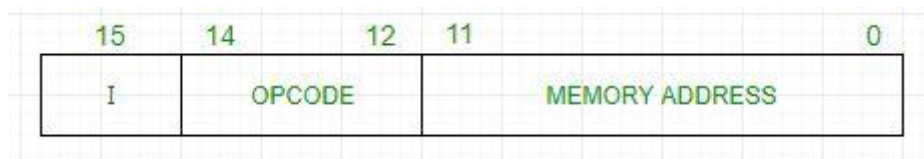
- **Call:** Transfers control to a subroutine (a set of instructions performing a specific task) and returns afterward.
- **Return:** Brings back control to the main program after a subroutine is executed.

### 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.

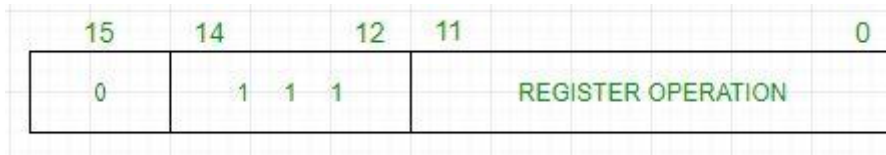
#### Memory Reference Instructions

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.

$AC \leftarrow AC + M[0001]$

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

...

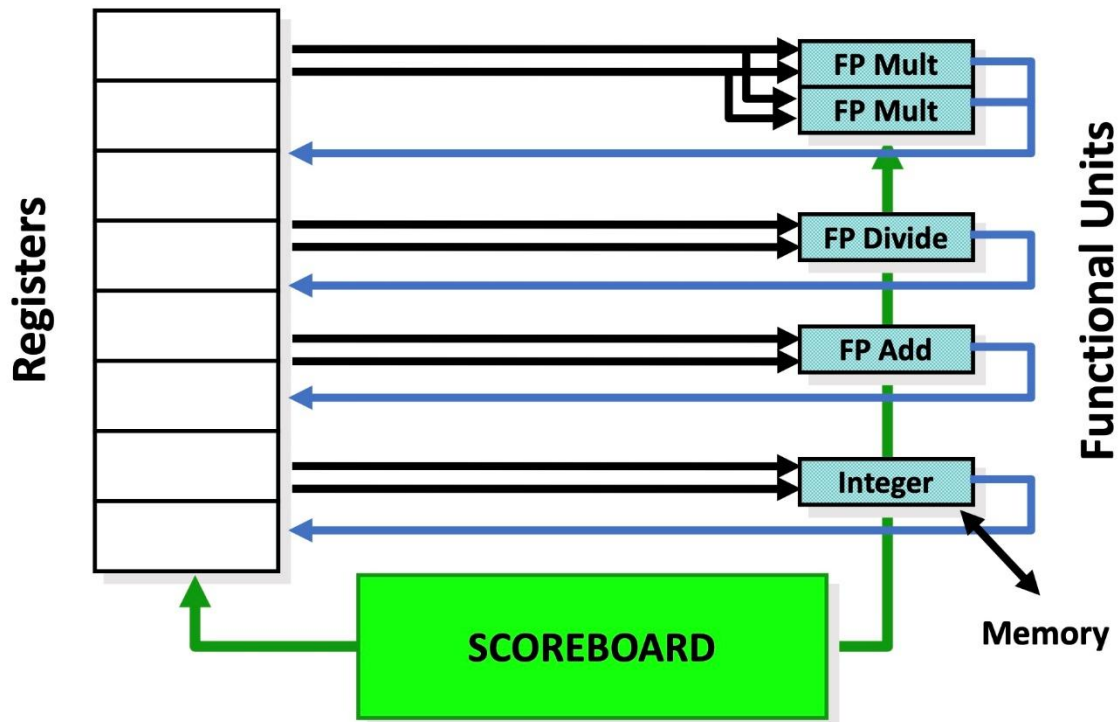
Next: SUB R7,R6,R4

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	$\forall f((Fj(f) \neq Fi(FU) \text{ or } Rj(f) = \text{No}) \& (Fk(f) \neq Fi(FU) \text{ or } Rk(f) = \text{No}))$	$\forall f(\text{if } Qj(f) = \text{FU} \text{ then } Rj(f) \leftarrow \text{Yes};$ $\forall f(\text{if } Qk(f) = \text{FU} \text{ then } Rk(f) \leftarrow \text{Yes};$ Rg.Result(Fi(FU)) ← 0; Busy(FU) ← No

**Instruction status:**

Instruction	<i>j</i>	<i>k</i>	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 <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU</i> <i>Qj</i>	<i>FU</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

**Register result status:**

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
<i>FU</i>									

# Scoreboard Example: Cycle 1

**Instruction status:**

Instruction	<i>j</i>	<i>k</i>	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 <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU</i> <i>Qj</i>	<i>FU</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

**Register result status:**

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
1				Integer					

## Scoreboard Example: Cycle 2

### Instruction status:

Instruction	<i>j</i>	<i>k</i>	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 <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU</i> <i>Qj</i>	<i>FU</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

### Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
2	<i>FU</i> 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	<i>j</i>	<i>k</i>	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 <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU</i> <i>Qj</i>	<i>FU</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
	Integer	Yes	Load	F6		R2				No
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

### Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
3	<i>FU</i> 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	Read			Write	
			Issue	Oper	Comp	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		FU	FU	Fj?	Fk?
				Fi	Fj				
	Integer	No							
	Mult1	No							
	Mult2	No							
	Add	No							
	Divide	No							

### Register result status:

Clock	FU										
	F0	F2	F4	F6	F8	F10	F12	...	F30		
4											

## Scoreboard Example: Cycle 5

### Instruction status:

Instruction	j	k	Read			Write	
			Issue	Oper	Comp	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		FU	FU	Fj?	Fk?
				Fi	Fj				
	Integer	Yes	Load	F2		R3			Yes
	Mult1	No							
	Mult2	No							
	Add	No							
	Divide	No							

### Register result status:

Clock	FU										
	F0	F2	F4	F6	F8	F10	F12	...	F30		
5				Integer							

# Scoreboard Example: Cycle 6

## Instruction status:

Instruction	<i>j</i>	<i>k</i>	Read Exec Write				
			Issue	Oper	Comp	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	<i>Busy</i>	<i>Op</i>	<i>dest</i>		<i>S1</i>	<i>S2</i>	<i>FU</i>	<i>FU</i>	<i>Fj?</i>	<i>Fk?</i>
				<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>	
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	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
6									
<i>FU</i>	Mult1	Integer							

# Scoreboard Example: Cycle 7

## Instruction status:

Instruction	<i>j</i>	<i>k</i>	Read Exec Write				
			Issue	Oper	Comp	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	<i>Busy</i>	<i>Op</i>	<i>dest</i>		<i>S1</i>	<i>S2</i>	<i>FU</i>	<i>FU</i>	<i>Fj?</i>	<i>Fk?</i>
				<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>	
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	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
7									
<i>FU</i>	Mult1	Integer			Add				

Two instructions in Issue; No forwarding

# Scoreboard Example: Cycle 8a (part 1)

## Instruction status:

Instruction	<i>j</i>	<i>k</i>	Read Exec Write				
			Issue	Oper	Comp	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		S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj						
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	<i>j</i>	<i>k</i>	Read Exec Write				
			Issue	Oper	Comp	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		S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj						
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	<i>j</i>	<i>k</i>	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 <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU</i> <i>Qj</i>	<i>FU</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
	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	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
9	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.

o o o

# Scoreboard Example: Cycle 12

## Instruction status:

Instruction	<i>j</i>	<i>k</i>	Read Exec Write				
			Issue	Oper	Comp	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		S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj						
	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 forgot 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.

o o o

# Scoreboard Example: Cycle 17

## Instruction status:

Instruction	j	k	Read Exec Write			
			Issue	Oper	Comp	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		FU	FU	Fj?	Fk?
				Fi	Fj				
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		No	Yes

## Register result status:

Clock	FU								
	F0	F2	F4	F6	F8	F10	F12	...	F30
17	Mult1			Add		Divide			

? Why not write the result of ADDD?

🤔 There is a WAR Hazard.

# Scoreboard Example: Cycle 21

## Instruction status:

Instruction	j	k	Read Exec Write			
			Issue	Oper	Comp	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		FU	FU	Fj?	Fk?
				Fi	Fj				
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	FU								
	F0	F2	F4	F6	F8	F10	F12	...	F30
21				Add		Divide			

WAR Hazard is now gone...

# Scoreboard Example: Cycle 22

## Instruction status:

Instruction	<i>j</i>	<i>k</i>	Read Exec Write				
			Issue	Oper	Comp	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	S1	S2	FU	FU	Fj?	Fk?
			Fi	Fj	Fk	Qj	Qk	Rj	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	<i>j</i>	<i>k</i>	Read Exec Write				
			Issue	Oper	Comp	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	S1	S2	FU	FU	Fj?	Fk?
			Fi	Fj	Fk	Qj	Qk	Rj	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, ...

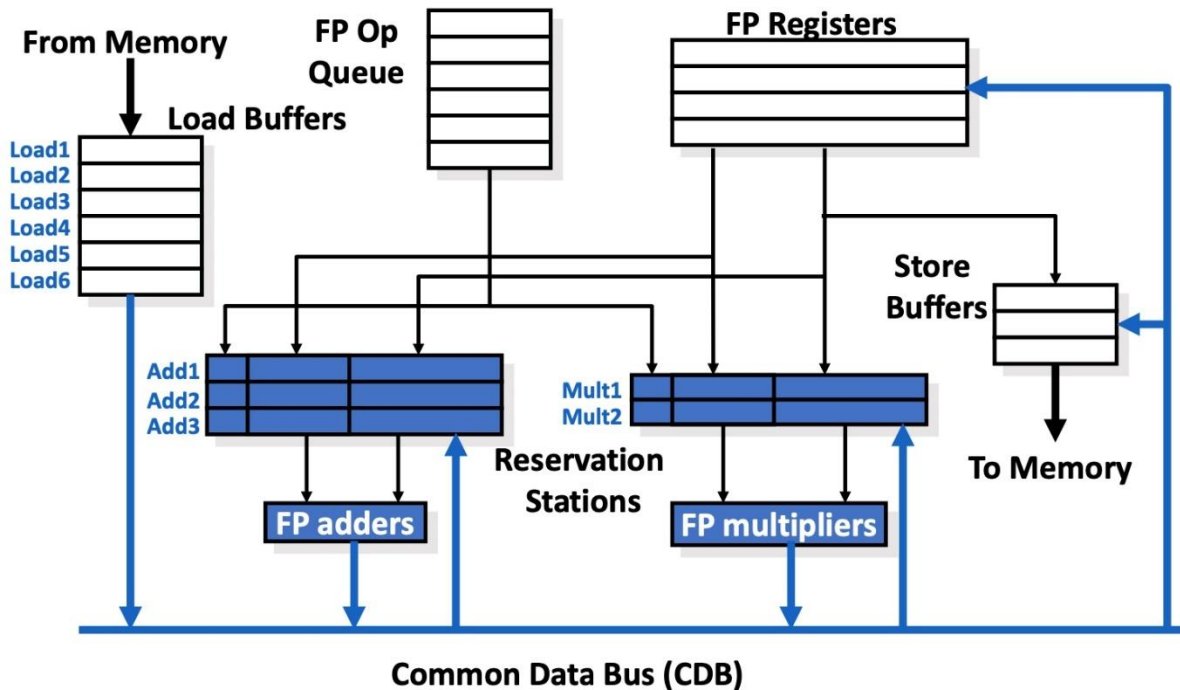
### Tomasulo Algorithm

- 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)
- $V_j, V_k$ : Value of Source operands
  - Store buffers has  $V_j$  field, result to be stored
- $Q_j, Q_k$ : Reservation stations producing source registers (value to be written)
  - Note:  $Q_j, Q_k=0 \Rightarrow$  ready
- Busy: Indicates reservation station or FU is busy
- Register result status ( $R_g.Result$ ) — 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)

## Instruction stream

### Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Rj</i>	<i>Rk</i>	Exec		Write
					Issue	Comp	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				

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

3 Load/Buffers

### Reservation Stations:

Time	Name	Busy	<i>Op</i>	<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
				<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

FU count  
down

3 FP Adder R.S.  
2 FP Mult R.S.

### Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
0									

Clock cycle counter

## Tomasulo Example Cycle 1

### Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Rj</i>	<i>Rk</i>	Exec		Write
					Issue	Comp	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				

	Busy	Address
Load1	Yes	34+R2
Load2	No	
Load3	No	

### Reservation Stations:

Time	Name	Busy	<i>Op</i>	<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
				<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

### Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
1				Load1					

# Tomasulo Example Cycle 2

## Instruction status:

Instruction	<i>j</i>	<i>k</i>	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	<i>Op</i>	<i>S1</i> <i>Vj</i>	<i>S2</i> <i>Vk</i>	<i>RS</i> <i>Qj</i>	<i>RS</i> <i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

## Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
2		Load2			Load1				

Note: Can have multiple loads outstanding

# Tomasulo Example Cycle 3

## Instruction status:

Instruction	<i>j</i>	<i>k</i>	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	<i>Op</i>	<i>S1</i> <i>Vj</i>	<i>S2</i> <i>Vk</i>	<i>RS</i> <i>Qj</i>	<i>RS</i> <i>Qk</i>
	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		Load2	Mult1			Load1			

Note: MULT issued; registers names are removed ("renamed") in Reservation Stations

# Tomasulo Example Cycle 4

## Instruction status:

Instruction	<i>j</i>	<i>k</i>	Issue	Exec Comp	Write Result	Load1	Busy	Address
LD	F6	34+	R2	1	3	4	No	
LD	F2	45+	R3	2	4		Yes	45+R3
MULTD	F0	F2	F4	3			No	
SUBD	F8	F6	F2	4				
DIVD	F10	F0	F6					
ADDD	F6	F8	F2					

## Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Oj	RS Ok
	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	Mult1	Load2		0000	Add1				

# Tomasulo Example Cycle 5

## Instruction status:

Instruction	<i>j</i>	<i>k</i>	Issue	Exec Comp	Write Result	Load1	Busy	Address
LD	F6	34+	R2	1	3	4	No	
LD	F2	45+	R3	2	4	5	No	
MULTD	F0	F2	F4	3			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 Oj	RS Ok
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	Mult1	0000			Add1	Mult2			

Timer starts down for Add1, Mult1

# Tomasulo Example Cycle 6

## Instruction status:

Instruction	<i>j</i>	<i>k</i>	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	<i>S1</i> <i>Vj</i>	<i>S2</i> <i>Vk</i>	<i>RS</i> <i>Qj</i>	<i>RS</i> <i>Qk</i>
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	<i>j</i>	<i>k</i>	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	<i>S1</i> <i>Vj</i>	<i>S2</i> <i>Vk</i>	<i>RS</i> <i>Qj</i>	<i>RS</i> <i>Qk</i>
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	<i>j</i>	<i>k</i>	<i>Rj</i>	<i>Rk</i>	Exec		Write	Load1	Load2	Load3	Busy	Address
					Issue	Comp	Result					
LD	F6	34+	R2	R2	1	3	4	No	No	No		
LD	F2	45+	R3	R3	2	4	5	No	No	No		
MULTD	F0	F2	F4	F4	3			No	No	No		
SUBD	F8	F6	F2	F2	4	7	8					
DIVD	F10	F0	F6	F6	5							
ADD	F6	F8	F2	F2	6							

## Reservation Stations:

Time	Name	Busy	Op	<i>S1</i> <i>Vj</i>	<i>S2</i> <i>Vk</i>	<i>RS</i> <i>Qj</i>	<i>RS</i> <i>Qk</i>
	Add1	No					
2	Add2	Yes	ADD	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	<i>j</i>	<i>k</i>	<i>Rj</i>	<i>Rk</i>	Exec		Write	Load1	Load2	Load3	Busy	Address
					Issue	Comp	Result					
LD	F6	34+	R2	R2	1	3	4	No	No	No		
LD	F2	45+	R3	R3	2	4	5	No	No	No		
MULTD	F0	F2	F4	F4	3			No	No	No		
SUBD	F8	F6	F2	F2	4	7	8					
DIVD	F10	F0	F6	F6	5							
ADD	F6	F8	F2	F2	6	10	11					

## Reservation Stations:

Time	Name	Busy	Op	<i>S1</i> <i>Vj</i>	<i>S2</i> <i>Vk</i>	<i>RS</i> <i>Qj</i>	<i>RS</i> <i>Qk</i>
	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	<i>j</i>	<i>k</i>	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	<i>S1</i> <i>Vj</i>	<i>S2</i> <i>Vk</i>	<i>RS</i> <i>Qj</i>	<i>RS</i> <i>Qk</i>
	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	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
15	FU	Mult1				Mult2			

# Tomasulo Example Cycle 16

## Instruction status:

Instruction	<i>j</i>	<i>k</i>	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	<i>S1</i> <i>Vj</i>	<i>S2</i> <i>Vk</i>	<i>RS</i> <i>Qj</i>	<i>RS</i> <i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	M2*F4		R(F6)	

## Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
16	FU	0000				Mult2			

Just waiting for Mult2 (DIVD) to complete

# Tomasulo Example Cycle 57

## Instruction status:

Instruction	<i>j</i>	<i>k</i>	Exec			Write	Busy	Address
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	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	<i>Op</i>	<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
				<i>V<sub>j</sub></i>	<i>V<sub>k</sub></i>	<i>O<sub>j</sub></i>	<i>O<sub>k</sub></i>
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

## Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
56						0000			

Once again: In-order issue, out-of-order execution and out-of-order completion.

## Drawbacks

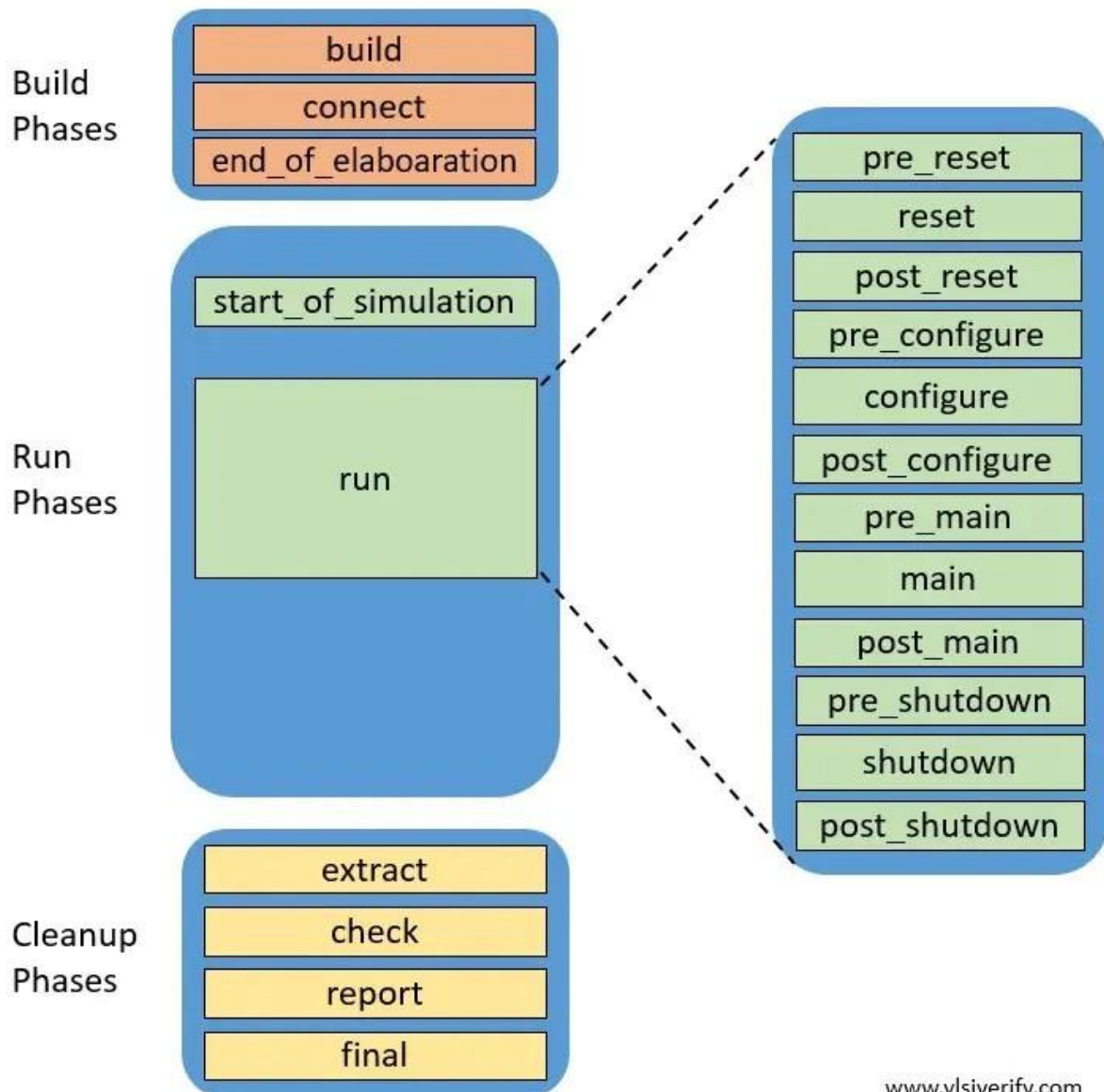
- High Complexity (large circuit, high power consumption)
- Many stores (CDB) at high speed resulting limit on performance by Common Data Bus
  - Each CDB must go to multiple functional units =» high wiring density, high capacitance
  - Number of functional units that can complete per cycle limited to 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:

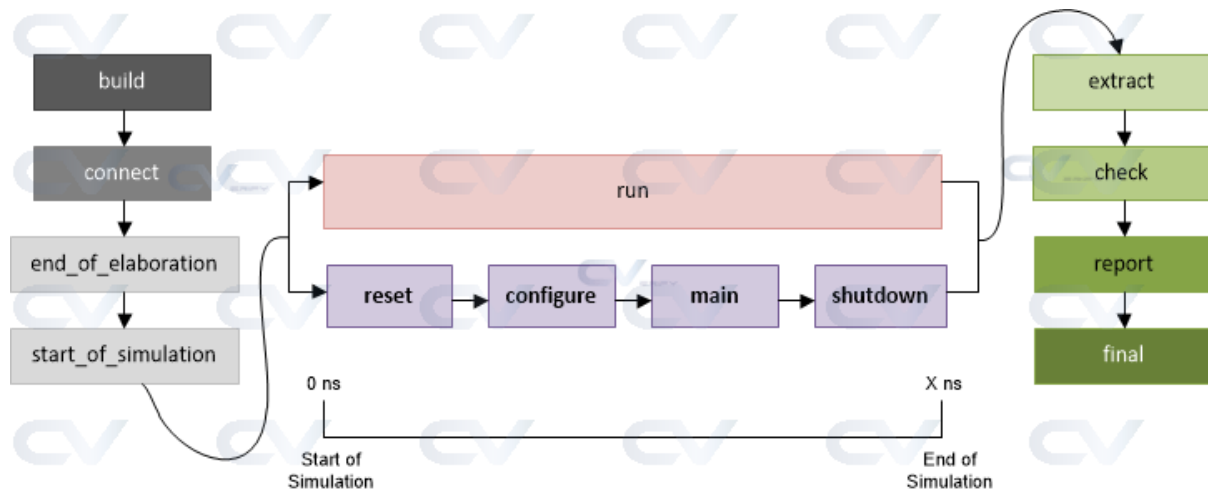


[www.vlsiverify.com](http://www.vlsiverify.com)

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	<code>build_phase</code>	function	Used to build testbench components and create their instances
	<code>connect_phase</code>	function	Used to connect between different testbench components via TLM ports
	<code>end_of_elaboration_phase</code>	function	Used to display UVM topology and other functions required to be done after connection
	<code>start_of_simulation_phase</code>	function	Used to set initial run-time configuration or display topology
Run	<code>run_phase</code>	task	Actual simulation that consumes time happens in this UVM phase and runs parallel to other UVM run-time phases.

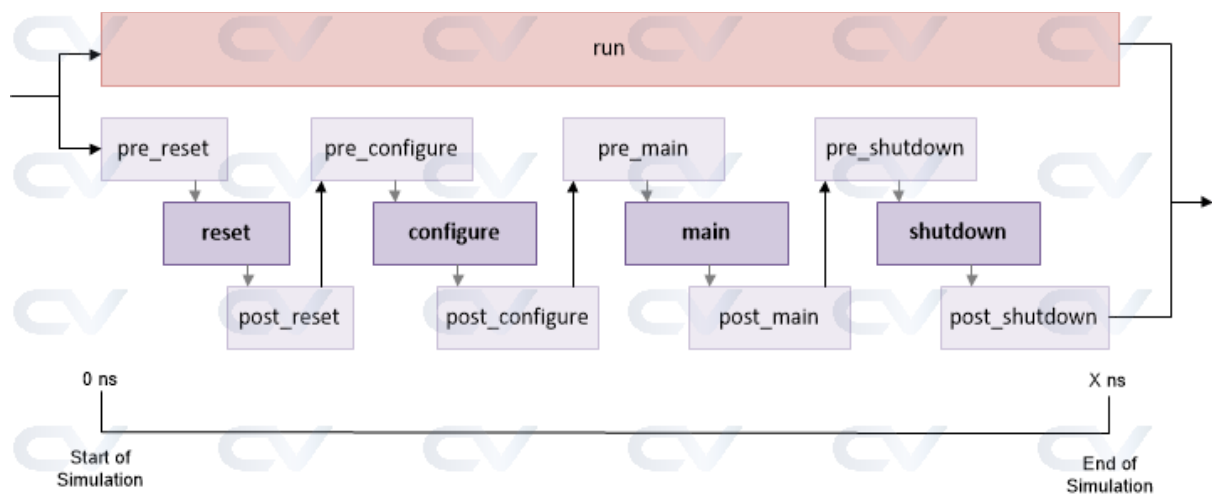
Clean	extract_phase	function	Used to extract and compute expected data from scoreboard, an
	check_phase	function	Used to perform scoreboard tasks that check for errors between expected and actual values from design
	report_phase	function	Used to display result from checkers, or summary of other test 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.

<b>Phase</b>	<b>Upon Entry</b>	<b>Typical Uses</b>	<b>Exit Criteria</b>
--------------	-------------------	---------------------	----------------------

build	Top-Level components are instantiated under uvm_root	<ul style="list-style-type: none"> <li>• Instantiate sub-components</li> <li>• Instantiate register model</li> <li>• Get configuration values for components being built</li> <li>• Set configuration values for sub-components</li> </ul>	<ul style="list-style-type: none"> <li>• All uvm_components have been instantiated.</li> </ul>
connect	All components have been instantiated	<ul style="list-style-type: none"> <li>• Connect TLM ports and exports, initiator sockets and target sockets</li> <li>• Connect register model to adapter components</li> <li>• Setup explicit phase domains</li> </ul>	<ul style="list-style-type: none"> <li>• Connections between various components have been established</li> <li>• All independent phase domains are set</li> </ul>
end_of_elaboration	Verification environment has been assembled	<ul style="list-style-type: none"> <li>• Display environment topology</li> <li>• Open files</li> <li>• Define additional configuration settings</li> </ul>	None
start_of_simulation	<ul style="list-style-type: none"> <li>• External tools/simulation engines have started and synced</li> <li>• Verification environment</li> </ul>	<ul style="list-style-type: none"> <li>• Display environment topology</li> <li>• Set debugger breakpoint</li> </ul>	None

	is configured and ready to be started	<ul style="list-style-type: none"> <li>Set initial run-time configuration values</li> </ul>	
run	<ul style="list-style-type: none"> <li>Indicates that power has been applied</li> <li>No active edges of clock should have happened before (x&gt;1)</li> </ul>	Components run over various run-time phases	<ul style="list-style-type: none"> <li>DUT no longer needs to be simulated</li> <li>post_shutdown_phase is ready to end</li> </ul>
extract	<ul style="list-style-type: none"> <li>DUT no longer needs to be simulated</li> <li>Simulation time will no longer advance</li> </ul>	<ul style="list-style-type: none"> <li>Extract data/final state information from scoreboard/testbench components</li> <li>Probe the DUT (hierarchical reference/backdoor or access)</li> <li>Calculate statistics and summaries</li> <li>Display final state information</li> <li>Close files</li> </ul>	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	<ul style="list-style-type: none"> <li>Report test results</li> </ul>	End of test

		<ul style="list-style-type: none"><li>• Write results to file</li></ul>	
final	All test-related activity has completed	<ul style="list-style-type: none"><li>• Close files</li><li>• Terminate co-simulation engines</li></ul>	Ready to exit simulator