ECE 551
Digital Design And Synthesis

Fall ‘21

UVM & Object Oriented Test Benches

Administrative Matters

- Should be working on Project Stuff
  - TourLogic
  - TourCmd
  - Top Level Testbench
  - Synthesis of Toplevel

UVM

What is UVM ?

- **U**niversal **V**erification **M**ethodology
- Object Oriented approach for reusability
- Created for the need to automate verification
    - As an example - UVM allows dynamically configurable test-bench
    - Allows compiling test-bench once and run with different arguments, stimulus to cover all scenarios
    - Big Designs - 1000 tests * 5 min compile time per test = That's 5000 min saved !!

UVM : We can only scratch the surface

Universal Verification Methodology(UVM) based on:

- **System Verilog Object-Oriented Programming**
- Dynamically generated objects to specify TB
- Transaction level - communication between objects
- Stimulus – UVM sequences

- *We only have enough time to introduce some OOP portions of system Verilog in relation to UVM.*

SV: Object Oriented Programming

Enables OOP through **class** data type

- Code Re-use
  - Implement functionality in TB, re-use it to create more complex functionality without knowing internal details - Encapsulation
- Code maintainability
  - Write common code in one place, access it anywhere from your test-bench

SV Classes

A class is a type

- Contains data referred to as class properties
- Contains subroutines (task/functions) referred to as class **methods**

```
typedef enum reg[1:0] {IDLE,RUN, ...} cmd_t;

class Packet;
   cmd_t Command;
   int Status;
   logic [31:0] Data [0:255];
   function int GetStatus();
     return(Status);
   endfunction : GetStatus
   task SetCommand (input cmd_t a);
     Command = a;
   endtask : SetCommand
endclass : Packet
```

SV Classes

Classes are dynamically created objects (class instance)

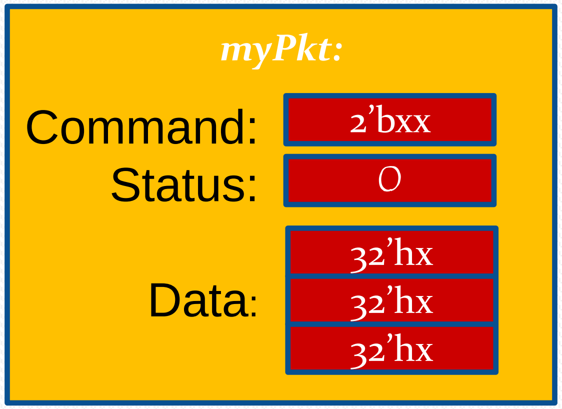
- Every class has a method *new()* call the constructor
- It can be explicitly specified or built-in
- It can take input aruguments or not

```
typedef enum reg[1:0] {IDLE,RUN, ...} cmd_t;

class Packet;
   cmd_t Command;
   int Status;
   logic [31:0] Data [0:255];
   function int GetStatus();
     return(Status);
   endfunction : GetStatus
   task SetCommand (input cmd_t a);
     Command = a;
   endtask : SetCommand
endclass : Packet
```

Class *Packet* has no explicit method *new()*

```
Packet myPkt = new;
```

Invoking constructor *new* creates an instance of class *Packet* called *myPkt*

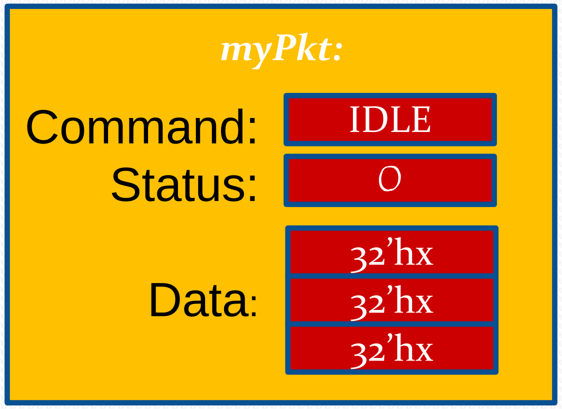


SV Classes (more constructor examples)

```
typedef enum reg[1:0] {IDLE,RUN, ...} cmd_t;

class Packet;
   cmd_t Command;
   int Status;
   logic [31:0] Data [0:255];
   function new();
     Command = IDLE;
   endfunction
   function int GetStatus();
     return(Status);
   endfunction : GetStatus
   task SetCommand (input cmd_t a);
     Command = a;
   endtask : SetCommand
endclass : Packet
```

```
Packet myPkt = new;
```

Invoking constructor *new* creates an instance of class *Packet* called *myPkt*



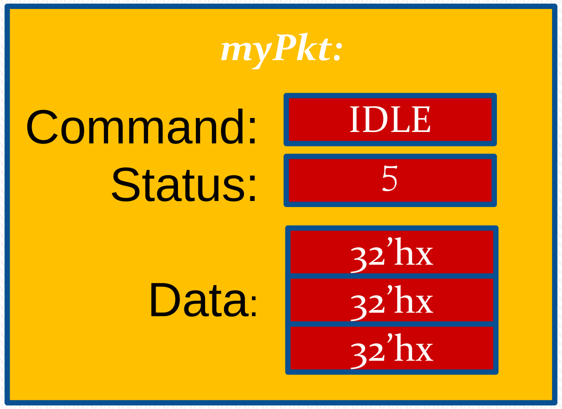
- In this example the class contains an explicit constructor function *new* that initializes the member *Command* to IDLE.

SV Classes (more constructor examples)

```
typedef enum reg[1:0] {IDLE,RUN, ...} cmd_t;

class Packet;
   cmd_t Command;
   int Status;
   logic [31:0] Data [0:255];
   function new(input int a);
     Command = IDLE;
     Status = a;
   endfunction
   function int GetStatus();
     return(Status);
   endfunction : GetStatus
   task SetCommand (input cmd_t a);
     Command = a;
   endtask : SetCommand
endclass : Packet
```

```
Packet myPkt = new(5);
```

Invoking the constructor with an input argument.



- In this example the constructor function takes an input argument that allows flexibility in the initialization of the instance created.

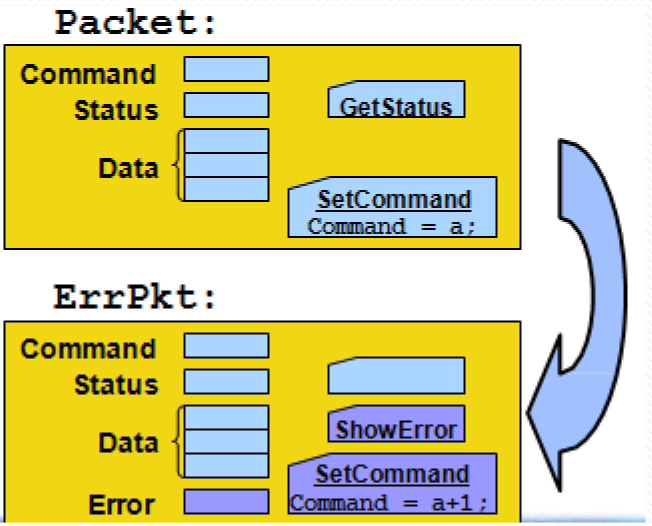
Inheritance

Classes can inherit properties and methods from other classes:

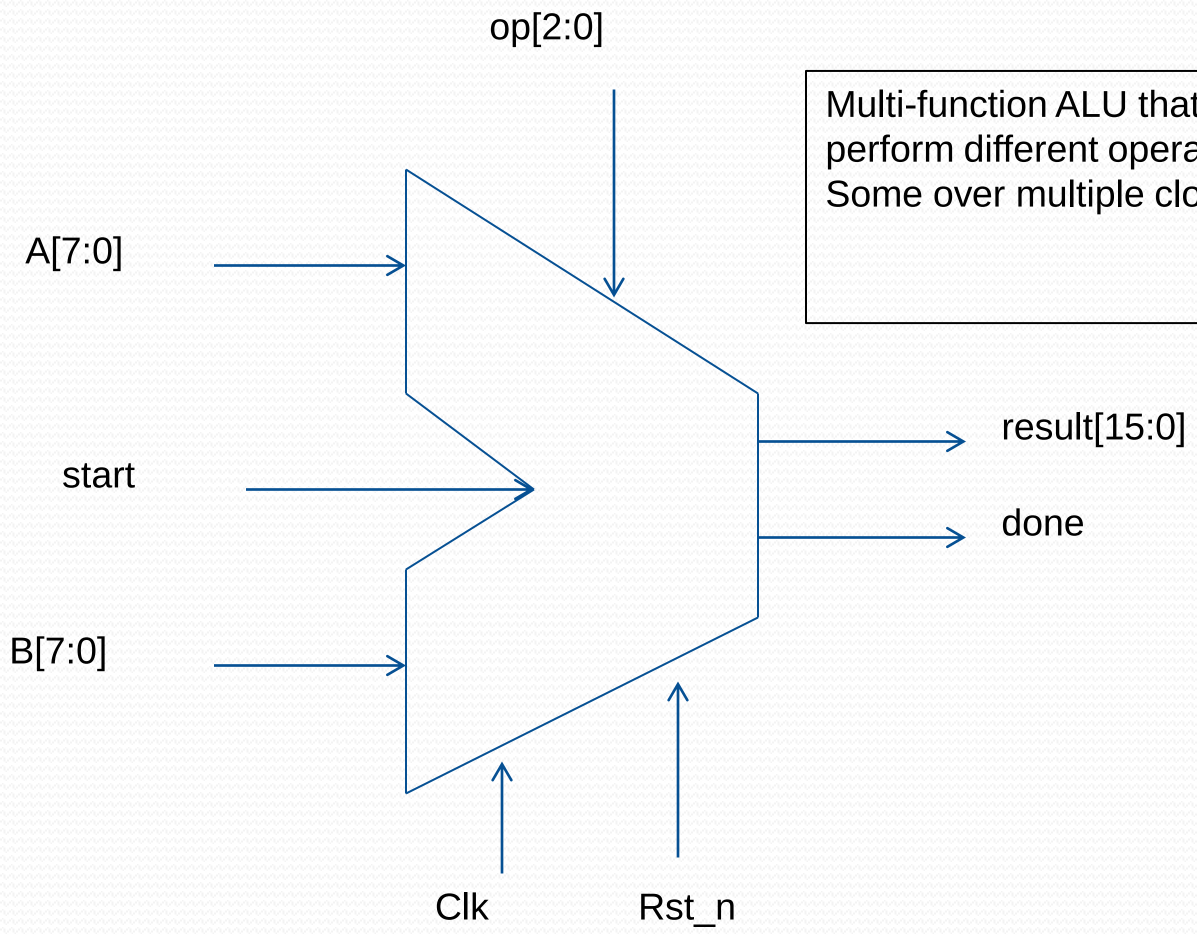
- Derived class
- Allows customization in derived class without modifying known good functionality of parent class

ErrPkt is derived class of Packet with addition of Error bit

```
class ErrPkt extends Packet;
  bit Error;
     .
     .
  function bit ShowError();
    return(Error);
  endfunction
     .
     .
endclass
```



Design Example : ALU



Multi-function ALU that can perform different operations. Some over multiple clock cycles.

Design Example : ALU

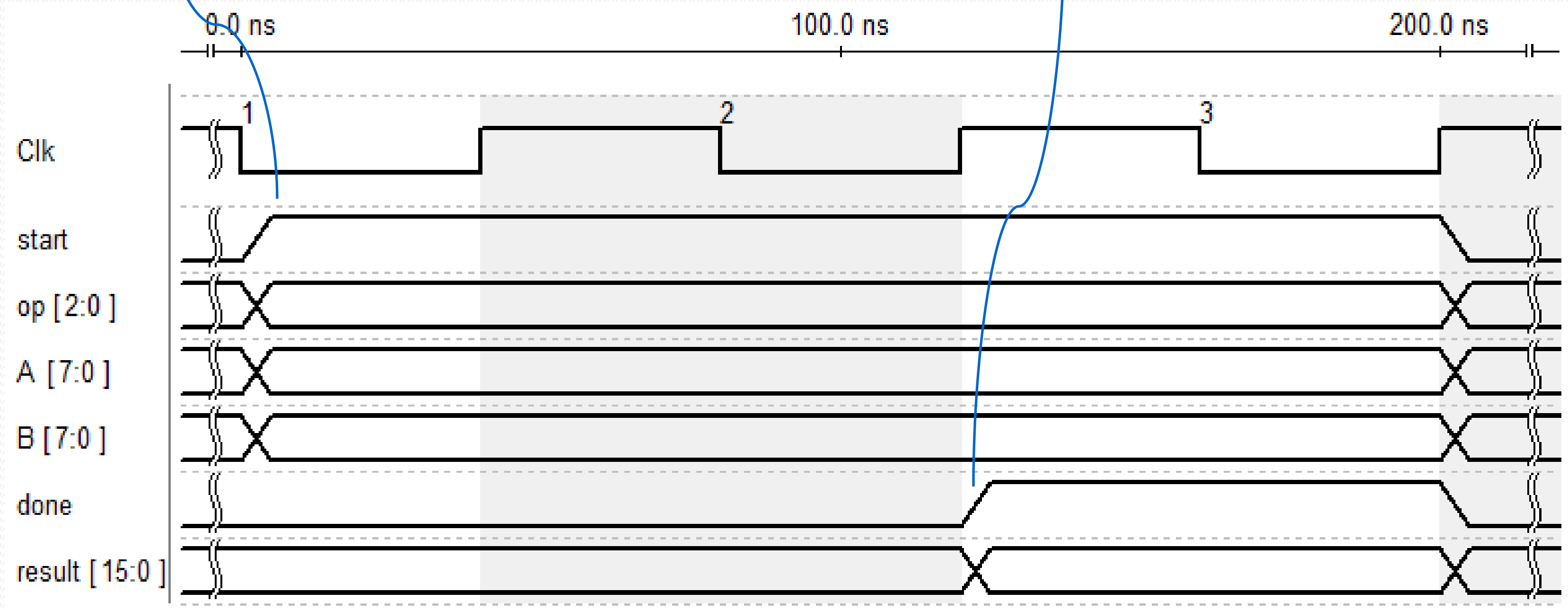
- Opcode description for ALU –

Operation	Opcode
No_op	3'b000
Add_op	3'b001
And_op	3'b010
Xor_op	3'b011
Mul_op	3'b100
Unused	3'b101 – 3'b111

- For this design let's create Object Oriented based test-bench

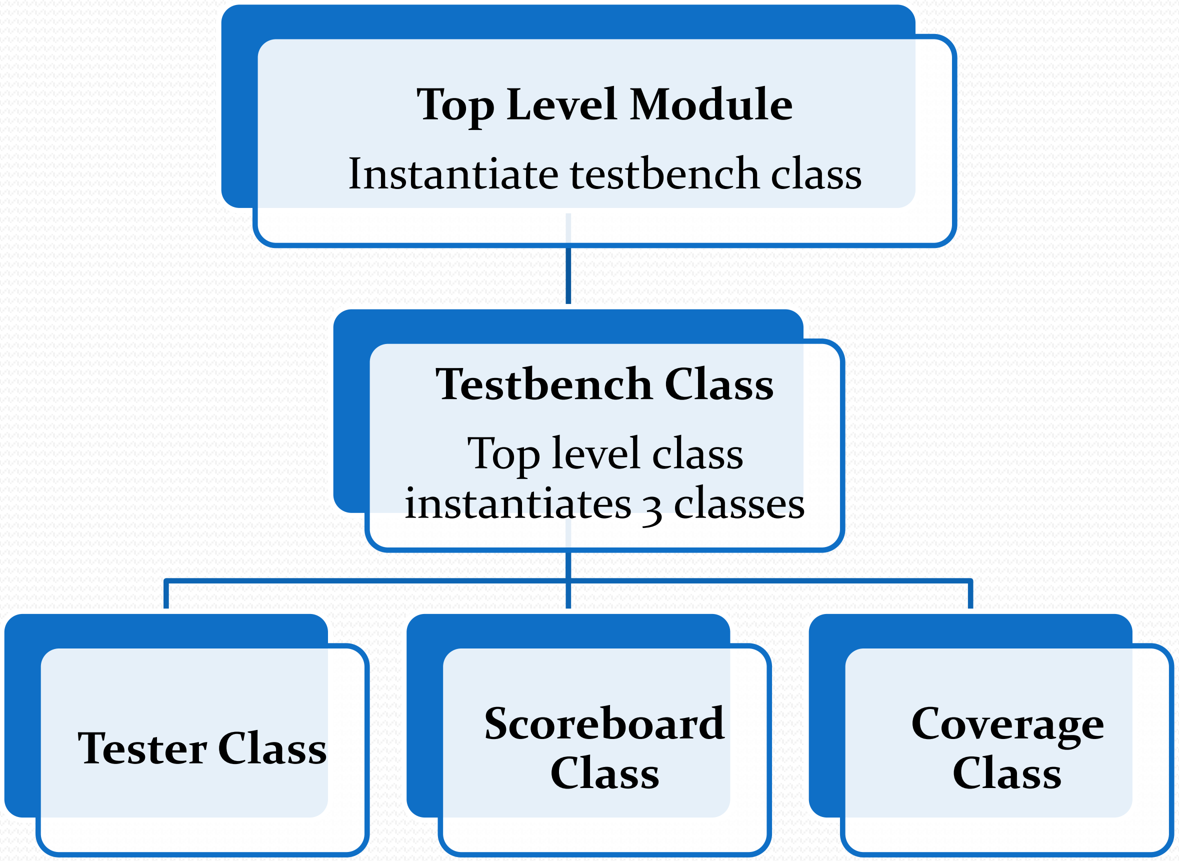
Design Example : ALU

- Waveform for ALU –
  - Start must remain high and operator , operands remain stable until ALU raises done signal and result is available



Object Oriented TB for ALU

Testbench contains one module and four classes –



System Verilog: **Interface** (allows one to encapsulate an interface)

- System Verilog **interface** encapsulate port signals in a test-bench – First step in modular test-bench
- This makes it easy to share signals between modules/objects

```
interface tiny_alu_bfm;
  bit [7:0]    A;
  bit [7:0]    B;
  bit          clk;
  bit          reset_n
  bit [2:0]    op
  bit          start
  wire         done
  wire [15:0] result;
      .
      .
      .
endinterface
```

Interface to ALU is defined and can be reused.

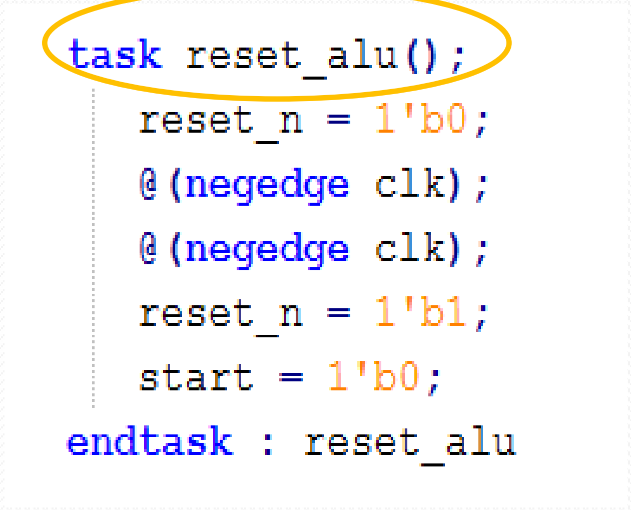
Interface is named: *tiny_alu_bfm*

- bfm = Bus Functional Model
- **BFM** is a term you will hear a lot and typically refers to an *interface* and a set of tasks associated with that interface that allow you to drive stimulus and check response.

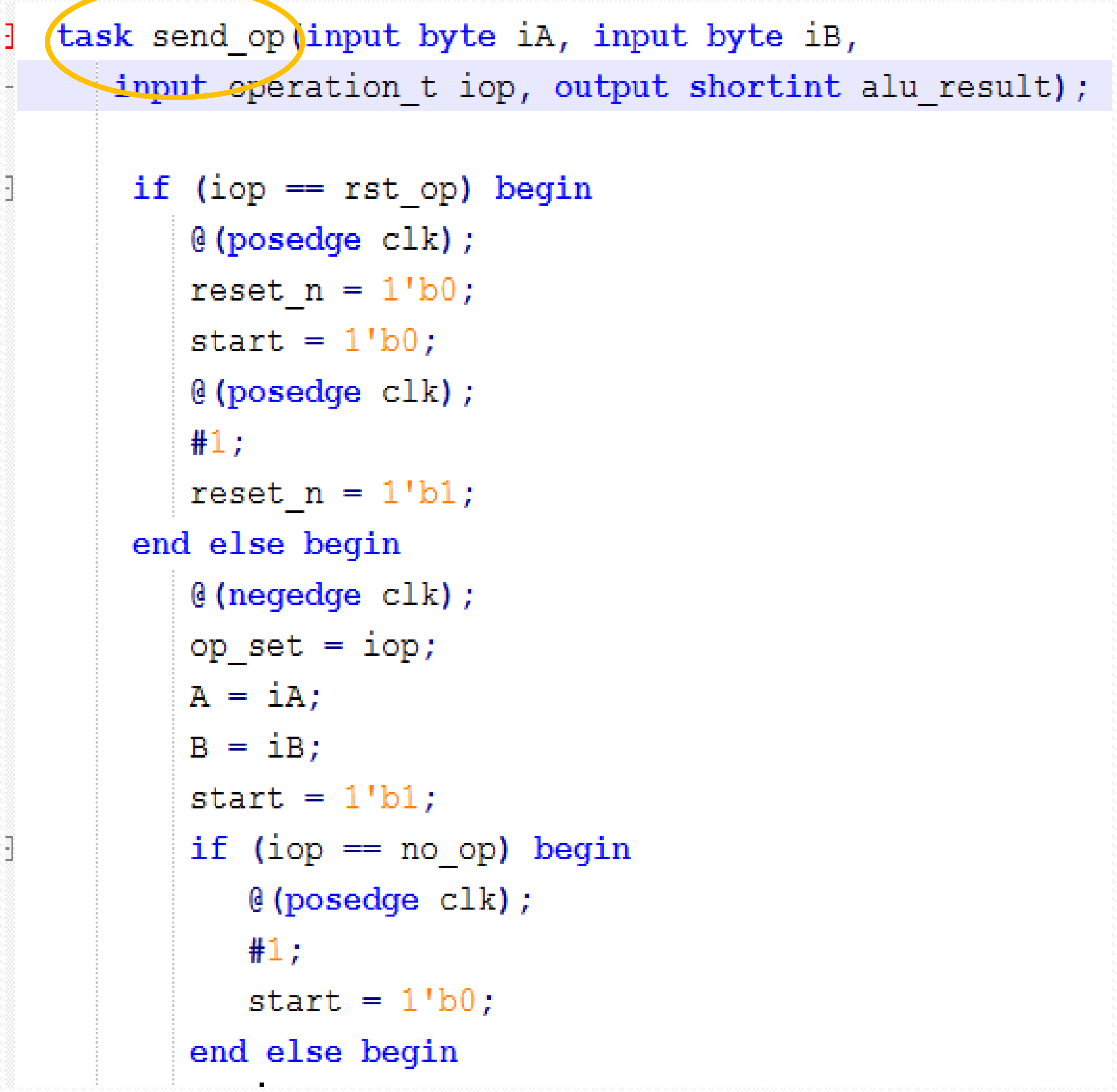
BFM : interface & tasks/functions for stimulus/response

```
interface tiny_alu_bfm;
  bit  [7:0]      A;
     .
     .
     .
  wire            done
  wire [15:0]     result;
```

BFM tasks called inside TB classes to drive/monitor

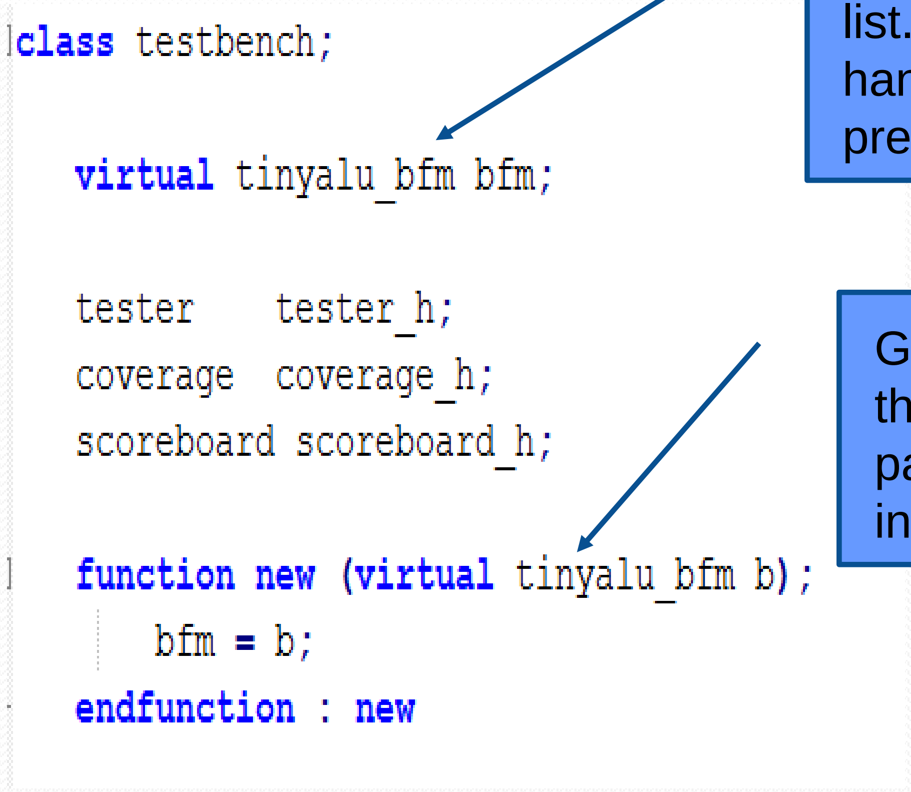


```
task reset_alu();
   reset_n = 1'b0;
   @(negedge clk);
   @(negedge clk);
   reset_n = 1'b1;
   start = 1'b0;
endtask : reset_alu
```



```
task send_op(input byte iA, input byte iB,
     input operation_t iop, output shortint alu_result);

      if (iop == rst_op) begin
         @(posedge clk);
         reset_n = 1'b0;
         start = 1'b0;
         @(posedge clk);
         #1;
         reset_n = 1'b1;
      end else begin
         @(negedge clk);
         op_set = iop;
         A = iA;
         B = iB;
         start = 1'b1;
         if (iop == no_op) begin
            @(posedge clk);
            #1;
            start = 1'b0;
         end else begin
                .
                .
                .
endinterface;
```

Testbench class



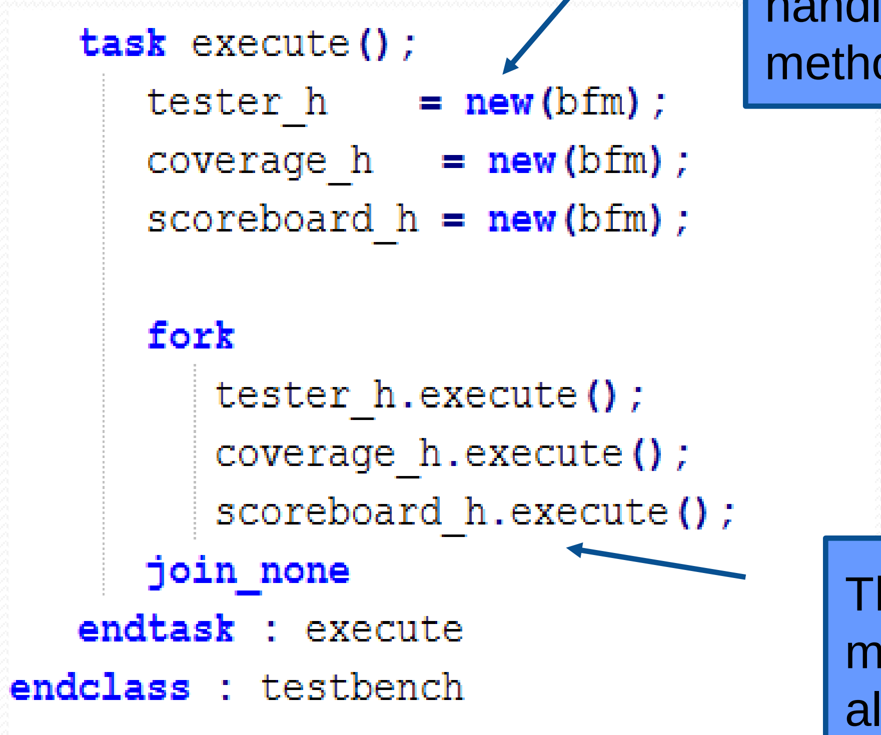
```
class testbench;

   virtual tinyalu_bfm bfm;

   tester     tester_h;
   coverage   coverage_h;
   scoreboard scoreboard_h;

   function new (virtual tinyalu_bfm b);
      bfm = b;
   endfunction : new
```

Object's world equivalent of module's port list. Object access signal by getting a handle to system verilog interface defined previously

Getting the handle to the interface into the bfm variable. Handle would be passed by top level module which instantiates testbench class

Testbench class : continued

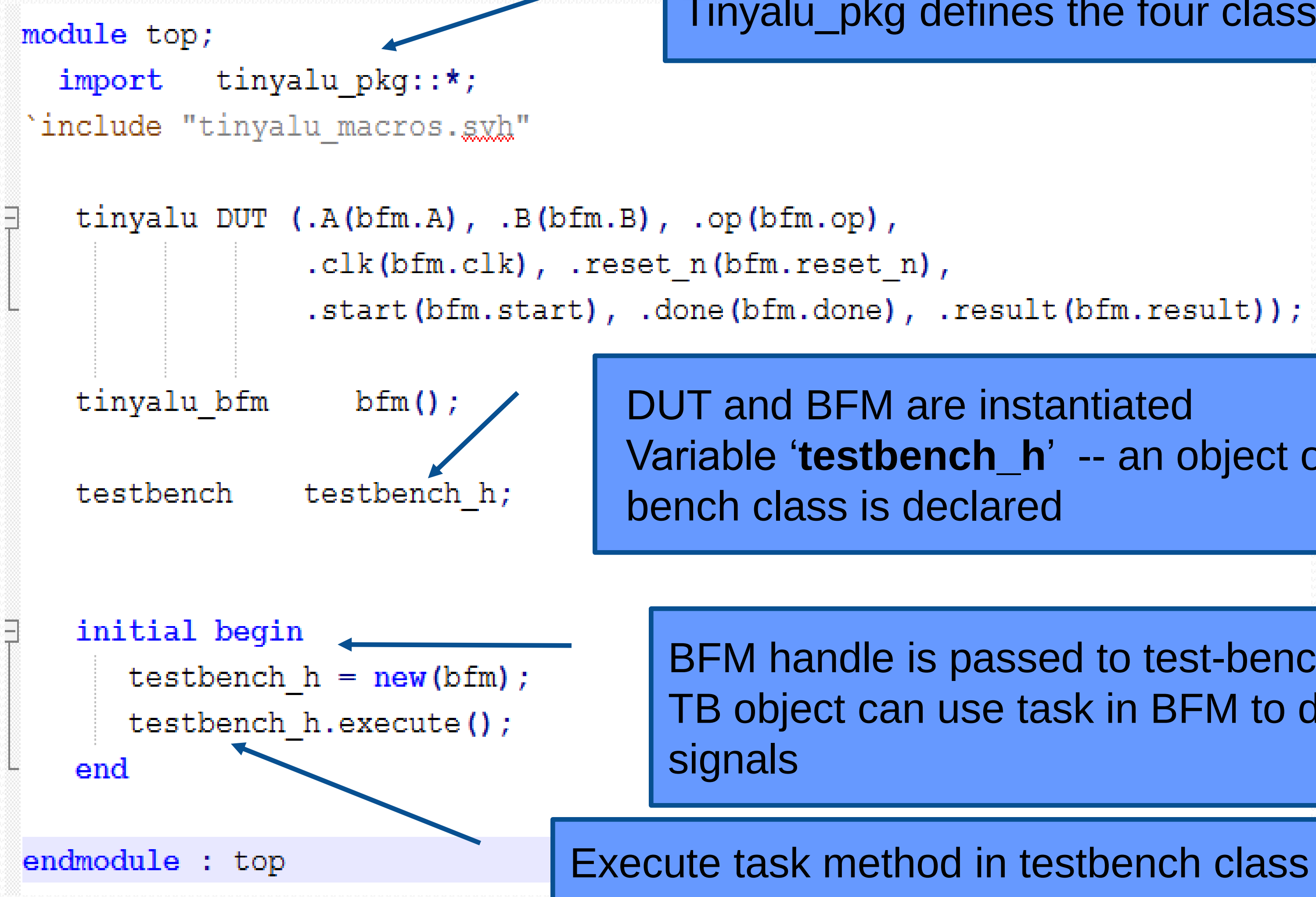
Execute task instantiates the lower level class objects passing bfm interface handle and also launches their execute method



```
   task execute();
      tester_h     = new(bfm);
      coverage_h    = new(bfm);
      scoreboard_h = new(bfm);

      fork
         tester_h.execute();
         coverage_h.execute();
         scoreboard_h.execute();
      join_none
   endtask : execute
endclass : testbench
```

This is same as instantiating three modules, each with its own initial or always block

Top Level Module



```
module top;
  import   tinyalu_pkg::*;
`include "tinyalu_macros.svh"

   tinyalu DUT (.A(bfm.A), .B(bfm.B), .op(bfm.op),
                .clk(bfm.clk), .reset_n(bfm.reset_n),
                .start(bfm.start), .done(bfm.done), .result(bfm.result));

   tinyalu_bfm     bfm();

   testbench    testbench_h;


   initial begin
      testbench_h = new(bfm);
      testbench_h.execute();
   end

endmodule : top
```

Tinyalu_pkg defines the four classes

DUT and BFM are instantiated
Variable '**testbench_h**' -- an object of test-bench class is declared

BFM handle is passed to test-bench object
TB object can use task in BFM to drive, monitor signals

Execute task method in testbench class is called

Tester class (drives the stimulus)

```
class tester;

   virtual tinyalu_bfm bfm;

   function new (virtual tinyalu_bfm b);
      bfm = b;
   endfunction : new
```

```
 task execute();
   byte          unsigned         iA;
   byte          unsigned         iB;
   shortint      unsigned         result;
   operation_t                    op_set;
   bfm.reset_alu();
   op_set = rst_op;
   iA = get_data();
   iB = get_data();
   bfm.send_op(iA, iB, op_set, result);
   op_set = mul_op;
   bfm.send_op(iA, iB, op_set, result);
   bfm.send_op(iA, iB, op_set, result);
```

Execute task generates random transaction and drive them using BFM interface send_op defined earlier

Get_data() is just a task that assigns a random byte for input stimulus. Defined elsewhere (in code we didn't look at)

Tester class is not bothered about protocol level details of sending command !!
Taken care by one piece of code i.e. bfm interface

Scoreboard class (The self checking part)

```
class scoreboard;
     virtual tinyalu_bfm bfm;

   function new (virtual tinyalu_bfm b);
     bfm = b;
   endfunction : new

   task execute();
      shortint predicted_result;
      forever begin : self_checker
         @(posedge bfm.done)
       #1;
          case (bfm.op_set)
            add_op: predicted_result = bfm.A + bfm.B;
            and_op: predicted_result = bfm.A & bfm.B;
            xor_op: predicted_result = bfm.A ^ bfm.B;
            mul_op: predicted_result = bfm.A * bfm.B;
          endcase // case (op_set)

         if ((bfm.op_set != no_op) && (bfm.op_set != rst_op))
           if (predicted_result != bfm.result)
             $error ("FAILED: A: %0h  B: %0h  op: %s result: %0h",
                     bfm.A, bfm.B, bfm.op_set.name(), bfm.result);

     end : self_checker
   endtask : execute
```

Scoreboard class checks that DUT is working. Usual declarations as before

Waits on posedge of done and checks the predicted output by monitoring signals via BFM interface

Putting it together

- Coverage class would perform code coverage metrics. Not covered here.
- We created simple test-bench using objects instead of modules
- Top level module declares object, instantiate and launch them all in their own thread.
- Now, this test-bench has the flexibility and re-use power of OOP
- UVM is a complex topic worthy of its course. It has become widely used in industry.

References

- UVM primer Book by Ray Salemi
- Course - http://www.cerc.utexas.edu/~jaa/ee382m-verif/
- Slides created from content of Kushagra Garg