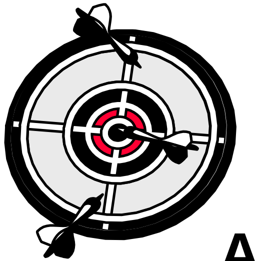
Agenda

DAY 2

5	Concurrency
6	Object Oriented Programming (OOP) – Encapsulation
7	Object Oriented Programming (OOP) – Randomization

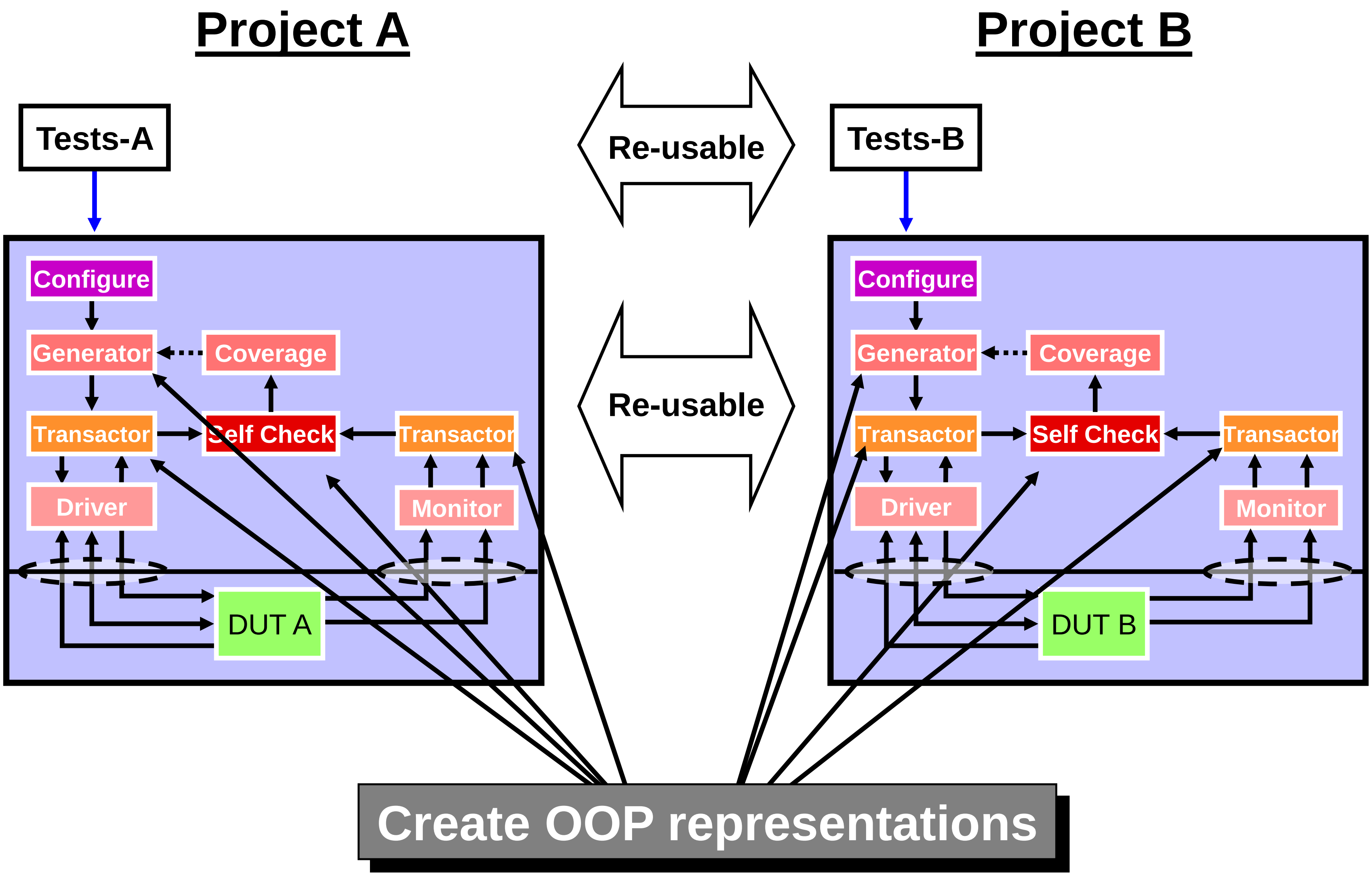
Unit Objectives



After completing this unit, you should be able to:

- **Raise level of abstraction by building data structure with self-contained functionality:**
  - Object-Oriented Programming(OOP) encapsulation
- **Protect integrity of data in OOP data structure:**
  - OOP data hiding
- **Simplify data initialization process:**
  - OOP constructor
- **Define a parameterized class**
- **Define and use packages**

Abstraction Enhances Re-Usability of Code



SystemVerilog OOP Program Constructs

- **Building SystemVerilog OOP structure is similar to building Verilog RTL structure:**

	RTL	OOP
Block definition	`module`	`class`
Block instance	instance	object
Block name	instance name	object handle
Data types	registers and wires	variables
Functionality	tasks, functions behavioral blocks (`always`, `initial`)	subroutines (tasks, functions)

Unlike in a module, nothing executes automatically in an object.
Some subroutine in the object must be called to perform an action.

OOP Encapsulation (OOP Class)

- **Similar to a `module`, an OOP `class` encapsulates:**
  - Variables (properties) used to model a system
  - Subroutines (methods) to manipulate the data
  - Properties & methods are called members of class

Class properties and methods are visible inside the class

```
class Packet;
  string    name;
  bit[3:0]  sa, da;     //copy of Packet properties
  bit[7:0]  payload[];  //copy of Packet properties
  task send();
    send_addrs();
    send_pad();
    send_payload();
  endtask: send
  task send_addrs();    ...  endtask
  task send_pad();      ...  endtask
  task send_payload();  ...  endtask
endclass: Packet
```

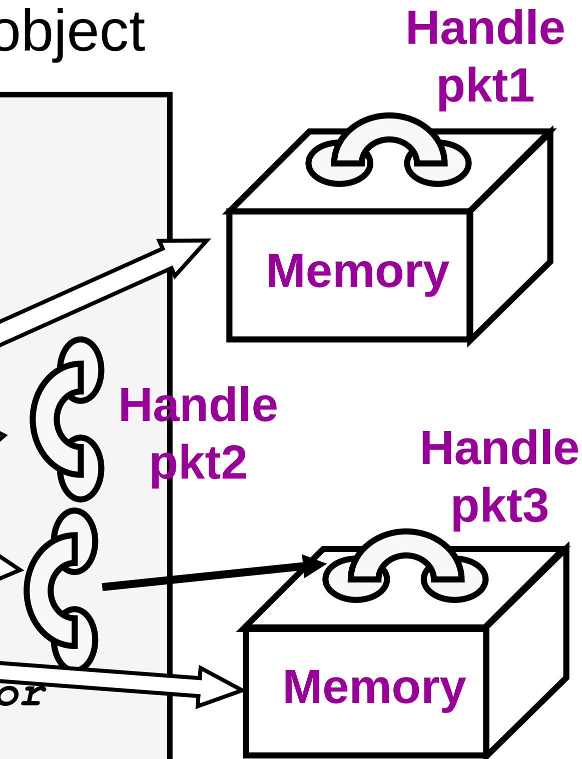
`module` vs. `class`

- **Why use `class`?**
  - Objects are dynamic, modules are static
    - Objects are created and destroyed as needed
  - Instances of classes are objects
    - A handle points to an object (class instance)
    - Object handles can be passed as arguments
    - Object memory can be copied or compared
    - Instances of modules can not be passed, copied or compared
  - Classes can be inherited, modules can not
    - Classes can be modified via inheritance without impacting existing users
    - Modifications to modules will impact all existing users

Constructing OOP Objects

- **OOP objects are constructed from `class` definitions:**
  - Similar to instance creation from **`module`** definition
- **Object memory is constructed by calling `new()`**
  - Handle used to refer to object

```
program automatic test;
  class Packet;
    bit[3:0]  sa, da;
    byte payload[];
    task send();
  endclass: Packet
  Packet pkt1 = new();
  Packet pkt2;
  Packet pkt3;
  initial begin
    pkt3 = new();
    pkt2.sa = 3; //runtime error
  end
endprogram: test
```



Accessing Object Members

- **Object memory is created via a call to `new()`**
- **Object members are accessed via the object handle:**
  - Similar to accessing instance signals and subroutines

```
program automatic test;
  class Packet;
    bit[3:0]  sa, da;
    byte payload[];
    task send();
    ...
  endclass: Packet
  Packet pkt;
  initial begin
    pkt = new();
    pkt.sa = 3; // access property
    pkt.da = 7; // access property
    pkt.send(); // access method
  end
endprogram: test
```

Initialization of Object Properties

- **Define constructor `new()` to initialize properties:**
  - No return type in declaration
  - Executes immediately after object memory is allocated
  - Not accessible via dot (.) notation

```
program automatic test;
  class Packet;
    bit[3:0] sa, da;
    bit[7:0] payload[];
    function new(bit[3:0] init_sa, init_da, int init_payload_size);
      sa = init_sa;
      da = init_da;
      payload = new[init_payload_size];
    endfunction: new
  endclass: Packet
  initial begin
    Packet pkt1 = new(3, 7, 2);
    pkt1.new(5, 8, 3);          // syntax error!
  end
endprogram
```

Initialization of Object Properties: this

- `this` **keyword**
  - An object's handle to itself
  - Unambiguously refers to class properties and methods of the current instance (object)
    - More readable – allows method arguments to have same name as class variables

```
program automatic test;
  class Packet;
    bit[3:0] sa, da;
    bit[7:0] payload[];
    function new(bit[3:0] sa, da, int payload_size);
      this.sa = sa;
      this.da = da;
      this.payload = new[payload_size];
    endfunction: new
  endclass: Packet
  …
endprogram: test
```

OOP Data Hiding (Integrity of Data) 1/3

- **Unrestricted access of object properties can cause unintentional data corruption**

```
program automatic test;
  class driver;
    int max_err_cnt = 0, err_cnt = 0;
    task run();
      …
      if (error_cond()) err_cnt++;
      if ((max_err_cnt != 0) && (err_cnt >= max_err_cnt))
          $finish;
    endtask
    function new(); // details not shown
  endclass: driver
  initial begin
    driver drv = new();
    drv.max_err_cnt = -1; // directly set max_err_cnt
    drv.run();            // Will this work?
  end
endprogram: test
```

Are all class data correct?

OOP Data Hiding (Integrity of Data) 2/3

- **Properties & methods can be protected using `local`**
  - Object members are **`public`** by default
  - **`local`** members of object can be accessed only in class

```
program automatic test;
  class driver;
    local int max_err_cnt = 0, err_cnt = 0;
    task run();… endtask
  endclass: driver
  initial begin
    driver drv = new();
    drv.max_err_cnt = -1; // Compile error!
    drv.run();
  end
endprogram: test
```

OOP Data Hiding (Integrity of Data) 3/3

- **Create `public` class method to allow users to access `local` members**
  - Ensure data integrity within the method

```
program automatic test;
  class driver;
    local int max_err_cnt = 0, err_cnt = 0;
    task run();… endtask
    function set_max_err_cnt(int max_err_cnt);
      if (max_err_cnt < 0) begin
        this.max_err_cnt = 0;
        return;
      end else this.max_err_cnt = max_err_cnt;
    endfunction
  endclass: driver
  initial begin
    driver drv = new();
    drv.set_max_err_cnt(-1); // No Compile error
    drv.run();
  end
endprogram: test
```

Ensure integrity of object data

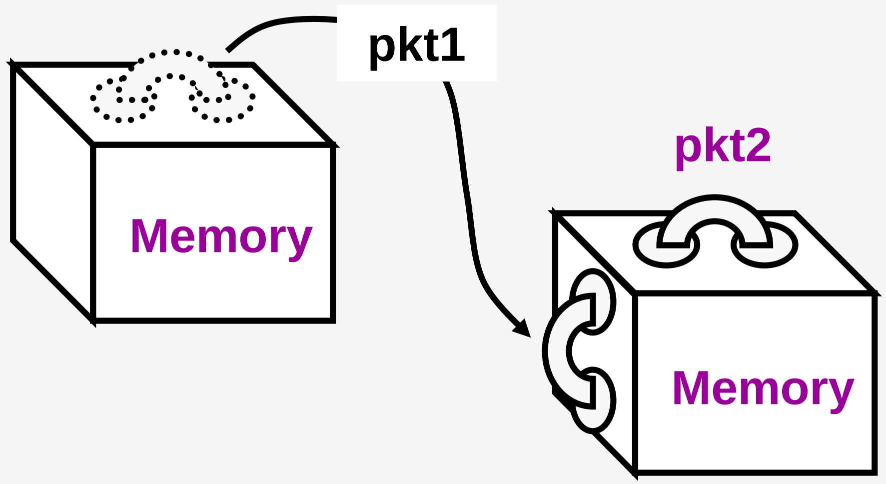
Working with Objects – Handle Assignment

What happens when one object handle is assigned to another?

Like any variable, the target takes on the value of the source.

```
class Packet;
    int payload_size;
    ...
endclass: Packet

…
Packet pkt1 = new();
Packet pkt2 = new();
…
pkt1 = pkt2;
pkt1.payload_size = 5; // whose payload_size is set ?
```



What happens to the pkt1 object memory?

Working with Objects – Garbage Collection

- **VCS *garbage collector* reclaims memory automatically:**
  - When an object memory is no longer accessible
  - And, the object has no scheduled event
- **Object can be manually de-referenced**

```
pkt1 = null;
```

- **Making an exact duplication of object memory:**

```
class Packet;
  int count;
  Payload p; // encapsulated object
endclass: Packet
…
Packet pkt1 = new();
Packet pkt1_copy;          // handle only
pkt1.p = new();
pkt1_copy = new pkt1;   // construct pkt1_copy
                        // and copy contents of pkt1 to pkt1_copy
                        // shallow copy (encapsulated objects not copied)
                        // object pkt1 must exist
```

This method of copying is not recommended. Normally every class that needs it should provide a `copy()` (or similar) method

```
pkt1_copy = pkt1.copy();
```

Working with Objects – Static members

- **Static members: use `static` keyword**
  - Associated with the class, not the object
  - Shared by all objects of that class
  - Variables and subroutines can be **`static`**
    - Static subroutines can only access static members

```
class Packet;
 static int count = 0;
 int id;
 static function int get_count ();
   return count;
 endfunction
 function new();
      this.id = count++;
 endfunction
 endclass: Packet
```

- Static members allocated and initialized at compile
- Static subroutines cannot be overridden

```
program automatic test;
initial begin
    Packet pkt0 = new();
    Packet pkt1 = new();
    $display("pkt0 id is: %0d", pkt0.id);
    $display("pkt1 id is: %0d", pkt1.id);
  end
endprogram: tes
```

What values get printed?

Working with Objects – const Properties

- Constant properties: can not be modified
  - use **`const`** keyword
    - Global constant – typically also declared static
    - Instance constant – can not be static

```
program automatic test;
  class Packet;
    static int count = 0;
    const int id; // instance constant
    static const string type_name = "Packet";// global constant
    function new();
      this.id = count++; // instance constant can only be assigned in new()
    endfunction
  endclass: Packet
  initial begin
    Packet packet0 = new();
    packet0.id = 0; // Compile error – can not change const property
    packet1.type_name = "newPacket"; // Compile error
  end
endprogram: test
```

Working with Objects – Array Methods

```
class Packet;
  rand bit [7:0] payload[]; // Data
  rand bit [2:0] pr;        // user-defined priority 0-7
  rand bit [15:0] addr;     // Address
endclass: Packet

Packet pq[$];  // Queue of packet handles
initial begin
  int len;
  generate_packet_queue(pq);

  // Sort objects according to user-defined priority
  pq.sort(pkt) with (pkt.pr);  // pkt is user-defined iterator
                               // pkt is auto-declared
  // Find total length of all payloads
  len = pq.sum() with (item.payload.size());
                     //item is default iterator
end
```

See Note

Working with Objects – Concurrency

- **Classes can not have `initial` or `always` blocks**
- **Spawn a process similar to an `always` block with `fork-join_none`**

```
class Driver;
…
task run();//thread start method
fork //emulate always block
    forever
        send();
join_none
endtask: run
…
endclass: Driver
```

- **Standard methodology**
  - Program calls ***run()*** method of the various OOP testbench components
    - Generator, Monitor, Driver, Scoreboard etc.

Parameterized Classes

- **Written for generic type and/or values**
  - Parameters passed at instantiation, just like parameterized modules
  - Allows reuse of common code

```
class stack #(type T = int,
       bit[11:0] depth = 1024);
 protected T items[$:depth];
 function void push( T a );
 ...
 function T pop( );
 function int size(); ...
endclass: stack
```

```
program automatic test;
  stack addr_stack; //default type
  stack #(Packet, 128) data_stack;
initial begin
  ...
  repeat(addr_stack.size()) begin
    Packet pkt = new();
    if(!pkt.randomize())
        $finish;
    pkt.addr = addr_stack.pop();
    data_stack.push(pkt);
   end
end
endprogram: test
```

Class `typedef`

- **Often need to use a class before declaration**
  - e.g. two classes need handle to each other
    - Use **`typedef`**

```
typedef class child;
class parent;
  child c1;
  ...
endclass: parent

class child;
  parent p1;
  ...
endclass: child
```

This is a compile error if typdef is missing

Best Practices (1/2)

- **Methods can be placed outside of the class block:**
  - Inside class block, declare a extern prototype
  - Outside class block, use a pair of colons :: to associate the method with its class
  - Double-colon is a scope/name resolution operator

```
class node;
  static int count = 0;
  string str;
  node next;
  ...
  task ping();
    ...
  endtask: ping
endclass: node
```

```
class node;
  static int count = 0;
  string str;
  node next;
  ...
  extern task ping();  // prototype
endclass: node
task node::ping();
  ...
endtask: ping
```

Place class name and double-colons before method name

Best Practices (2/2)

- **Useful methods for Data classes (user defined):**
  - **`display()`**
    - Print object variables to console - Helpful for debugging
  - **`compare()`**
    - Returns match, mismatch, other status based by comparing object variables to variables of another object
      - Simplifies self-check
  - **`copy()`**
    - Copy selected variables or nested objects
      - Allows you to do deep copy if required
- **Use `typedef` to create shortcuts**
  - **`typedef`** ***`stack#(Packet) pkt_stack`***;
    - Now use ***`pkt_stack`*** instead of ***`stack#(Packet)`***

Virtual Interfaces

- **Classes need to drive/sample signals of interface**
  - Interfaces can not be created at object construction
    - Need to create a **virtual** reference to interface

```
class Driver;
  virtual router_io.TB rtr_io;
  ...
  function new(virtual router_io.TB rtr_io);
    this.rtr_io = rtr_io;
  endfunction: new

  task send_addrs();
    this.rtr_io.cb.frame_n[sa] <= 1'b0;
    for(int i=0; i<4; i++) begin
      this.rtr_io.cb.din[sa] <= da[i];
      @(this.rtr_io.cb);
    end
  endtask: send_
endclass: Driver
```

Create virtual reference to interface

Pass virtual connections via constructor argument*

Drive/Sample signals using virtual interface

```
program automatic test(router_io.TB rtr_io);
...
Driver drv = new(rtr_io); // pass interface
```

SystemVerilog Packages

- **Packages are a mechanism for sharing among modules, programs and interfaces the following:**
  - Parameters
  - Data – variables and nets
  - Type definitions
  - Tasks & functions
  - Sequence and property declarations
  - Classes
- **Declarations may be referenced within modules, interfaces, programs, and other packages**

Packages: Example

ComplexPkg.sv

```
package ComplexPkg;
class Complex;
  float i, r;
  extern virtual task display();// not shown
endclass: Complex
// standalone functions
function automatic Complex add(Complex a, b);
  add = new();
  add.r = a.r + b.r; add.i = a.i + b.i;
endfunction: add

function automatic Complex mul(Complex a, b);
  mul = new();
  mul.r = (a.r * b.r) - (a.i * b.i);
  mul.i = (a.r * b.i) + (a.i * b.r);
endfunction: mul

endpackage: ComplexPkg
```

Rules Governing Packages

- **Packages are explicitly named scopes appearing at the outermost level of the source text (at the same level as top-level modules and primitives)**
- **Packages must not contain any processes**
  - Wire declarations with implicit continuous assignments are not allowed
- **Packages can not have hierarchical references**
- **Variable declaration assignments within the package must occur before any initial, always, always_comb, always_latch, or always_ff blocks are started**

> package subroutines are static unless explicitly automatic.
> Classes are always automatic.

Using Packages

- **Directly reference package member using class scope resolution operator ::**

```
ComplexPkg::Complex cout = ComplexPkg::mul(a,b);
```

- **import package into appropriate scope**
  - Explicit import of specific symbols

    ```
    import ComplexPkg::Complex;
    import ComplexPkg::add;
    ```

  - Implicit import of all symbols in package

    ```
    import ComplexPkg::*;
    ```

    - Now all symbols in ComplexPkg are visible
  - OK to import same package in multiple locations
    - **`` `include ``** cannot be used in multiple places

Using Packages: Example (1/2)

ComplexPkg.sv

```
package ComplexPkg;
class Complex;
   float i, r;
   extern virtual task display();// not shown
endclass: Complex
// standalone functions
function automatic Complex add(Complex a, b);
   add = new();
  add.r = a.r + b.r; add.i =
endfuncti

function
   mul = new(
```

import specific symbols

```
// import of specific symbol
program automatic
   test(if.tb_port tb_io);
import ComplexPkg::Complex ;
...
endprogram: test
```

import whole package

```
// implicit import all symbols
module dut(if.dut_port dut_io);
import ComplexPkg::* ;
Complex l,m,n;
...
endmodule: dut
```

Direct reference using ::

```
// Direct reference
class harmonix;
ComplexPkg::Complex i,j ;
...
endclass: harmonix
```

Using Packages: Example (2/2)

- **Packages can be imported by other packages.**
- **To allow a package imported by one package to be imported along with the importing package, export it.**
  - export follows same syntax as import

```
package signal_analysis;

import ComplexPkg::*;
// export with signal_analysis
export ComplexPkg::*;

class harmonix;
Complex alpha, beta, gamma;
...
endclass: harmonix

endpackage: signal_analysis
```

Quiz Time

OOP: Quiz 1

```
program automatic test1;

class abc;
   int a = 10;
   function new(int a);
       a = a;
   endfunction
endclass

abc o1;

initial
begin
   o1 = new(5);
   $display("a = %0d",o1.a);
end
endprogram: test1
```

1. What will the program display?
2. Did it display what you expected?
3. How will you fix this?

OOP: Quiz 2

```
program automatic test1;

class abc;
        int a;
endclass
initial begin
abc o1, o2;

o1 =new(); o2 = new();
o1.a = 5;
o2.a = 50;

$display("A: %0d %0d", o1.a, o2.a);
o2 = o1;
o1.a = 500;
$display("B: %0d %0d", o1.a, o2.a);
end
endprogram: test1
```

1. **What will the program display?**
2. **Why?**
3. **How many objects each at first and second display lines?**
   - **Why?**
4. **If number of objects is less what happened to missing objects? If it is more how did more objects get constructed?**

OOP: Quiz 3

- **What is the difference between a `public` and `local` member of a `class`?**
- **Can `local` members be `static`?**
- **What is the `::` operator?**
  - Give some examples where it can be used
- **List two uses of `typedef`**

Unit Objectives Review

Having completed this unit, you should be able to:

- **Raise level of abstraction by building data structure with self-contained functionality:**
  - Object-Oriented Programming(OOP) encapsulation
- **Protect integrity of data in OOP data structure:**
  - OOP data hiding
- **Simplify data initialization process:**
  - OOP constructor
- **Define a parameterized class**
- **Define and use packages**

Appendix

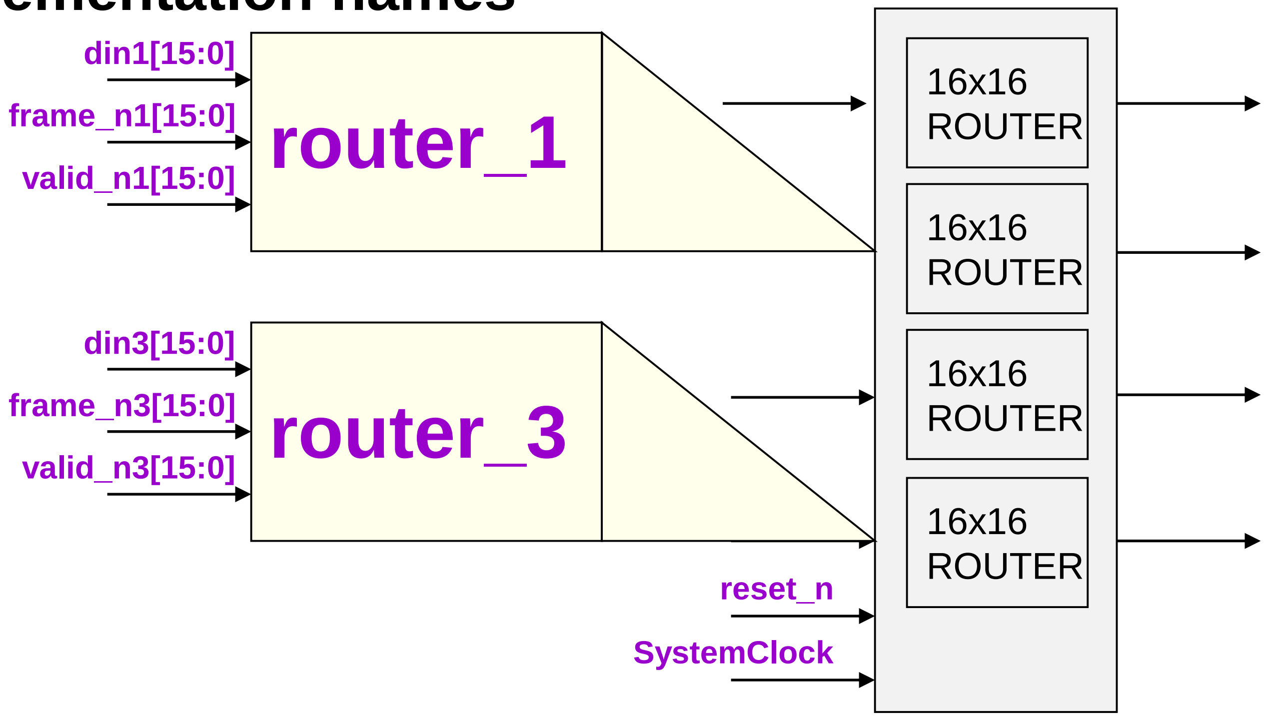
SystemVerilog Virtual Interface

Singleton Objects

SystemVerilog Virtual Interface

Virtual Interfaces (1/5)

- **Allow grouping of signals by function**
- **Create a handle to an interface**
  - Virtual interfaces can be passed via routine argument
- **Promotes reuse by separating testbench from implementation names**



Virtual Interfaces (2/5)

- **STEP 1: Define a physical interface**
  - Similar to creating interface for just the single instance
  - The difference is that common connections are removed

```
interface router_io(input bit clock);
// logic reset_n;
   logic [15:0] din, frame_n, valid_n;
   logic [15:0] dout, valido_n, frameo_n;

   clocking cb @(posedge clock);
     output din, frame_n, valid_n;
     input dout, valido_n, frameo_n;
   endclocking: cb

   modport TB(clocking cb);
// instead of modport TB(clocking cb, output reset_n);
endinterface: router_io
```

Virtual Interfaces (3/5)

- **STEP 2: Connect the interface**

```
module router_test_top;
  bit     SystemClock;
  logic   reset_n;
  router_io io_0(SystemClock);
  router_io io_1(SystemClock);
  router_io io_2(SystemClock);
  router_io io_3(SystemClock);
  test t(io_0, io_1, io_2, io_3, reset_n);
  router dut(
     .clock      (SystemClock);
     .reset_n    (reset_n);
     .din0       (io_0.din),
     .frame_n0   (io_0.frame_n),
     .valid_n0   (io_0.valid_n),
     ...
     .din1       (io_1.din),
     .frame_n1   (io_1.frame_n),
     .valid_n1   (io_1.valid_n),
     ...
  );
endmodule: router_test_top
```

One interface per instance

Connect common signals separately

Connect unique signals with a specific interface instance

Virtual Interfaces (4/5)

- **STEP 3: Pass virtual interface in via constructor**
- **STEP 4: Drive/Sample signals with virtual interface**
  - This class is now re-useable for any router instance

```
class Driver;
             string        name;
   virtual router_io.TB rtr_io;
   ...
   function new(string name = "Driver",
                virtual router_io.TB router);
     this.name = name;
     this.rtr_io = router;
   endfunction: new
   virtual task send_addrs();
     rtr_io.cb.frame_n[sa] <= 1'b0;
     for(int i=0; i<4; i++) begin
       rtr_io.cb.din[sa] <= da[i];
       @(rtr_io.cb);
     end
   endtask: send_addrs
endclass: Driver
```

Create reference to virtual interface

Pass virtual connections via constructor argument

Drive/Sample signals using virtual interface

Virtual Interfaces (5/5)

- **STEP 5: Connect Virtual to physical**

```
program automatic test(router_io.TB r0, r1, r2, r3
                                   output logic reset_n);
  class BFM_environment;
    DriverClass driver[16];
    function new(virtual router_io.TB rtr_io);
    ...
    endfunction: new
  endclass: BFM_environment
  BFM_environment bfm[4];
  initial begin
    bfm[0] = new(r0);
    bfm[1] = new(r1);
    bfm[2] = new(r2);
    bfm[3] = new(r3);
   ...
  end
  task reset();
    reset_n <= 1'b0; ...;
  endtask: reset
endprogram: test
```

Connect Virtual to physical

```
module router_test_top;
 logic SystemClock, reset_n;
 router_io
   io_0(…),io_1(…),io_2(…),io_3(…);
 test
   t(io_0,io_1,io_2,io_3,reset_n);
 router dut(...);
 ...
endmodule: router_test_top
```

Singleton Objects

Singleton Objects

- **A singleton object is a globally accessible static object which provides customized service methods**
  - Created at compile-time
  - Globally accessible at run-time
  - Can have static and non-static members
  - For convenience only

```
class service_class;
  static service_class me = get();
  static function service_class get();
    if (me == null) me = new(); return me;
  endfunction
  protected function new();
  endfunction
  extern function void error (string msg);
endclass
```

```
service_class service_object = service_class::get();
service_object.error("A different error");
```

Singleton object **me** created at compile-time

Globally accessible at run-time

non-static function

Static members accessed using `::`