SystemVerilog for Verification

Developed & Presented by

Willamette HDL for Mentor Graphics

Version 4.7

Course Outline

Verification 5
- AVM Testbench Structure 7
- Transactions 9

Data Types 18
- User defined types 21
- Enumeration 23
- Parameterized types 31

Arrays & Structures 33
- Dynamic Arrays 34
- Associative Arrays 37
- Queues / Lists 39
- Structures 42

SV Scheduler 63

Program Control 68

Hierarchy 79
- Implicit port connections 85
- Packages 88
- Compilation Units 91

Tasks & Functions 98
- Task enhancements 99
- Function enhancements 101

Dynamic Processes 108
- fork-join none 110

Interprocess Sync & Communication 114
- semaphore 116
- mailbox 118
- Lab 120

Classes
- Classes 123
- Constructors 126
- Lab 135

Interfaces 141
- Simple bundled 145
- Modports 148
- Virtual Interfaces 153
- Router design 156
- Lab 159

Course Outline (cont.)

Classes (cont) 163
- Inheritance 164
- Virtual methods 171
- Protection 178
- Parameterized classes 182
- Polymorphism 185
- Virtual Classes 187
- Lab 189

Randomization & Constraints 197
- Stimulus Generation Methodologies 198
- Constraint blocks 204
- Randomize 213
- Random sequences 221
- Lab 228
- Reference slides 231

Functional Coverage 242
- Covergroup 249
- Coverpoint 252
- Cross 261
- Coverage methods,options 265
- Lab 273
- Reference slides 278

SVA 279
- Immediate assertions 281
- Concurrent assertions 282
- Concurrent assertions basics 285
  - Boolean Expressions 288
  - Sequences 289
  - Property Block 293
  - Verification Directives 295
  - Lab 302
- Sequence blocks 304
  - Sequence Operators 306
  - Repetition operators 308
  - Other methods & operators 319
  - Worked Example 326
  - Lab 334
- Sequence expressions 337
- Property block 343
- Local Data values 351
  - Lab 357
- Verification Directives 359
  - Bind Directive 364
  - Lab 373

Sample Solutions 374

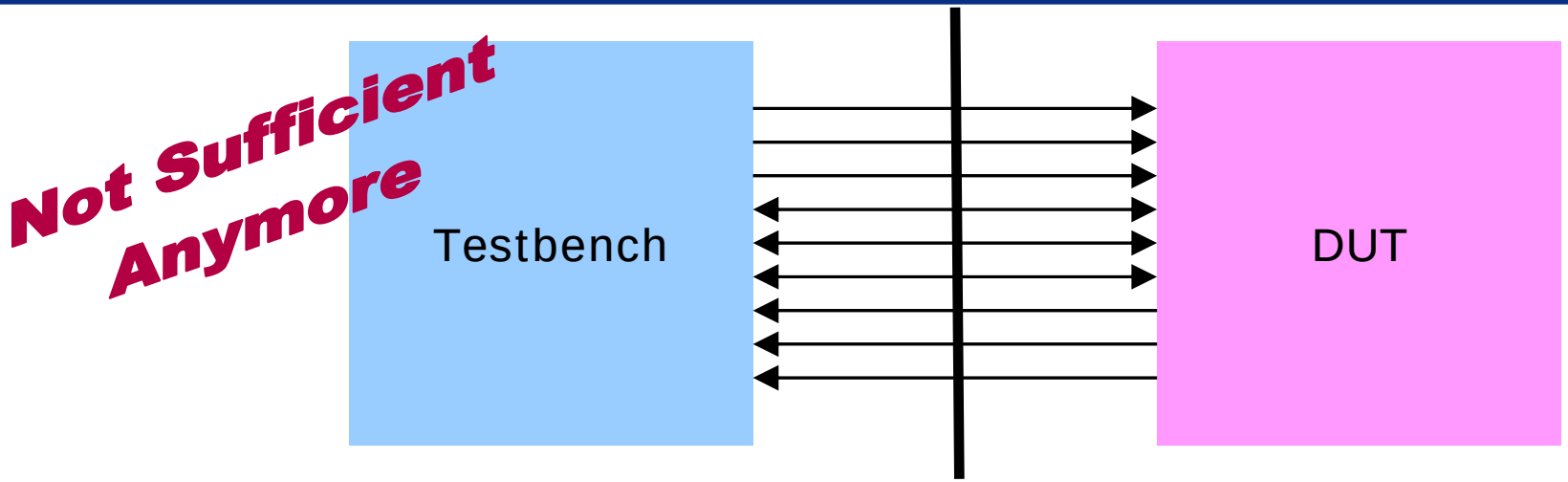
Verification

In this section

SystemVerilog

- Advanced Verification Methodology
- Testbench Structure
- Transactions

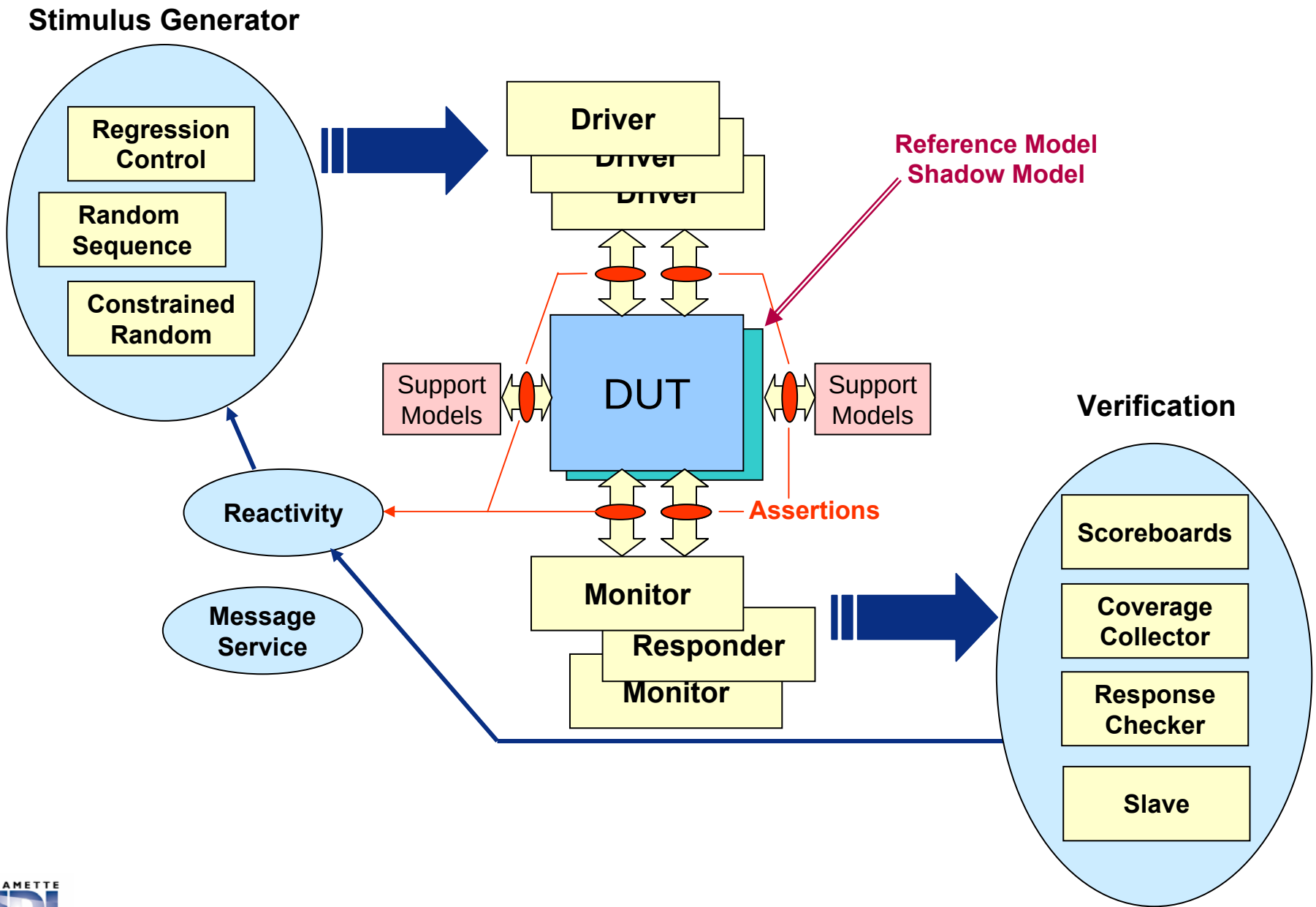
Traditional Testbench Structure



- Testbench and DUT are completely separate
  - DUT is often treated as a black box
- Interaction only through one (potentially large) interface
  - Stimulus is applied and results are measured from external pins only
- OK for simple designs
- For complex designs:
  - It is virtually impossible to predict all potential input sequences
  - It IS impossible to generate such stimulus by hand
  - How do you know when you are done?
- PLI-based view of the world
- Highly non-reusable

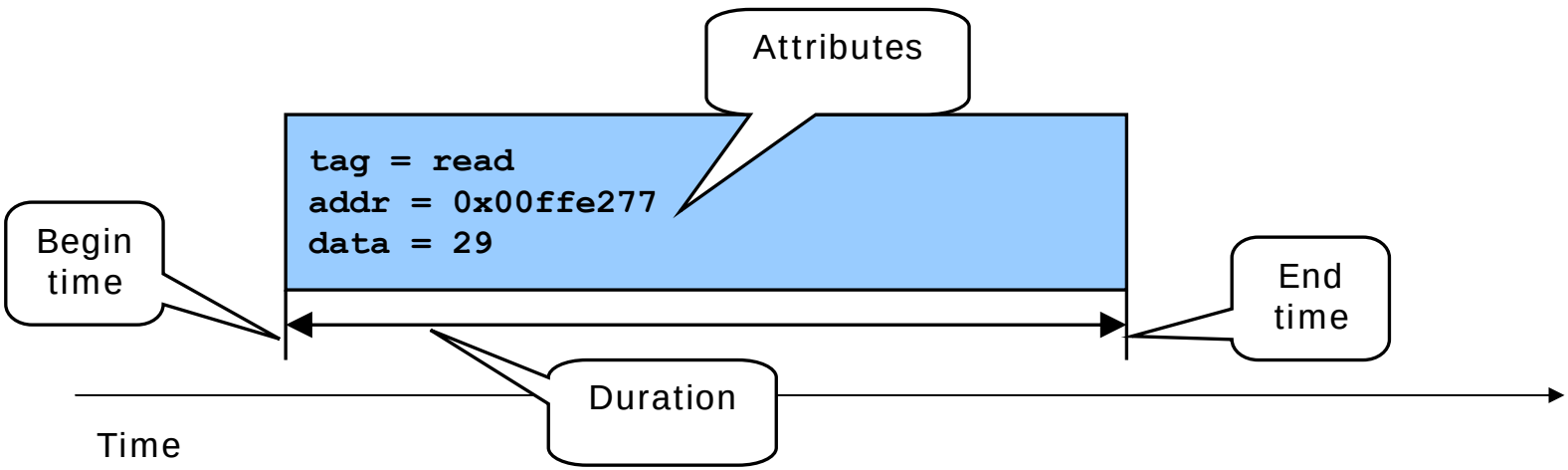
Advanced Testbench Structure (AVM)

- AVM combines many techniques/ideas to form a reusable framework



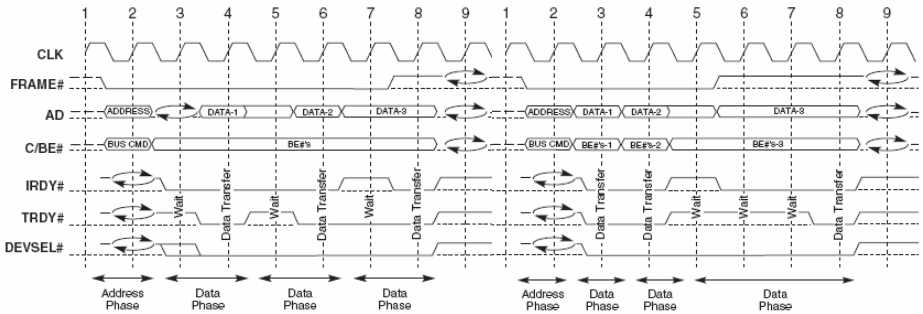
Transactions - Definition

- **Representation of arbitrary activity in a device**
  - **Bounded by time**
  - **Has attributes**



Transaction Metaphors

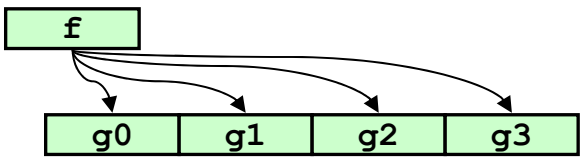
Bus Operations



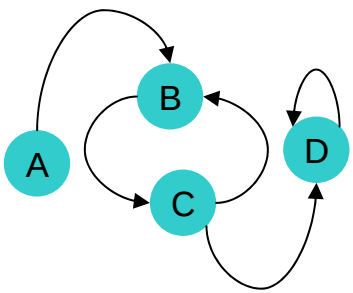
read	write

Function Calls

```
f() {
  for(i=0; i<4; i++)
    g(i)
}
```

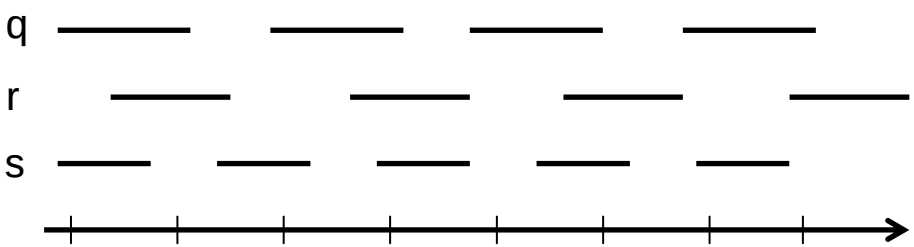


FSM



A	B	C	B	C	D

Gantt Chart



resource occupation

Transaction Level Modeling

- Transaction Level Model (TLM)
  - A model written at the transaction level
    - Interface is *not* pin-level
      - May or may not be cycle accurate
    - Internal description typically algorithmic
      - May or may not be cycle accurate
- Transactions provide a medium for moving data around without doing low level conversions
  - Transaction traffic can be recorded or otherwise observed
- TLM interfaces provide a standard way of building verification components

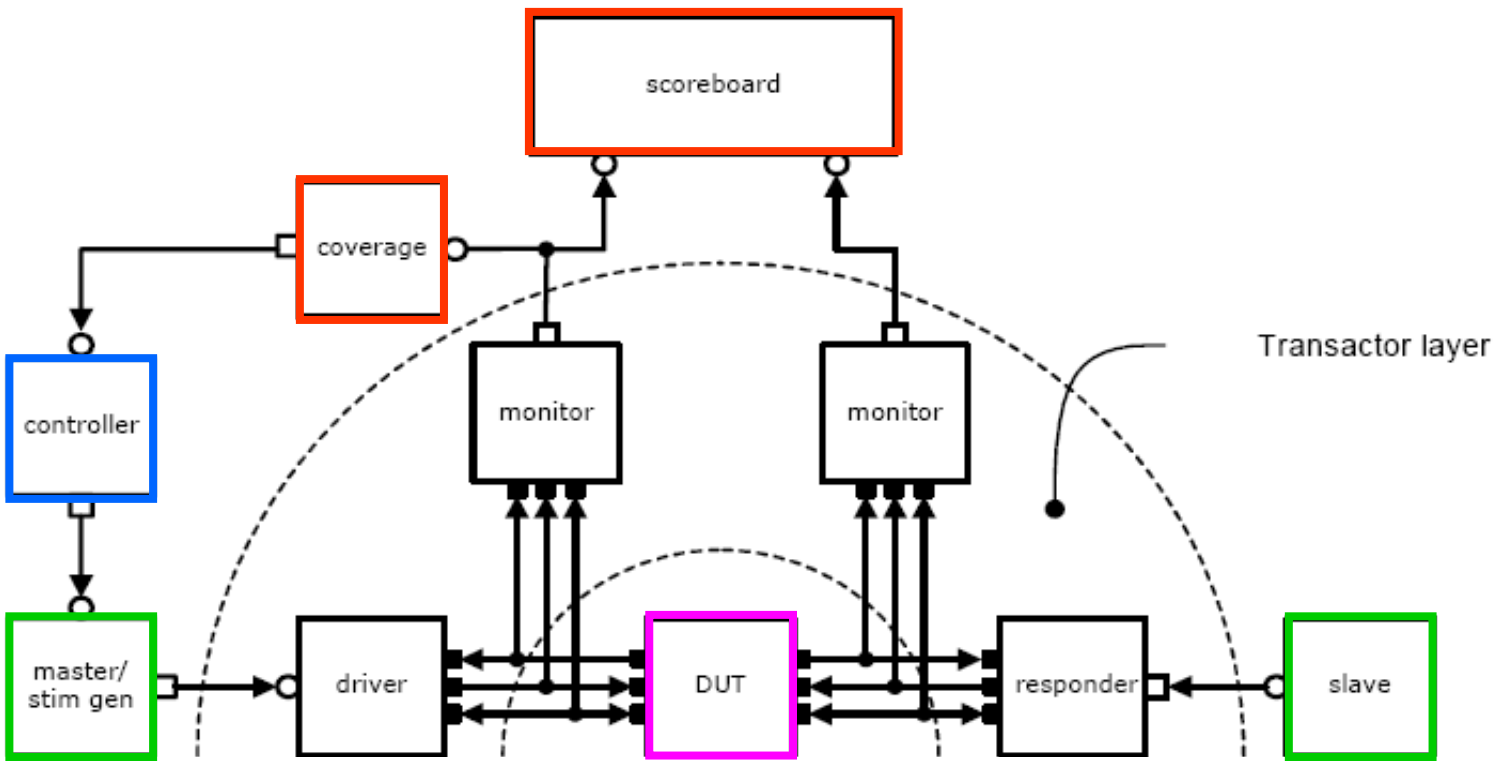
Objective: keep as much of the testbench structure at the transaction level as possible

Why Transactions?

- Easier to write transaction level components than RTL
- Easier to debug transaction level components than RTL
- More opportunities for reuse
  - standard interfaces
  - standard APIs

Mentor AVM 2.0 layers

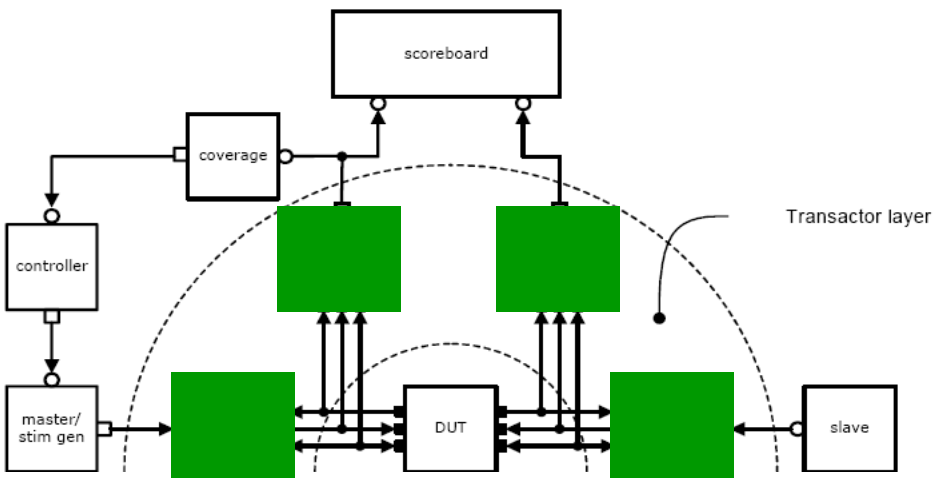
Control	test controller
Analysis	coverage, scoreboard etc
Environment	stim_gen, master-slave, constraints
Transactors	driver, monitor, responder
DUT	pin-level design



AVM Transactor level

Reference Page

- **Driver**
  - **Converts transaction stream to pin-level activity**
- **Responder**
  - **Converts pin-level activity to transaction stream(s)**
- **Monitor**
  - **Monitors pin-level activity for incorrect behavior**
  - **Totally passive: no affect on DUT**



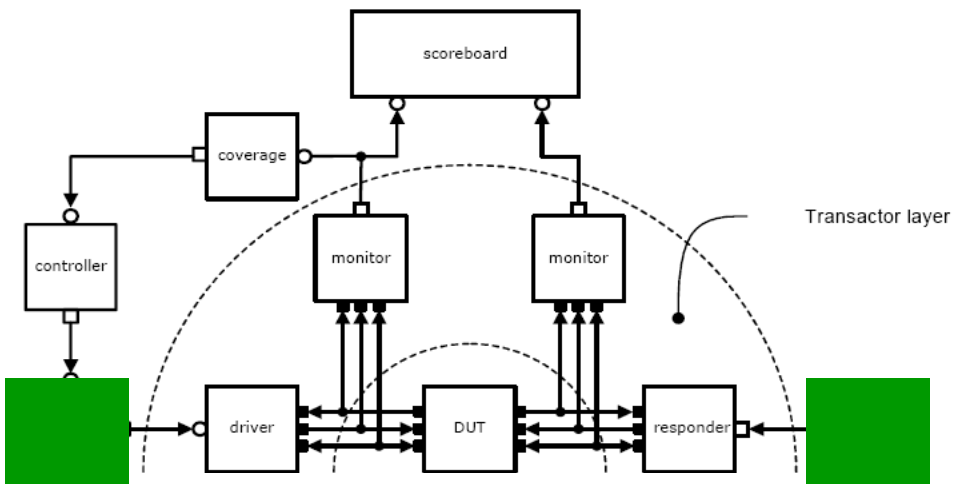
AVM Environment level

Reference Page

- **Stimulus Generator**
  - Generates transaction level stimulus
  - Contains randomization algorithms and constraints

- **Master**
  - Bidirectional component: sends requests and receives responses
  - Initiate activity
  - May use responses to steer requests

- **Slave**
  - Transaction level device driven by a responder
  - Doesn't initiate activity but responds to it appropriately



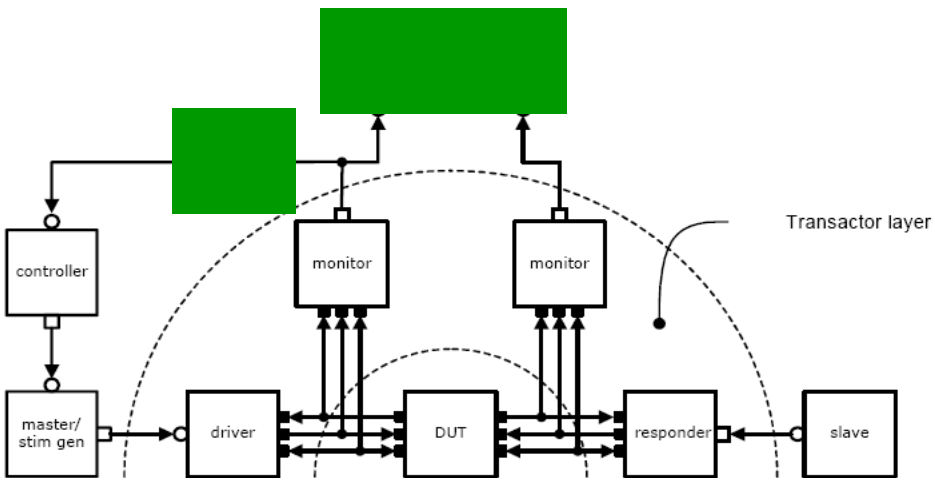
AVM Analysis level

Reference Page

- **Coverage Collector (Functional Coverage)**
  - Monitor and determine completeness of simulation
  - Has counters organized into bins
  - Counts transactions and puts counts into appropriate bin

- **Scoreboard**
  - Tracks transaction level activity from multiple devices
  - Keeps track of information that shows if DUT is functioning properly

- **Analysis port**
  - Pass information from transactors (or lower) to analysis layer
  - Link Transactors, Drivers & Monitors to Scoreboard
  - Abstract, so Scoreboard does not interfere (slow, add wait-state etc) with correct behavior of the DUT



AVM Control level

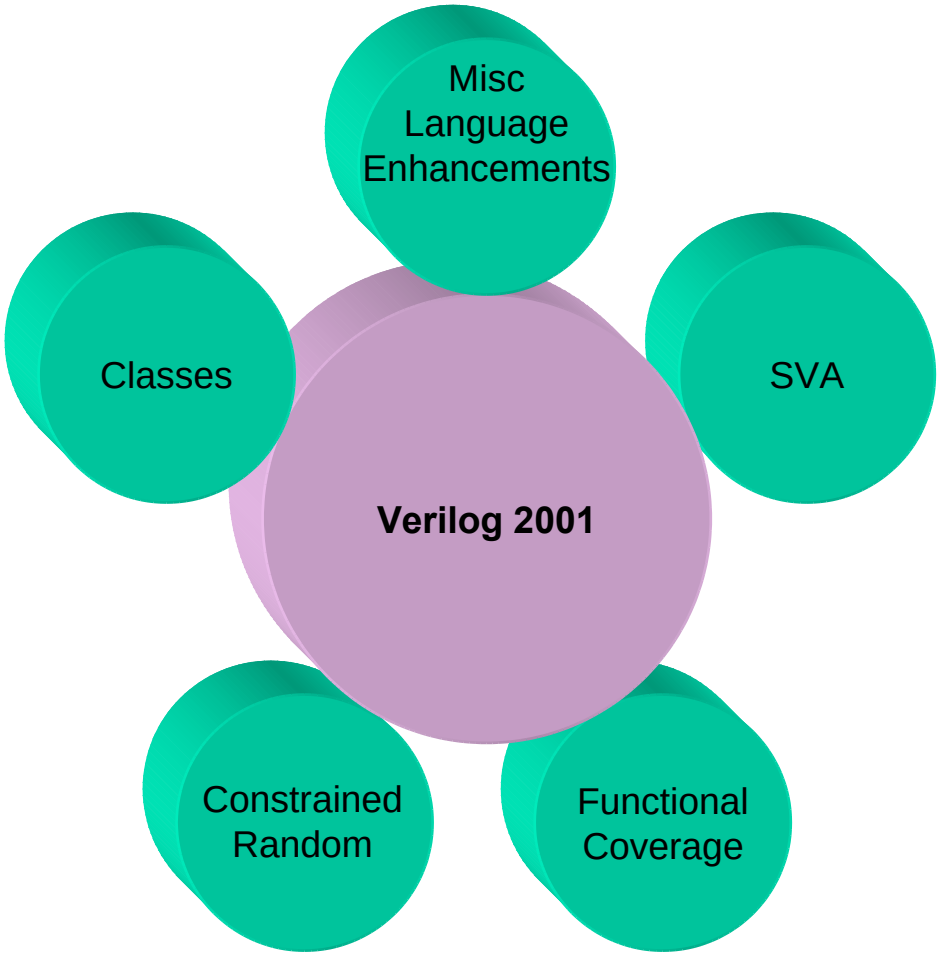
Reference Page

- Controller
  - Main thread of a test :- orchestrates all activity
  - Receive information from scoreboards and coverage collectors
  - Send commands to environment components.

  - Scenario 1:
    1. Controller starts a stimulus generator running
    2. Wait for coverage collector to say test is complete
    3. Controller stops stimulus generator.

  - Scenario 2
    1. Controller starts stimulus generator with initial constraints
    2. Wait for coverage collector to say when certain goals are met
    3. Controller sends stimulus generator modified/new constraints

SV for Verification



Data Types

In this section

Data types
Static & Dynamic Variables
Enumeration

SystemVerilog Data Types

Basic data types

time	64-bit integer, defaults to seconds
real	from Verilog, like C double, 64-bits
shortreal	from C float, 32-bits
string	variable size array of characters
void	non-existent data, used for functions

shortint int longint byte bit	2-state (1, 0) defaults to 0
reg logic integer	4-state (1, 0, X, Z) defaults to x

Integer data types

shortint	16-bit integer
int	32-bit integer
longint	64-bit integer
byte	8-bit integer (ASCII character)
integer	32-bit from Verilog (sized: 0, 1, X, Z)

default to signed

int unsigned usig; *// make unsigned*

bit	0 or 1
reg	from Verilog (unsized: 0, 1, X, Z)
logic	like reg

default to unsigned

reg signed ssig; *// make signed*

Initialization of variables

- SystemVerilog offers more sophistication than Verilog 2001

  Examples:

```
int a = 0;
```

Equiv:
int a;

initial
a = 0;

```
int b = a+2;

bit[31:0] c = $urandom_range(0,152)  // c between 0 and 152 inclusive
```

User Defined Types

- SystemVerilog supports a new keyword: **typedef**

Syntax:

```
typedef <base_data_type> <type_identifier>
```

```
typedef int typ ;          // typ becomes a new type
typ a = 1, b = 10;         // these are 2 new variables of type 'typ'
```

Literals

- SystemVerilog allows easy specification of un-sized literal values with the (') apostrophe:

```
'0, '1, 'x, 'X, 'z, 'Z // Notice, no base specifier needed
reg [23:0] a = 'z;     // All bits of a are set to "z"
```

- String literals are written as in Verilog, between double quotes
- SV adds new escaped characters:

```
" \v "     // vertical tab
" \f "     // form feed
" \a "     // bell
" \x02 "   // hexadecimal number
```

Enumeration

Syntax:

```
enum [enum_base_type] { enum_name_declaration
{,enum_name_declaration} }

enum_base_type:  default is int
```

- Enumeration is a useful way of defining abstract variables.

NOTE:

Default assigned values start at zero

```
        0        1          2
enum  {red, green, yellow} lite;
```

- Define an enumeration with " enum "

```
enum  {red, green, yellow} traf_lite1, traf_lite2;
```

- Values can be cast to integer types and auto-incremented

```
enum  { a=5, b, c} vars;    // b=6, c=7
```

- A sized constant can be used to set size of the type

```
enum bit[3:0] {bronze=4'h3, silver, gold} medal;
```

// All medal members are (must be) 4-bits

Enumerated Types

- **Enumerated Type**
  - Define a new type

```
typedef enum  {NO, YES} boolean;     // boolean is NOT a SystemVerilog type
boolean myvar;                       // but it just became one ☺
```

"myvar" will be checked for valid values in all assignments

```
module enum_types;
typedef enum {red, green, blue, yellow, white, black} colors ;
colors my_colors;

initial begin
  my_colors = green;
//  my_colors = 1;            // invalid,  needs to typecast
  my_colors = colors'(2);   // cast as blue
  case (my_colors)
    red:  $display ("my_color is red");
    blue:  $display ("my_color is blue");
    default:  $display ("my_color is not blue or red");
  endcase
end
endmodule
```

Enumeration Examples

- Consider this enumeration

```
enum  {red, green, yellow} lite1, lite2;  // anonymous int type assumed
```

- Default anonymous int type cannot be assigned ‘x’ or ‘z’

```
enum  { a, b = 'x, c = 'z} vars;          // ILLEGAL assignment to x or z
enum logic { a, b = 'x, c = 'z} vars;     // legal assignment to logic type
```

All SV types supported

- **Example enum in Finite State machines**
  - Replace parameters as symbolic names
  - Strongly-typed preventing mis-assignment

```
enum logic [1:0] { S0, S1, S2 } state, next_state;
```

Enumeration Methods

- SystemVerilog provides some methods to allow easy manipulation of enumerated types
  - `function enum first()`
    - returns the value of the first member of the enumeration enum.
  - `function enum last()`
    - returns the value of the last member of the enumeration enum.
  - `function enum next(int unsigned N = 1)`
    - returns the Nth next enumeration value (default is the next one) starting from the current value of the given variable.
    - Note: next() "wraps" to the first value when applied to the last
  - `function enum prev(int unsigned N = 1)`
    - returns the Nth previous enumeration value (default is the previous one) starting from the current value of the given variable.
    - Note: prev() "wraps" to the last value when applied to the first
  - `function int num()`
    - returns the number of elements in the given enumeration.
  - `function string name()`
    - returns the string representation of the given enumeration value

Enumeration Methods Use

Example:

```
module enum_methods;
typedef enum {red, green, blue, yellow, white, black} colors ;
colors c = c.first();
initial begin
  forever begin
    $display("enum declaration %0d = %0s", c, c.name());
    if(c == c.last())
      $finish;
    c = c.next();
  end
end
endmodule
```

Output:

enum declaration 0 = red
enum declaration 1 = green
enum declaration 2 = blue
enum declaration 3 = yellow
enum declaration 4 = white
enum declaration 5 = black

Static (compile-time) Casting

Syntax:
`<type>' (<value>)`

```
reg [31:0] a ,b;
typedef int mytype;
mytype green;
```

- Convert between data types with the cast (forward-tick):

```
int' (2.0*3.0);    // Cast result to integer
10' (a+b);         // implies a number of bits (10 here)
signed' (a);       // works for sign-changing too
```

- User defined types may also be cast:

```
green = mytype' (b);
```

- The following Verilog functions are still supported:
  `$itor()`, `$rtoi()`, `$bitstoreal()`, `$realtobits()`, `$signed()`, `$unsigned()`

Dynamic Casting - $cast

Syntax:

```
function int $cast(  singular_dest_var, source_exp );
```

or

```
task $cast( singular_dest_var, source_exp );
```

NOTE 1
$cast is a runtime check
NOT compile time

NOTE 2
Singular means any type except
unpacked struct, union, array

- Both forms attempt to assign source_exp to dest_var
  - **Function**: If assign is successful return 1, else return 0 and leave dest_var unchanged
  - **Task**: If assign is not successful, trigger a runtime error and leave dest_var unchanged

```
typedef enum { red, green, blue, yellow, white, black }
Colors;
Colors col;
$cast( col, 2 + 3 );
```

This code assigns 5 (or black) to col. Without $cast, this assignment is illegal

-OR-

Use the function form of $cast to check if the assignment will succeed:

```
if ( ! $cast( col, 24 ) )      // This is an invalid cast
   $display( "Error in cast" );
```

String Data Type

- String data type is a variable size array of characters, indexed from 0 to N-1 (N is array length)
  - Every element of the array is also a string

```
string st1, st2;
st1 = "";                    // null string
st2 = "abc";                 // assigned from a string literal
st1 = { st1, st2 };          // concatenation, st1 becomes "abc"
st2[0:1] = "et";             // st2 becomes "etc"
```

a	b	c
0	1	2

e	t	c
0	1	2

- Supports relational operators (`==, !=, <, <=, >, >=`), concatenation (`{ }`) and replication (`{n{ }}`)
  - e.g. (a<b) is true because a precedes b alphabetically
- By means of the . operator, strings support special methods:
  - `len(), putc(), getc(), toupper(), tolower(), compare(), icompare(), substr(), atoi(), atohex(), atooct(), atobin(), atoreal(), itoa(), hextoa(), octtoa(), bintoa()` **and** `realtoa()`

Parameterized types

- SystemVerilog extends Verilog parameters to support types.

  This example shows a fifo, whose width, depth AND TYPE are parameterized:

```
module fifo  #( parameter depth = 16,          // default width/depth
                parameter width = 8,           // default width/depth
                parameter type ft = bit )      // fifo is type bit (2-state) by default
               ( input  ft [width-1:0] datin, input bit r_w, clk,
                output  ft [width-1:0] datout );

ft [width-1:0] mem [depth-1:0];          // declare fifo array & type (bit by default)
. . .
endmodule

module use_fifo;
  logic [31:0] i, o;
  bit clk, r_w;

  fifo   #( .depth(64), .width(32),      //  override both parameter default values
                .ft(logic) )             //  override parameter ft to be type logic (4-state)
        U1(.datout(o), .datin(i), .clk(clk), .r_w(r_w));

endmodule
```

Tip

An easy way (as here) to switch between 2-state and 4-state operation of your design.

Const

- The **const** keyword effectively means the variable may not be changed by user code
  - Value is set at run-time and it can contain an expression with any hierarchical path name

```
const logic option = a.b.c ;
```

- A **const** may be set during simulation within an automatic task (discussed later)

Arrays & Structures

In this section

Dynamic Arrays
Associative Arrays
Queues
Structure / Union

Multi-dimensional
Using Arrays
Supported data types & operations
Querying functions

Dynamic Arrays

- Dynamic declaration of one-dimensional arrays

Syntax:

```
data_type array_name[] ;
```

Declares a dynamic array *array_name* of type *data_type*

```
data_type array_name[] = new[ array_size ] [(array)] ;
```

Allocates a new array *array_name* of type *data_type* and size *array_size*

Optionally assigns values of *array* to *array_name*

If no value is assigned then element has default value of *data_type*

Declaration

Resize

Declare & size

```
bit [3:0] nibble[ ];     // Dynamic array of 4-bit vectors
int data[ ];             // Dynamic array of int

initial begin
  nibble = new[100];     // resize to 100-element array
  data = new [256];      // resize to 256-element array
end

int addr[ ] = new [50]; // Create a 50-element array
```

Dynamic Arrays - Methods

- **function int size()**
  - Returns the current size of the array
- **function void delete()**
  - Empties array contents and zero-sizes it

```
int my_addr[ ] = new[256];

initial begin
  $display("Size of my_addr = %0d", my_addr.size() );
  my_addr.delete();
  $display("Size of my_addr (after delete) = %0d", my_addr.size());
end
```

output:
 # Size of my_addr = 256
 # Size of my_addr (after delete) = 0

Dynamic Arrays – Array Assignments

- Assignment of a dynamic array to a dynamic array
  - Creates a new dynamic array the size of the RHS array

```
int dyn_array[] = new[100];
int  an_array[];

initial begin
  an_array = new[dyn_array.size()](dyn_array);   // same as next line
  an_array = dyn_array;                          // same as line above
  an_array[2] = 222;                             // init location  2
  an_array = new[150];                           // resize array - lose contents
  $display("1: an_array[2] = %0d",an_array[2]);
  an_array[3] = 333;                             // init location 3
  an_array = new[200](an_array);                 // resize array - save contents
  $display("2: an_array[3] = %0d",an_array[3]);
end
```

output:
 # 1: an_array[2] = 0
 # 2: an_array[3] = 333

- Assignment of a fixed size array to a dynamic array
  - OK if data type and array size are the same

```
int odd_size[] = new[87];
int fix_sz_array[100];

initial begin
  dyn_array = fix_sz_array;      // OK  same type and size
  odd_size = fix_sz_array;       // type check error - different size
end
```

Arrays – Associative

- Associative arrays (sometimes called indexed arrays)
  - Support situations where data set size is totally unpredictable and elements may be added or removed individually to grow/shrink the array
  - Implemented as a look up table and so require an index.

Syntax:

```
data_type array_id [ index_type ];   // index type is the datatype to use as index
                                     // examples include string, int, class, struct
```

Example:

```
bit i_array[*];                 // associative array of bits (unspecified index)
                                // unspecified index (*) implies any integral value
bit [7:0] age [string];         // associative array of 8-bit vectors, indexed by string

initial begin
   string tom = "tom";
   age [tom] =  21;
   age ["joe"] =  32;
   $display("%s is %d years of age ", tom, age[tom], "[%0d ages available]", age.num());
end
```

tom is 21 years of age [2 ages available]

- 7 new methods support associative arrays (see reference section)

```
num(), delete(), exists(), first(), last(), next(), prev()
```

Associative Array Methods

```
function int num()
```

- Returns the number of entries in the array, if empty returns 0

```
function void delete( [input index] )
```

- Index is optional
- If index is specified, deletes the item at the specified index
- If index is not specified the all elements in the array are removed

```
function int exists ( input index );
```

- Checks if an element exists at the specified index within the given array.
  - Returns 1 if the element exists, otherwise it returns 0

```
function int first( ref index )
```

- Assigns to the given index variable the value of the first (smallest) index in the associative array
  - It returns 0 if the array is empty, and 1 otherwise

```
function int last( ref index )
```

- Assigns to the given index variable the value of the last (largest) index in the associative array
  - It returns 0 if the array is empty, and 1 otherwise.

```
function int next( ref index );
```

- finds the entry whose index is greater than the given index. If there is a next entry, the index variable is assigned the index of the next entry, and the function returns 1
  - Otherwise, index is unchanged, and the function returns 0

```
function int prev( ref index );
```

- finds the entry whose index is smaller than the given index. If there is a previous entry, the index variable is assigned the index of the previous entry, and the function returns 1
  - Otherwise, the index is unchanged, and the function returns 0

Queues & Lists

- SV has a built-in list mechanism which is ideal for queues, stacks, etc.
- A list is basically a variable size array of **any** SV data type.

```
int q1[$];          // $ represents the 'upper' array boundary
int n, m, item;

q1 = '{ n, q1 };    // uses concatenate syntax to write n to the left end of q1
q1 = '{ q1, m };    // uses concatenate syntax to write m to the right end of q1

item = q1[0];       // read leftmost ( first ) item "n" from list
item = q1[$];       // read rightmost ( last ) item "m" from list

n = q1.size();      // determine number of items on q1

q1 = q1[1:$];       // delete leftmost ( first ) item of q1
q1 = q1[0:$-1];     // delete rightmost ( last ) item of q1

for (int i=0; i < q1.size(); i++)// step through a list using integers (NO POINTERS)
  begin … end

q1 =  '{ };         // clear the q1 list
```

' (tick) signifies cast overload and is required

Queue Methods

function int size()

- Returns the number of items in the queue. If the queue is empty, it returns 0.

function void insert (int index, queue_type item)

- Inserts the given item at the specified index position
- `Q.insert (i, e)  =>  Q = '{Q[0:i-1], e, Q[i:$]}`

function void delete (int index)

- Deletes the item at the specified index position
- `Q.delete (i)  =>  Q = '{Q[0:i-1], Q[i+1:$]}`

function queue_type pop_front()

- Removes and returns the first element of the queue
- `e = Q.pop_front ()  =>  e = Q[0]; Q = Q[1:$]`

function queue_type pop_back()

- Removes and returns the last element of the queue
- `e = Q.pop_back ()  =>  e = Q[$]; Q = Q[0:$-1]`

function void push_front (queue_type item);

- Inserts the given element at the front of the queue
- `Q.push_front (e)  =>  Q = '{e, Q}`

function void push_back (queue_type item);

- Inserts the given element at the end of the queue
- `Q.push_back (e)  =>  Q = '{Q, e}`

Queue Example: Simple FIFO

```
module q_fifo( input  logic [7:0] data_in, output bit empty, full,
               logic [7:0] data_out,  input event write_fifo, read_fifo );

logic [7:0] q [$];                      //  Declare the queue q
always @ (write_fifo)                   //  the write_fifo event
  if(full)
    $display ("tried to write a full fifo");
  else begin
    q.push_front(data_in);              // write to left end of q
    if(q.size > 7)
      full = 1;                         // set as full
    empty = 0;                          // set as not empty
  end
always @ (read_fifo)                    // the read_fifo event
 if(empty)
   $display("tried to read an empty fifo");
 else begin
   data_out <= q.pop_back();              // remove from right end
   full = 0;                              // not full after a read
   if(q.size() == 0)
     empty = 1;                           // set as empty
 end
endmodule
```

Structures

- Think of a structure as an object containing data members of any SV type

```
struct{
  bit[7:0]   my_byte;
  int  my_data;
  real pi;
} my_struct;    // my_byte, my_data and pi are "members" of my_struct
```

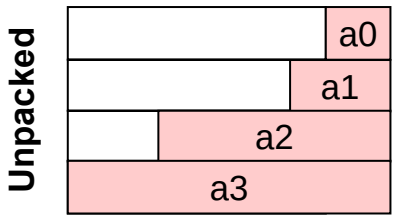
- *Data members can be referenced individually (using the . operator) or altogether as a unit*

```
initial begin
    my_struct.my_byte = 8'hab;
    my_struct = '{0, 99, 3.14};
end
```

Structures may:
- be packed or unpacked.
- be assigned as a whole
- pass to/from a function or task as a whole
- contain arrays

Unpacked vs Packed Structures

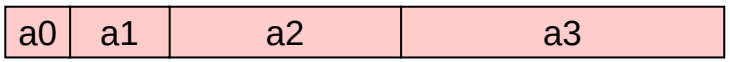
Unpacked




```
struct {bit[1:0]a0;
        bit[2:0] a1;
        bit[5:0] a2;
        bit[8:0]a3;
       } u_pkt;
```

- Default for structs
- Tool dependant implementation.
- Think of as a logical grouping of member elements
- Each member may be assigned differently

Packed




```
struct packed  {
        bit[1:0] a0;
        bit[2:0] a1;
        bit[5:0] a2;
        bit[8:0] a3;
        } p_pkt;
```

- Much more useful in hardware
- Easily converted to bit-vectors
- May be accessed as a whole
- First member specified is most significant
- May be declared as signed for arithmetic

Limitations

- Only integer types (bit, logic, int, reg, time)
- Unpacked arrays/structures are NOT allowed here
- ***If any member is 4-state, all members are cast to 4-state***

Unpacked vs Packed Structures -2

- Either type of structure may be typedef'd:

```
typedef struct  { byte RED,GRN,BLU; } RGB;      // Named structure
RGB screen [640][400];                          // screen of 640x400 pixels
```

- Packed structures may also be declared signed:

```
struct packed signed
    { int a; shortint b; byte c; bit[7:0] d;} pack1;    // signed, 2 state
struct packed unsigned
    { time a; logic[7:0] b;} pack2;                     // unsigned, 4 state

typedef struct packed {     // Packed struct/unions may also be typedef'd
    bit[3:0] f1;
    bit[7:0] f2;
    bit[11:0] f3;
    bit f4;
    bit[63:0] f5;
} a_packed_struct;
```

Uses of structures

- Structures are ideal way to encapsulate data "packets"
- Use as data or control abstraction in architectural models
- Use as abstraction for top-down I/O design
  - **Behavioral:**
    - pass structures through ports, as arguments to tasks/functions, etc.
    - Use to refine structure size/content
  - **RTL:**
    - break structure apart & define final I/O
  - **Verification:**
    - to encapsulate constraint/randomization weights

Multidimensional Arrays

Reference Page

- SystemVerilog supports multi-dimensional arrays just like Verilog…

```
bit [7:0] mem [4:1];     // byte-wide memory with 4 addresses, like Verilog
mem[ i ] [6:1] = 0;      // 2D+ indexing supported (like Verilog 2001)
```

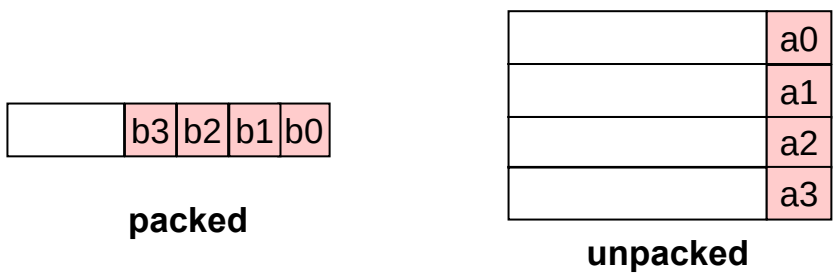
- Also from Verilog 2001 we get multi-multi-dimensions… phew!

```
bit [a:b] [n:m] [p:q] mem [ t:u] [v:w] [x:y]; // arbitrary dimensions
```

packed

unpacked

- The terms packed and unpacked to refer to how the data is actually stored in memory
  - packed => 8-bits to a byte, unpacked => 1 bit per word

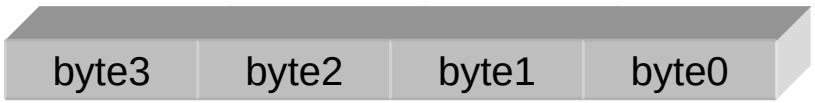


Multidimensional Array Examples

Reference Page

- The packed dimensions describe how the data is arranged (or packed)
- The unpacked dimensions describe how we map this data to a multi-dimension address

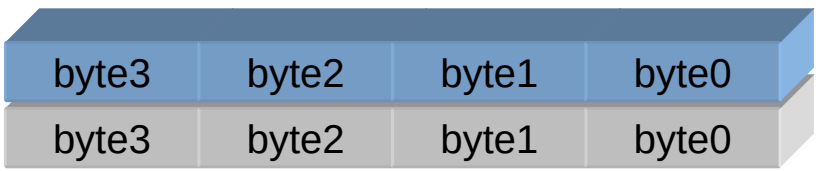
```
bit [3:0] [7:0] aa ;
```
// a single, packed 4-byte data word (4x8 = 32 bits)



aa

```
aa[0] = aa[0] + 1;
```
// byte increment

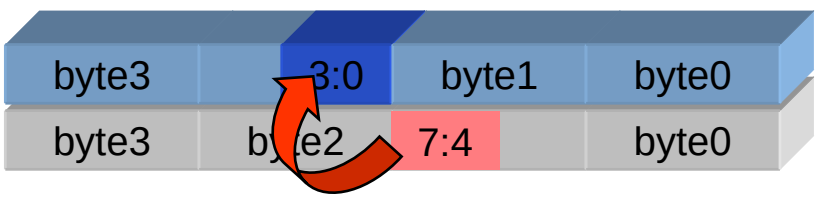
```
bit [3:0] [7:0] bb [1:0];
```
// 2-deep array of packed 4-byte data words



bb[1]

bb[0]

```
bb[1] = bb[0];
```
// word assignment



bb[1]

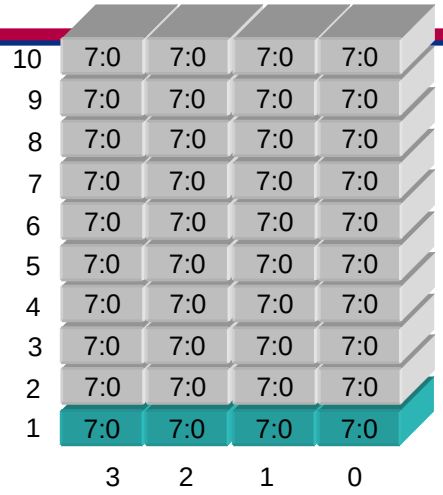
bb[0]

```
bb[1][2][3:0] = bb[0][1][7:4];
```
// nibble copy

More Multidimensional Examples

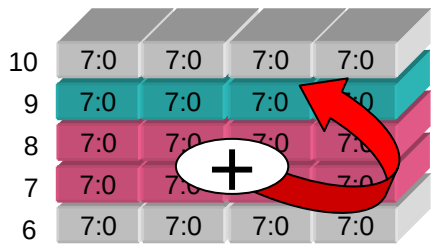
Reference Page

```
bit [3:0] [7:0] aa ;                // packed 4-byte variable
bit [3:0] [7:0] bb [10:1] ;         // array of 10  4-byte words
```

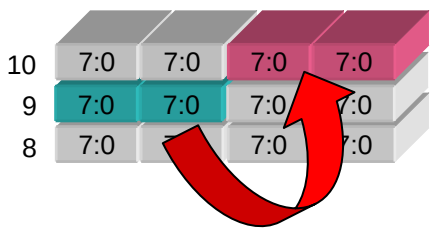


- To access: start with unpacked dimensions proceeding left to right then continue with the packed dimensions, also proceeding left to right…

```
bb[9] = bb[8] + bb[7];      // add 2 4-byte words
```




```
bb[10][1:0] = bb [9][3:2] ;    // copy 2 MS bytes
                               //  from word 9 to word 10 (LS bytes)
```

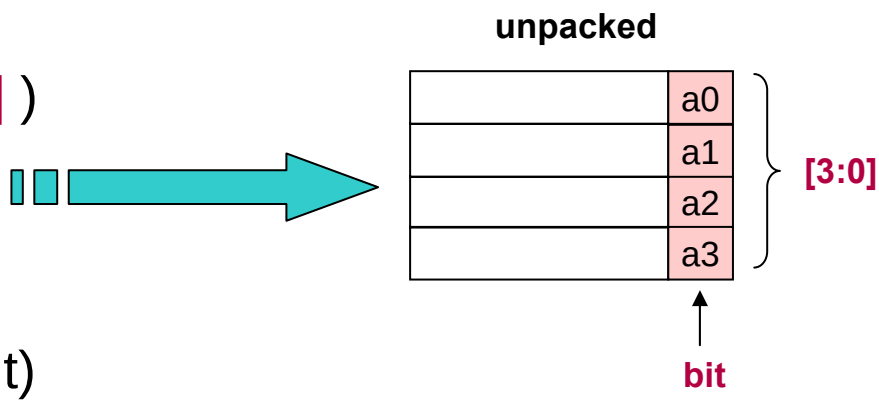


Array Data Types

Reference Page

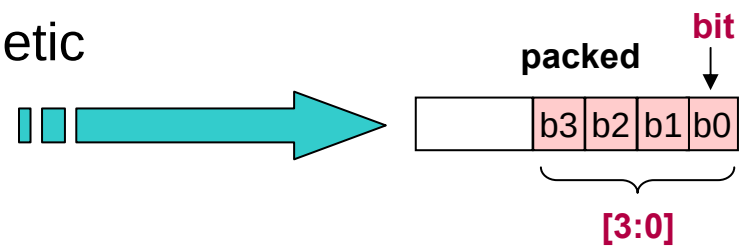
- **Unpacked**
  - Familiar from Verilog, may be ANY Datatype
  - Only access a single element at a time
    - Although whole arrays may be copied
  - One element may be assigned procedurally while another is assigned continuously
  - Specified as a range (int mem [256:1])

```
bit a [3:0];    // unpacked array of bits
```



- **Packed**
  - Only bit-level types (reg, wire, logic, bit)
  - Access whole array or slice as a vector
  - All elements must be assigned identically
  - Compatible with $monitor/$display etc.
  - Allows arbitrary length integers/arithmetic

```
bit [3:0] b;    // packed array of bits
```



Array Examples

Reference Page

```
bit [2:0][7:0] m =  24'b0;          // packed 3-byte vector

bit [7:0] nn [0:5];   // unpacked array of 6 bytes
bit [7:0] nn [6];     // unpacked array of 6 bytes
```

Equivalent
[N] => [0:N-1]

```
typedef bit [1:6] bsix;             // 6-bit packed vector
bsix [1:10] mine;                   // 60-bit packed vector
```

equiv: `bit [1:10][1:6] mine;`

Arrays – Supported Operations

Reference Page

```
bit[7:0] PA, PB ;
int var;

initial begin
  var = 3;
  PA = PB;
  PA[7:4] = 'hA;
  PA[var -:4] = PA[var+1 +:4];
    equiv: PA[3:0] = PA[7:4];

end
```

Packed & unpacked arrays support:

// Read/write
// Read/write of a slice
// Read/write of a variable slice

Verilog 2001 Syntax

```
[ M -: N ]    // negative offset from bit index M, N bit result
[ M +: N ]    //  positive offset from bit index M, N bit result
```

```
bit[3:0][7:0] MPA;

initial begin
   MPA = 32'hdeadbeef;
   MPA = MPA+1;

end
```

<u>Only</u> packed arrays allow:

// Assignment from an integer
// Treatment as an integer in an expression

Unpacked Array Literals

Reference Page

- Assigning literal values to SV arrays:

```
int  k [1:3][1:4] = '{default: 5};            // All elements "5"
```

- For more control, consider the dimensions of the array and use { } to match those dimensions exactly

```
int k [1:3][1:4] = '{'{1,2,3,4}, '{5,6,7,8}, '{9,10,11,12}};// 3 groups of 4
```

```
int m [1:2][1:3] =  '{'{0,1,2}, '{3{4}}}; // 2 groups of 3
```

Arrays and $readmem h/b

Reference Page

```
module array_ops2;

bit[7:0] PA [3:0];
bit[1:0][7:0] PB [3:0];   // two packed dimensions
byte UA [7:0];
byte UB [7:0][1:0];       // two unpacked dimensions

initial
 begin
   #10 $readmemh("hex.dat",PA);
       for(int i=0; i<=3;i++)
          $display("PA[%0h",i,"]: %b",PA[i]);

   #10 $readmemh("hex.dat",PB);
       $display("");
       for(int i=0; i<=3;i++)
          $display("PB[%0h",i,"]: %b",PB[i]);

   #10 $readmemh("hex.dat",UA);
       $display("");
       for(int i=0; i<=3;i++)
          $display("UA[%0h",i,"]: %b",UA[i]);

 #10 $readmemh("hex_multi.dat",UB);
       $display("");
       for(int i=0; i<=3;i++)
         for(int j=0; j<=1;j++)
           $displayh("UB[",i,"][",j,"]: ",UB[i][j]);
 end
endmodule
```

```
00: 00
01: 01
02: AA
03: FF
```

hex.dat

```
[1:0]
[7:0]
00 01
10 11
20 21
30 31
40 41
50 51
60 61
70 71
```

hex_multi.dat

```
PA[0]: 00000000
PA[1]: 00000001
PA[2]: 10101010
PA[3]: 11111111

PB[0]: 0000000000000000
PB[1]: 0000000000000001
PB[2]: 0000000010101010
PB[3]: 0000000011111111

UA[0]: 00000000
UA[1]: 00000001
UA[2]: 10101010
UA[3]: 11111111

UB[00000000][00000000]: 00
UB[00000000][00000001]: 01
UB[00000001][00000000]: 10
UB[00000001][00000001]: 11
UB[00000002][00000000]: 20
UB[00000002][00000001]: 21
UB[00000003][00000000]: 30
UB[00000003][00000001]: 31
```

Array manipulation methods

Reference Page

- SV supports a variety of methods to search, order and reduce arrays.

Syntax:

expression.array_method_name [(list_of_arguments)]
[with (expression)]

Search

find()	returns all the elements satisfying the given expression
find_index()	returns the indexes of all the elements satisfying the given expression
find_first()	returns the first element satisfying the given expression
find_first_index()	returns the index of the first element satisfying the given expression
find_last()	returns the last element satisfying the given expression
find_last_index()	returns the index of the last element satisfying the given expression

min()	returns the element with the minimum value
max()	returns the element with the maximum value
unique()	returns all elements with unique values
unique_index()	returns the indexes of all elements with unique values

with clause is optional under certain conditions

```
string SA[10], qs[$];
int IA[*], qi[$];

qi = IA.find( x ) with ( x > 5 );          // Find all items greater than 5
qs = SA.unique( s ) with ( s.tolower );    // Find all unique lowercase strings
```

Array manipulation methods - 2

Reference Page

Order

reverse()	reverses all the elements of the array (packed or unpacked)
sort()	sorts the unpacked array in ascending order
rsort()	sorts the unpacked array in descending order
shuffle()	randomizes the order of the elements in the array

```
string s[] = { "teacher", "apple", "student" };
s.reverse();                    // s becomes { "student", "apple", "teacher" };

struct { byte red, green, blue; } video [512];
video.sort() with ( item.red );          // sort video using the red byte only
video.sort( sel ) with ( sel.blue << 8 + sel.green );      // sort by blue-green algorithm
```

Reduce

sum()	returns the sum of all the array elements
product()	returns the product of all the array elements
and()	returns the bitwise AND (&) of all the array elements
or()	returns the bitwise OR (|) of all the array elements
xor()	returns the logical XOR (^) of all the array elements

```
byte b[] = { 1, 2, 3, 4 };
int y;

y = b.sum() ;          // y becomes 10 => 1 + 2 + 3 + 4
y = b.product() ;      // y becomes 24 => 1 * 2 * 3 * 4
```

Array Querying functions

Reference Page

$bits, $left, $right, $low, $high, $increment, $size, $dimensions

Return type is `integer` for all querying functions

`$dimensions( array_id)`

Returns the number of dimensions in the array (0 if scalar)

```
bit [3:0] [7:0] aa;
initial $display( $dimensions(aa) ); // prints " 2 "
```

`$bits(array_id):`

Returns the number of bits in array or struct

initial $display($bits(aa)); *// prints " 32 "*

`${left|right}(array, N = 1):`

Returns bounds of a dimension // $left => msb, $right => lsb

```
bit [3:1] [7:4] cc [2:0];
initial  $display( $left(cc) , $right(cc,2) );   // prints " 2    1"
```

`${low|high} (array_id, N = 1)`

Returns the min|max bounds of a variable // min|max of $left and $right of dimension

`$increment(array_id, N = 1)`

Returns 1 if $left is greater than or equal to $right

Returns –1 if $left is less than $right

`$size(array_id, N = 1)`

returns number of elements in the dimension ($high - $low +1) **(was `$length` (now deprecated))**

Array Query function quiz

Reference Page

```
bit   [7:0][5:0] e [3:0][31:0];

initial
   $display( $bits(e),,
                 $dimensions(e),,
                         $size(e,1),,
                                 $size(e,2),,
                                         $size(e,3),,
                                                 $size(e,4));
```

6144	4	4	32	8	6

Unions

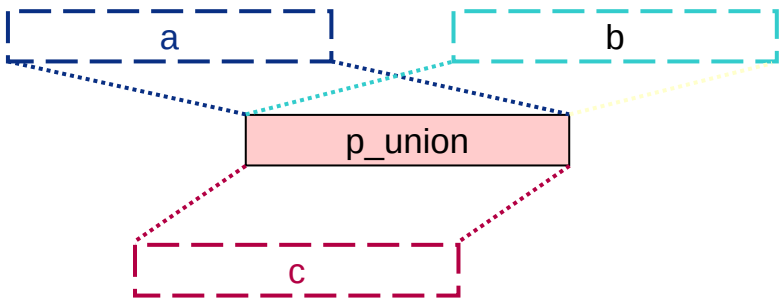
- Also borrowed from C, SV supports packed and unpacked forms
- C unions are essentially the same as SV unpacked unions
  - Union is a specialized form of struct
  - Memory footprint is the size of the largest member
  - Used to reduce memory consumption of a design
  - All members begin at the same memory address
  - The member read must also be the most recently written
- Their limitations make unpacked unions less useful for h/w
- Packed unions HOWEVER, are VERY useful

Packed Unions

Reference Page

- A packed union contains 1 or more packed members
  - All members are the same size and occupy the same space
  - Data may be written via one member, read by another
  - Unlike C/C++ SV unions are independent of platform byte ordering
  - The union may be accessed as a whole

```
union packed  {
int     a;
integerb;
reg[31:0]  c;
} p_union;
```



- Characteristics
  - Easy conversion to bit-vectors
  - May be accessed as a whole
  - First member specified is most significant
  - May be declared as signed for arithmetic
  - Non-integer datatypes (e.g. real) are NOT allowed
  - Unpacked arrays/structures are NOT allowed
  - ***if any member is 4-state, all members are cast to 4-state***

Packed Union - example

Reference Page

- Since all members are same size and occupy the same memory space
  - The same data may be accessed via any of the members
  - Permits one data element to exist in multiple name-spaces

```
typedef struct packed {        // Default unsigned
          bit [7:0] f1, f2;
          bit [11:0] f3;
          bit [2:0] f4;
          bit f5;
          bit [63:0] f6;
          } my_packed_struct;

typedef union packed {         // Default unsigned
          my_packed_struct mps;
   3      bit [95:0] bits;
members   bit [11:0] [7:0] bytes;
          } my_packed_union;

my_packed_union u1;
```

Given:

```
byte b;

b = u1.bits[87:80];
b = u1.bytes[10];
b = u1.mps.f2;
```

equivalent!

Structure Expressions

Reference Page

```
module mod1;

typedef struct {
   logic [7:0] a;
   int b;
   } my_struct;

my_struct s1 ;

initial begin
    $monitor("my_struct s1.a: %h, s1.b: %h",s1.a, s1.b);

    #10 s1 =  '{5, 6};                    // assign by position
    #10 s1 =  '{b:5, a:6};                // assign by name
    #10 s1 =  '{default: 7};              // default: new SV operator
    #10 s1 =  '{int:9, default:1};        // assign by type, others default

  end
endmodule
```

NOTE

SV distinguishes between structure expressions (as shown here) and ordinary concatenations by the **'{** syntax.

Simulator output

```
my_struct s1.a: xx, s1.b: 00000000
my_struct s1.a: 05, s1.b: 00000006
my_struct s1.a: 06, s1.b: 00000005
my_struct s1.a: 07, s1.b: 00000007
my_struct s1.a: 01, s1.b: 00000009
```

Bit-stream casting

Reference Page

- A bit-stream data type is any data type that may be represented as a serial stream of bits. This includes any SV type except handle, chandle, real, shortreal, or event.
- Bit-casting allows easy conversion between bit-stream types, for example to model packet transmission over a serial communication stream.

```
typedef struct {  shortint c;
                  byte payload [2] ; } pkt_t;

pkt_t  pkt_a = '{ c:42, default:1};
pkt_t  pkt_b;

int int_c;        // bit-stream is modeled here as a simple int

initial
   begin
     int_c = int'(pkt_a);        // cast to stream
     #1 $stop;
   end

always @(int_c)
   begin
     pkt_b = pkt_t'(int_c);      // cast from stream
     $display("Received: c: %0d, payload[0]: %0d, payload[1]: %0d",
              pkt_b.c, pkt_b.payload[0], pkt_b.payload[1]);
   end
```

Must be same width

NOTE
There are rules and limitations to the use of bit-stream casting.
See the LRM for more information.

Simulator output

```
# Received: c: 42, payload[0]: 1, payload[1]: 1
# Break at bit_cast.sv line 18
```

SV Scheduler

In this section

- Processes
- Events
- Time Slot regions

Processes

- An SV description consists of connected threads of execution called processes
- Processes are objects
  - Can be evaluated
  - Can have state
  - Respond to changes on inputs
  - Produce outputs
- All processes are concurrent
  - Order of execution is indeterminate
- SV processes:
  - Procedural blocks
    - `initial, always, always_ff, always_comb, always_latch`
  - Primitives, continuous assignments, asynchronous tasks

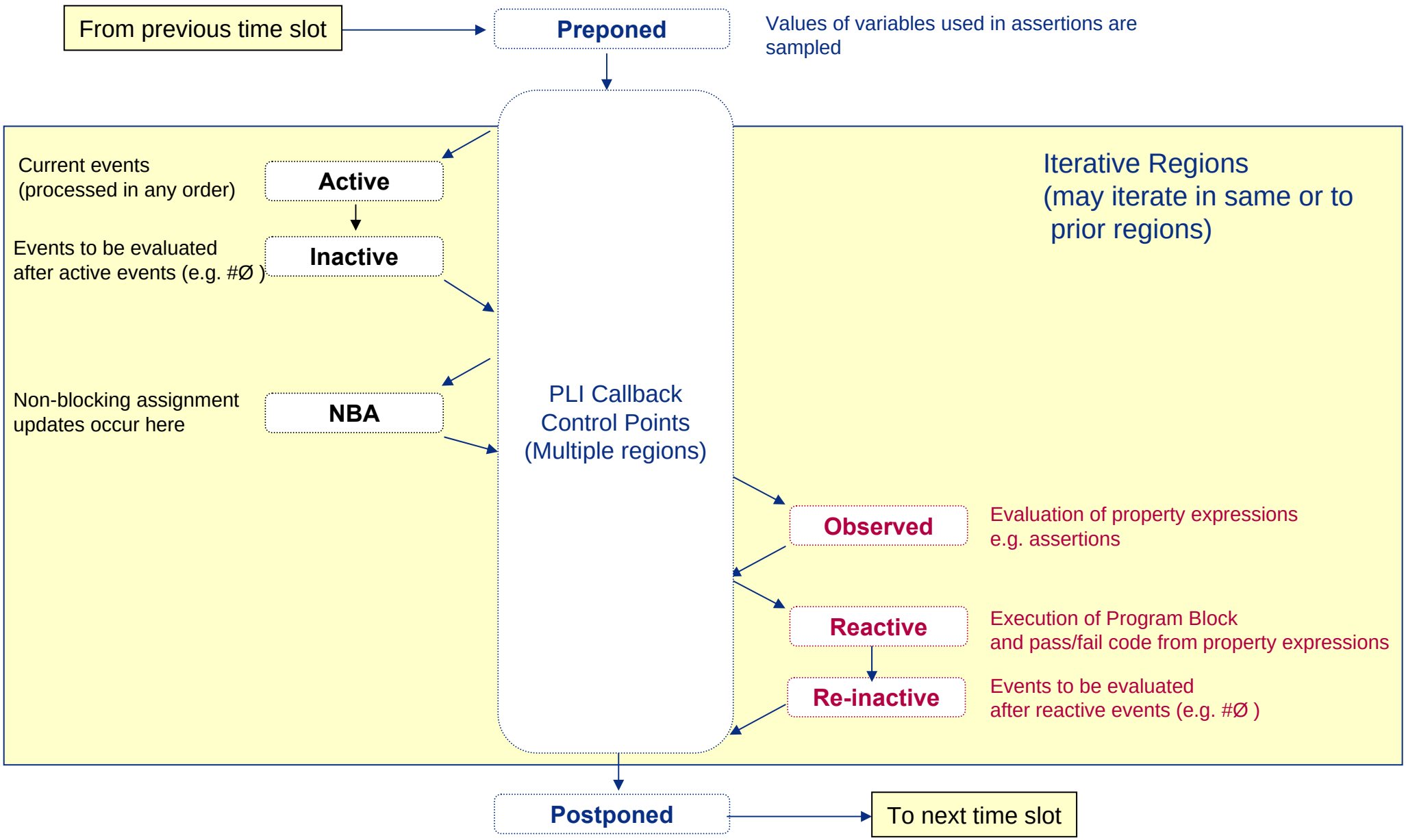
Events

- SV simulation is event based
  - **update event**
    - A change in a variable or net
  - **evaluation event**
    - The evaluation of a process
    - PLI callback
      - Points where PLI application routines can be called from the simulation kernel
  - Processes are sensitive to update events
    - When an update event occurs
      - All the processes that are sensitive to that event are considered for evaluation in an arbitrary order

Time

- Simulation time is a value maintained by the simulator
  - Models the actual time it would take for the system description being simulated
- All events scheduled at a particular time define a ***time slot***
- Each time slot is divided into multiple ***regions***
  - Events may be scheduled in these regions
  - Provides for an ordering of particular types of events
  - Allows for checkers and properties to sample data in a stable state
- Every event has only one simulation execution time
  - May be in the current time slot
  - May be in a future time slot
- Simulation proceeds by executing and removing the events in the current time slot
- Time advances
  - When all the events are executed and removed from the current time slot time
  - To the next non-empty time slot
- Simulation completes when no more events are scheduled

SV Time Slot Regions



Verilog 95 Regions — **PLI Regions** (Verilog 2001+) — **SystemVerilog Regions**

Program Control

In this section

SystemVerilog

- C Operators
- Loops
- disable / break / continue

Operators

- In addition to the standard Verilog operators, SystemVerilog adds C operators:
  - **Assignment**:
    - Blocking assignments only

```
+=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, <<<=, >>>=
```

  - **Bump**:
    - Blocking assignments only & NO RHS timing constructs

```
++a, --a, a++, a- -
```

  - **Power**:

```
**
```

do loop

Syntax:
```
do <statement> while <expression>
```

- Like a while loop, but evaluates after the loop executes rather than before

```
while (i < 10)
   begin
      …
      i = i +1;
   end
```

Loop would NEVER execute if i >= 10
So programmer has to setup value of i before loop

Guaranteed to execute AT LEAST once

```
do
   begin
      …
      i = i +1;
   end
while (i < 10) ;
```

for - Loop Variable

- Verilog 2001:

```
int j;
initial
   for ( j = 0; j <= 20; j = j+1)
      // j is global and could be 'accidentally'
      // modified elsewhere
     $display(j);
```

- SystemVerilog allows the loop variable to be declared within the loop:

```
initial
   for ( int j = 0; j <= 20; j++)   // j is local
      $display(j);
```

for – multi-assignments

- SystemVerilog allows multiple initialization or update assignments within the loop.
- Uses a simple comma-separated syntax.
- NOTE: All or none of the variables must be local

```
int k = 99;          // This declaration will be ignored by the following loop

initial
   for ( int j = 0, k = 10; j <= k; j++, k-- )  // j and k are local
       #10 $display("  j: %0d  k: %0d",j,k);

initial
  #40 $display("    Global k: %0d", k);
```

Simulator output

```
#  j: 0  k: 10
#  j: 1  k: 9
#  j: 2  k: 8
#    Global k: 99
#  j: 3  k: 7
#  j: 4  k: 6
#  j: 5  k: 5
```

foreach loop

Syntax:

```
foreach ( <array name>[<loop variables>] ) <statement>
```

- Iterates over all elements of an array
  - Array can be fixed-size, dynamic, or associative

```
int Ary[10];

foreach (Ary[i])
  $display("Ary[%d] = %d",i,Ary[i]);
```

i iterates from 0 to 9

foreach – examples

- Multiple loop variables correspond to nested loops
- If used with associative arrays, the type of the loop variable is auto-cast to the type of the array index

```
int mem[8][4];

foreach (mem[i,j])
  $display("mem[%d][%d] = %d",i,j,mem[i][j]);
```

j iterates first from 0 to 3, then i from 0 to 7 as if in a nested for loop

```
logic[7:0] binmem [4][2];

foreach (binmem[row,col,b])

  $display("Bit %d of binmem[%d][%d] is %z",b,row,col,binmem[row][col][b]);
```

b iterates first from 7 down to 0, then col from 0 to 1, then row from 0 to 3

```
int assoc[string];

foreach (assoc[s])
  $display("assoc[%s] = %d",s,assoc[s]);
```

s is cast as a string and iterates through all keys in the associative array

disable, break & continue

- The use of **disable** to terminate a loop is discouraged, mostly for style reasons ☺
- Instead, System Verilog adds C-like constructs: **break** and **continue**
  - These offer similar functionality but do NOT require an explicit block/task name.

```
reg[3:0] b, a;
…
for ( int i=0; i<52; i++ )
  begin
    a = 1;
    #5 case(b)
         1:  continue;
         0:  break;

       endcase
    a = 0;
  end
…
```

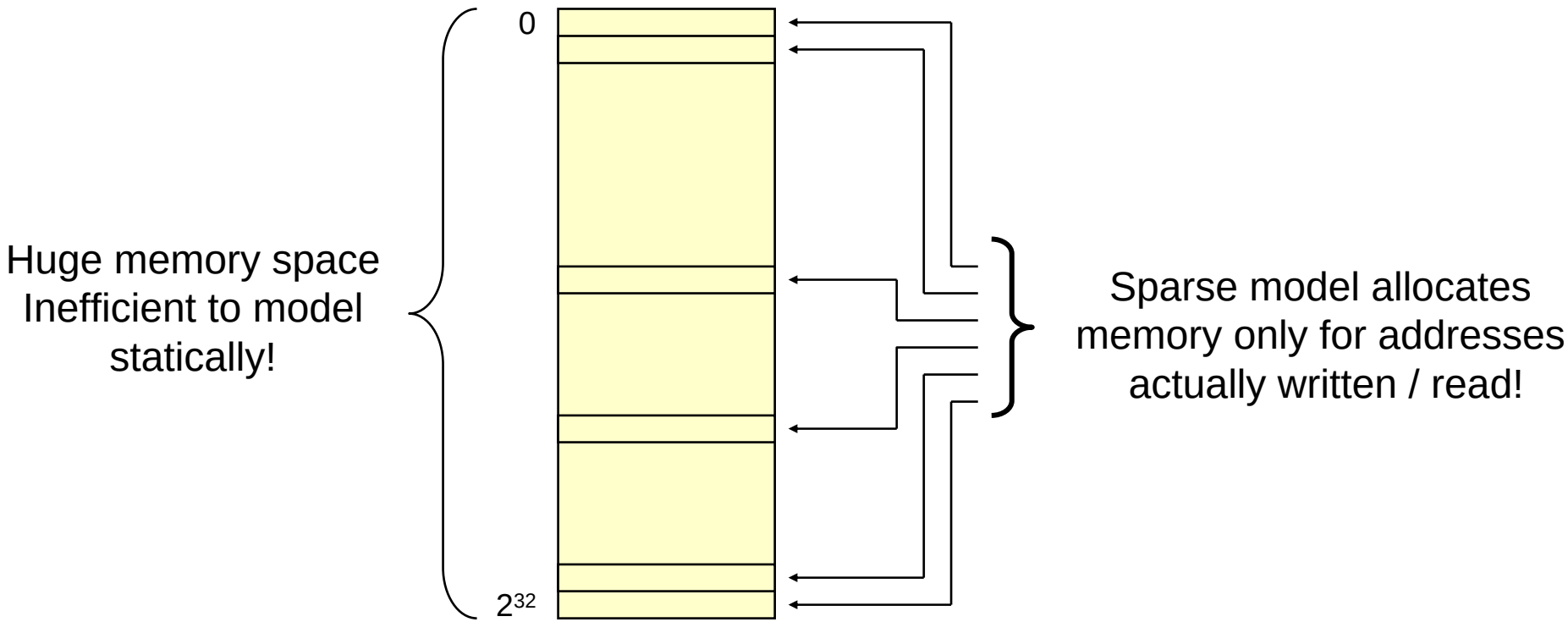
jump to next iteration of loop

exit the loop (as in C)

NOTE
break and **continue** may only be used inside a loop

Lab – Sparse Memory: Introduction

- How to model a memory with a large address space?
  - For example a 32 bit address
  - Impractical to allocate the entire memory
- One technique: describe a sparse memory model
  - Associative arrays are a way to implement a sparse memory model



Lab – Sparse Memory: Instructions

- Working directory: **arrays**
- Instructions
  - In file sparse_mem.sv edit the module sparse_mem
    - Create an enumerated type called "boolean" ("FALSE" and "TRUE“)
    - Create an associative array called "big_mem"
      - data type is boolean
      - index type is an unsigned 32 bit value
      - Default value of each entry should be false
    - Write to big_mem
      - Write a random number of entries (max 25), each at a random location
        - » Use **$random()** to generate random numbers
      - Write the value TRUE at these random locations
    - Display the following information about big_mem:
      - How many entries it has with the value TRUE
      - What is the smallest index with the value TRUE
      - What is the largest index with the value TRUE
      - The index value of all entries with the value TRUE
- Compile and run

Lab – Sparse Memory: Sample Output

```
# big_mem has 23 entries
# the smallest index is 15983361
# the largest index is 3883308750
# Here are the addresses:
#   15983361
#  112818957
#  114806029
#  512609597
#  992211318
# 1177417612
# 1189058957
# 1206705039
# 1924134885
# 1993627629
# 2033215986
# 2097015289
# 2223298057
# 2301810194
# 2302104082
# 2985317987
# 2999092325
# 3151131255
# 3230228097
# 3574846122
# 3807872197
# 3812041926
# 3883308750
```

Sol

Hierarchy

In this section

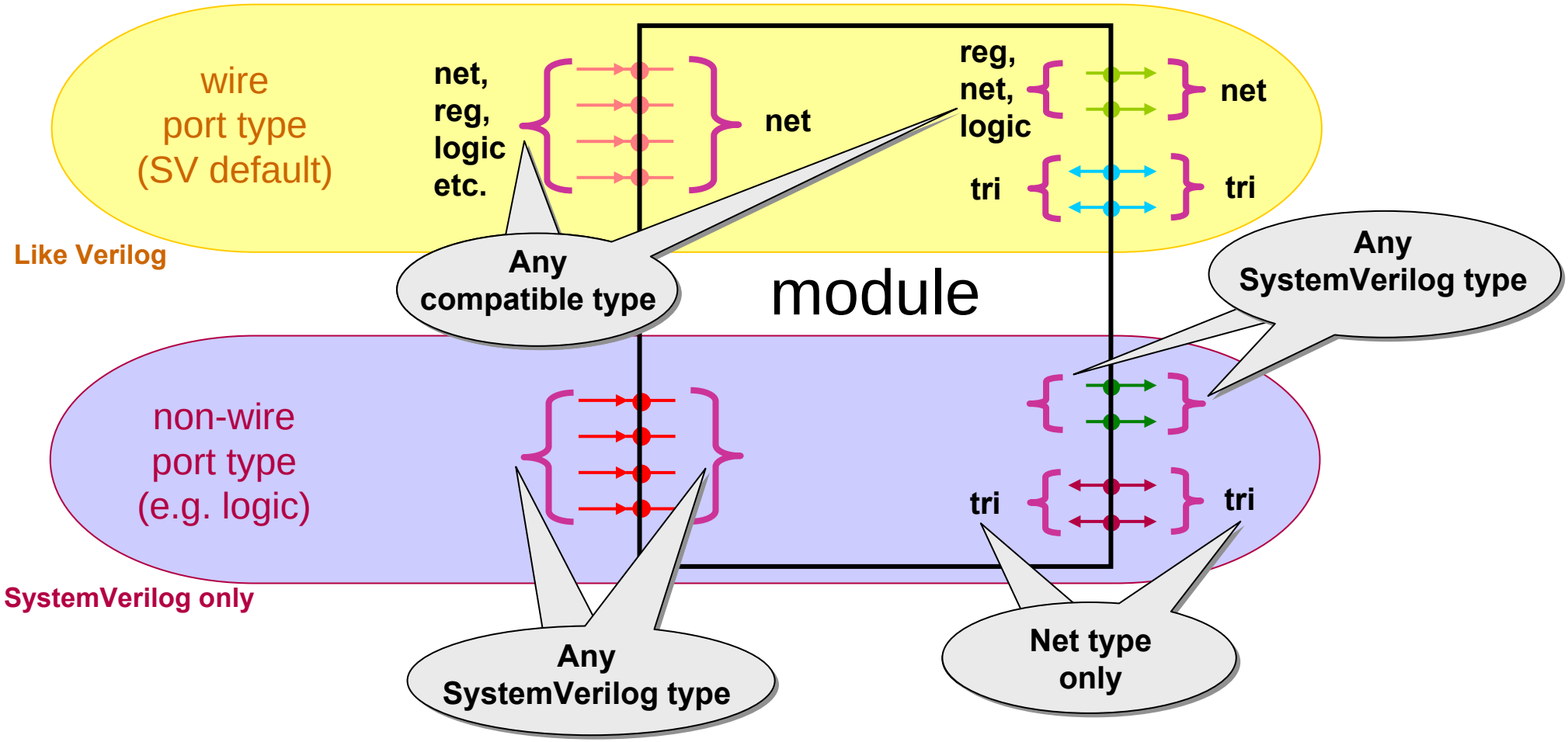
SystemVerilog

- Ports
- Driving SV variables
- Time specifiers
- Packages
- Compilation Units

Port Connection Rules

Synth

Since ports default to a net type (like Verilog) classic Verilog rules apply.
BUT for ports declared as non-net type (e.g. logic) things are different…



Module Ports

- SV port declarations can be made within the parentheses of the module declaration

```
module MUX2 (  output logic [1:0] out,
               input  logic [1:0] in_a, in_b,
               input        [1:0] sel  ) ;
```

Recommended

- Ports may be of ANY SystemVerilog type including events, structs, arrays, etc.

```
typedef struct {
    bit isfloat;
    struct { int i; shortreal f; } n;
} tagd;                                  // nested structure

module mh1 (input event e1, input int in1, output tagd out1);
...
endmodule
```

Driving an SV variable

- Verilog has only 1 resolved type – net
  - Multiple assignments to same net get resolved
  - Non-net types are unresolved so last assignment wins

- In SV, a variable (i.e. not a net) of any type may be driven by just **one** of the following:

Choose only one

- Arbitrary number of procedural assignments (like reg in Verilog)
- Single continuous assignment
- Single primitive/module output

Allows the simulator to spot a common Verilog problem of multiple drivers to a node

Absolute rule because none of these types are resolved

Hierarchy

- SystemVerilog adds several enhancements to design hierarchy:
  - `timeunit` and `timeprecision` specifications bound to modules
  - Simplified named port connections, using .name
  - Implicit port connections, using `.*`
  - Interfaces to bundle connections between modules

Module-Centric Time Specifiers

- In Verilog, the `timescale directive has always caused problems
  - e.g a module can accidentally 'inherit' the `timescale of a previously compiled module
- SystemVerilog resolves this with an alternative to `timescale :

```
module timespec;
timeunit 1ns;
timeprecision 0.01ns;

initial
   #5.19 $display("Current time is %f", $realtime);

endmodule
```

NOTE

`timeunit` & `timeprecision` are local to a module, they are not `directives that can affect modules compiled later in sequence.

Precedence if timeunit / timeprecision not specified in a module:

1. **If the module is nested, inherit from the enclosing module.**
2. **Use the time units of the last `timescale specified within the compilation unit.**
3. **Use the compilation unit's time units specified outside all other declarations.**
4. **Set to the compiler's default time units.**

Questa vsim command has switches for globally setting the timeunit and timeprecision

Implicit Port Connections

- SystemVerilog adds two new ways to reduce the chore of instantiation for the common situation where port and connected signal match in name and size
- Consider this 4:1 mux:

```
module mux (output logic [2:0] out,
            input logic [2:0] a, b, c, d, sel);
```

- Verilog 'by-name' instantiation (port d unconnected)

```
mux U1 (.out(out), .a(a), .b(b), .c(c), .d(), .sel(sel1));
```

- `.name` syntax instantiation

```
mux U2 ( .out, .a, .b, .c, .d(), .sel(sel1) );
```

- `.*` syntax instantiation

```
mux U3 ( .*, .d(), .sel(sel1));
```

Unconnected port using 'by-name' syntax

Implicit Port Connections – 2

- Both new SystemVerilog styles of instantiation require:
  1. Ports and their connecting variables must have same name and same width
  2. Ports and their connecting variables must be of compatible type
  3. Ports outside of the implicit list must be connected by port name (not order)
- So, which to use, `.name` or `.*` ?
  - `.name` allows implicit connections but shows port & signal names for documentation
  - `.*` allows full wildcarding where listing port & signal names is not required

SV and hierarchy

- **Packages**
  - A mechanism for sharing parameters, data, type, task, function, sequence and property declarations amongst modules, interfaces and programs.
- **Compilation units**
  - A collection of one or more SystemVerilog source files compiled together
- **Compilation-unit scope**
  - A scope local to the compilation unit, containing all declarations that lie outside of any other SV scope
- **$unit::**
  - A name used to explicitly access the identifiers in the compilation-unit scope

Packages -1

- Packages are explicitly named scopes declared at top of hierarchy
  - May contain types, variables, tasks, functions, sequences, and properties

```
package p;
   typedef enum { FALSE, TRUE } BOOL;
   const BOOL c = FALSE;
endpackage
```

```
package q;
   const int c = 0;
endpackage
```

Declaration assignments are performed ***before*** any always/initial/program blocks execute

- Package items may be referenced within modules, interfaces, etc. (even other packages)
  - Using fully resolved name (use the scope resolution operator "::")

```
module foo;
  wire a = q::c;
endmodule
```

:: is the scope resolution operator (q::c → c inside of q)

Packages -2

- Packages are explicitly named scopes declared at top of hierarchy
  - May contain types, variables, tasks, functions, sequences, and properties

```
package p;
   typedef enum { FALSE, TRUE } BOOL;
   const BOOL c = FALSE;
endpackage
```

```
package q;
   const int c = 0;
endpackage
```

- Instead of a fully resolved reference an entire package or a package item may be imported
- However, if the identifier of an imported item is already used in the importing scope, that item is silently NOT imported

***** is a wildcard, allowing ANY identifier in package to be referenced

```
module foo2;
  import q::*;
  wire a = c;        // no need to reference package q since c is "imported" already
//  import p::c;  // Import c from package p,  BUT compile error: c already exists

  import p::*;      // c silently NOT imported, no error/warning posted

  wire b = p::c; // No error because fully referenced
endmodule
```

Package example

```
package pkg_rmac;

    typedef struct {bit[55:0] preamble ; bit [7:0] sfd ;} h_type;

    typedef enum {
            _,
            Preamble,
            SFD,
            Destination_MAC,
            Source_MAC,
            Length,
            Packet_Num,
            Payload,
            CRC
    } fids;

    // Typedefs for stream_mp3_file task
    typedef struct {
        int n64;
        int n244;
        int n428;
        int n608;
        int n792;
        int n972;
        int n1156;
        int n1340;
        int n1500;
    } distro;

    typedef struct {
        int sendmp3;
        int noise;
    } frm;

    typedef struct {
        int normal;
        int long;
        int short;
        int random;
    } gap;

    typedef struct {
        frm frame;
        distro dis;
        gap ifg;
    } stream_set;

endpackage
```

```
`include "tb_eth_defines.v"
`include "eth_defines.v"
`include "timescale.v"

module tb_rmac(
    input wire [3:0]MTxD,      // NOT USED
    . . .

    //==============================
    //===  REFERENCE SV PACKAGE  ===
    //==============================
    import pkg_rmac::*;

    stream_set set; // used to configure…

    . . .
```

Question

What is the advantage of using a pkg instead of tb_eth_defines.v above?

Compilation Unit (CU)

Reference Page

- A CU is a collection of SV source files intended to compile together
- Its purpose is to eliminate problems that have plagued Verilog for years:
  - e.g. modules can “inherit” unexpected `define values simply by file compilation order
- The syntax to declare a CU is tool-specific
  - CU scope ($unit) is defined as local to the compilation unit, it contains all declarations that lie outside of any other scope

    Two use-models are supported:

    1) Each file is a separate CU (so declarations in each CU scope are accessible only within its corresponding file)
       - **LRM calls this the default use model!**
    2) All files on a given compilation command line make a single CU
       - Declarations within those files are accessible anywhere else within the implied CU)

CU Example -1

Reference Page

code.sv

```
typedef struct{ logic [9:0] addr;
                  logic[31:0] data;
                } global_pkt;
global_pkt unit_pkt;

task global_task (input global_pkt in1);
   $display($stime,,"addr: %h, data: %h",in1.addr, in1.data);
endtask

module code;
logic [7:0] x;

global_pkt y;

initial
  begin
   #20;    y.data = 2;   y.addr = 2;   global_task(y);
   #10;
   unit_pkt.data = 3;
   $unit::unit_pkt.addr = 3;
   global_task(unit_pkt);

  end

endmodule
```

NOTICE
$unit:: is redundant

<os> vlog code.sv
<os> vsim –c code

```
# Loading work.code_sv_unit
# Loading work.code
VSIM 1> run -all
#   20000000 addr: 002, data: 00000002
#   30000000 addr: 003, data: 00000003

VSIM 2>
```

According to the default use model in SV each file is its own Compilation Unit…

so the typedef and task declarations are within the implicit CU of module code.

CU Example -2

Reference Page

code.sv

```
typedef struct{ logic [9:0] addr;
                logic[31:0] data;
              } global_pkt;
global_pkt unit_pkt;

task global_task (input global_pkt in1);
   $display($stime,,"addr: %h, data: %h",in1.addr, in1.data);
endtask

module code;
logic [7:0] x;

global_pkt y;

initial
  begin
   #20;
   y.data = 2;
   y.addr = 2;
   global_task(y);

   #10;
   unit_pkt.data = 3;
   unit_pkt.addr = 3;
   global_task(unit_pkt);

  end

endmodule
```

<os> vlog code.sv test_1.sv ← ERROR!!

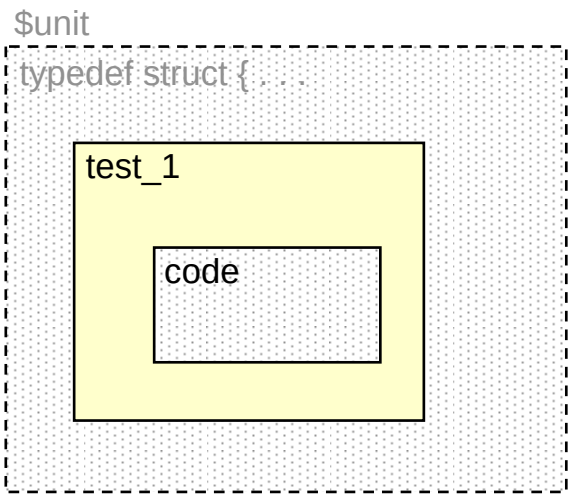
```
** Error: test_1.sv(10): Component name 'unit_pkt' does not refer to a scope.
** Error: test_1.sv(10): Component name 'unit_pkt' does not refer to a scope.
** Error: test_1.sv(10): Component name 'unit_pkt' does not refer to a scope.
** Error: test_1.sv(11): Component name 'unit_pkt' does not refer to a scope.
** Error: test_1.sv(11): Component name 'unit_pkt' does not refer to a scope.
** Error: test_1.sv(11): Component name 'unit_pkt' does not refer to a scope.
```

This compile fails because test_1 cannot see the declaration of global_task() since a local variable of the same name hides the external declaration of unit_pkt.

test_1.sv

```
module test_1;

logic unit_pkt;  // local variable hides global

code U1();

initial
  begin
   #10;
   unit_pkt.data = 1; // not compat. with local variable type
   unit_pkt.addr = 1;
   global_task(unit_pkt);
  end
endmodule
```

NOTICE



CU Example -3

Reference Page

code.sv

```
typedef struct{ logic [9:0] addr;
                logic[31:0] data;
              } global_pkt;
global_pkt unit_pkt;

task global_task (input global_pkt in1);
   $display($stime,,"addr: %h, data: %h",in1.addr, in1.data);
endtask

module code;
logic [7:0] x;

global_pkt y;

initial
  begin
   #20;
   y.data = 2;
   y.addr = 2;
   global_task(y);

   #10;
   unit_pkt.data = 3;
   unit_pkt.addr = 3;
   global_task(unit_pkt);

  end

endmodule
```

<os> vlog code.sv test_2.sv ← ERROR!!

```
** Error: test_2.sv(10): (vlog-2164) Class or package '$unit' not found.
** Error: test_2.sv(11): (vlog-2164) Class or package '$unit' not found.
** Error: test_2.sv(12): (vlog-2164) Class or package '$unit' not found.
```

This compile fails because the implicit $unit scope works only within the same file.

However, if we put both modules in a single file…

<os> vlog all_in_one.sv
<os> vsim –c test_2

```
# Loading work.code_sv_unit
# Loading work.test_2
run -all
#   10000000 addr: 001, data: 00000001
#   20000000 addr: 002, data: 00000002
#   30000000 addr: 003, data: 00000003
```

test_2.sv

```
module test_2;

logic unit_pkt;  // local variable hides global

code U1();

initial
  begin
   #10;
   $unit::unit_pkt.data = 1;    // $unit:: needed to "skip" local variable
   $unit::unit_pkt.addr = 1;
   global_task($unit::unit_pkt);
  end
endmodule
```

NOTICE
added **$unit::**

CU Example -4

Reference Page

code.sv

```
typedef struct{ logic [9:0] addr;
                  logic[31:0] data;
                } global_pkt;
global_pkt unit_pkt;

task global_task (input global_pkt in1);
   $display($stime,,"addr: %h, data: %h",in1.addr, in1.data);
endtask

module code;
logic [7:0] x;

global_pkt y;

initial
  begin
   #20;
   y.data = 2;
   y.addr = 2;
   global_task(y);

   #10;
   unit_pkt.data = 3;
   unit_pkt.addr = 3;
   global_task(unit_pkt)

  end

endmodule
```

Since putting multiple modules in a single file is not a workable approach… the QuestaSim™ compiler switch **–mfcu** defines a custom CU:

vlog –mfcu <file1.sv> <file2.sv>…<fileN.sv>

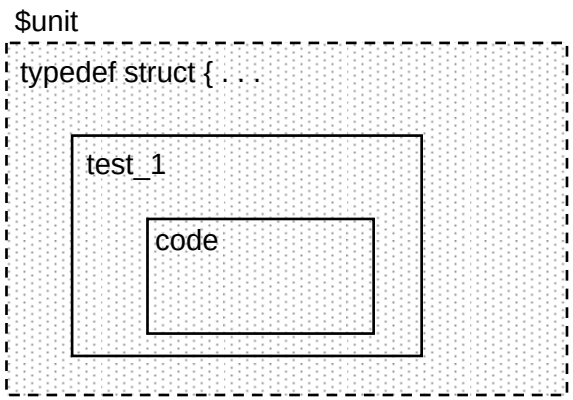
CU

<os> vlog –mfcu code.sv test_2.sv
<os> vsim –c test_2

test_2.sv

```
module test_2;

logic unit_pkt;  // local variable hides global

code U1();

initial
  begin
   #10;
   $unit::unit_pkt.data = 1;    // $unit:: needed here
   $unit::unit_pkt.addr = 1;
   global_task($unit::unit_pkt);
  end
endmodule
```

```
# Loading work.code_sv_unit
# Loading work.test_2
run -all
#   10000000 addr: 001, data: 00000001
#   20000000 addr: 002, data: 00000002
#   30000000 addr: 003, data: 00000003
```



Tasks & Functions

In this section

- Task enhancements
- Function enhancements
- Recursion
- Default arguments
- Explicit calls
- Pass by reference
- Data scope and lifetime

Static/Dynamic Variables

- Remember, SystemVerilog is 100% backward compatible with both Verilog 1995 and 2001.

- **Static** (Verilog 1995)
  - Memory-allocation / initialization once, at compile time
  - Exists for the entire simulation

- **Automatic** (Verilog 2001)
  - Stack based
  - Reallocated / initialized each time a block/task/function is entered
  - Supports recursion in blocks, tasks and functions

- May NOT be used to trigger an event
- May NOT be assigned by a non-blocking assignment
- May not be used to drive a port

Tasks & Functions

- Fully Verilog compatible
  - Default type is '`logic`' but all SV types are supported
- Extensions to address limitations in Verilog 95
  - System Verilog type support
  - Dynamic memory allocation (recursion allowed)
  - Default input values
  - Default argument values
  - Argument pass-by-reference

Tasks (new features)

- SystemVerilog makes a number of extensions to basic Verilog syntax.
  - ANSI-C style formal declarations (plus a new one: **ref**)
  - Arguments can be any SystemVerilog type (default is logic)
  - Default direction is input
  - **task-endtask** implies a **begin-end** structure
  - **return** statement can end the task call before **endtask**

```
task write_vector( input int  data,
                           logic[11:0] addr);
  @(negedge clk);
  if (bus_error) return; // cancel operation if bus_error true
  data_bus = data;
  addr_bus = addr;
  write = 1;
  @(posedge clk);
  #5 write = 0;
endtask
```

return** is supported in tasks but NOT **return(value)

Tasks (Recursion)

- SystemVerilog makes major extensions to basic Verilog syntax.
  - Supports **automatic** keyword (from Verilog 2001)
  - Unlike Verilog, all local variables are dynamically allocated at call time.
  - Full recursion is supported (automatic variables/arguments stored on stack)
    - Can do concurrent calls
    - Can do recursive calls

```
task automatic my_task(
    input int local_a,
    int local_b);

  if (local_a == local_b)              // detect where arguments are identical
    begin
      my_task(local_a-1,local_b+1);    // fix by inc/decrementing
      return;                          // end 'this' copy of task
    end
  global_a = local_a;                  // drive the outputs
  global_b = local_b;
endtask
```

Functions

- Extends the basic Verilog function syntax.
  - ANSI-C style formal declarations (plus new one: `ref`)
  - Arguments can be any SystemVerilog type
  - Return value can be a structure or union
  - Default direction is `input`, but also supports `output`
  - `Function-endfunction` implies a `begin-end` structure
  - `return` statement supported as well as assignment to function name
  - Supports `automatic` keyword, allowing recursion just like tasks
  - Supports return type of `void`

```
function automatic int factorial (int n);
    if (n==0) return (1);    // factorial 0 is 1
    else return (factorial(n-1)*n);
endfunction
```

Data Type: void

- The void datatype represents a non-existent value.
  - It is used in function declarations to indicate that they do NOT return a value

```
function void invert();        // Invert the image ( pos -> neg, neg -> pos )
  for(int ver=1; ver <= 150; ver++)
  for(int hor=1; hor <= 400; hor++)
    begin
      vidbuf[ver][hor].R = 255 - vidbuf[ver][hor].R;   // invert
      vidbuf[ver][hor].B = 255 - vidbuf[ver][hor].B;   // invert
      vidbuf[ver][hor].G = 255 - vidbuf[ver][hor].G;   // invert
    end
endfunction

initial
  invert();              // function called like a task
```

Task/Function – Default Arguments

- SystemVerilog allows specification of default argument values for subroutines
  - If an argument is passed to the task/function it overrides the default

```
task do_this( input int  data = 0, addr = 0,        // both default to 0
   logic[1:0] ctrl = 2'b10);                        // default to 2
…
endtask
```

- Notice Verilog placeholder syntax has a new significance for default arguments:

```
initial
  begin
     do_this(5,2,0);        // data(5), addr(2), ctrl(0)
     do_this(5,2);          // data(5), addr(2),   default: ctrl(2)
     do_this( , ,0);        // ctrl(0)   default: data(0), addr(0),
```

- SV also supports explicit call by argument name:

Recommended!

```
     do_this(.data(5),.addr(2) );      // data(5), addr(2), ctrl(2)
  end
```

Task/Function – pass by reference

- Tasks and functions can be passed an argument in 2 ways:
- Pass by value (Verilog compatible)
  - Copy each argument into the subroutine 'space'
  - If subroutine is automatic, then subroutine retains a local copy on stack
  - If arguments change within subroutine, this is invisible outside the subroutine
  - At end of subroutine, inout/output arguments are returned by value
  - Obviously inefficient for larger arguments
- Pass by reference (C++ style, *less confusing than C-style using pointers*)
  - Arguments passed by reference are not copied into the subroutine 'space'
  - Subroutine accesses the argument data via the reference
  - If arguments change within subroutine, original updates immediately
  - Reference is indicated by **`ref`** keyword

```
task/function <name> ( input/output/ref [type] <name>, … );
…
end{task/function}
```

Tasks also support inout

Pass by reference examples

Task/Functions with ref arguments must be automatic

Passing a large array by value is very inefficient so pass by `ref` instead

```
byte byte_array[1000:1];

function automatic int crc( ref byte packet [1000:1] );
  for( int j= 1; j <= 1000; j++ ) begin
    crc ^= packet[j];
  end
endfunction

initial
  int a = crc(byte_array);
```

const is used here to prevent modification of the reference/original and is legal for both tasks and functions

```
task automatic show ( const ref bit[7:0] data );
   for ( int j = 0; j < 8 ; j++ )
        $display( data[j] );  // data can be read but not written
endtask
```

Functions (direct calls)

Reference Page

```
module super_func;
int n;
parameter MAX = 10;

initial
  begin
      $display("The factorials from 1 to %d", MAX);
      for (n=0; n<=MAX; n=n+1)
        $display("%d! = %d", n, factorial(n));
      $display("And again, the factorials from 1 to %d", MAX);
      v_factorial(MAX);
  end

function automatic int factorial (int n);
  if (n==0) return (1); // factorial 0 is 1
  else return (factorial(n-1)*n);
endfunction

function automatic void v_factorial (int m);
  for (n=0; n<=m; n=n+1)
     $display("%d! = %d", n, factorial(n));
endfunction

endmodule
```

```
The factorials from 1 to        10
      0! =          1
      1! =          1
      2! =          2
      3! =          6
      4! =         24
      5! =        120
      6! =        720
      7! =       5040
      8! =      40320
      9! =     362880
     10! =    3628800
And again, the factorials from 1 to        10
      0! =          1
      1! =          1
      2! =          2
      3! =          6
      4! =         24
      5! =        120
      6! =        720
      7! =       5040
      8! =      40320
      9! =     362880
     10! =    3628800
```

Function calls itself recursively

Static vs Automatic

Reference Page

static storage allocated on instantiation and never de-allocated

automatic stack storage allocated on entry to a task, function or named block and de-allocated on exit.

```
module lifetime;

static int svar;
// automatic int avar;    // illegal outside of procedural block or a task/function

initial begin
    static int svar2;     // redundant
    automatic int avar2;  // also allowed inside static tasks/functions
  end

task automatic autotask;
  automatic int avar3;    // redundant!  automatic by default
  static int svar3;       // static across simultaneous calls to task
endtask

initial $monitor (svar); // svar2, svar3 & avar2, avar3 are not visible
endmodule
```

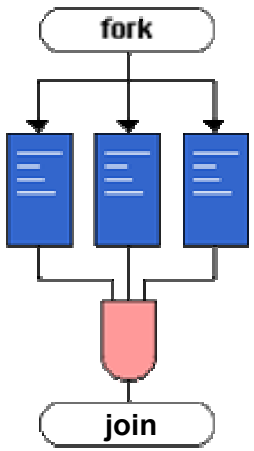
Dynamic Processes

In this section

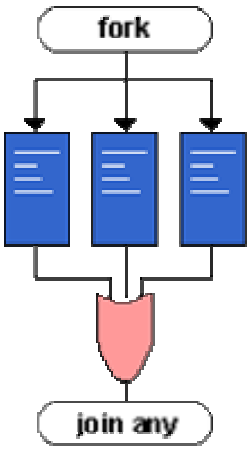
Dynamic processes
Process control

Dynamic Processes

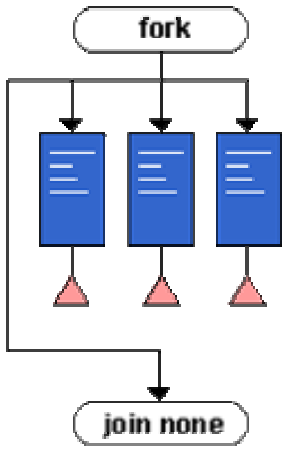
- SystemVerilog defines 2 new special cases of fork…join with associated keywords **join_any** & **join_none**




```
fork
   …
   …
   …
join      // all blocks finished
```




```
fork
   …
   …
   …
join_any  // any block finished
```

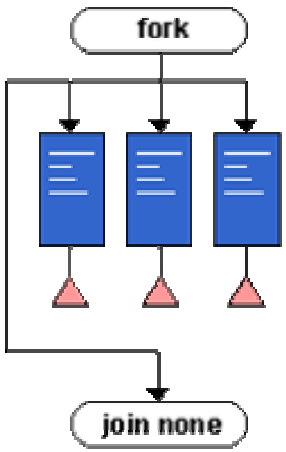



```
fork
   …
   …
   …
join_none  // no waiting at all
```

fork…join_none

- SystemVerilog replaces process by **join_none**
  - This allows any number of processes to be spawned simultaneously without any impact on the flow of the main process

```
task run_test;

   fork
    timeout( 1000 );
    apply_stimulus();
    verify_response();
   join_none

   @(sig);        // blocking statement, allows child processes to start

endtask
```



NOTE
The child processes spawned by a fork…join_none do not start executing until the parent process hits a blocking statement

Process Control – `wait fork`

- With Dynamic processes SystemVerilog needed to provide more global detection that spawned processes have completed
- The `wait fork` statement is used to ensure that all child processes (spawned by the process where it is called) have completed execution

```
begin
  fork
    task1();
    task2();
  join_any        // continue when either task completes

  fork
    task3();
    task4();
  join_none       // continue regardless

  wait fork;      // block until tasks 1-4 complete
end
```

Process Control – `disable fork`

- The `disable fork` statement terminates ***all*** active child processes of the process where it is called,
  - Termination is recursive, in other words it terminates child processes, grandchild processes, etc.

```
task test_with_timeout;
  fork    // spawn off two child tasks in parallel, 1st to finish triggers off the join_any
    run_test();
    timeout( 1000 );
  join_any
  disable fork;    // kills the slower task
endtask
```

```
task test_with_timeout;    // Verilog 1995
  fork
    begin
      run_test();
      disable timeout;
    end
    begin
      timeout( 1000 );
      disable run_test;
    end
  join
endtask
```

At first glance, this code may appear to do the same thing as the disable fork example above.

However, what if the timeout task was a global one, used in many places? The disable timeout line would terminate this occurrence of timeout but ALSO any other occurrences that happen to be executing elsewhere in the system.

disable fork terminates only copies of the timeout task spawned by the current block

Gotcha! – disable fork

- Remember, **disable fork** **recursively** terminates ***all*** active child processes of the process where it is called,

```
module disable_fork;

task semicolon;
  forever #10 $write(";");
endtask

task gosub;
  fork
    semicolon;
  join_none
  #100;
  disable fork;
  $write("DISABLE");
  #100 $display("");
  $stop;
endtask

initial : comma
  fork
    forever #10 $write(",");
  join_none


initial
  begin
    fork : full_stop
      forever #10 $write(".");
    join_none
    gosub;
  end

endmodule
```

Simulation output

```
,;.,;.,;.,;.,;.,;.,;.,;.DISABLE,,,,,,,,,,
```

Interprocess Synch & Communication

In this section

SystemVerilog

- Mailboxes
- Semaphores

Interprocess Synch. & Communication

- Dynamic processes and OOP coding styles require more sophistication than Verilog provides
- SystemVerilog introduces:
  - **Semaphores**
    - Synchronization/arbitration for shared resources (keys)
    - Mutex control
    - Methods: `new()`, `get()`, `put()`, `try_get()`
  - **Mailboxes**
    - FIFO queuing mechanism between threads (bounded/unbounded)
    - Default type is singular (packed)
    - Methods: `new()`, `num()`, `put()`, `get()`, `peek()`, `try_put()`, `try_get()`, `try_peek()`

Semaphore

semaphore S1 = new([# of keys]); // default is 0 keys

- new() function Prototype

  function new(int `keyCount = 0`);)

  - **`keyCount`** is the initial number of keys
  - **`keyCount`** may increase beyond its initial value

- put() task Prototype e.g. S1.put(3);

  ```
  function void put( int keyCount = 1);
  ```

  - **`keyCount`** = is the number of keys returned to the semaphore (default 1)

- get() task Prototype e.g. S1.get();

  ```
  task get( int keyCount = 1);
  ```

  - **`keyCount`** = is the number of keys to obtain from the semaphore (default 1)
  - Process <u>blocks</u> on a FIFO basis if keyCount keys are not available

- try_get() function Prototype e.g. S1.try_get(3);

  ```
  function int try_get( int keyCount = 1);
  ```

  - **`keyCount`** = is the number of keys to obtain from the semaphore (default 1)
  - Process returns 0 if keyCount keys are not available

Semaphore Example

```
module semaphores;
  semaphore s1 = new(1);

  task t1();
    for(int i = 0; i<3; i++) begin
      s1.get(1);
      #5;
      $display("t1 has semaphore");
      s1.put(1);
      #5;
    end
  endtask

  task t2();
    for(int i = 0; i<3; i++) begin
      s1.get(1);
      #5;
      $display("t2 has semaphore");
      s1.put(1);
      #5;
    end
  endtask
```

```
initial begin
    fork
      t1();
      t2();
    join_none
    #0;
  end
endmodule
```

```
Output:
# t1 has ownership of semaphore
# t2 has ownership of semaphore
# t1 has ownership of semaphore
# t2 has ownership of semaphore
# t1 has ownership of semaphore
# t2 has ownership of semaphore
```

Mailbox

mailbox [#(type)] MB1 = new([bound]); // Mailboxes default to singular (packed) type
// **bound** is mailbox depth, default unbounded

- **new() Prototype** e.g. **mailbox #(Packet) channel = new(5);**

```
function new( int bounded = 0);
```

- **num() Prototype** e.g. **some_int_variable = MB1.num();**

```
function int num();          // returns # of messages currently in mailbox
```

- **put() Prototype** e.g. **MB1.put(sent_message);**

```
task put( message);
```

  - **Store message in mailbox in FIFO order, <u>block</u> if mailbox full (bounded mailboxes only)**
- **get() Prototype** e.g. **MB1.get(rcvd_message);**

```
task get( ref message);
```

  - **Remove message from mailbox in FIFO order, <u>block</u> if mailbox empty**
- **peek() Prototype**

```
task peek( ref message);
```

  - **Copy message from mailbox in FIFO order (don't remove), <u>block</u> if mailbox empty**

- **try_put(), try_get(), try_peek() Prototype**

```
function int try_put( message);
function int try_get( ref message);
function int try_peek( ref message);
```

  - **<u>Non-blocking</u> forms, return 0 if mailbox full/empty**

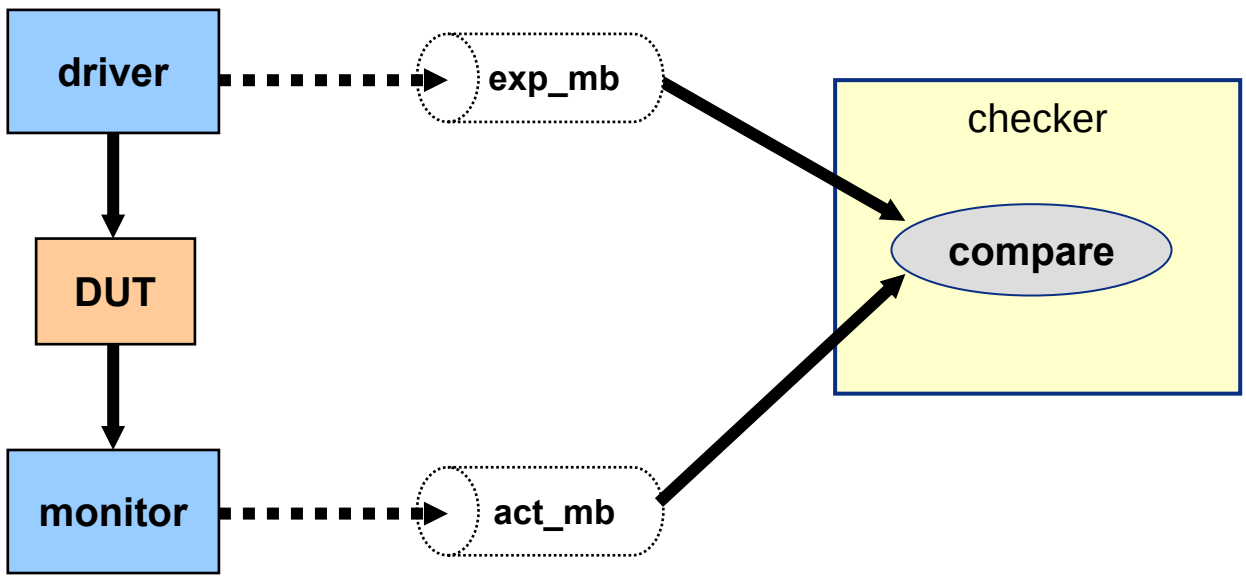
Mailbox Example

```
module mailboxes;
  mailbox #(int) m = new();  // create mailbox

  task t1();
    for(int i = 0; i<4; i++) begin
     m.put(i);
     $display("T1 sent: %0d",i);
     #5;
    end
  endtask

  task t2();
    int temp;
    while(1) begin
     m.get(temp);
     $display("T2 received: %0d",temp);
    end
  endtask

initial begin
    fork
      t1();
      t2();
    join_none
    #0;
  end
endmodule
```

```
Output:
# T1 sent: 0
# T2 received: 0
# T1 sent: 1
# T2 received: 1
# T1 sent: 2
# T2 received: 2
# T1 sent: 3
# T2 received: 3
```

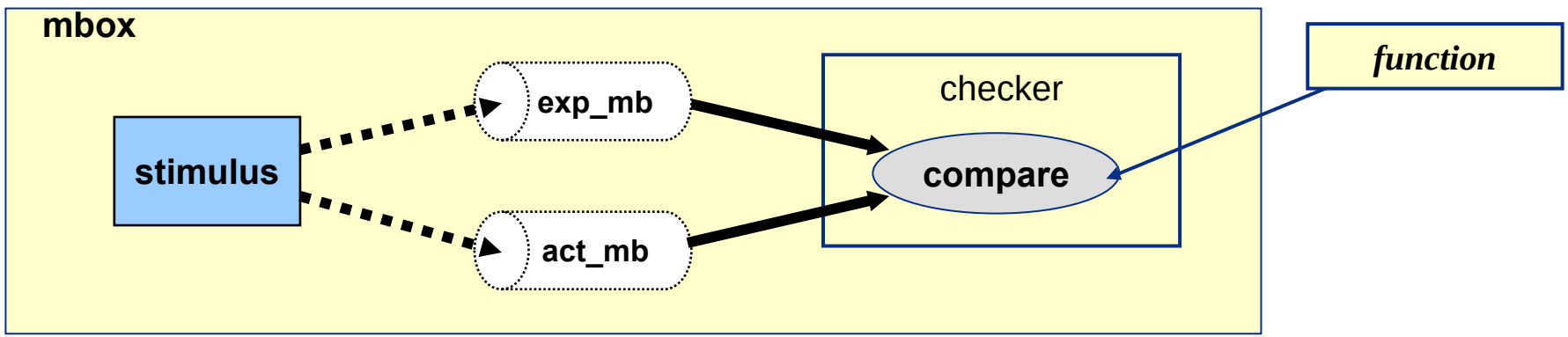
Lab – mboxes: Introduction

- Simple HDL testbenches can combine stimulus generation and response checking in a single code block. More complex verification environments tend to split these functions (and others) among separate autonomous blocks.
- This requires a clean simple mechanism for blocks to communicate together with minimal overhead and no lost messages. Mailboxes are ideal for this.
- For example: driver and monitor can report the packets they exchange with the DUT to an external checker via mailboxes…



Lab – mboxes: Instructions - 1

- Working directory: **mboxes**
- Instructions
  - In file `types.sv` declare a package called `types`
    - This package contains a new type called `packet` which is a struct containing a single field (`int pid;`)
  - Edit the file `mbox.sv` per the diagram below:
    - Declare 2 mailboxes (exp_mb and act_mb) and initialize them.
    - Edit the `stimulus` task where indicated
      - Write packet stim_pkt to both mailboxes
    - Add a `compare` function with two arguments of type `packet`
      - Compares the packets received (`pid` fields should match)
      - Returns a 1 if packets match, 0 if not



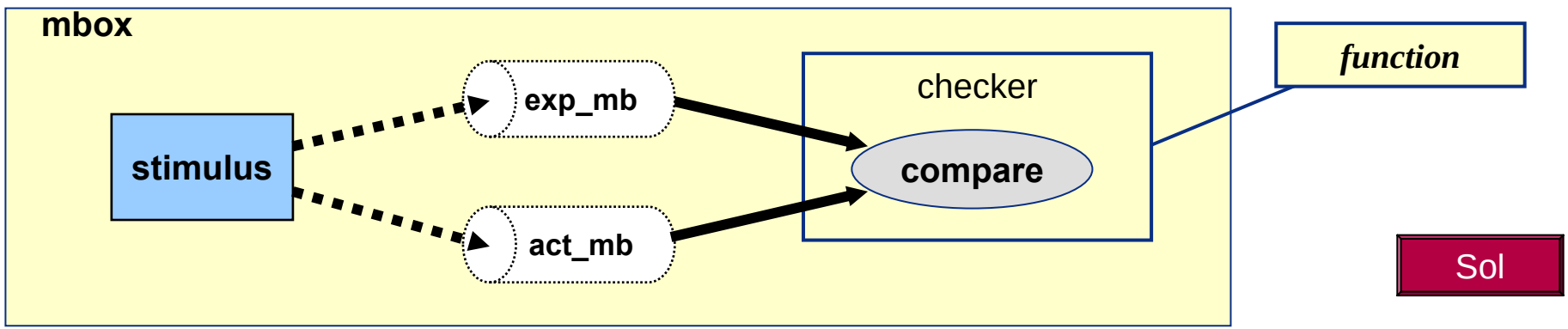
Lab – mboxes: Instructions - 2

- Continue to edit the file **`mbox.sv`**
  - Create the **`checker`** task
    - Reads packets from the 2 mailboxes (exp_mb and act_mb)
    - Calls compare function to see that packets match
    - Reports any errors and ends simulation when 256 pairs of packets have been compared
  - Start stimulus & checker tasks in a fork/join_none thread

- Compile and run - you should get no comparison errors!

- **EXTRA CREDIT**: Verify your code by deliberately inserting an error in one packet

```
vlog types.sv mbox.sv
vsim mbox
```



Sol

Classes

In this section

- OOP concepts
- Inheritance
- Virtual Methods
- Reference
  - Encapsulation
  - Parameterization
  - Polymorphism

SV Classes - Overview

- Object Oriented design is a common programming paradigm
  - Data and the means to manipulate is described together in a formal structure called a class
- A **`class`** is a datatype, similar to a **`struct`**
  - Has data elements (called **`properties`**)
  - But also contains functions and tasks (called **`methods`**) through which class properties may be manipulated
- An instance of a **`class`** is referred to as an **`object`**
- SystemVerilog objects are dynamically created and destroyed
  - Memory allocation and deallocation (garbage collection) is handled automatically
  - Since pointers are a key ingredient in the flexibility of classes, SystemVerilog implements them too, but in a safer form, called **`handles`**
- Code minimization and reuse is facilitated through inheritance, parameterization and polymorphism

Classes

- **Class**
  - Formal description
  - Members
    - Properties (data elements)
    - Methods (functions and tasks)
  - Constructor (**new()**)
- **Object**
  - Instance of a class (e.g. **pkt**)
- **Properties & Methods**
  - Accessed using "." operator

```
$display(pkt.command);
pkt.clean();
```

```
class Packet ;
  //properties
  cmd_type command;
  int unsigned address;
  status_type status;

  // methods
  task clean();
    command = IDLE; address = 0;
  endtask
endclass

Packet pkt = new();
```

Class Constructors - `new()`

- Special member function called when an instance of a class is created
  - Name of function is `new()` (reserved keyword)
    - No return type specified in declaration
    - May have arguments - Allows for run-time customization
  - If no constructor is specified, compiler will create one
  - Only one constructor for each class
- Used for initialization of the instance

```
class Packet ;
  //properties
  cmd_type command;
  int unsigned address;
  status_type status;

  // initialization
  function new(int addr);
    command = READ;
    address = addr;
    status = OK;
  endfunction
endclass
```

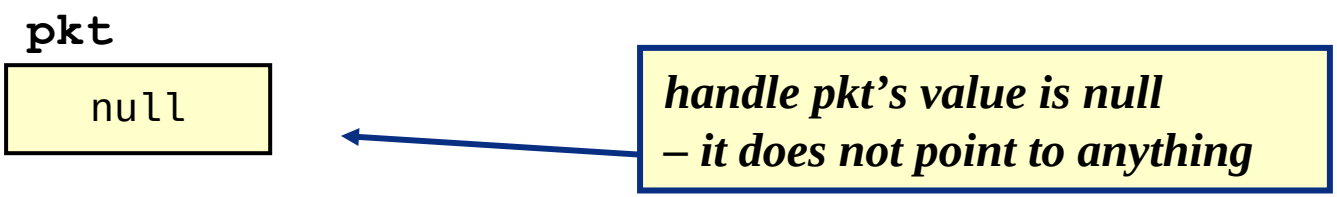
user-provided Constructor

Constructor args (optional), allow for run-time customization

Class Object Creation (Instantiation) - 1

- To create a class object
  - **First** declare an object `handle` of the class type
    - Like a pointer

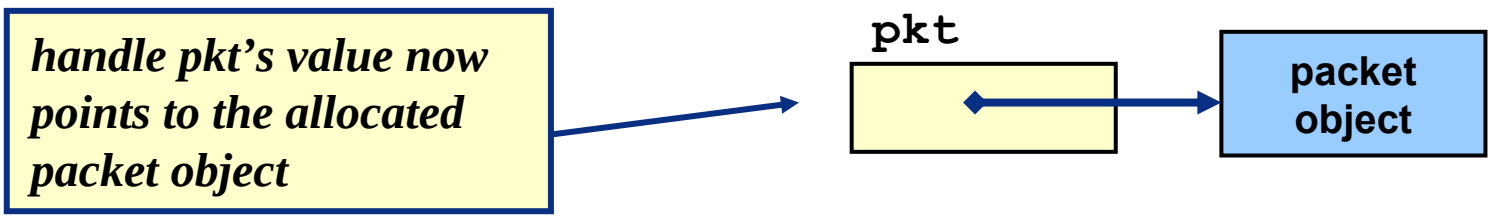
```
module test;
  Packet pkt;   // declare object handle, does not create object itself, its value is null
initial
  …
endmodule
```



Class Object Creation (Instantiation) - 2

- **Second** call the function **new()** and assign the return to the object handle
  - Dynamically allocates (creates) the object by calling the constructor for the class
  - Type of left hand side determines return type of **new()**

```
module test;
   Packet pkt = new(0);          // declare object handle and initialize it
  …
endmodule
```

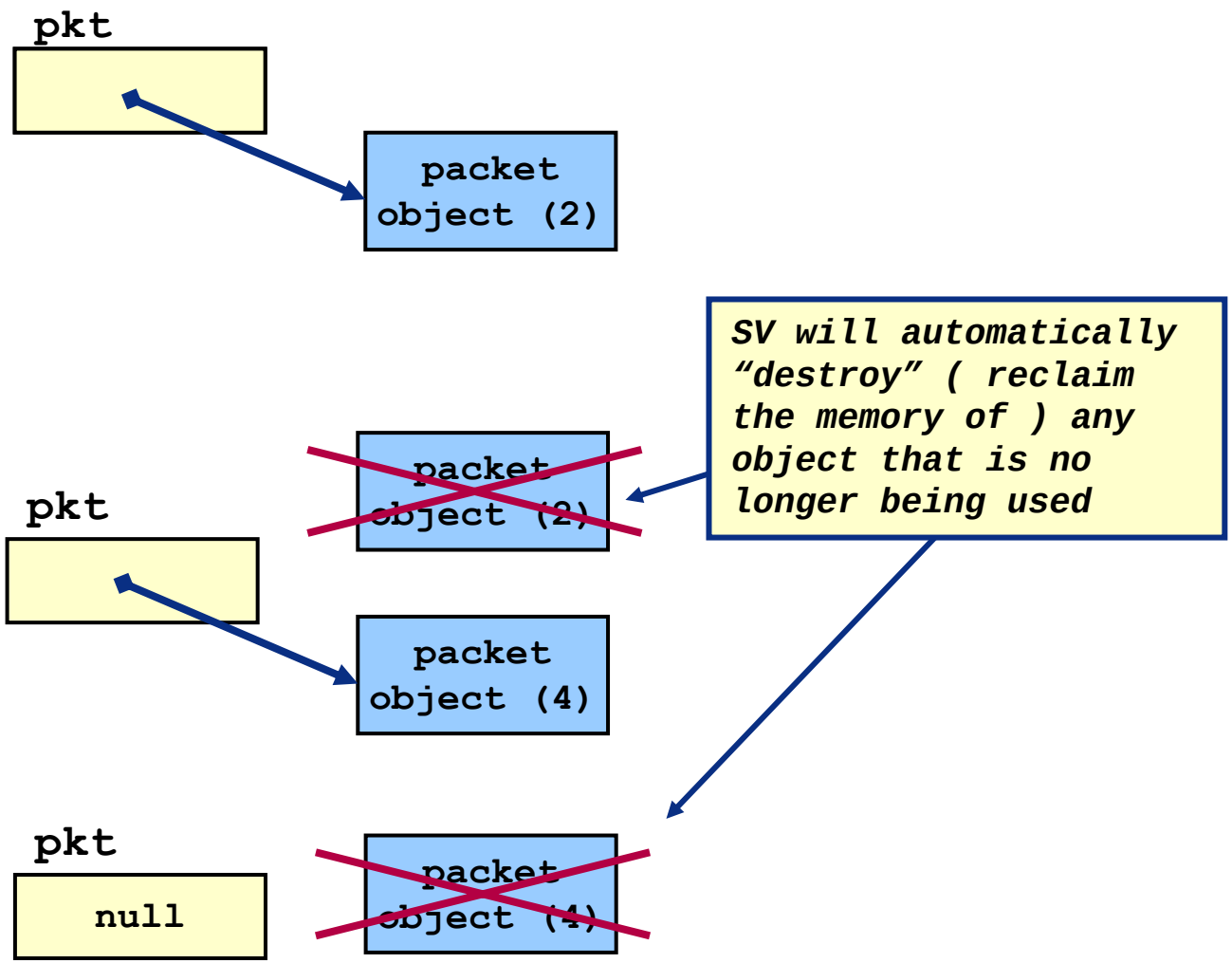


- **new()** may also be called procedurally… a very powerful concept!

```
module test;
   Packet pkt;          // declare object handle, does not create object itself, just the handle

initial
   pkt = new(0);        // create object and assign to the handle pkt (pkt "points" to object created)
endmodule
```

Re-Assigning Handles to New Objects

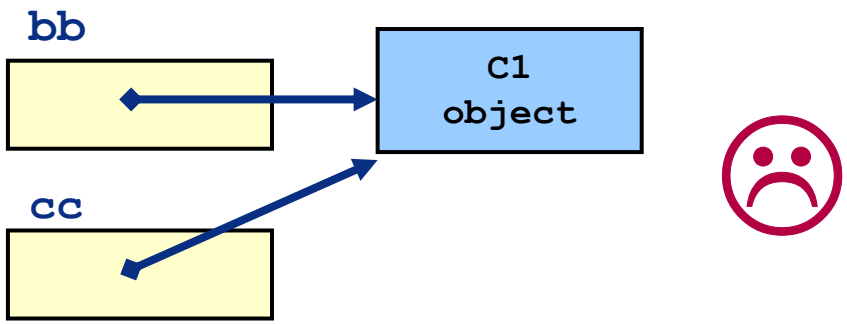
```
module test;
  Packet pkt;

initial begin
  pkt = new(2);

  ...
  pkt = new(4);

  ...
  pkt = null;
  end
endmodule
```



Copying Objects

- OK, we can create/destroy objects… How about duplicating one?

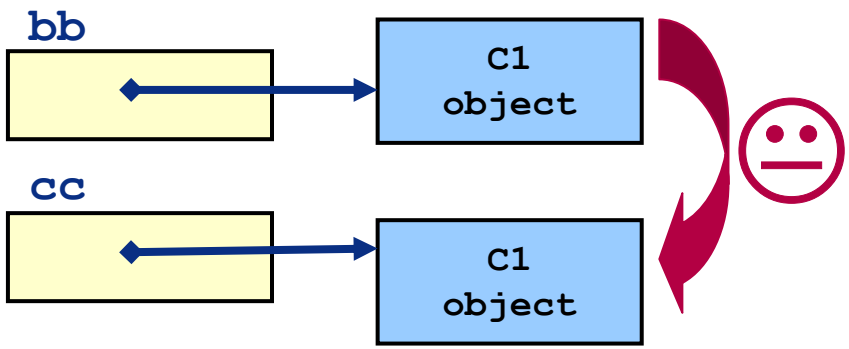
```
class C1 ;
  Packet pp;
endclass

C1 bb;
bb = new();
C1 cc;
cc = bb;
```



Two handles pointing to the same object

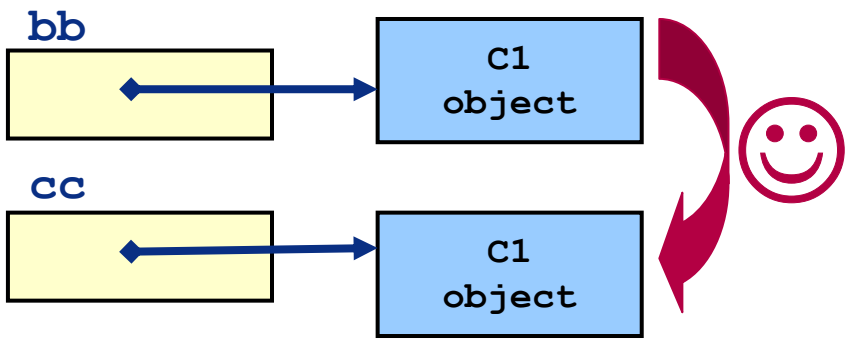
```
C1 bb, cc;
bb = new();
cc = new bb;
```

NOTE SYNTAX
No ( )'s allowed



Shallow Copy: Any nested objects will NOT be duplicated , only the members themselves (handle pp will be copied but not the object it points to)

```
C1 bb, cc;
bb = new();
cc = new();
cc.copy( bb );
```

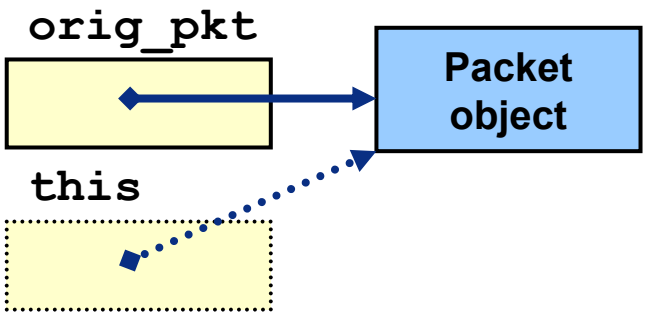


Had to write a custom method "copy"

this

- A predefined object handle that refers to the current object
  - Provides unambiguous reference to the object

```
class Packet;
 integer status;

 virtual function Packet clone();
   Packet temp = new this;          // create new Packet object
   return(temp);                    // return cloned object
 endfunction

endclass

Packet orig_pkt, cloned_pkt;

initial begin
  Packet orig_pkt = new();
  orig_pkt.status = 55;
  cloned_pkt = orig_pkt.clone();
  $display("cloned_pkt.status = %0d", cloned_pkt.status);
end
```



Simulation output

```
# cloned_pkt.status = 55
```

Class example

```
class ether_packet;

// Ethernet Packet Fields
  bit[55:0] preamble = 'h55555555555555;
  bit [7:0] sfd = 'hab;
  bit[47:0] dest, src;
  bit[15:0] len;
  bit [7:0] payld [ ];

  function new(int i);
    payld = new[i];  len = i;
  endfunction : new

  task load_frame( input [47:0] lsrc, ldest,
                   input  [7:0] start_dat);
    src = lsrc;  dest = ldest;
    len = payld.size();
    if(start_dat > 0)
       for(int i = 0; i < len; i++)
           payld[i] = start_dat +i;
  endtask : load_frame

  function void print;
    $displayh("\t      src: ", src);
    $displayh("\t     dest: ", dest);
    $displayh("\t      len: ", payld.size);
    for(int i = 0; i < len; i++)
       $displayh("\t payld[%0h]: %0h",i,payld[i]);
    $displayh("");
  endfunction : print

endclass : ether_packet
```

Ethernet Packet

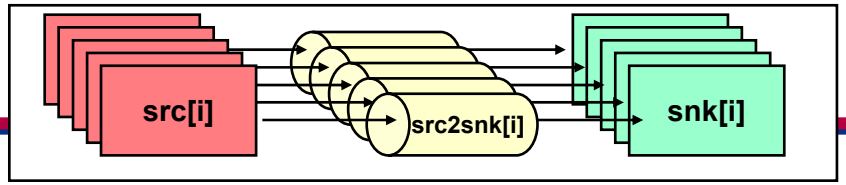
preamble
sfd
destination
source
length
payload

```
module test_ether();

ether_packet ep ;

initial
  begin
    ep = new (4);
    ep.load_frame('h55,'h66,'h77);
    ep.print();

//    . . .

    ep = new(1);
    ep.load_frame('h22,'h33,'h44);
    ep.print();

//    . . .

    ep = null;
  end

endmodule
```

Simulation output

```
     src: 000000000055
    dest: 000000000066
     len: 00000004
payld[0]: 77
payld[1]: 78
payld[2]: 79
payld[3]: 7a

     src: 000000000022
    dest: 000000000033
     len: 00000001
payld[0]: 44
```

Class example #2 [part1]



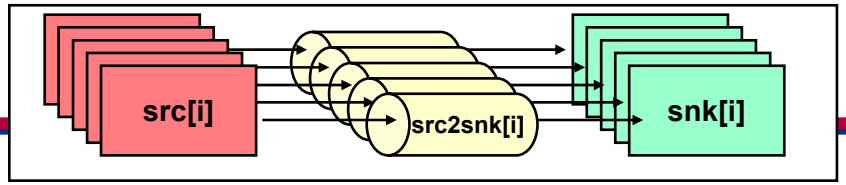

```
module array_handles;

class Packet;
 int field1;
 function new(int i);
   field1 = i;
 endfunction
endclass : Packet

                    class sink;
                      mailbox #(Packet) in_chan;
                      Packet stim_pkt;
                      int id;

                      function new(int i);
                        id = i;
                      endfunction

                      task run();
                        while(1) begin
                          in_chan.get(stim_pkt);
                          $display("sink[%0d]:   Received packet with field1 = (%0d)",
                              id, stim_pkt.field1);
                        end
                      endtask

                    endclass : sink
```

Notice:

- **3 classes, Packet, sink and source (on next slide)**
- **Each source & sink object has a local mailbox handle.**

  Those handles are "null" Why?

Class example #2 [part 2]




```
class source;
  mailbox #(Packet) out_chan; // null handle
  Packet pkt_to_send;
  int id;

  function new(int i);
    id = i;
  endfunction

 task run();
    for(int i = 0; i <= id; i++) begin
      pkt_to_send = new(i);
      out_chan.put(pkt_to_send);

    end
  endtask
endclass : source

sink        snk[];
source     src[];
mailbox #(Packet)   src2snk[];
. . .
endmodule
```

Dynamic Arrays of handles

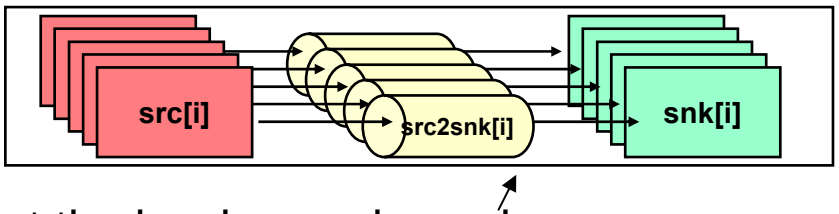
Notice:

- **3 dynamic arrays:**
  src, snk and src2snk
- **Dynamic arrays, once initialized can hold multiple handles of each object type**
- **So, put all this together and…..**

We have a Lab exercise!

Lab – Classes & Mailboxes: Instructions - 1

- Lab directory: **class**
- Purpose: Learn how different Verilog can be when using classes



- Instructions:
  - Using the code on the previous slide, implement the hardware shown here
  - Complete the supplied module array_handles
  - Edit the file array_handles.sv:
    - Add an initial block that implements the following:
      - Initialize each of the 3 dynamic arrays to a size of 5 elements (5x snk, 5x src & 5x src2snk)
      - Create 5 objects to fill each array (snk[4]-snk[0], src[4]-src[0], src2snk[4]-src2snk[0])

        NOTE: Initialize the snk and src object id's with their corresponding index

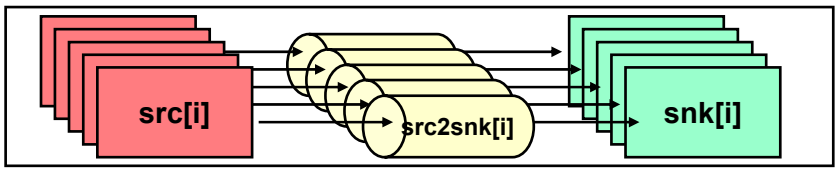
        HINT: Simple for-loops will be handy here

- Continued on next page -

Lab – Classes & Mailboxes: Instructions - 2

- Lab directory: **class**
  - Next, still within an initial block in the file array_handles.sv :
    - In procedural code, dynamically connect the 5 data pipelines shown by copying handles as necessary (i.e. map src to snk thru mailboxes):

src[0] -> src2snk[0] -> snk[0]
src[1] -> src2snk[1] -> snk[1]
src[2] -> src2snk[2] -> snk[2]
src[3] -> src2snk[3] -> snk[3]
src[4] -> src2snk[4] -> snk[4]



EXTRA CREDIT: Do this step by adding an argument to the constructor of sink/source classes

- Call the **run()** method for all src and snk objects

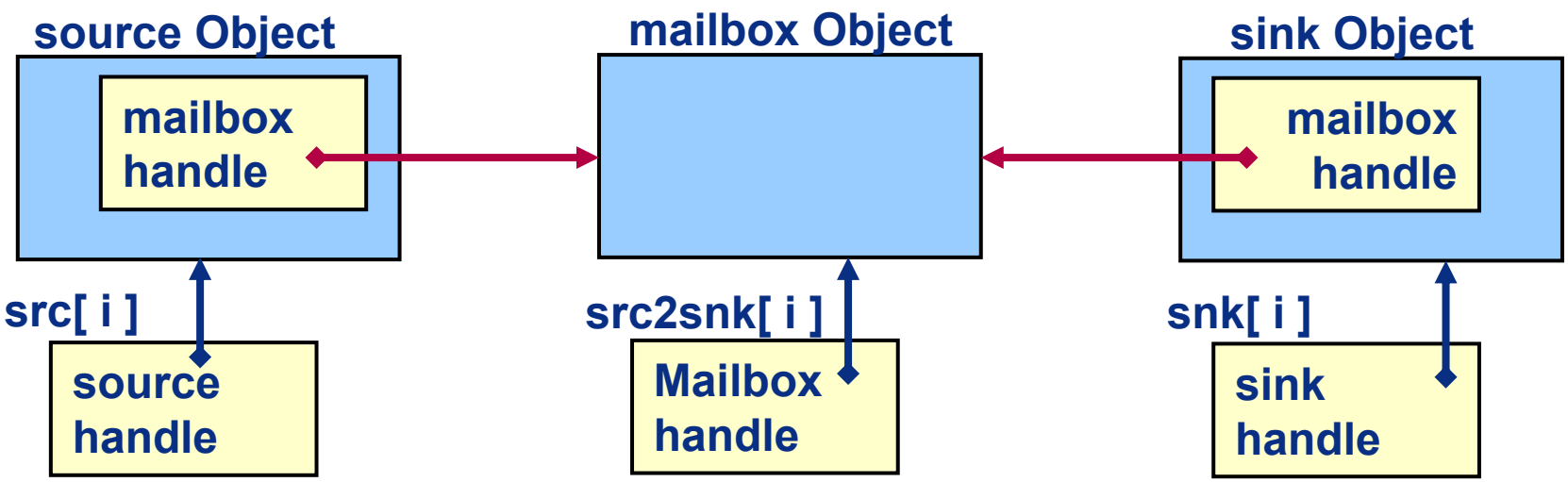
HINT: Start corresponding objects simultaneously
src[0] & snk[0], src[4] & snk[4] etc.

- Compile & simulate

Expected Simulator output

```
# source[0]: Sent packet with field1 = (0)
# sink[0]:   Received packet with field1 = (0)
# source[1]: Sent packet with field1 = (0)
# source[1]: Sent packet with field1 = (1)
# sink[1]:   Received packet with field1 = (0)
# sink[1]:   Received packet with field1 = (1)
# source[2]: Sent packet with field1 = (0)
# source[2]: Sent packet with field1 = (1)
# source[2]: Sent packet with field1 = (2)
# sink[2]:   Received packet with field1 = (0)
# sink[2]:   Received packet with field1 = (1)
# sink[2]:   Received packet with field1 = (2)
# source[3]: Sent packet with field1 = (0)
# source[3]: Sent packet with field1 = (1)
# source[3]: Sent packet with field1 = (2)
# source[3]: Sent packet with field1 = (3)
…
```

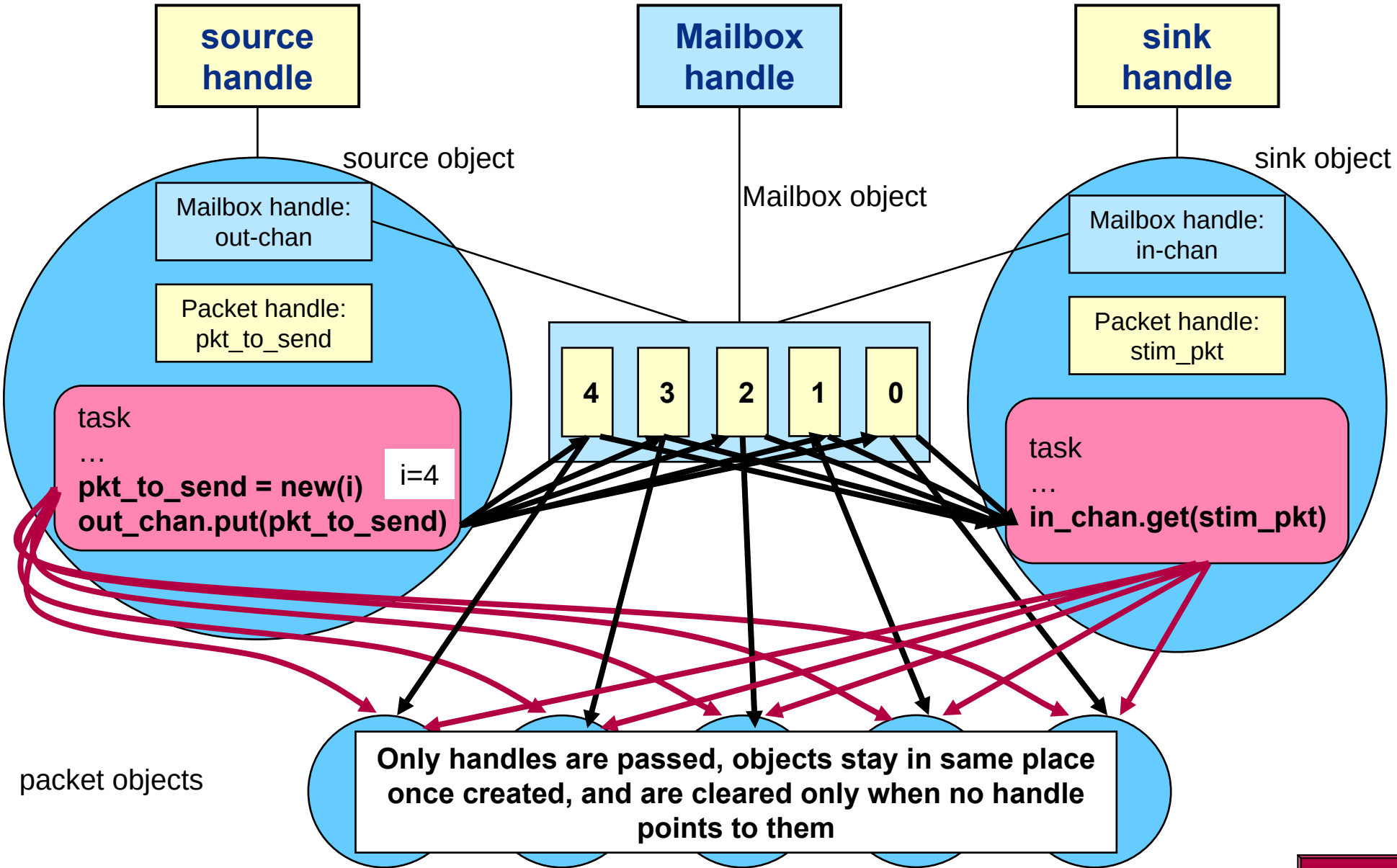
Lab – Classes & Mailboxes: Instructions - 3



- Create handles (initialize the arrays)
- Create objects and assign to handles (using new())
- Connect objects (by assigning (copying) mailbox handles)

Sol

Lab – Classes & Mailboxes: Wrap-up



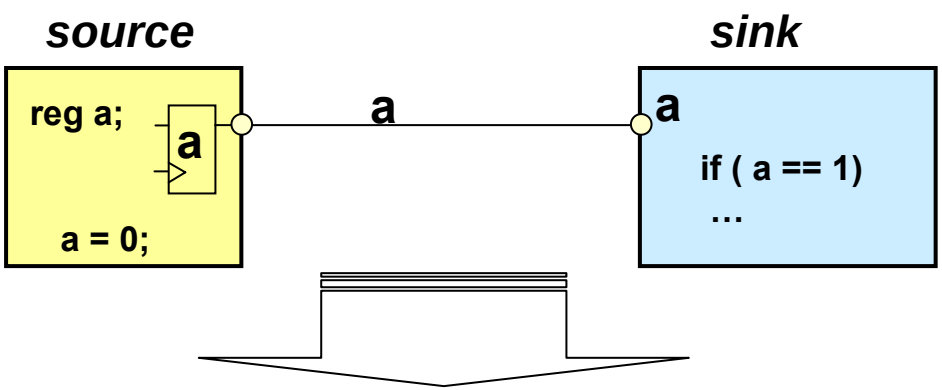
Sol

Interfaces

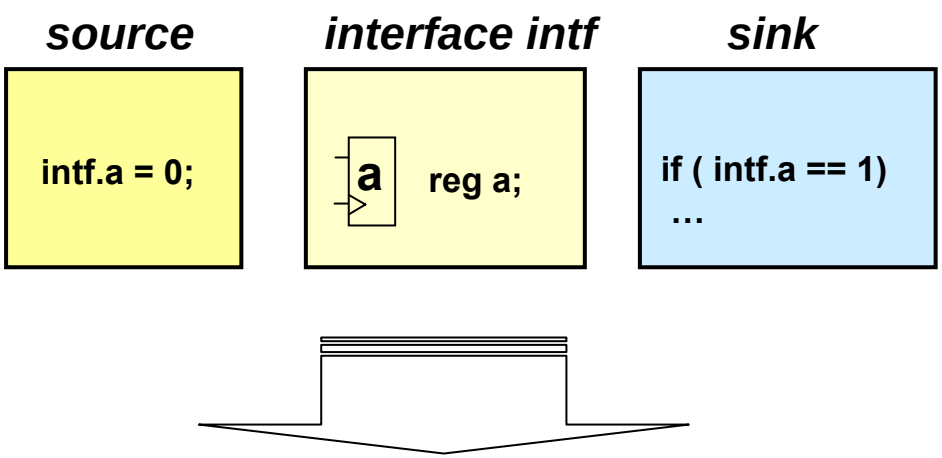
In this section

- Interface concepts & characteristics
- Simple bundled
- Methods
- Modports
- Importing methods

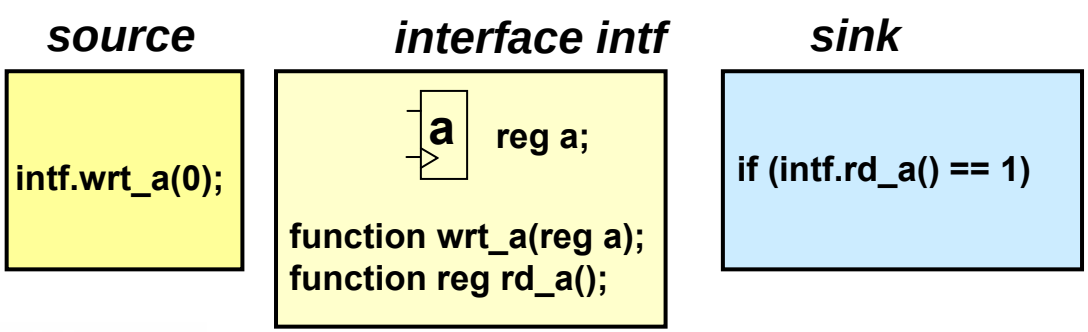
IO abstraction



- Traditional Verilog approach
  - Simple netlist-level IO
  - source/sink can be abstracted but IO must stay at low level
  - IO operations are cumbersome



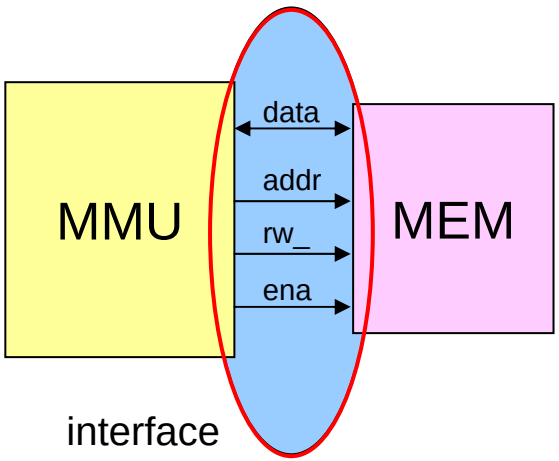
- Simple “bundle” interface
  - All accesses are through interface
  - Simplifies source/sink declarations



- Enhanced interface with methods
  - source/sink only call methods
  - source/sink don’t see low-level “details” like variables/structure, etc
  - Easy to swap interface abstractions without any effect on source/sink

Interfaces

- Great simulation efficiency can be achieved by modeling the blocks of a system at different levels of abstraction, behavioral, rtl, gate, etc.
- In Verilog, the I/O between blocks has always remained at the lowest “pin” level
- High-performance system-level simulation requires abstract inter-block communication.



At its simplest an interface is like a module for ports/wires

```
module mmu(d, a, rw_, en);
  output [15:0] a;
  output rw_, en;
  inout [7:0] d;
  . . .
endmodule

module mem(d, a, rw_, en);
  input [15:0] a;
  input rw_, en;
  inout [7:0] d;
  . . .
endmodule

module system;
  tri [7:0] data;
  wire [15:0] addr;
  wire ena, rw_;

  mmu U1 (data, addr, rw_, ena);
  mem U2 (data, addr, rw_, ena);
endmodule
```

Traditional Verilog

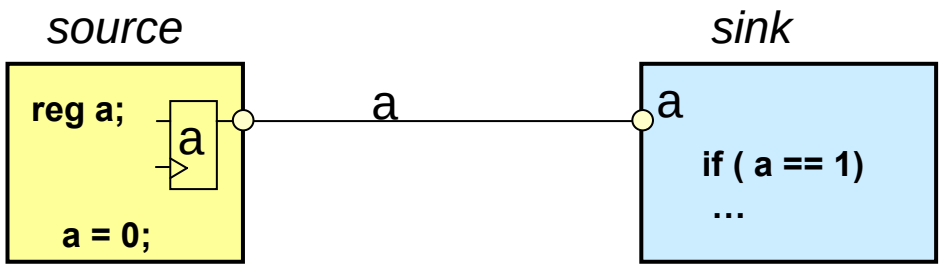
```
interface interf;
  tri [7:0] data;
  logic [15:0] addr;
  logic ena, rw_;
endinterface

module mmu(interf io);
  io.addr <= ad;
  . . .
endmodule

module mem(interf io);
   adr = io.addr;
 . . .
endmodule

module system;
  interf i1;
  mmu U1 (i1);
  mem U2 (i1);
endmodule
```

SystemVerilog

Interface characteristics

- Interfaces bring abstraction-level enhancements to ports, not just internals
- An interface may contain any legal SystemVerilog code except module definitions and/or instances
  - This includes tasks, functions, initial/always blocks, parameters etc.
- Bus timing, pipelining etc. may be captured in an interface rather than the connecting modules
- Interfaces are defined once and used widely, so it simplifies design
- Interfaces are synthesizable

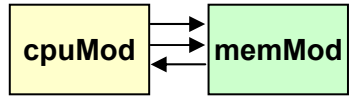
Traditional Verilog (no interfaces)



- Traditional Verilog approach
- Familiar, simple netlist-level IO

- source/sink can be abstracted but IO (a) must stay at low level
- IO operations are cumbersome, especially for abstract source/sink modules

Design Example – Traditional Verilog




```
module memMod (input bit req,  bit clk,  bit start,
                    logic[1:0] mode,  logic[7:0] addr,
                   inout logic[7:0] data,
               output bit gnt,  bit rdy          );
   logic avail;
   ...
endmodule
```

module definition

Complete portlist…

```
module  cpuMod ( input bit clk,  bit gnt,  bit rdy,
                inout logic [7:0] data,
               output bit req, bit start,
                   logic[7:0] addr,
                   logic[1:0] mode  );
   ...
endmodule
```

module definition

Complete portlist…

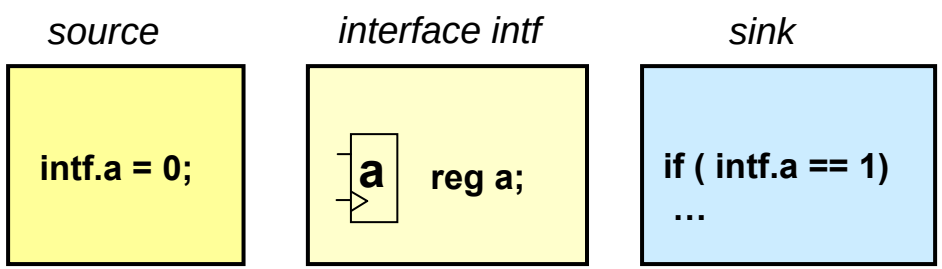
```
module top;
   logic req, gnt, start, rdy; // req is logic not bit here
   logic clk = 0;
   logic [1:0] mode;
   logic [7:0] addr, data;

   memMod mem(req, clk, start, mode, addr, data, gnt, rdy);
   cpuMod cpu(clk, gnt, rdy, data, req, start, addr, mode);

endmodule
```

Top-level module definition

Signals to interconnect instances

Instantiate/connect everything

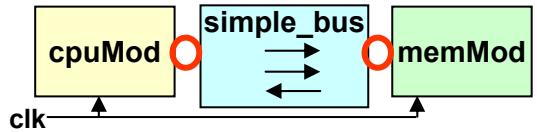
Simple bundle Interface



- Simple “bundle” of signals
- No port clutter
- All accesses are hierarchical (through interface)
- Simplifies source/sink declarations

- With no ports, who drives and who samples “a”
- All variables in interface are accessible
  (no privacy/control)

Simple Bundle interface




```
interface simple_bus; // Define the interface
  logic req, gnt;
  logic [7:0] addr, data;
  logic [1:0] mode;
  logic start, rdy;
endinterface: simple_bus
```

When an interface is used like a port… all the signals of that interface are assumed to be **inout** ports

simple_bus Interface definition

signals making up simple_bus

```
module memMod (simple_bus a,     // port a is of 'type' simple_bus
               input bit clk);   // and separately hook up clk
  logic avail;
 // a.req is the req signal in the 'simple_bus' interface

  always @(posedge clk) a.gnt <= a.req & avail;

endmodule
```

Declare a module and name its simple_bus interface 'a'

Access interface signals by name.signal

```
module cpuMod( simple_bus b, input bit clk);
  ...
endmodule
```

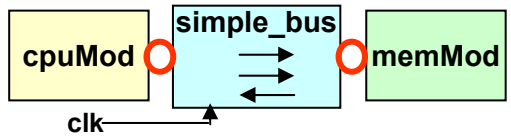
Declare a module and name its simple_bus interface 'b'

Top-level module

Instantiate/connect everything

```
module top;
  logic clk = 0;

  simple_bus sb_intf;          // Instantiate the interface
  memMod mem (sb_intf, clk);   // Connect the interface to the module instance
  cpuMod cpu (.b( sb_intf), .clk(clk));   // Either by position or by name

endmodule
```

Interface Ports



- Ports may be defined to an interface
  - This allows external connections to the interface to be made and hence automatically to all modules which bind to the interface
    - Typically used for clk, reset, etc.

```
interface simple_bus (input bit clk); // Define the interface
   logic req, gnt, start, rdy;
   logic [7:0] addr, data;
   logic [1:0] mode;
endinterface: simple_bus
-------------------------------------------------------
module memMod( simple_bus a );    // Uses just the interface
   logic avail;
   always @(posedge a.clk)        // the clk signal from the interface
      a.gnt <= a.req & avail;     // a.req is in the 'simple_bus' interface
endmodule
-------------------------------------------------------
module cpuMod(simple_bus b);  ...  endmodule
-------------------------------------------------------
module top;
   logic clk = 0;
   simple_bus sb_intf1(clk);      // Instantiate the interface
   simple_bus sb_intf2(clk);      // Instantiate the interface
   memMod mem1(.a(sb_intf1));     // Connect bus 1 to memory 1
   cpuMod cpu1(.b(sb_intf1));
   memMod mem2(.a(sb_intf2));     // Connect bus 2 to memory 2
   cpuMod cpu2(.b(sb_intf2));
endmodule
```

simple_bus Interface with one input port 'clk'

Module with interface 'a'

2 simple_bus instances

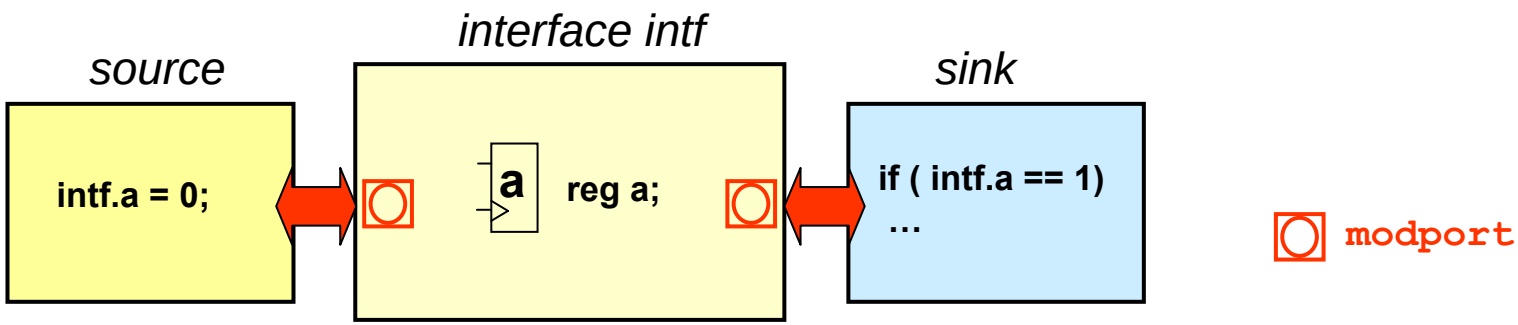
cpu/memory pair 1

cpu/memory pair 2

Interface Modports

Modport derives from keywords “module” and “port”

They identify signal ownership from point of view of a module using that modport

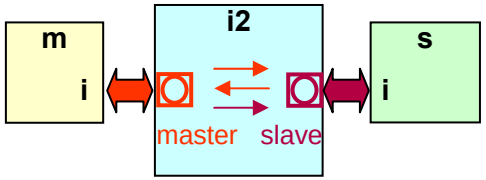


- Modports specifically control variable access
- source/sink connect to a specific modport
- source/sink can’t use resources not listed in their modport

- Multiple modules can connect to interface using the same modport so we may need to consider this in our implementation

Modports – Example

"modport" keyword implies how the ports are accessed from the point of view of the module…




```
interface i2;
   logic a, b, c, d, e, f;
   modport master (input a, b, output c, d, e);
   modport slave (output a, b, input c, d, f);
endinterface
```

Interface 'i2' with 'master' and 'slave' modports

The modport list name can be specified in a couple ways:

Style 1 ***… in the module header…***

```
module m (i2.master i);
   ...
endmodule

module s (i2.slave i);
   ...
endmodule
--------------------

module top;
   i2 itf;
   m u1(.i(itf));
   s u2(.i(itf));
endmodule
```

i is port name of interface i2

Modport indicates direction

Interface name acts as a type

Style 2 ***… in the module instantiation…***

```
module m (i2 i);
   ...
endmodule
-----------------------

module s (i2 i);
   ...
endmodule
-----------------------------------

module top;
   i2 itf;
   m u1(.i(itf.master));
   s u2( .i(itf.slave));
endmodule
```

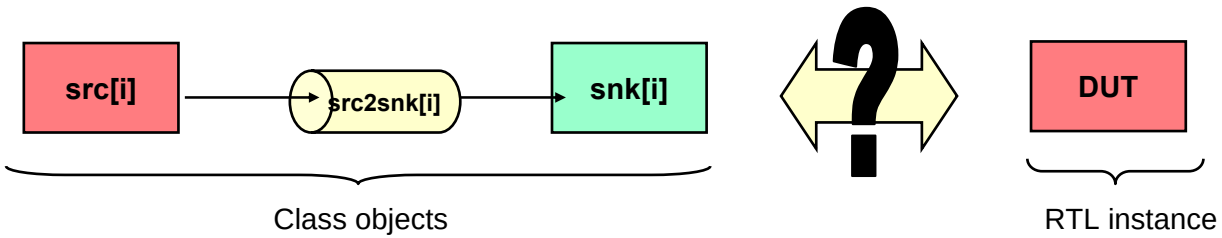
HINT: Any number of modules may connect to an interface via the same modport

Of what use are Interfaces in Verification ?

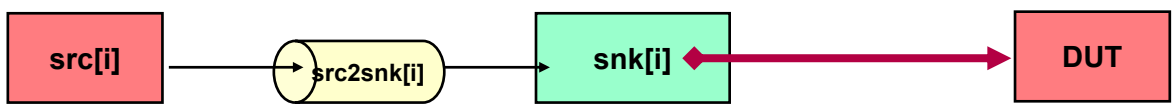
- Interface
  - Bundle signals into appropriate groups
  - Efficient way for objects to talk to modules/ports
  - Used to connect between class objects and modules (via virtual interface)
- Modport
  - Subgroup signals by purpose, timing and direction
  - Specify direction of asynchronous signals
  - Note: signals may appear in multiple modports

Objects Communicating with Module Instances

- How do you "talk" between a class object and a module?
  - Module instance has RTL ports
  - Class object does not have ports

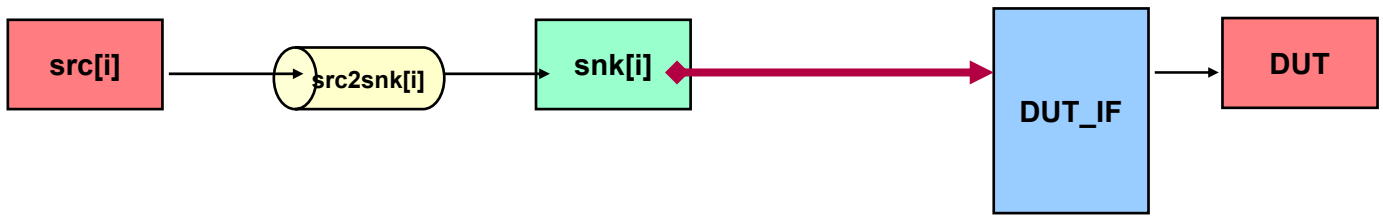


- Simple solution is to hierarchically address the module's ports (or signals connected to the ports) from the class
  - Issues
    - Must hard code into class definition a hierarchical path to the port or signal
    - Limits reusability of class
      - What if you have multiple DUT / snk's to connect?



Classes & Interfaces

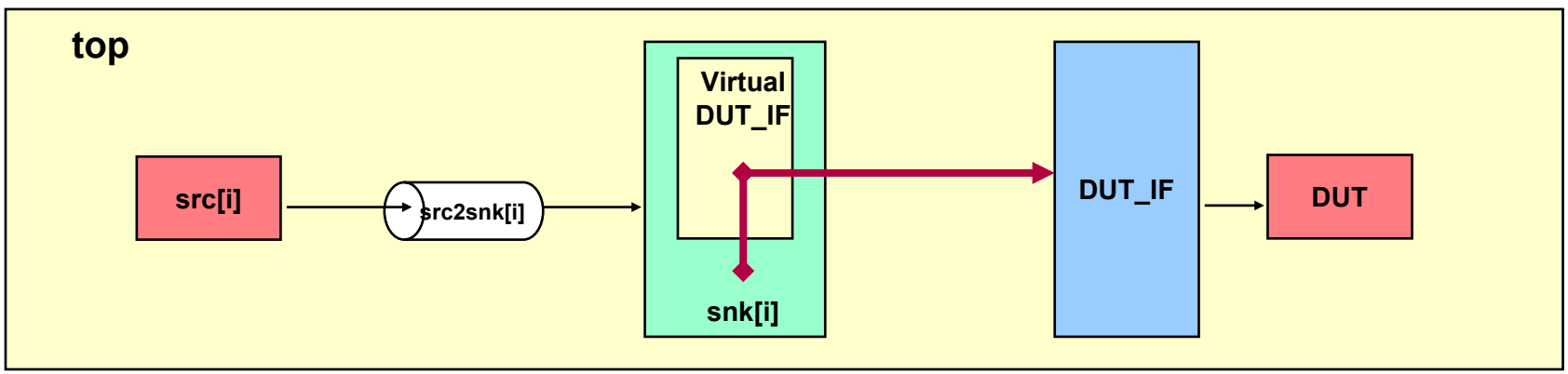
- An Interface can simplify the class object ← → module instance communication
  - Must still hard code into the class the hierarchical path to the interface instance



- Need a better approach….
  - If only interfaces were object oriented…

Classes & Virtual Interfaces

- **Virtual Interface**
  - Mechanism for separating abstract models and test programs from the actual signals that make up the design
  - Can be declared as a class property
    - **Think of it as a handle to an interface**
    - May be initialized
      - Procedurally
      - By constructor argument
    - May be changed dynamically
    - Instead of referring directly (via a hierarchical pathname) to the actual signals in an interface the user manipulates virtual signals through the variable



Declaration syntax: `virtual interface_type v_if_name ;`

Steps to Virtual Interfaces

- **It will most often be desirable to create a virtual interface connection between the OOP testbench and the DUT without modifying the DUT**
- **Steps to a virtual interface connection (reference next slide)**
  1. Create an interface (dut_if) with variables which map to the DUT's ports
  2. Instantiate the interface (d_if) at the same level of hierarchy as the DUT
  3. Instantiate the DUT and connect its ports via hierarchical reference to the variables in the dut_if that map to the DUT ports (from step 1)
  4. In the transactor class (sink) add a property (v_dut_if) of type *virtual interface_type* (virtual dut_if)
  5. In the transactor class add an argument (real_dut_if) to the constructor of type *virtual interface_type* (virtual dut_if)
  6. In the body of the constructor assign the argument real_dut_if to the property v_dut_if
  7. Instantiate the transactor class passing the interface instance (d_if) as the constructor argument

Virtual Interface Example – Source/Sink

```
interface dut_if;
int data_rcvd;
  bit clk = 1;

  always #25 clk = !clk;

endinterface
```

1

```
module dut( input int data_rcvd,
            input bit clk );
 always @(posedge clk)
 begin
   $display("%m received data:  %0d",
   data_rcvd);
 end
 endmodule
```

```
module top;
import types::*;

dut_if d_if();

dut DUT(.data_rcvd (d_if.data_rcvd),
        .clk ( d_if.clk )
);

sink        snk = new(d_if);

source     src = new(2);
mailbox #(Packet) src2snk = new(1);

endmodule
```

2

3

7

```
class sink;
  mailbox #(Packet) in_chan; // null handle
  virtual dut_if  v_dut_if;
  int id = 1;

  function new ( virtual dut_if  real_dut_if );
     // map virtual interface to real interface
     v_dut_if = real_dut_if;
  endfunction

  task run();
    while(1) begin
      in_chan.get(stim_pkt);
    // access real interface via virtual IF
    @( negedge v_dut_if.clk )
        v_dut_if.data_rcvd = stim_pkt.field1;
    $display ("sink: Received packet with field1 = (%0d)",
                          id, stim_pkt.field1);
    end
  endtask
endclass : sink
```

4

5

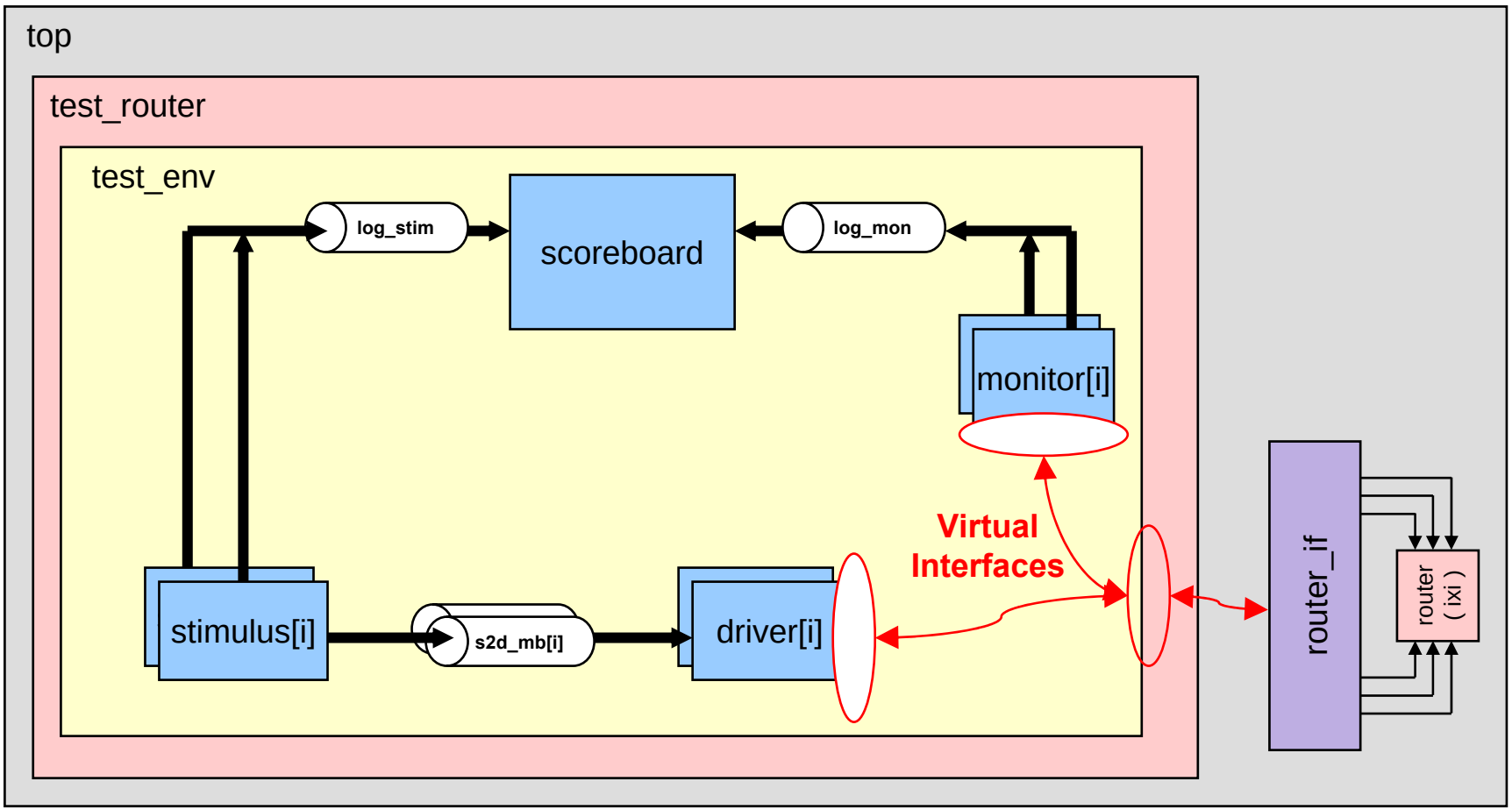
6

Router design

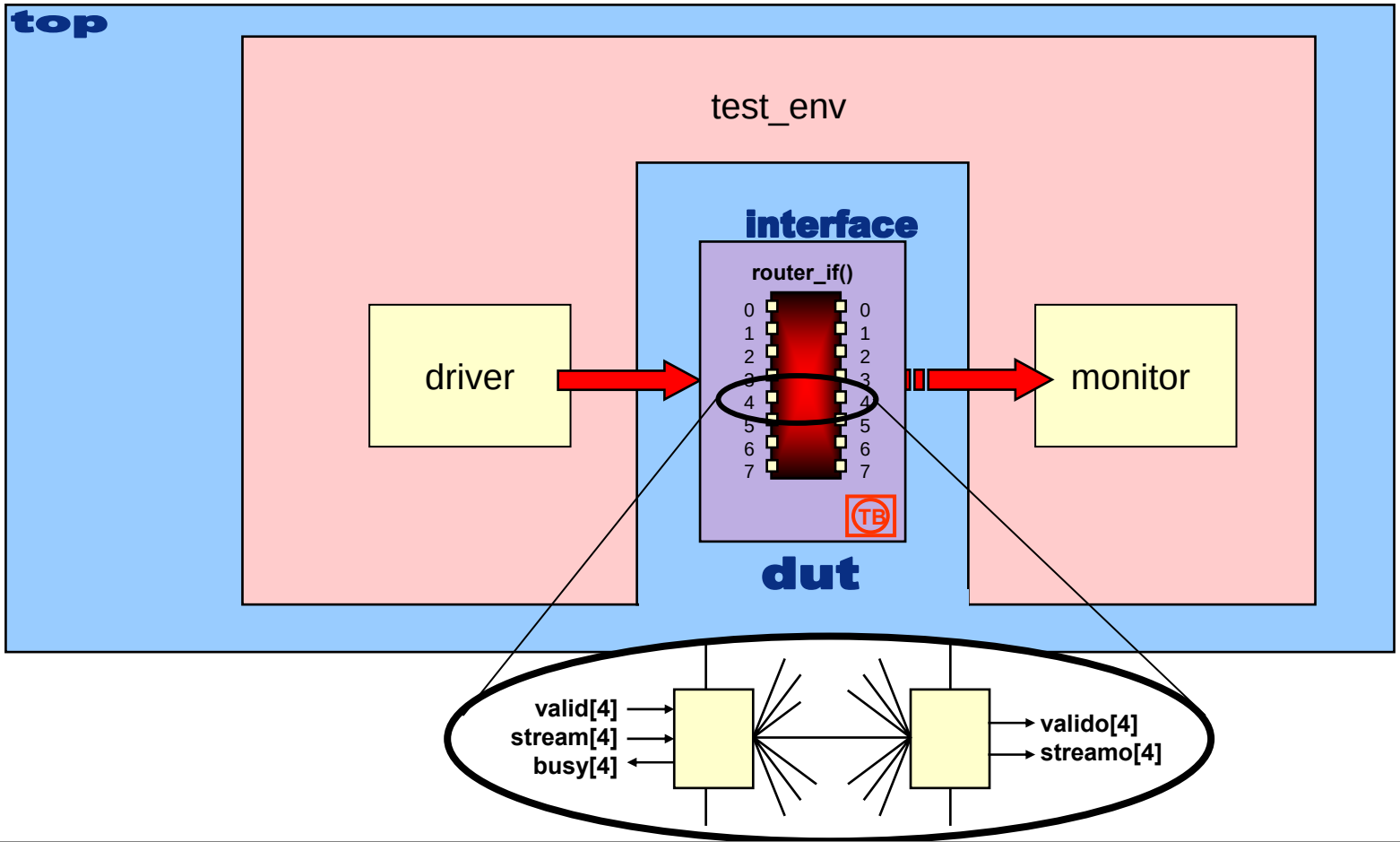
SystemVerilog

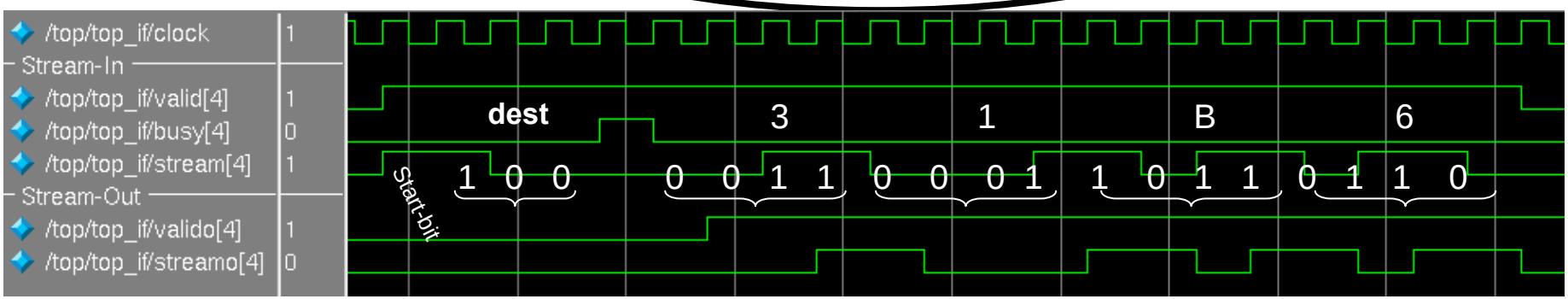
Router Design & Testbench

- **Modules:** top, test_router, router
- **Classes:** test_env, stimulus, driver, monitor, scoreboard
- **Interfaces:** router_if
- **mailboxes:** log_stim, log_mon, s2d_mb



Router Schematic

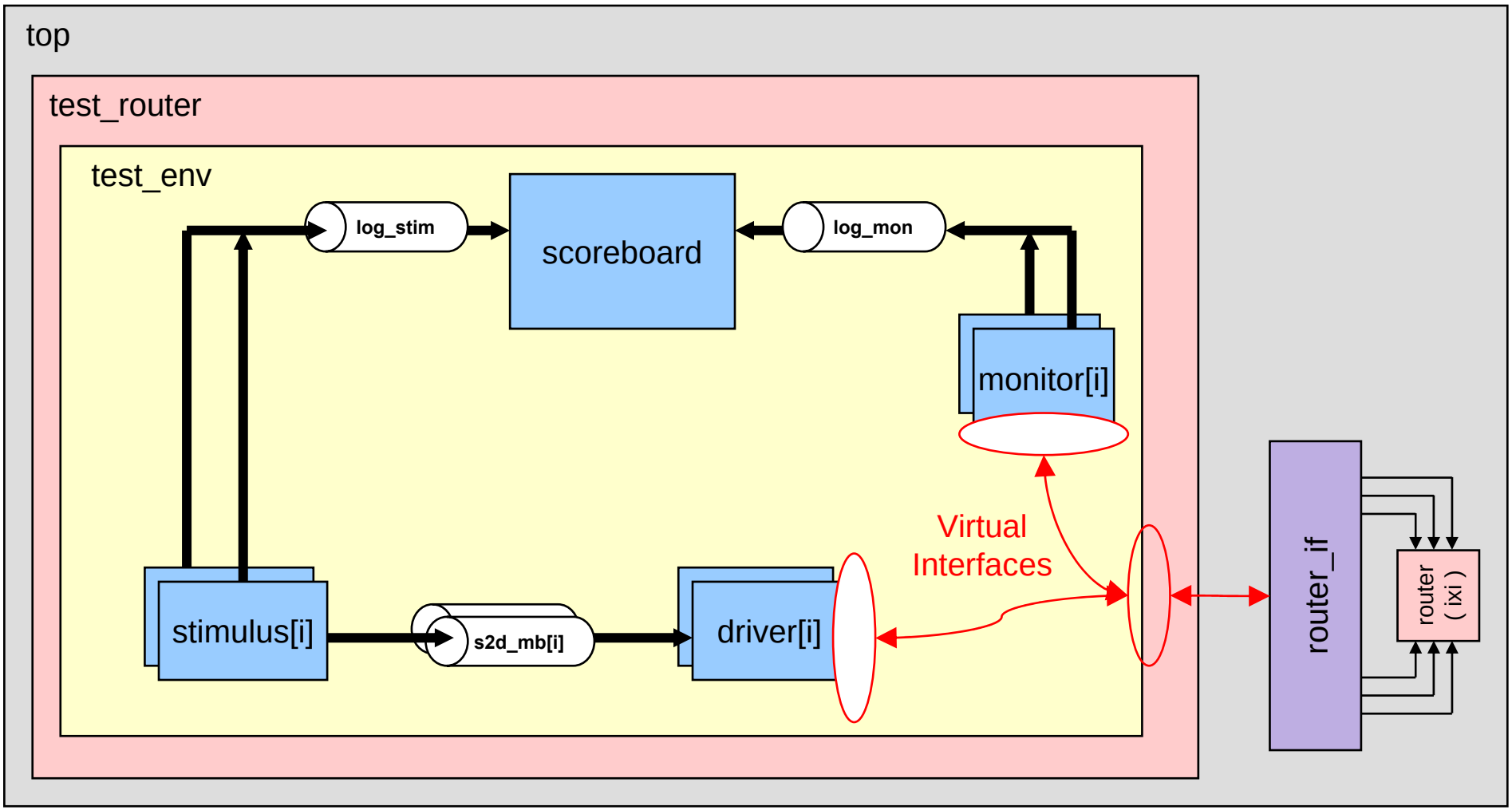




Example Waveform: shows packet #31 (Payload 1 byte: B6) entering port 4 routed to port 4

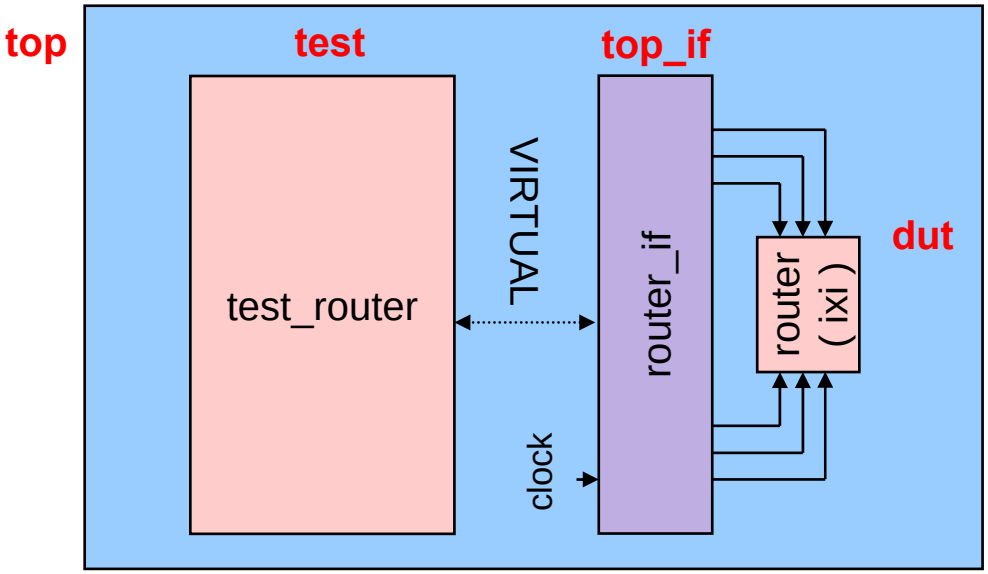
Lab – Virtual Interface: Introduction - 1

- Lab directory: **Router/virtual_interface**
- Purpose: Connect abstract objects to a DUT using a virtual interface



Lab – Virtual Interface: Introduction - 2

```
interface router_if(input bit clk);

  logic  rst ;
  logic [7:0] valid ;
  logic [7:0] stream ;
  logic [7:0] streamo ;
  logic [7:0] busy ;
  logic [7:0] valido ;

endinterface: router_if
```

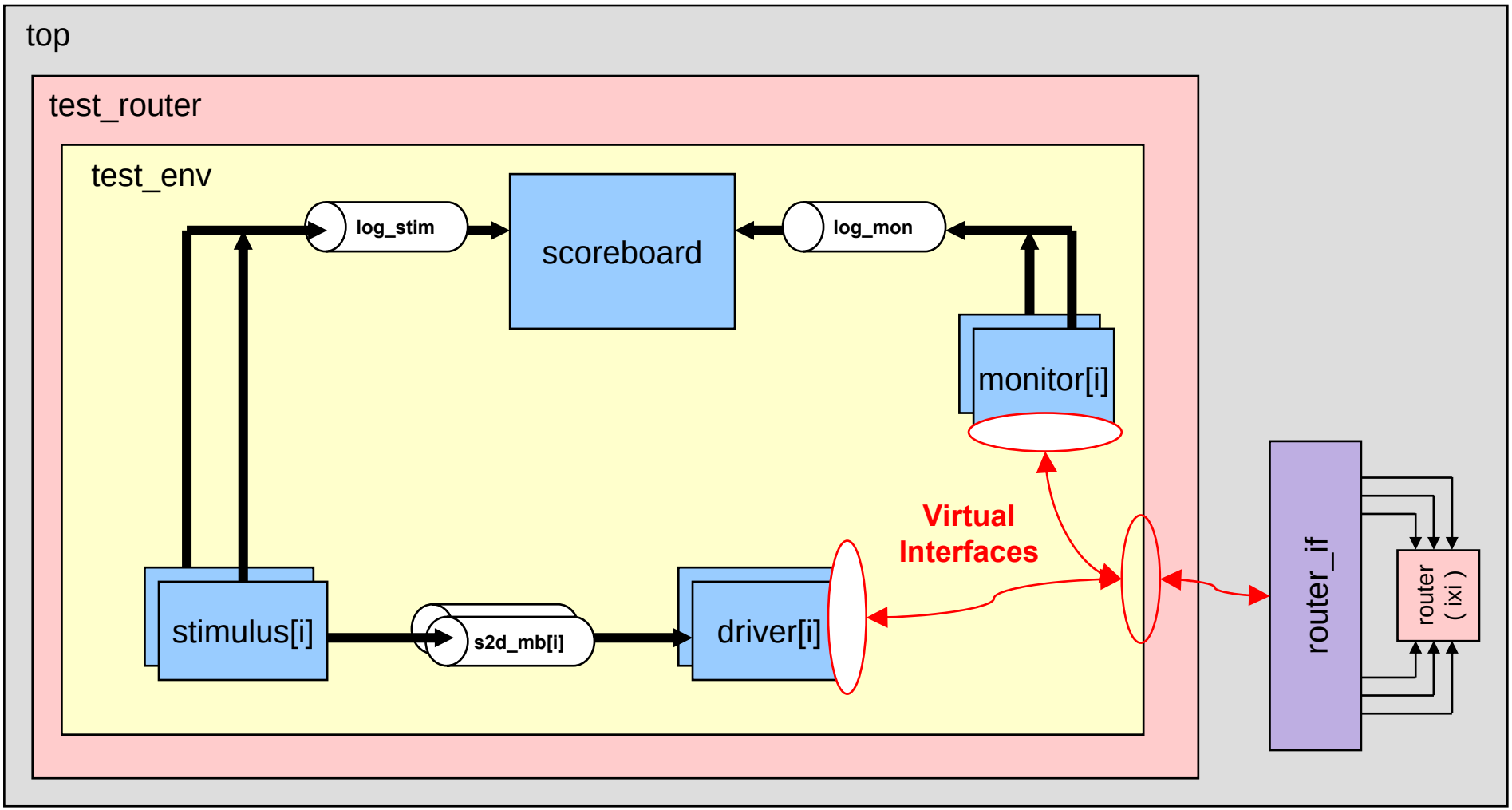


- File top.sv
  - Contains instantiations for:
    - the interface router_if "top_if"
    - the router module "dut"
    - the test_router module "test"

- What's missing? The virtual interface connections within test_router.sv

Lab – Virtual Interface: Instructions - 1

- In this exercise you will configure virtual "router_if" interfaces in each of the *test_env()*, *driver()* and *monitor()* classes



Lab – Virtual Interface: Instructions - 2

- Edit test_router.sv
  - Add a virtual router_if interface called r_if to each of the classes: test_env(), driver() and monitor()
    - Modify the constructor in each of these classes to:
      - accept a virtual router_if interface argument.
      - assign that input argument to the local virtual interface r_if
    - Modify the test_env() class to:
      - Pass the correct virtual interface argument to driver() and monitor() contructors (d.new() and m.new())
  - Modify the test_router module to:
    - Pass the real interface instance name (r_if) to the constructor of test_env() when you instantiate it.
  - Compile & run the code by typing:
    - make
    - -or- make gui

Expected Simulator output

```
…

#
# 588 packets sent, 500 packets received, 0 errors
#
```

Sol

Classes (cont.)

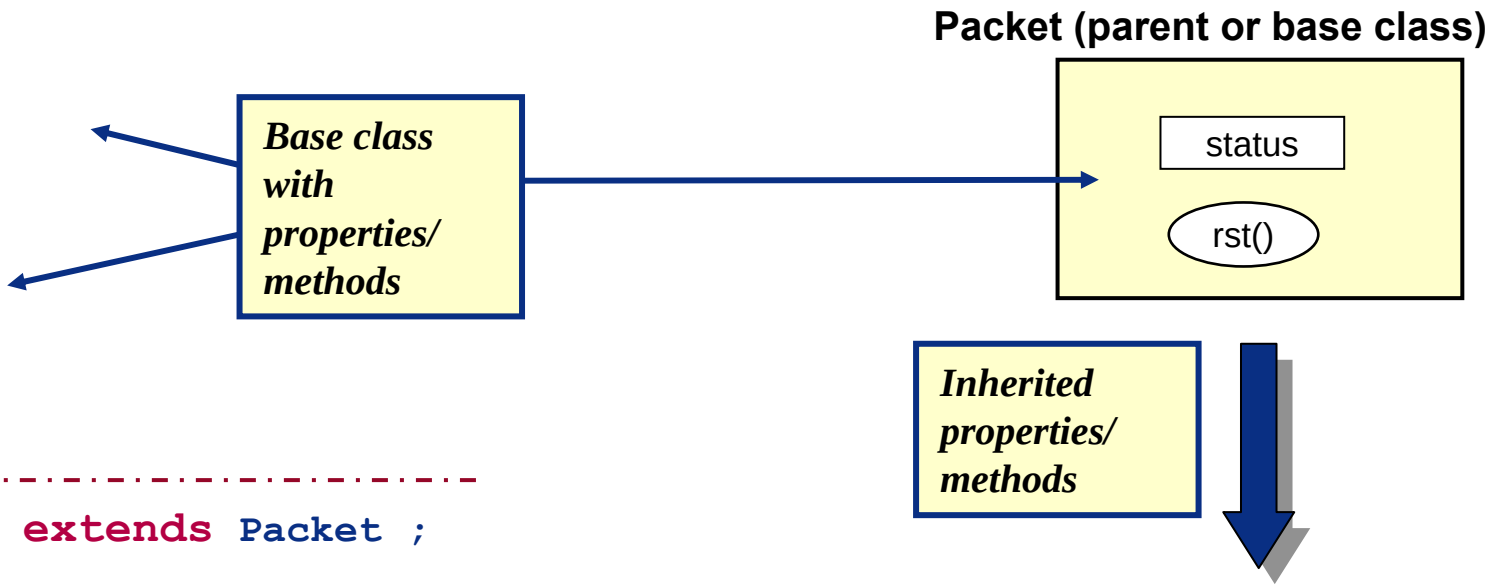
In this section

- Parameterization
- Inheritance
- Virtual Methods
- Reference
  - Encapsulation
  - Parameterization
  - Polymorphism

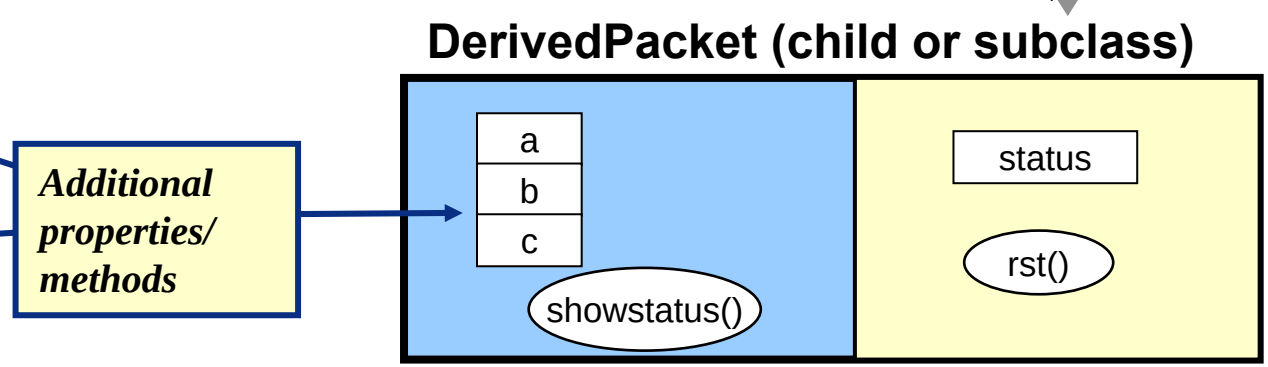
Inheritance

- Rather than add additional functionality to a class by copying and modifying we "extend" classes by inheritance

```
class Packet ;
  //properties
  integer status;

  // methods
  task rst();
    status = 0;
  endtask
endclass
```



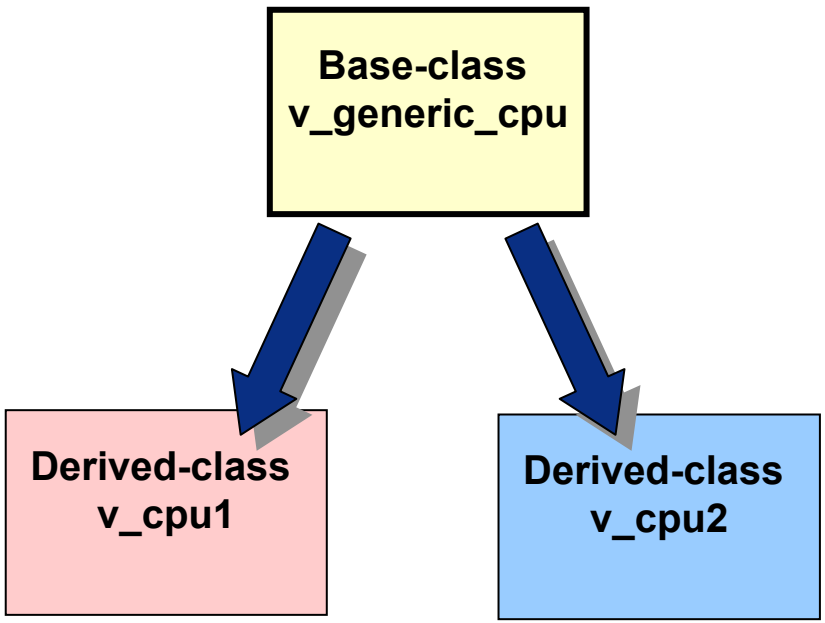

```
class DerivedPacket extends Packet ;
  //properties
  integer a,b,c;

  // methods
  task showstatus();
    $display(status);
  endtask
endclass
```



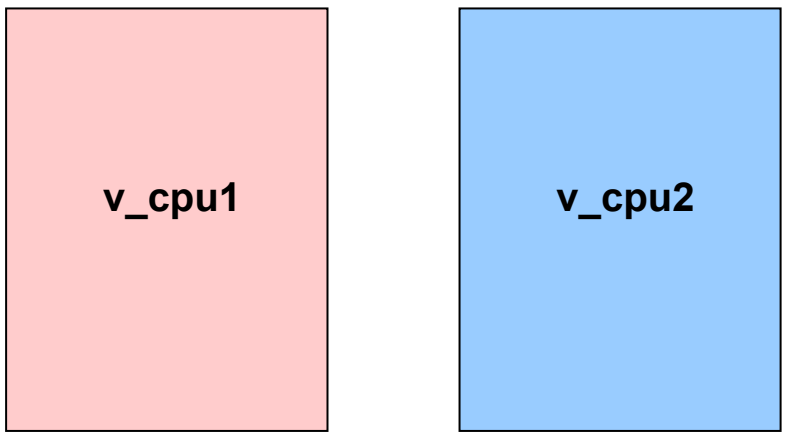
- **DerivedPacket inherits all of Packet, that is to say DerivedPacket "*is a* " Packet**
- ***PLUS* DerivedPacket has its own additional properties/methods**

Inheritance Example

- Inheritance approach is ideal for verification
- Non-inheritance approach involves duplicated effort



vs.



- Base class with all common "stuff"
  - Customized v_cpu1
  - Customized v_cpu2
- Easy to make a class for verification of a 3rd type cpu

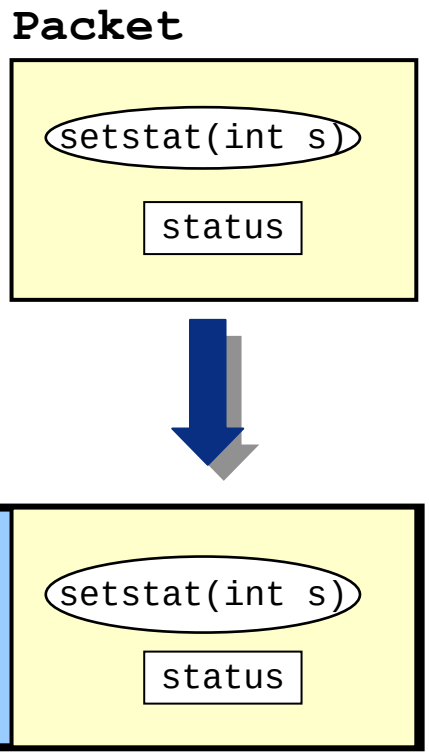
- Two different custom v_cpu's
  - Lots of duplicate effort
- Lots of work to create a third
  - Even more duplicate effort

Inheritance – overriding

- Derived classes may “override” the definition of a member inherited from the base class
  - **"Hides"** the overridden property or method
  - To override a method the child method's signature must match the parent's exactly

```
class Packet ;
  int status = 4;

  function void setstat(int s);
    status=s;
    $display ("parent setstat");
  endfunction
endclass
```

Base class with properties/ methods




```
class DerivedPacket extends Packet

  function void setstat(int s);
    status=s;
    $display ("child setstat");
  endfunction
endclass
```

DerivedPacket overrides method of Packet

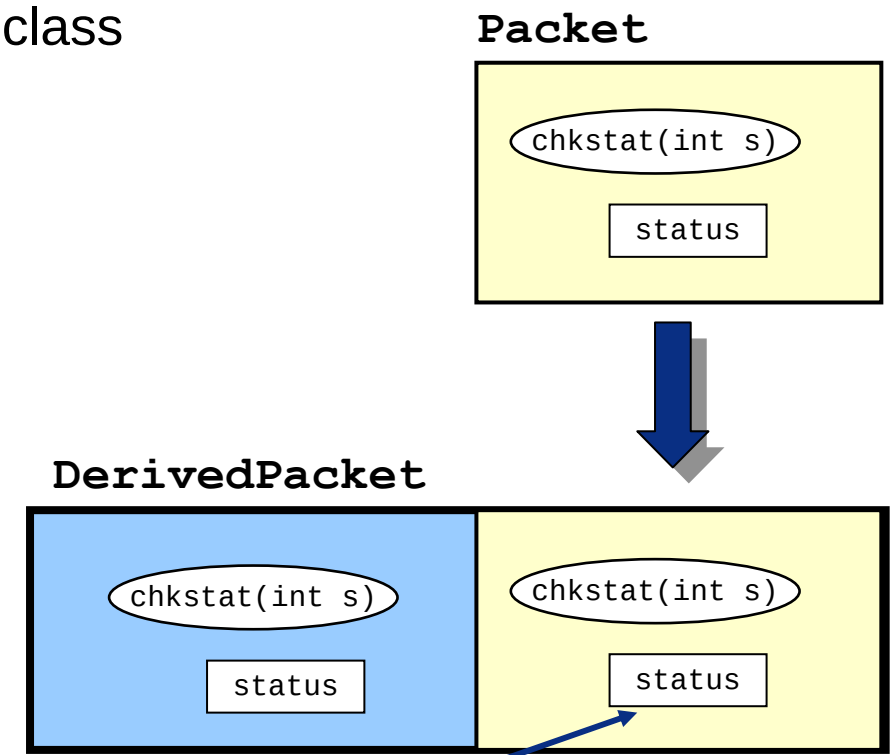
```
DerivedPacket p = new();
initial begin
  p.status = 44;     // OK to access inherited property
  p.setstat(33);     // output is "child setstat"
end
```

which setstat() is called?

super

- The **super** keyword is used to refer to members of the base class
  - Required to access members of the parent class when the members are overridden in the child class
  - You may only "reach up" one level (super.super is not allowed)
  - Only legal from within the derived class

```
class Packet ;
  int status = 4;

  function bit chkstat(int s);
    return (status == s);
  endfunction
endclass
-----------------------------------
class DerivedPacket extends Packet ;
  int status = 15;

  function void chkstat(int s);
    $display(super.status);
  endfunction
endclass
```



Prints "4" because it explicitly refers to the base class member status

Cannot access status outside of object. super keyword legal only inside an object

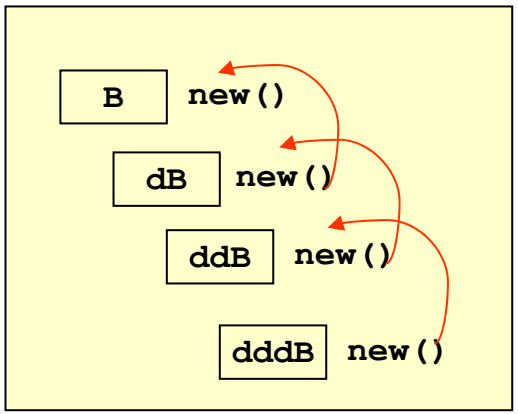
```
DerivedPacket p = new();
p.super.status = 88;  //illegal!
```

Constructors & Inheritance - 1

- All classes have a default constructor which may be overridden if specific initialization is required.

```
class Packet ;
  //properties
  integer status;

  // methods
  function new();
    status = 0;
  endfunction
endclass
  ---------------------------------
   class DerivedPacket extends Packet ;
     //properties
     integer a,b,c;

     // methods
     function new();
       a = 0; b = 0; c = 0;
     endfunction

   endclass
```

Base class constructor



Derived class with its own constructor

When invoked, & before running any of its own code, new() invokes the new() method of its super-class, and so on up the hierarchy. Execution is root downward...

Constructors & Inheritance - 2

- The default SV constructor has no arguments. What if a base class constructor requires arguments not present in the child class constructor?

```
class Packet;
  //properties
  integer status;
  int a,b,c;
  // methods
  function new(int i);
    status = 0;
    a = i; b = i; c = i;
 endfunction
endclass
```

Base class constructor

```
class DerivedPacket extends Packet;
  //properties
  // methods
  function new();
  endfunction

endclass
```

ERROR
implicit call of parent class constructor new() does not match parent class constructor signature

```
class DerivedPacket1 extends Packet;
  //properties
  // methods
  function new();
    super.new(5);
  endfunction

endclass
```

Solution 1

```
class DerivedPacket2 extends Packet(5);
  //properties
  // methods
  function new();
  endfunction

endclass
```

Solution 2
(pass constructor args)

Solution 3: Add default argument values to the Base class constructor

Inheritance Puzzle #1

- Inheritance is quite straightforward but can raise some confusing situations…Consider:

```
class Packet ;

  task build_payld();
      $display("Packet payld");
  endtask

  task build_packet();
      …
      build_payld() ; // which build_payld() will this call?
      …
  endtask

endclass
```

```
class DerivedPacket extends Packet ;

   task build_payld() ;  // over-ridden method
      $display("DerivedPacket payld");
   endtask

endclass
```

```
module poly1;

DerivedPacket der = new();

initial
  der.build_packet();

endmodule
```

Simulation output

```
Packet payld
```

The base class does not know anything about the derived class. So here build_packet() will call build_payld() in the base class even though it is overridden in the derived class

Question

How can we have the base class method call the overridden method in the derived class instead?

Virtual Methods

- A virtual method in a base class is like a prototype for derived classes
- Variation on previous slide:

```
class Packet ;
  virtual task build_payld();
      $display("Packet payld");
  endtask

  task build_packet();
      …
      build_payld();
      …
  endtask

endclass
```

```
class DerivedPacket extends Packet ;

   task build_payld();
      $display("DerivedPacket payld");
   endtask
endclass
```

```
module poly3;

DerivedPacket der = new();

initial
  der.build_packet();

endmodule
```

Simulation output

```
DerivedPacket payld
```

Here, program calls base method build_packet() which calls build_payld() but because build_payld() is declared virtual in the base class…
the local derived class method is called.

Think of Virtual methods this way:

The implementation of a virtual method that is used is always the last one in a descent of the hierarchy (or local to DerivedPacket in this case).

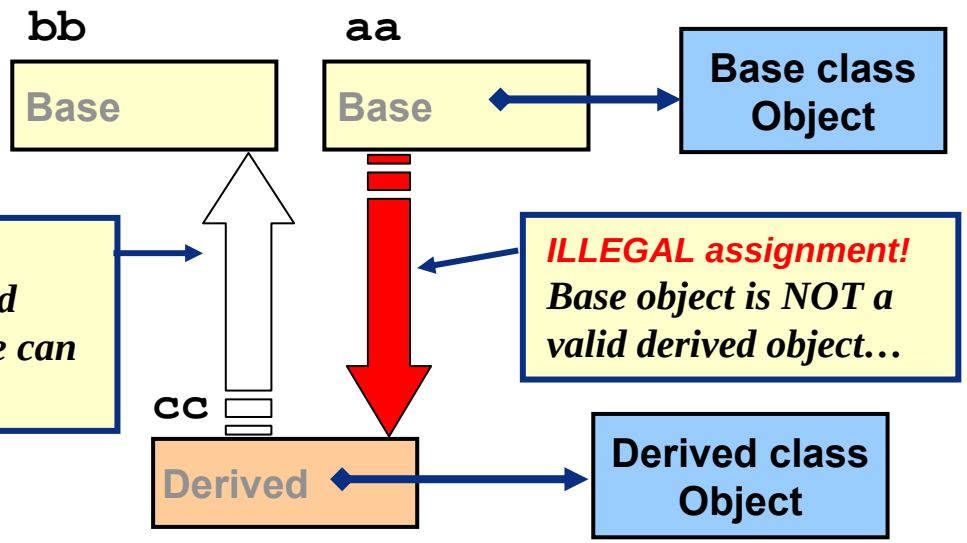
More on " is a "

- We said earlier that a derived-class object is also a 100% valid object of the base class
- Let's see…

```
module is_a;

  Base aa, bb;
  Derived cc;
```

```
class Base ;
   integer status;
endclass
```

```
class Derived extends Base;
   integer fred;
endclass
```



LEGAL assignment!
Derived object "is a" valid base object so base handle can point to derived objects

ILLEGAL assignment!
Base object is NOT a valid derived object…

```
  initial begin

    aa = new();
    cc = new();

    bb = cc;   // legal

    cc = aa;   // Compile time Error !

  end
endmodule
```

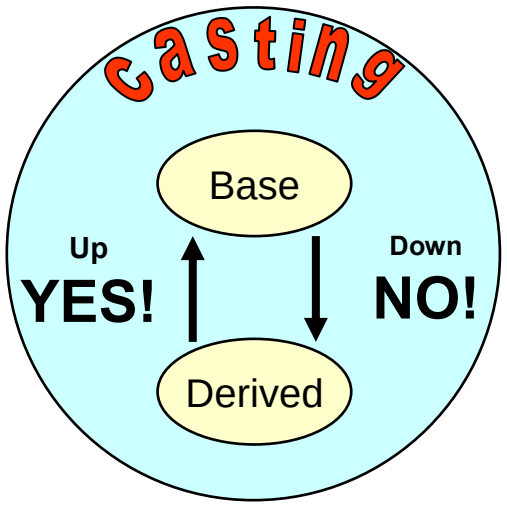
Dynamic Casting & OOP - $cast

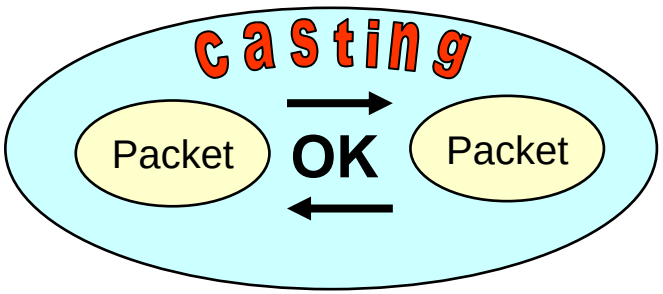
- How can you tell if two objects are of the same or compatible classes?

```
module same_class;
  Base aa, bb;

  initial begin
    aa = new(); bb = new();
    if ( $cast (aa, bb) )          // returns 1
      $display("Objects are of compatible type");
   else
      $display("Should never see this");
  end
endmodule
```

```
class Base ;
  integer status;
endclass
```

```
class Derived  extends Base;
   integer fred;
endclass
```






```
module base2derived;
  Base aa; Derived cc;
  initial begin
    aa = new(); cc = new();
    if (!( $cast(cc, aa) ))   // ILLEGAL
         // Illegal cast - from base to derived

    if ( $cast(aa,cc) )       // LEGAL
         // Legal -  from derived to base

    if (( $cast(cc, aa) ))    // LEGAL
         // Now legal!!! - back to derived
    cc = aa;   // But this is ALWAYS a compile error!
  end
endmodule
```

Lab – OOP: Instructions – 1

- Lab directory: **Router/oop**
- **Overview**:
  - This version of the router is missing some functionality. In this lab, you will complete the router testbench by extending the base class (BasePacket) to add a compare function (`compare()`).
- Edit the Packet definition file (defs.sv)
  - From the base class **BasePacket**, declare a derived class (**Packet**) (declare it after class **BasePacket** in the same file) which has the following characteristics:
    - A new function `compare()`:
      - `function bit compare(Packet to);`
      - return type `bit`
      - 1 input argument called "to":
        - » argument "to" is of type Packet and is the value to compare against

- Continued on next page -

Lab – OOP: Instructions – 2

- Lab directory: **Router/oop**

  - `compare()` function should perform the following checks:
    - Input Packet “to” should be a valid handle (i.e. not null)
    - Payload size and contents of "to" should match local payload size and contents
    - `compare()` function returns status (1 for success, 0 for fail)
  - Add function new()
    - takes an argument (p_id) of type bit[7:0] with a default value of 1
    - Uses argument (p_id) to initialize the pkt_id field of the inherited base class

  - Compile & run the code by typing:
    - make
    - -or- make gui

What’s missing?
We need better randomization of the packets
Coming up: Object Oriented Randomization

Sol

static Properties/Methods

- By default each instance of a class has its own copy of the properties and methods
- Use of the keyword **static** makes only one copy of the property or method that is shared by all instances
- **static property**
  - A property that is shared across all objects of the class
  - Static properties may be accessed before an object of the class is created
- **static method**
  - A method shared across all objects of the class
  - May only access static properties
  - Callable from outside of the class, even before an object of that class is created

```
class static_ex;
  static integer fileId =
         $fopen( "data", "r" );
  static int val;
  static function void print_val();
    $display("val = %0d", val);
  endfunction
endclass

module test;
static_ex s1;   // Only handle is required
static_ex s2;
bit[7:0] c;
initial begin
 c = $fgetc( s1.fileId );
 $display("%0s", c);
 s1.val = 22;
 s1.print_val();
 s2.val = 44;
 s1.print_val();
end
endmodule
```

File: **data**

```
a
```

Sim. Output

```
# a
# val = 22
# val = 44
```

const Properties

O 6.2 Ignored

- The keyword **const** inside a class is consistent with elsewhere in SV
- A property that is declared **const** is protected from any modification.
- Two sub-types of **const** are defined:
  - Global constant (declaration specifies value)
  - Instance constant

```
class gPacket;
// global constant protected from modification
   const int size = 512;
   byte payload [size];
endclass
```

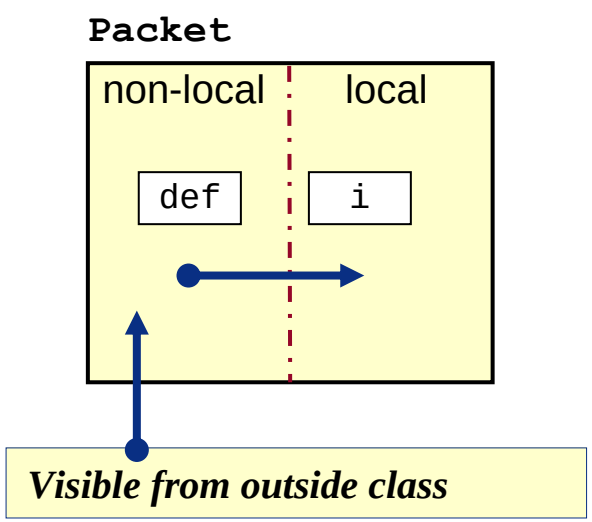
```
class iPacket;
  const int size;                  // instance constant protected from all but constructor
  byte payload [];
  function new();
    size = $random % 4096;  // only one assignment (in constructor) is allowed
    payload = new[ size ];
  endfunction
endclass
```

No initial value!

Data Hiding – `local`

- In OOP it is common to want to control access to object members.
  - Preventing member corruption
  - Hiding the implementation details from users who might hard-code their assumptions and prevent later implementation changes
  - Provide a public "interface" for the class
- SV defines `local` and `protected` (more later) keywords
- `local`
  - A local member is only visible within that class or by hierarchical reference from other instances of the same class

```
class Packet;
   local int i;    // visible only from within class Packet
   int def;
endclass

Packet p = new();

p.def = 8;    // OK
p.i = 5;      // ERROR!
```



Default is “non-local” (like public in C++) - open to anyone!

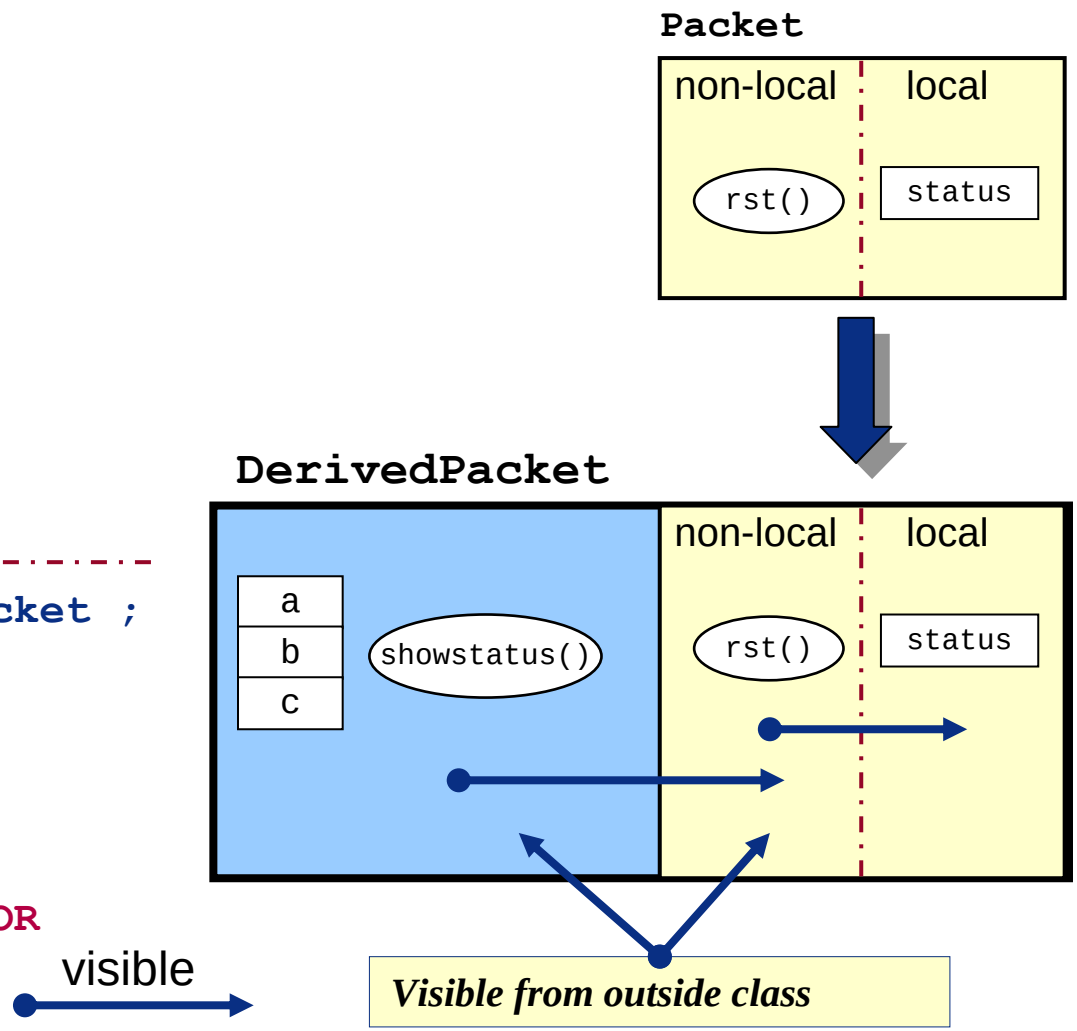
visible

local & Inheritance

- Properties and methods which are declared local in the parent class are not "visible" in the child or subclass which inherits from the parent
  - Present but not visible from subclass
  - May only be accessed by methods in the parent class

```
class Packet ;
  //properties
  local integer status;

  // methods
  task rst();
    status = 0;
  endtask
endclass
```

```
class DerivedPacket extends Packet ;
  //properties
  integer a,b,c;

  // methods
  task showstatus();
    $display(status);    // ERROR
  endtask
endclass
```

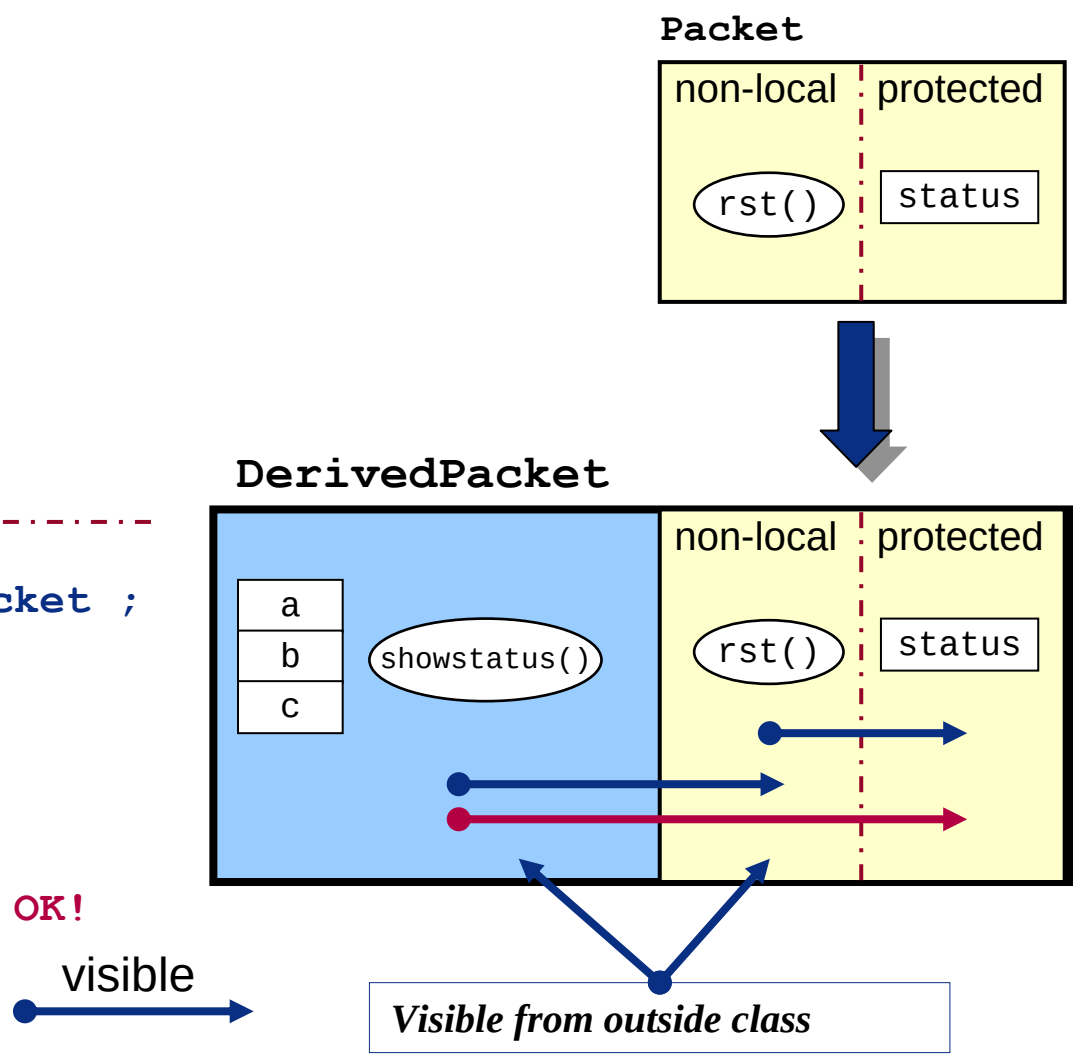


protected & Inheritance

- **protected** members of the parent class
  - Treated as local members in the parent class but visible in child classes
  - Inherited as protected and treated as local members in the child class

```
class Packet ;
  //properties
  protected integer status;

  // methods
  task rst();
    status = 0;
  endtask
endclass
```

```
class DerivedPacket extends Packet ;
  //properties
  integer a,b,c;

  // methods
  task showstatus();
    $display(status);    // Now OK!
  endtask
endclass
```



extern

- Allows separation of the declaration of a method (in the class) from its' definition outside of the class

```
class Packet;
  ...
  extern protected virtual function int send(int value);
endclass
```

DECLARE the method

```
function int Packet :: send(int value);

// body of method
...
endfunction
```

DEFINE the method

Class Scope Resolution operator ::

Parameterized Classes: Specialization

- A class which has parameters is called a ***generic class***
  - A generic class in and of itself is *not* considered a new type
  - An instance of a generic class with a unique set of parameters is called a ***specialization***
- A **specialization** *is* considered a new type
  - Each specialization with a different unique set of parameters defines a new type
  - A specialization of a generic class using the default parameter values/types is called a ***default specialization***
  - Two specializations of a particular generic class with *same* parameter values/types are considered the *same* type
- Handles of a particular specialization (type) may not point to objects of a different specialization (type)

```
AA aa_512 = new();                  // Create a default specialization of generic class AA
AA #(.size(256)) my_aa = new();     // Create a specialization of generic class AA (a new type)
initial begin
//  my_aa = aa_512;                 // Illegal - my_aa and aa_512 are different types
```

Questa runtime output from above code:
** Fatal: Illegal assignment to object of class AA__1 from object of class AA__2

Parameterized Classes

- Constructors allow for customization of objects at run time
- Parameters allow for customization of objects at compile time
- Parameters can be value parameters or type paramters
- Parameterization of a value:

```
class AA #( int size = 512 );
  byte payload [size];
endclass
AA #(.size(256)) my_aa = new();      // my_aa has array size of 256
```

size substituted by the value 512 at compile time

- Parameterization of a type:

```
class BB #( type T = int );
  T payload [512];
endclass
BB bb_int = new();                   // object containing 512 ints
BB #(string) bb_string = new();      // object containing 512 strings
```

T substituted by the type int at compile time

- Multiple parameters:

```
class CC #( type T = int, int size = 512 );
  T payload [size];
endclass
CC #(.T(integer), .size(1024)) cc_handle = new();  // type integer, size 1024
```

Parameterized Classes: Inheritance

- Parameterized classes may be inherited

```
class D_base #( type T = int );
  T payload [512];
endclass
```

Generic class

T is int

```
//class D_1 derived from default specialization of D_base
class D_1 extends D_base ;   endclass
D_1 D_1_handle = new();
```

T is string

```
//class D_2 derived from specialization of D_base
class D_2 extends D_base #(.T(string) );   endclass
D_2 D_2_handle = new();
```

T is int
R is bit

```
//generic class D_3 derived from default specialization of D_base
class D_3 #( type R = bit ) extends D_base;  endclass
D_3  D_3_handle = new();
```

T is R
which is
byte

```
//generic class D_4 derived from specialization of D_base
class D_4 #( type R = bit ) extends D_base #(.T(R));
endclass
D_4 #(.R(byte))  D_4_handle = new();
```

T is R

Polymorphism

- Consider a base class that is multiply-derived into sub-classes, each of which over-rides a common "virtual" base method
- Polymorphism allows that any derived-class object referenced via the base class will invoke the appropriate method without any special programmer intervention.

```
class Display;
integer v;

  virtual task Print();
     $display("v (dec): ", v);
  endtask

endclass
```

Base class is simply used for its handle… …never instantiated…

```
class HexDisplay extends Display ;

    task Print(); // over-ridden method
      $displayh("v (hex) : ",v);
    endtask

  endclass
```

```
class OctDisplay extends Display ;

   task Print(); // over-ridden method
     $displayo("v (oct) : ",v);
   endtask

endclass
```

```
module poly5;

HexDisplay hx = new();
OctDisplay  oc = new();
Display poly;

initial begin
  hx.v = `habcd;
  oc.v = `habcd;

  poly = hx;
  poly.Print();

  poly = oc;
  poly.Print();
end

endmodule
```

Simulation output

```
v (hex) : 0000abcd
v (oct) : 00000125715
```

Virtual Classes – Class Interfaces

- To define a class interface (or API) create a virtual (or abstract) class
  - When a class is declared as virtual it cannot be instantiated, only inherited
  - Methods defined in the class do not need an implementation – only a prototype – typically overridden in a derived class (see pure virtual slide)
    - This provides the API or interface definition

```
virtual class Packet ;

  virtual function bit CRC();
  endfunction

endclass
```

Virtual classes CANNOT be instantiated

Only a virtual class allows methods without an implementation

Virtual function in an abstract class. Default implementation provided… do nothing

Override base class function

```
class Ether_Packet  extends Packet;
    …

  // methods
  function bit CRC();
    $display("Ethernet CRC");
    return(1);
    // Verify CRC according to
    // 802.3 (Ethernet) standard
  endfunction
endclass
```

```
class Token_Packet  extends Packet;
    …

  // methods
  function bit CRC();
      $display("Token-ring CRC");
      return(1);
    // Verify CRC according to
    // 802.5 (Token-ring) standard
  endfunction
endclass
```

Virtual Classes - 2

```
class CRC_checker;
  integer idx = 0;
  Packet  pkt_array[512]; // Array of ANY kind of packet

  function void add_to_array(Packet p);
    pkt_array[idx] = p;
    idx++;
  endfunction

  function void check_CRC();
    for (i = 0; i <  512; i++; )
      begin
        if (!pkt_array[i].CRC())
          $display ("CRC Error");
      end
  endfunction
endclass
```

Arrays and other containers like lists, etc. usually require that all elements be of the same type.

Inheritance allows either an Ether_Packet or a Token_Packet to be passed in as an argument and stored into the array.

Polymorphism means that regardless of which type of packet is stored in the array, the CORRECT CRC function will be called.

What's the big deal about polymorphism?

It allows for more generic (i.e. reusable) code, where multiple alternative method definitions in derived classes can be dynamically bound at run time via a variable of the base (or super) class.

Pure Virtual Methods

- A pure virtual method is a method declared as a prototype only.
- Any derived class MUST provide an implementation

Pure virtual methods MUST be overridden within a deriving class, or a compile error will be triggered

```
virtual class Packet ;

  pure virtual function bit NoDefault();

  virtual function bit Do_Nothing_by_Default()
  endfunction

endclass
```

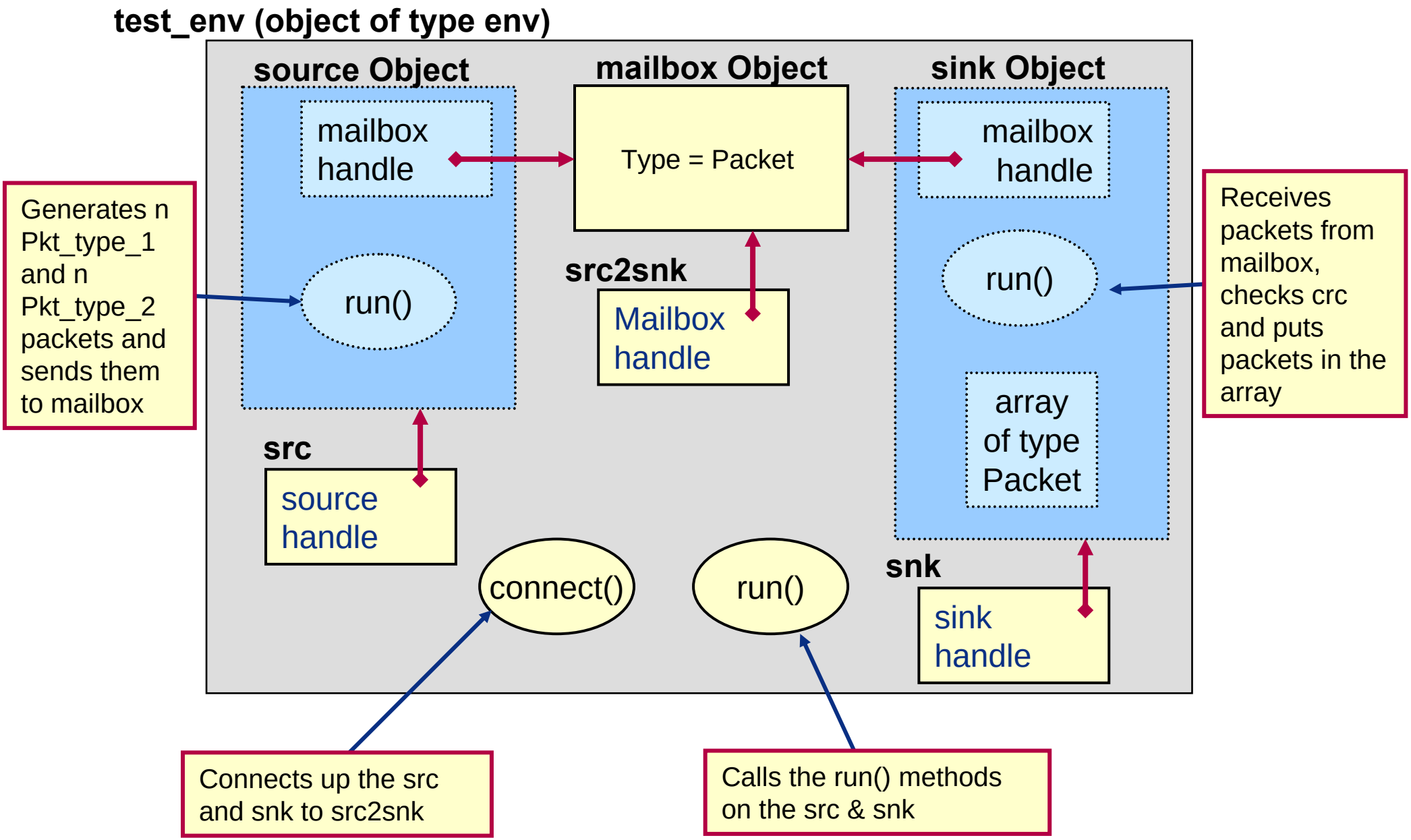
Empty virtual methods are also allowed.
However, in a deriving class, these are interpreted as having a default implementation – DO NOTHING!

Lab – Polymorph: Introduction

- This lab challenges you to write a simple polymorphic system.
- The system sends packets from a source object to a sink object through a mailbox and is structured similar to previous examples in the course.
- The type of the mailbox connecting the source and the sink will be of the (provided) base packet class
  - This base ***Packet()*** class is declared virtual. (Cannot be instantiated)
- The packets that are generated and sent by the source and received by the sink will be of the two new types which you will write, derived from the base packet type. These two new types differ in how they generate crc check fields.
- The base class mailbox will carry only derived class packets and every packet received by the sink object must be crc-checked. The check_crc() function will be called polymorphically.

- Continued on next page -

Lab – Polymorph: Lab Architecture



Lab – Polymorph: Instructions – 1

- Lab directory: **polymorph**
- Edit the file packet_types_pkg.sv
  - Extend the virtual class Packet() to create two new derived classes **`Pkt_type_1()`** and **`Pkt_type_2()`**
    - Each will provide an implementation for the virtual functions **`gen_crc()`** and **`check_crc()`**
      - **`gen_crc()`** creates a crc per Pkt_type as follows:
        - » **`Pkt_type_1`**: crc = sum of payload array
        - » **`Pkt_type_2`**: crc = product of payload array
      - **`check_crc()`** verifies the received crc of the packet

Lab – Polymorph: Instructions – 2

- Edit the file type_pkg.sv
  - Create the class ***source()***, which features:
    1. A mailbox of the base ***Packet()*** class called `out_chan`
    2. A custom constructor that has two arguments of type `int`
       - Number of each type of derived Packet to send to the sink (type_1,type_2)
    3. A task `run()` which
       - Generates the requested number of packets (type_1 and type_2)
       - Writes the two packet streams simultaneously to the mailbox
       - Uses a base class pointer to write the derived class packets to the mailbox

Lab – Polymorph: Instructions – 3

- Continue editing the file type_pkg.sv

  - Complete the class ***sink()*** which features:

    1. A mailbox of the base ***Packet()*** class called **`in_chan`**
    2. A custom constructor that has argument of type **`int`**
       - Total number of packets to receive (type_1 + type_2 streams)
    3. An associative array of the base class type indexed by the **`pkt_id`**
    4. A task **`run()`** which
       - Uses a base class pointer to read the derived class packets from the mailbox
       - Checks the **`crc`** of the received packets (use **`check_crc()`**)
       - Places good packets into the associative array indexed by the **`pkt_id`**
       - Prints messages to indicate the status of each packet received

- Continued on next page -

Lab – Polymorph: top, env

```
module top();
  import types_pkg::*;
  env test_env = new();
  initial begin
    test_env.connect();
    test_env.run();
  end
endmodule
```

```
class env;
  // create  channel between source & sink
  mailbox #(Packet) src2snk = new();
  source src = new(5,5);   // create source obj
                           // send 5 of each type of Packet
  sink snk = new(10);      // create sink obj - receive 10 Packets

  function void connect();
    src.out_chan = src2snk;  //connect up src to mailbox
    snk.in_chan = src2snk;   //connect up snk to mailbox
  endfunction

  task automatic run();
    fork
      snk.run();  // start up sink
      src.run();  // start up source
    join_none
  endtask
endclass :env
```

Lab – Polymorph: Packet_types_pkg.sv

```
virtual class Packet ;
  local byte unsigned payload[];
  local byte unsigned crc;
  int pkt_id;
  static int num_pkts = 1;

  function new();
    pkt_id = num_pkts++;
  endfunction

  virtual function void gen_crc();  endfunction
  virtual function bit check_crc(); endfunction

  function void print_payload();
    for (int i=0; i<payload.size(); i++)
      $display(payload[i]);
  endfunction

  function void init_pkt(int sz);
    payload = new[sz];
    for (int i = 0; i<sz; i++)
      payload[i] = $random() % 256;
    gen_crc();
//    crc++; // insert error by un commenting this line
  endfunction
endclass
```

call gen_crc in the derived class

payload values randomized

Lab – Polymorph: Sample Output

```
VSIM 1> run -all
# source: Sent packet, id = 1
# source: Sent packet, id = 2
# source: Sent packet, id = 3
# source: Sent packet, id = 4
# source: Sent packet, id = 5
# sink: Received a good packet, id = 1
# sink: Received a good packet, id = 2
# sink: Received a good packet, id = 3
# sink: Received a good packet, id = 4
# sink: Received a good packet, id = 5
# source: Sent packet, id = 6
# source: Sent packet, id = 7
# source: Sent packet, id = 8
# source: Sent packet, id = 9
# source: Sent packet, id = 10
# sink: Received a good packet, id = 6
# sink: Received a good packet, id = 7
# sink: Received a good packet, id = 8
# sink: Received a good packet, id = 9
# sink: Received a good packet, id = 10
```

Your output may look different - why?

Sol

Randomization & Constraints

In this section

SystemVerilog

- Stimulus Generation Methodologies
- Constraint blocks
- Randomize
- Random sequences
- Random Number Generation
- Scoreboarding

Stimulus Methodologies

- 3 common means of generating stimulus
  - **Exhaustive stimulus**
    - Usually algorithm-driven based on implementation
    - Breaks down on complex designs - impracticably large testspace
    - Highly redundant by definition
  - **Directed stimulus**
    - Works for well-understood or simple designs
    - Reduces redundancy but still very hard to get 100% coverage
  - **Random stimulus**
    - Usually transaction based (well defined transactions => good coverage)
    - Spots corner cases humans couldn't predict
    - Highly redundant for unconstrained random numbers

Random Testing

- Concept:
  - Define the set of transactions needed for verification
  - Randomize transaction parameters to achieve coverage
- Theory:
  - Random testing will find all bugs identified by other means and more
  - Beware: While this is theoretically true, it is also misleading
    - Can lead to sloppy no-thought testing
    - May require very long runs to achieve coverage
- Best Strategy:
  - **Directed Randomization**
    - Steer random paths by means of probabilities (weights)
    - Weigh interesting cases more than uninteresting ones
    - Initialize the system to interesting states before randomization

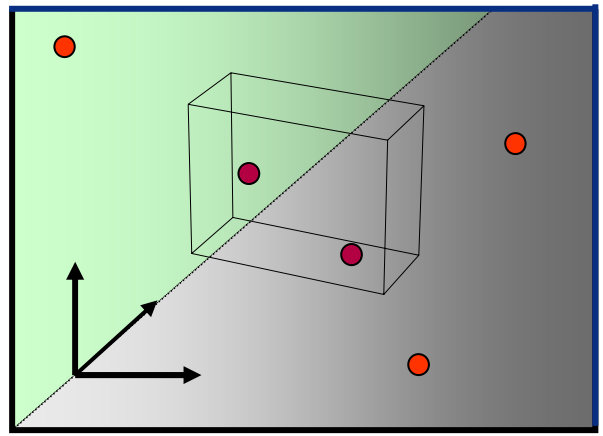
Directed Random Testing

- **True random testing**
  - Rather a blunt instrument, generates illegal as well as legal conditions
  - 100% coverage is achievable in theory but how long will it take?
  - May be a high level of redundancy

- **Directed random testing**
  - More focused approach, avoids illegal/uninteresting conditions
  - Steer random choices with constraints and weighted probabilities
  - Target rare or low-likelihood conditions by assigning higher weight
  - Challenge is to constrain (focus) without limiting breadth of coverage
  - More complex to write, harder to debug
  - Tracking of test-space covered is VITAL

Randomness

- **True random numbers**
  - Unpredictable sequence of random numbers
  - In use, generates legal & illegal conditions
  - Failing tests are not repeatable
  - Usually harder to debug



Random vector "miss"

Random vector "hit"

- **Pseudo random**
  - Generate a predictable series of highly random numbers
  - Reuse the sequence in regression tests
  - Reapply the same sequence in different tests

- **Constrained random**
  - Start with true or pseudo random number streams
  - Constraints use system knowledge to remove illegal numbers

- **Most methodologies mix/match all 3 types as needed**

Object Based Randomization

- SV randomization capabilities are object (class/endclass) based
  - Objects are ideal for aggregate structures like MP3 frames or Ethernet packets
  - Objects may contain variable declarations and constraint definitions

```
class bus;
  randc bit[15:0] addr;
  rand bit[31:0] data;

  constraint word_aligned {addr[1:0] == 2'b00;}

endclass
-----------------------------------
bus mybus = new();

initial
  repeat(50) begin
    if ( mybus.randomize() == 1 )
        $display ("addr: %16h , data: %h\n", mybus.addr, mybus.data);
    else
        $display ("Randomization failed");
  end
```

Object containing variables to be randomized

constraint block "bounds" the randomization

Type modifiers indicate randomizable Variables (more on next slide)

object

Builtin method to trigger randomization (more later)

`rand`, `randc` Type Modifiers

- SV identifies class variables as randomizable by means of two type modifiers:
  - `rand` – standard random values uniformly distributed (may repeat)

    rand bit[7:0] len;
  - `randc` - random "cyclic" values cover all possible values before repeating

    randc bit[7:0] cyc;
- Type modifiers may be applied to any integral types:
  - Individual variables
  - Static arrays, associative arrays
    - all elements are randomized
  - Dynamic arrays
    - All elements randomized
    - Array size randomized if size is constrained

NOTE
`randc` variables are assigned values before `rand` variables.
Constraining a `randc` variable with a rand property is illegal

Constraint Block

- A constraint block is a class member that controls randomization of the random variables in that class.

```
class ex1;
  rand int a,b;

  constraint con1 { a > b; }
endclass
```

Expression(s)

Note the syntax!!!

Constraint name

- Expressions within a constraint are legal SV code but:
  - Operators with side-effects (++N, N--) are not allowed
  - Use of functions is limited (automatic, no outputs or side-effects, etc.)
  - SV set membership syntax is supported
- In addition, some special constructs are provided (to be discussed shortly):

`dist`	- distribution/weights
`->`	- implication
`if…else`	- alternate style of implication

Constraint Block Examples

```
bit [31:0] f;

class ex2;
  rand int a,b,c,d,e;

  constraint con_ab { a > b; a < 256; (a - b) >= 64; }

  constraint con_c { c inside { [1:5], 12, [(a-b):(a+b)] }; }

  integer primes[0:4] = '{2,3,5,7,11};
  constraint con_d { d inside primes;  }

  function automatic int count_ones ( bit [31:0] w );
     for( count_ones = 0; w != 0; w = w >> 1 )
          count_ones += w & 1'b1;
  endfunction

  constraint con_ef { e == count_ones( f ) ; }

endclass
```

Multiple constraints are "anded" together

Implicitly: a,b must be solved before c

Functions are called before constraints are solved and their return values act like state variables

HINT: Use function for parity generation etc.

Constraint Block : Overriding

- Overriding is a major benefit of the OO nature of constraints:

```
class ex1;
  rand int a,b;

  function new();
    …
  endfunction

  constraint con1 { a >10;
}

endclass
```

```
class ex2 extends ex1;

  constraint con1 { a != 0; }
  constraint con2 { b < 30; }

endclass
```

Overridden constraint

New "additional" constraint

- This allows constraints to be reused and modified without editing the "base" set
  - Enhance a test without rewriting (breaking) the original classes
  - Reuse methods with different stimulus

Constraint Block : Iteration

- The **foreach** construct supports iteration through an array
  - Array may be multidimensional
  - Loop variables
    - Each corresponds to a dimension of the array
    - Type is implicitly declared – same as array index
    - Scope is the **foreach** construct

C1 constrains each element of the array A to be in the set [2,4,8,16].

```
class C;
  rand byte A[ ] = new[6] ;
  constraint C1 { foreach ( A [ i ] ) A[i] inside {2,4,8,16}; }
  constraint C2 { foreach ( A [ j ] ) A[j] > 2 * j; }
endclass
```

loop variable

C2 constrains each element of the array A to be greater than twice its index.

Dynamic Constraint Changes

- In the test system, randomized constraints aren't fixed static things
  - Need to change constraints:
    - for better design-space coverage
    - for end-case coverage
- Modifying constraints as a test runs is called "Dynamic Constraint Modification"
  - Fundamental to Reactive Testbenches (more later)
  - Quick easy way to create new stimulus and (effectively) new test cases
- It's generally better to modify constraints dynamically than to rewrite, recompile, and rerun
  - Alternative is to extend base class, instantiate and write code to use it

Dynamic Constraint Changes in SV

- SV mechanisms provided to achieve this:
  - `implication` and `if-else` within constraint blocks
  - Change `dist` weights in constraint blocks
  - `constraint_mode()` to turn on/off constraints
  - `rand_mode()` to turn on/off random variables

Constraint Block : Implication (`->, if else`)

- Implication is a way to declare conditional relationships

```
rand int w,x;
constraint con_wx
        { (w == 0) -> (x == 0) ;}
```

```
rand int w,x;
constraint con_wx { if (w == 0)
                        x == 0;   }
```

If w is zero, x will also be zero.

Q: What if w is non-zero? **A:** ……

```
rand int z;
constraint con_mode_z {
  mode == sm  -> {z < 10; z > 2;}
  mode == big -> z > 50;
}
```

```
rand int z;
constraint con_mode_z {
  if (mode == sm)
   {z < 10; z > 2;}
  else if (mode == big)
   z > 50;
}
```

If mode is sm, z will be less than 10 and greater than 2, if mode is big, z will exceed 50

Q: What about other modes? **A:** ……

Constraint Block: dist -1

- **dist** is a test for set membership
  - *Optional* weights (statistical distribution)

```
class test_dist;
rand int x;
int c = 1;
constraint con_x {
      x dist { [1:2] := c , 3 := 3 , 4 := 5 };
}
endclass

module dist1;
test_dist td = new();
bit rr;
int result[6];
initial begin
 for (int i=0; i<1000; i++) begin
   rr = td.randomize();
   result[td.x]++;
 end
 for (int i=1; i<5; i++)
   $display("%0d count = %0d",i,result[i]);
end
endmodule
```

Statistical weights

Weight is assigned to each value across range

```
Output:
1 count = 99
2 count = 111
3 count = 288
4 count = 502
```

Constraint Block: dist -2

```
class test_dist;
rand int x;
constraint con_x {
      x dist { [1:4] :/ 2 , 5 := 3 , 6 := 4, 7 };
}
endclass

module dist2;
test_dist td = new();
bit rr;
int result[9];
initial begin
 for (int i=0; i<1000; i++) begin
   rr = td.randomize();
   result[td.x]++;
 end
 for (int i=1; i<8; i++)
   $display("%0d count = %0d",i,result[i]);
end
endmodule
```

No weight: defaults to :=1

Weight is divided equally across range

```
Output:
1 count = 54
2 count = 54
3 count = 45
4 count = 57
5 count = 288
6 count = 397
7 count = 105
```

randomize()

- Randomize is a built-in virtual function that generates new random values for all active variables in an object. It *CANNOT* be overridden

`virtual function int randomize();`

  - returns 1 if successful, otherwise 0

Randomize() works within constraints

```
class bus;
  randc bit[15:0] addr;
  rand   bit[31:0] data;

  constraint word_aligned {addr[1:0] == 2'b00;}

endclass

  -.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-

bus mybus = new();

initial
  repeat(50) begin
    if ( mybus.randomize() == 1 )
        $display (“addr: %16h , data: %h\n”, mybus.addr, mybus.data);
    else
        $display (“Randomization failed”);
  end
```

Return value: 0 if solver fails to satisfy constraints

`randomize() with`

- **`randomize()with`** is a way to add constraints in-line.
  - In-line constraints act in parallel with embedded constraint blocks.

```
class Sum;
  rand bit[7:0] x,y,z;

  constraint con { z == x+y; }
endclass

-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-


bit [7:0] a,b;

task MyTask (Sum n );
    int succeeded;
    succeeded = n.randomize() with { x < y;  z >= b; };
endtask
```

Same rules apply

Object variables

Local variables

randomize() Inline Control

- So far `randomize()` has always been shown with no parameters.
  - The optional parameter list determines the variables to be randomized
    - *Regardless* of how they were declared.
- Ideal for randomizing the state variables to find end-cases, etc.

```
class Xmit;
   rand byte limit, length;
   byte max, min;

   constraint c1 {(length<max) && (limit>min);}
endclass
-----------------------------------------------------
Xmit x1 = new();
x1.randomize();                 //   limit, length      max, min
x1.randomize(limit);            //   limit              length, max, min
x1.randomize(max,min);          //   max, min           limit, length
x1.randomize(max, limit);       //   max, limit         length, min
```

NOTE: Only variables specifically declared `randc` are cyclic

Non-OO Randomization

- So far randomization has been described as object-based
- There is also a randomization protocol which does not require class(es)
- `std::randomize()` allows randomization of data within the current scope

OO Example

```
class stimc;
   rand bit [15:0] addr;
   rand bit [31:0] data;
   rand bit rd_wr;
   constraint c { addr < 1024; }
endclass
-----------------------------------

function bit gen_stim( stimc p );
  bit success;
  success = p.randomize();
  return p.rd_wr;
endfunction
```

Equivalent code

Non-OO Example

```
module stim;
   bit [15:0] addr;
   bit [31:0] data;
   bit rd_wr;

function bit gen_stim();
   bit success;
   success = randomize( addr, data,
                        rd_wr )
             with { addr < 1024 ; };
   return rd_wr ;
endfunction
...
endmodule
```

Call std::randomize()

Note syntax!

Random Variable Control

- Each object (or random variable within it) can be made active/inactive by the method **rand_mode()**

- Task Prototype

  task object.rand_mode(bit mode)

  - **mode** = 0 means inactive, **randomize()** ignores variable
  - **mode** = 1 means active, variable is **randomize()** –able

- Function Prototype

  function int object.rand_mode( )

  - returns the mode

```
class Packet;
  rand int x,y;
  int k;
  constraint con_x { x <= y; }
endclass
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-
Packet p = new();
int stat_y;
initial begin
  p.rand_mode(0);
  p.y.rand_mode(1);
  stat_y = p.y.rand_mode();
//  p.k.rand_mode(1);   // illegal !  no rand_mode for k
end
```

Set p object rand_mode to inactive

Set p.y rand_mode to active

Check active mode of p.y

Constraint Control

- Each object containing constraints also supports a method `constraint_mode()`
  - Allows constraints to be turned on/off at will. An inactive constraint will not affect randomization

- Task Prototype

  `task object.constraint_mode(bit mode)`
  - `mode` = 0 means inactive, `randomize()` ignores constraint
  - `mode` = 1 means active, constraint affects `randomize()`

- Function Prototype

  `function int object.constraint_mode( )`
  - returns the mode

```
class Packet;
  rand int x;
  constraint con_x { x dist { [1:4] := 1, [5:9] := 2, 10 := 3 }; }
endclass
   ---------------------------------------------------------------
function int toggle_con_x ( Packet p );
   if ( p.con_x.constraint_mode() )
      p.con_x.constraint_mode(0);
   else p.con_x.constraint_mode(1) ;

   return( p.randomize() );
endfunction
```

Pre and Post Randomization

- All classes provide internal `pre_randomize()` and `post_randomize()` methods
  - Automatically called before/after `randomize()`
  - May be overridden to handle multi-part randomization

Prototypes

```
function void pre_randomize();
function void post_randomize();
```

HINT: Use `pre_randomize()` for initialization
Use `post_randomize()` for cleanup, diagnostics, etc.

NOTE: To use either of these in a derived class it is compulsory to add a call to its corresponding parent class method.

```
module post_randomize;
class pkt;
  rand bit [7:0] addr;
  bit [7:0] last_addr;
  constraint not_equal {
    addr != last_addr;
  }
  function void post_randomize();
    last_addr = addr;
  endfunction
endclass

pkt p;

initial begin
 p = new();
 for (int i = 0; i<20; i++) begin
   if(p.randomize() ==1);
     $display("addr = %0d",p.addr);
 end
end
endmodule
```

Random Case

- The features discussed so far allow flexible randomization of variables
- But what about decision making?
  - `randcase` is a case statement that randomly selects one of its branches

```
module rand_case();
int results[3];
int w = 3;
initial begin
  for(int i=0; i<1000; i++)
    randcase
      1:  results[0]++;
      w: results[1]++;
      6:  results[2]++;
    endcase
  for (int i=0; i<3; i++)
    $display("%0d count = %0d",i,results[i]);
end
endmodule
```

Weights are non-negative
May be a variable

```
Output:
0 count = 93
1 count = 310
2 count = 597
```

A weight of zero (0) prevents that branch from ever happening

Random Sequences

- Derived from theory behind language parsers **randsequence**
  - Generates randomized sequences for test
  - Optional weights can be added

```
module randseq_1();
initial
  for (int i=0; i<10; i++)
    randsequence( main )
      main     :  first second done;
      first    :  add | dec  ;
      second   :  pop | push ;
      done     :  { $display("done");} ;
      add      :  { $write("add ");  } ;
      dec      :  { $write("dec ");  } ;
      pop      :  { $write("pop  "); } ;
      push     :  { $write("push "); } ;
    endsequence
endmodule
```

Start-point

production

Unchangeable sequence

"non-terminals"

Random choice: a | b

"terminals"

```
Output:
# dec push done
# add push done
# dec pop  done
# dec pop  done
# dec pop  done
# dec push done
# dec push done
# dec pop  done
# add pop  done
# dec push done
```

Weighted Random Sequences

- Optional weights allow statistical control over the choices within a sequence

```
module randseq_2();

int wt = 2;
initial
  for (int i=0; i<10; i++)
    randsequence( main )
      main     :  first second done;
      first    :  add := 1 | dec := wt;
      second   :  pop | push ;
      done     :  { $display("done"); } ;
      add      :  { $write("add ");  } ;
      dec      :  { $write("dec ");  } ;
      pop      :  { $write("pop  "); } ;
      push     :  { $write("push "); } ;
    endsequence
endmodule
```

Weighted random choice

```
Output:
# dec push done
# dec push done
# add pop  done
# dec pop  done
# dec pop  done
# dec push done
# dec push done
# add pop  done
# add pop  done
# dec push done
```

Random Sequence Conditionals

- Insert conditionals in a production by means of **`if-else`** or **`case`**

```
module randseq_3();
int wt = 2;  bit[1:0] status = 0;  bit up = 0;
initial
  for (int i=0; i<10; i++) begin
    status++;  up++;
    randsequence( main )
      main:  first second third;
      first: case (status)
                 0,1:  done;
                 2:    third;
                 default: illeg;
             endcase ;
      second: add := 1 | dec := wt;
      third: if (up) pop else push ;

      done : { $display("done"); } ;
      add  : { $write("add ");  } ;
      dec  : { $write("dec ");  } ;
      pop  : { $write("pop  "); } ;
      push : { $write("push "); } ;
      illeg: { $display("Illegal Status"); } ;
    endsequence
  end
endmodule
```

Case-Conditional (default is optional)

If-Conditional (else is optional)

```
# done
# dec pop  push dec push Illegal Status
# dec pop  done
# dec push done
# add pop  push dec push Illegal Status
# dec pop  done
# dec push done
# dec pop  push dec push
```

Random Sequence Jumps

- Insert jumps in a production by means of **break** or **return**

```
module randseq_4();
int status;
initial
  for (int i=0; i<10; i++) begin
    status = i;
    randsequence( main )
      main   : first second third;
      first  : add := 1 | dec := 2;
      second: { if (status == 2)
                    begin $write("RETURN "); return;
                    end } pop | push;
      third  : { if (status == 3)
                    begin $display("BREAK" ); break;
                    end} done  ;
      done   : { $display("done"); } ;
      add    : { $write("add ");  } ;
      dec    : { $write("dec ");  } ;
      pop    : { $write("pop  "); } ;
      push   : { $write("push "); } ;
    endsequence
end
endmodule
```

Return says quit the current production (don't do pop or push, i.e. goto third)

Break says exit the randsequence

```
# dec push done
# dec push done
# add RETURN done
# dec pop  BREAK
# dec pop  done
# dec push done
# dec push done
# add pop  done
# add pop  done
# dec push done
```

randsequence Example (1)


```
forever
  begin : loop     // Rand. seq. of data & 'noise' eth packets and Inter-Frame-Gaps
    randsequence ( traffic )

      traffic : ifg frame ;

        frame : data := set.frame.data    |
                noise:= set.frame.noise ;   // ratio of data to 'noise'

         data : {
             randcase  // Weighted random choice of packet size
                 set.dis.n64    :  siz = 64;
                 set.dis.n244   :  siz = 244;
                 set.dis.n428   :  siz = 428;
                 set.dis.n608   :  siz = 608;
                 set.dis.n792   :  siz = 792;
                 set.dis.n972   :  siz = 972;
                 set.dis.n1156  :  siz = 1156;
                 set.dis.n1340  :  siz = 1340;
                 set.dis.n1500  :  siz = 1500;
             endcase
             epkt.build_data_frame(siz, rfile);
             epkt.drive_MRx;        // Call of transactor method
         };
```

```
typedef struct {
    int n64;
    int n244;
    int n428;
    int n608;
    int n792;
    int n972;
    int n1156;
    int n1340;
    int n1500;
} distro;

typedef struct {
    int data;
    int noise;
} frm;

typedef struct {
    int normal;
    int long;
    int short;
    int random;
} gap;

typedef struct {
    frm frame;
    distro dis;
    gap ifg;
} stream_set;

stream_set set;
```

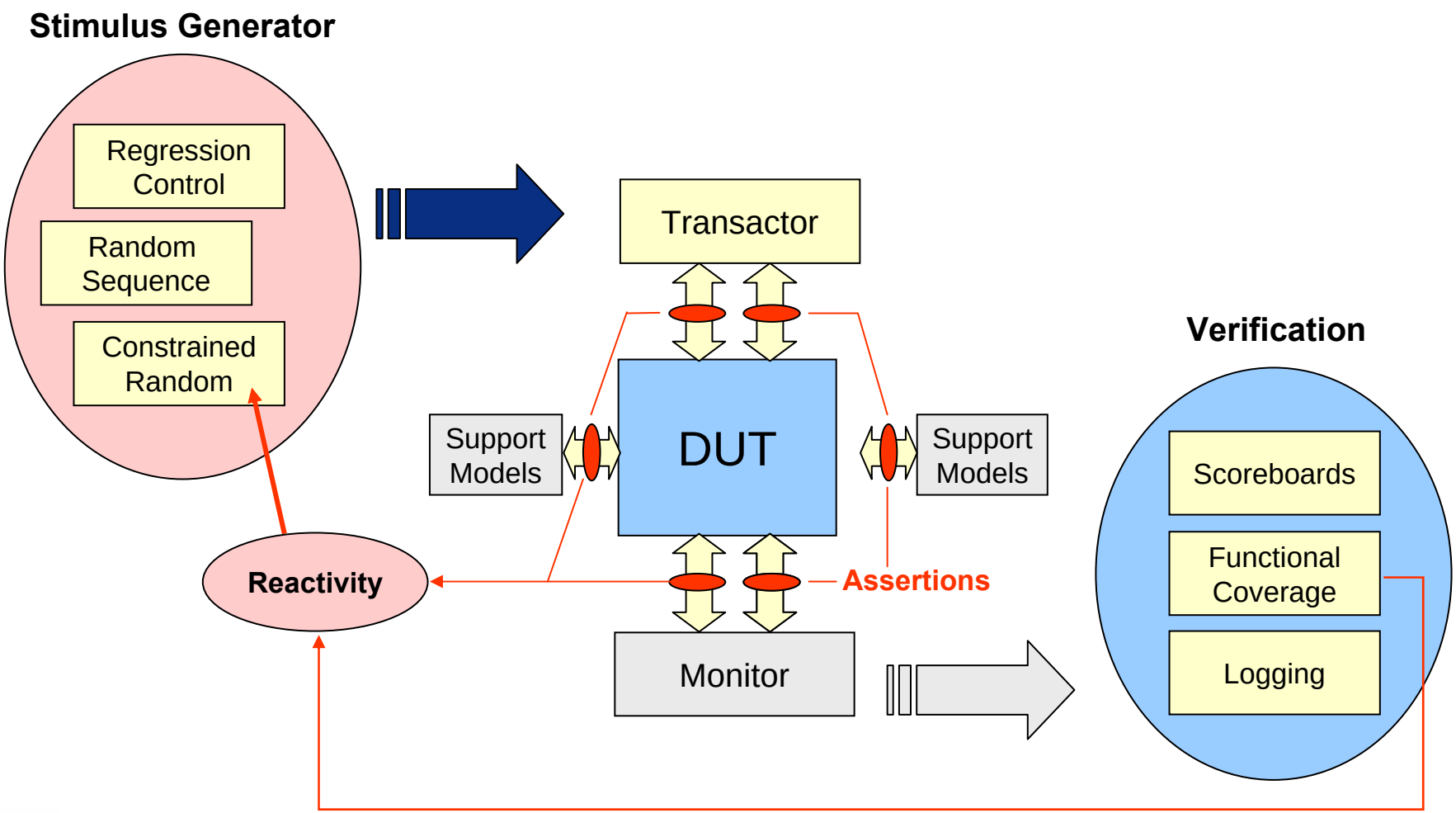
randsequence Example (2)

```
noise : {
          nul = epkt.randomize();
          epkt.drive_MRx;
         };

ifg : normal := set.ifg.normal |
      long   := set.ifg.long   |
      short  := set.ifg.short  |
      random := set.ifg.random ;

normal : {
         repeat (38)  @(posedge mrx_clk);
         };

long :   {
         repeat (100) @(posedge mrx_clk);
         };

short:   {
         repeat (5)   @(posedge mrx_clk);
         };

random : {
         repeat ( $urandom_range(10,100)) @(posedge mrx_clk);
         };
endsequence
```

```
typedef struct {
     int n64;
     int n244;
     int n428;
     int n608;
     int n792;
     int n972;
     int n1156;
     int n1340;
     int n1500;
 } distro;

 typedef struct {
     int data;
     int noise;
 } frm;

 typedef struct {
     int normal;
     int long;
     int short;
     int random;
 } gap;

 typedef struct {
     frm frame;
     distro dis;
     gap ifg;
 } stream_set;

 stream_set set;
```

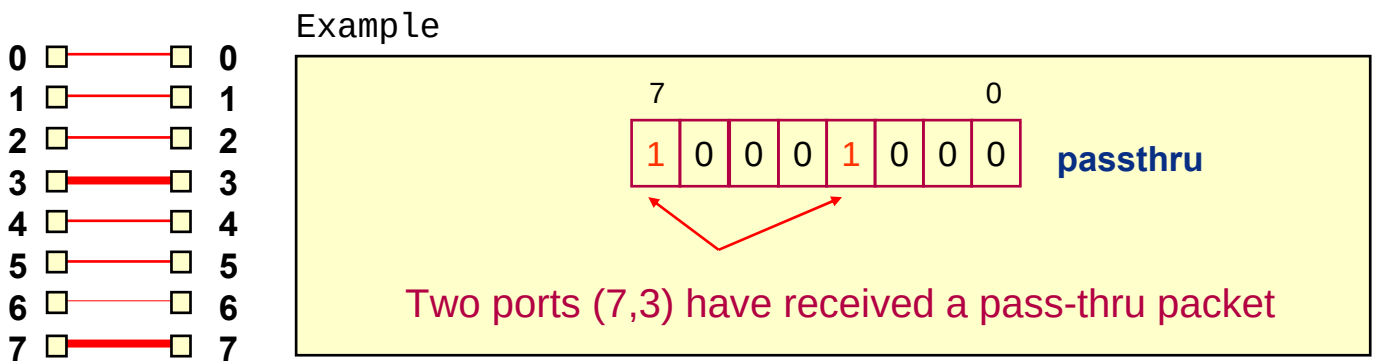
Reactive Randomization

- Reactivity refers to the use of feedback from some realtime DUT analysis (typically SVA or Functional Coverage) to control/refine the randomization
- We will discuss this in more detail in upcoming sections.



Lab - Randomization : Introduction

- This lab is a further development of the router testbench from earlier labs
- One change is the addition of some SV assertions to detect when a packet is passed straight-thru the router (source port # == dest port #)
  - The SVA code updates a register in pkg defs with feedback information
  - In pkg defs there is an 8-bit register called **passthru**
    - Each bit of this register represents whether the corresponding output port of the DUT has received an input packet from its opposite input port #



- Initially, the testbench will be configured to constrain randomization so that all packets are pass-thru (srce == dest) until such time as all ports have been targeted (register passthru == 255)
- At that time, the testbench will dynamically modify itself so that pass-thru packets are no longer generated and randomization will continue until end of simulation

Lab - Randomization : Instructions - 1

- Lab directory: **Router/random**
- Overview:
  - During this lab, you will complete the router testbench by editing and extending the BasePacket class to support reactive randomization.
- Edit the file **defs.sv**
  - In base class **BasePacket**
    - Add rand qualifiers to properties:
      - dest
      - payload
    - Add a constraint block to limit *dest* to legal values
    - Add a constraint to limit the size of *payload* (a dynamic array) to 1 minimum, 4 maximum
  - In derived class **Packet**:
    - Add a constraint to force all pass-thru (plus logic to disable it)
- Note in the file defs.sv
  - bit [7:0] passthru;
  - bit pt_mode; // pass thru mode flag (set to 1 at start up) for optional use

Lab - Randomization : Instructions - 2

Verilog 1995 style randomization

```
pkt2send.dest = ($random() % 8);
sz =  (($random() % 4) + 1);
pkt2send.payload = new[sz];
for(int i=0; i<$size(pkt2send.payload); i++)
  pkt2send.payload[i] = ($random() % 256);
```

- Edit the file **test_router.sv**
  - In class base_scoreboard, completete the task **`run1()`**
    - Note: initially, all random packets are pass-thru
    - Modify so when all passthru's have been tested (passthru == 8'hFF)
      - Display a statement to that effect (may want to stop simulation so you can see it!)
  - In class stimulus, rewrite the task **`run()`**
    - Remove the simple Verilog 1995 randomization code
    - Add a call to pkt2send.randomize()
  - Compile and run by typing:
    make
    -or- make gui

What's missing?
Some way to measure when we're finished
This version just declares a max # of packets (500)
Coming Up: Functional coverage

Sol

Constraint Solve Order

Reference Page

- By default, there is no "order" to solving constraints - all variables in constraint expressions are solved simultaneously.
- You can impose a solve order, which can affect the probability density of the result (but <u>does not</u> affect the solution space)

```
class ex1;
  rand bit[1:0] a;
  rand bit b;
  constraint con1 { (b==0) -> (a==0);
}
endclass
```

```
randomize()
```

a	b	Prob
0	1	.2
1	1	.2
2	1	.2
3	1	.2
0	0	.2

```
randomize() with
    { solve a before b; }
```

a	b	Prob
0	1	.125
1	1	.25
2	1	.25
3	1	.25
0	0	.125

```
randomize() with
   { solve b before a; }
```

a	b	Prob
0	1	.125
1	1	.125
2	1	.125
3	1	.125
0	0	.5

Custom Randomization

Reference Page

- Even if no class members are tagged **`rand`** or **`randc`**, SystemVerilog will still call the **`pre_randomize()`** function when you call **`randomize()`**.
- You can use this to gain complete control over the randomization of variables
  - E.g. You can use different distributions such as (from Verilog 2001)
    - `$dist_normal( seed, mean, std_deviation )`
    - `$dist_exponential( seed, mean )`
    - `$dist_poisson( seed, mean )`
    - `$dist_chi_square( seed, degree_of_freedom )`
    - `$dist_t( seed, degree_of_freedom )`
    - `$dist_erlang( seed, k_stage, mean )`
    - `$dist_uniform( seed, start, end )`

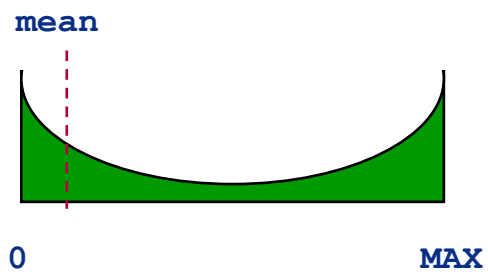
All arguments are integer values.

seed is an inout variable that is modified and reused by the algorithms

Custom Randomization Example

Reference Page

- It is often useful to create a distribution that favors extreme values over middle ones. This would be a U-shaped, or so-called "bathtub" distribution.




```
class Bathtub;
  int val;
  int seed = 1;
  int mean, MAX;

  function new(int mn = 10, int mx = 100);
    mean = mn ; MAX = mx;
  endfunction

  function void pre_randomize();
    val = $dist_exponential(seed,mean);
    val = (val > MAX) ? MAX : val;
    if ($urandom_range(1)) val = MAX - val;
  endfunction
endclass


bathtub b = new();


  b.randomize();          // Calls pre-randomize()
```

Generate a number on the "left" side of the U

50% of the time, mirror it to create the "right" side of the U

Stability – Why is it Important?

Reference Page

- Consider this scenario:
  - A bug is found during a long simulation run using constrained random tests
  - Thankfully, SV randomization is repeatable so the error condition should be easily recreated by rerunning the simulation
    - To aid verification, some small logging code is added (say 2 procedural blocks)
    - Simulation is rerun to fail & generate the additional logging info
    - Error conditions change or even worse the error disappears

- What happened?
    - Almost certainly, randomization changed due to code restructuring
    - **RNG environment wasn't stable!**

SV RNG Stability

Reference Page

- In SV each object or thread has an individual random number generator (**RNG**)
  - RNG's of different threads and objects do not interfere with each other
  - This is known as random stability
- RNG stability has a major influence on test
  - Tests must be controllable and reproducible
  - Editing the code for example should not affect the RNG sequences
- Test environment must take stability into account
- SV solution is called "Hierarchical Seeding"
  - RNG's are controlled by manual seeding of objects/threads
  - Seeds are passed down from the top level of hierarchy
  - Single seed at the root thread can control all lower levels

Object / Thread Stability

Reference Page

- Hierarchical seeding involves 4 properties:
  - **Initialization RNG's**
    - A simulator-specific ***default seed*** is used to seed the RNG within each instance of a module/program/interface or a package. These so-called "Initialization RNG's" provide seeds for subsequent objects and threads
  - **Object Stability**
    - each class instance has a unique RNG (accessed by: **<obj>.randomize**)
    - at **new** an objects' RNG is initialized with the next random value of its parent thread RNG
    - Stability is only guaranteed if we add/initialize new objects **after** current, so make code additions after existing code in a file
  - **Thread Stability**
    - each thread has a unique RNG (accessed by: **$urandom**)
    - As each new dynamic thread is created, its RNG is initialized with the next random value of its parent thread RNG.
    - Maintain same code order… add/initialize new threads **after** current.
  - **Manual Seeding**
    - Any non-Initialization RNG may be manually seeded using **$srandom**

RNG System Functions

Reference Page

- **Object randomization method**
  - `.randomize()`
    - Called via an object handle it randomizes properties according to constraints and other rules.
    - `.randomize` may be called from any appropriate thread
- **Thread randomization system calls** (Callable only from within the thread itself)
  - `$urandom [seed]`
    - returns a new 32-bit unsigned random every time called
    - seed is optional and works like Verilog 2001 $random
  - `$urandom_range([min,] max)`
    - returns a new 32-bit unsigned random number in the range
    - min is optional and defaults to 0
- **Manual randomization method**
  - `srandom(seed)` **- Note this is a method!!!**
    - initializes the RNG of an object or thread to seed

Manual Seeding of objects

Reference Page

- **srandom()** permits manual seeding of an objects' RNG

```
class Packet;
   rand bit[15:0] header;
   ...
   function new (int seed);
      this.srandom(seed);
      ...
   endfunction
endclass
```

GUARANTEES seed is set before any members get randomized

```
Packet p = new(200); // Create p with seed 200.
```

- **srandom()** can also be used outside of objects

```
p.srandom(300);        // Re-seed p with seed 300.
```

Manual Seeding

Reference Page

- Using **srandom()** to seed a threads RNG

std::process
(Outside the scope of this class)

A class defined in the std package.
Allows fine control over processes
spawned by fork…join etc.
*Think of **::self.** as similar to **this.***

```
integer x, y, z;
fork
    begin
        process::self.srandom(100);        // seeding at the start of a thread
        x = $urandom;
    end
    begin
        y = $urandom;
        process::self.srandom(200);        // seeding during a thread
    end
join
```

Manual seeding of a root thread makes the entire sub tree stable, allowing it to be moved within the source code for example…

Advanced Seeding

Reference Page

- The built-in seeding control mechanisms are sufficient for most users:
  - Stability maintained per Object / per thread
  - Manual `srandom()` seeding at top level propagated down

- For more sophisticated projects SV provides RNG state save/load via simple strings which can be saved/read from a file, etc. etc.
  - `get_randstate()` **// Note this is a process:: method!**
    - Returns state of RNG of a process as a *string*
  - `set_randstate(string)` **// Note this is a process:: method!**
    - Copies *string* into state of RNG

- NOTE: ***`string`*** is vendor-specific and NOT portable across simulators

QuestaSim™ and Root RNG

Reference Page

- By default, the root RNG seed is taken from the sv_seed variable in the modelsim.ini file.
- At simulation time, QuestaSim™ allows you to manually seed the root RNG

```
vsim -sv_seed 21
```

- If neither of these is provided, the default value is 0.

Functional Coverage

In this section

SystemVerilog

- Structural vs Functional Coverage
- Coverage Modeling
- Covergroup, Coverpoint, Cross
- Sequential & procedural sampling

Coverage

- Coverage attempts to get a numerical handle on toughest question in verification:

  "When can I get off this merry-go-round?"

- Two types of coverage are popular in Design Verification (DV):
  - **Structural Coverage (SC)**
    - Tool generated, e.g. Code Coverage, Toggle Coverage, etc.
  - **Functional Coverage (FC)**
    - Human generated metrics derived from Test Plan
    - Usually makes sense to start FC after SC goals are met

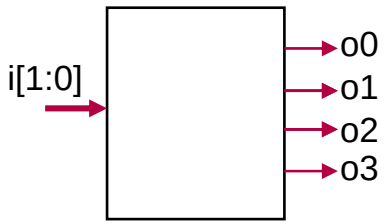
Structural Coverage

- HDL Structural Coverage has arisen from analysis of SW projects
  - Specifically the statistical analysis derived from instrumentation/analysis of executing code, including:
    - How many times each line of code executed
    - Paths, decisions, loops, procedures, etc.
- **Pros**:
  - White-box view of design from within
  - Tool generated, so less burden on designers
  - Finds areas of a program not exercised
  - Targets additional test cases to increase coverage
  - Provides a measure of code coverage and indirectly of quality
  - May identify redundant test cases that do not increase coverage
- **Cons**:
  - Derived from work on linear languages (C, C++)
  - What about concurrency of HDL's?

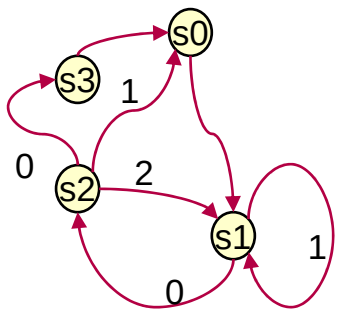
Functional Coverage

- HDL Functional Coverage comes at the problem from a user or system view
  - It asks questions like:
    - Have all typical operational modes been tested
    - All error conditions? All corner cases?
    - In other words: Are we making progress? Are we done yet?
- **Pros**:
  - Black-box, external view
  - Targets the stimulus/testbench more than the actual code
  - Adaptable to more efficient transaction-level analysis, not just signal level
  - Identifies overlap/redundancy in test cases
  - Fits excellently with Assertion Based Verification & Constrained Random
  - Fits with Transaction-based verification
  - Strong support built-in to SV
- **Cons**:
  - Considerable effort initially to implement

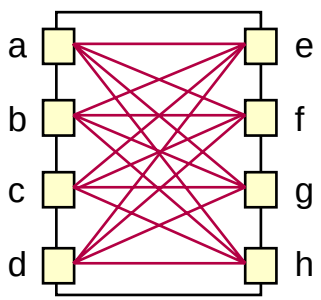
Functional Coverage examples



- **Mux/Demux block**
  - Check truth table



- **Finite State Machine**
  - Check all states entered / exited
  - Check all valid state transitions
  - Check all valid state sequences



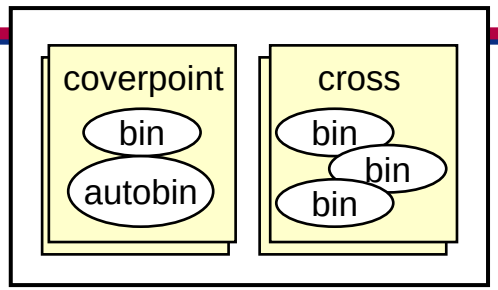
- **Crossbar**
  - Check all inputs have driven all outputs

Functional Coverage and SV

- Functional coverage has been used successfully for many years.
  - Home-grown custom tools (Perl, C++, etc.)
  - Commercial tools (Various vendors)
- SV features sophisticated Functional Coverage capability:
  - Cross-simulator support
  - Well understood standard approach
  - Database accessible from within SV (testbench reactivity/adaptability)
- SV Coverage technology allows:
  - Coverage of variables and expressions, cross coverage
  - Automatic and/or user-defined coverage bins
  - Bins track sets of values, transitions, or cross products
  - Filtering conditions at multiple levels
  - Events and sequences to automatically trigger coverage sampling
  - Procedural activation and query of coverage
  - Optional directives to control and regulate coverage

- Coverage bins maintain a record of state and transition events
- When all bins in all coverage blocks have reached their targets, then by definition, 100% of functionality has been tested!

SV Coverage Terminology



Keywords

- **`covergroup`**
  - A user-defined type like a class, which defines a coverage model
  - Composed of a number of sub-elements including the following…
- **`coverpoint`**
  - Describes a particular type of coverage event, how it will be counted as well as one or more bins to organize the count
- **`cross`**
  - Defines a new coverage event by "combining" two or more existing coverpoints or variables
- **`bins`**
  - A coverage event counting mechanism, automatically or user-defined

- **Clocking event**
  - Defines when a coverpoint is sampled. Usually a clock but can also be the start/end of a method/task/function or can be manually triggered within procedural code
- **Guard (`iff` <expr>)**
  - Optional condition that disables coverage at the covergroup, coverpoint, cross or bins level

Declaring a `covergroup`

- **`covergroup`**
  - Similar to a class, a covergroup instance is created via **`new()`**
  - Covergroups may be defined in a module, program, interface or class and even package and generate-blocks
- Syntax:

```
covergroup name [ ( <list_of_args> )] [ <clocking_event>  ] ;
        <cover_option. > ;
        <cover_type_option. > ;
        <cover_point> ;
        <cover_cross> ;
endgroup [ : identifier ]
```

covergroup Example

- covergroup is similar to a class
  - It defines a coverage model

```
module cover_group;
bit clk;
enum {sml_pkt, med_pkt, lrg_pkt} ether_pkts;
bit[1:0] mp3_data, noise, inter_fr_gap;

covergroup net_mp3 () @(posedge clk);
  type_option.comment = "Coverage model for network MP3 player";
  option.auto_bin_max = 256;
     Mp3 : coverpoint mp3_data;
     Junk: coverpoint noise;
     epkt:  coverpoint ether_pkts;
  Traffic: cross epkt, Mp3;        // 2 coverpoints
endgroup

net_mp3 mp3_1 = new();
net_mp3 mp3_2 = new();
…
endmodule
```

Coverage sampling clock

Options

Coverage points

Cross coverage between coverpoints

instances of coverage model (multi-instantiation is OK)

Covergroups Within Classes

- It is possible to embed one/more **covergroups** within a **class**
  - Provides coverage of class properties (regardless of property local/protected status)
  - May only have one variable of the **covergroup**

```
class packet;

// Ethernet Packet Fields
  bit[7:0] dest, src;
  bit[15:0] len;
  bit [47:0] payld [ ];
  bit valid;

  covergroup cov1 @(valid);      // embedded covergroup
     cp_dest : coverpoint dest;
      cp_src : coverpoint src;
  endgroup

  function new(int i);
    payld = new[i];  len = i;
    cov1 = new();
  endfunction : new

endclass : packet
```

Coverage sampling event is any transition of property valid

Constructor must instantiate covergroup!

Notice syntax! Handle of same name as CG is implicit

coverpoint

- **coverpoint** indicates an integral variable or an integral expression to be covered
- Events are counted in **bins** which are either auto or user-defined
- **coverpoint** syntax

guard expressions

```
[label :] coverpoint <expr> [ iff ( <expr> )]
        {
                [ <coverage_option> ]
        bins name [ [ ] ] = { value_set }  [ iff ( <expr> ) ];
        bins name [ [ ] ] = ( transitions )[ iff ( <expr> ) ];
        bins name [ [ ] ] = default [ sequence ][ iff ( <expr> ) ];
        ignore_bins  name = { value_set };
        ignore_bins  name = ( transitions );
        illegal_bins name = { value_set };
        illegal_bins name = ( transitions );
        }
```

- Two classes of **bins**:
  - *Value* **bins**
    - Increment when specific values are seen
  - *Transition* **bins**
    - Increment at end of predefined sequence of transitions

Covergroup, Coverpoint & Class Example

```
module cover_points;

event smpl;

class cpoints;

rand bit [7:0] a;
bit expr = 1;
bit  ok = 1;
int arg = 66;
covergroup cg_a (int val) @(smpl);
   cp_a : coverpoint a iff ( expr )
 {
   bins val_bin = { val }; // i.e. 66
 }
endgroup

function new();
  cg_a = new(arg);   // pass in argument to covergroup
                     // Instantiation may also be in a function called by the constructor
endfunction

endclass
```

Covergroup argument

Constructor must instantiate covergroup!

```
cpoints cp;

initial begin
  cp = new();
  for (int i=0; i<10; i++) begin
    void'(cp.randomize());
    -> smpl;
    $display("coverage = ",
         cp.cg_a.get_coverage() );
  end
end

endmodule
```

Auto bins

- If no user-defined **bins** are declared for a coverpoint,
  - **bins** are automatically created.
  - Automatically created **bins** are named: **auto[0] , auto[1] , auto[2],** etc.
    - The values inside of **[ ]** are the values of the expression

```
bit [7:0] a;
bit expr, clk, ok;

covergroup cg_a @(posedge clk);
    option.auto_bin_max = 16;
        cp_a : coverpoint a iff ( expr );
endgroup
```

No user-bins, so expect auto-bins
Q: How many bins here
and what distribution?

- To prevent auto-bin explosion, the number of auto **bins** created for a coverpoint is defined as the lower of:
  - 2^N *(where N is the number of **bins** required for full coverage of **coverpoint**)*
  - **auto_bin_max**, *(a predefined coverage option (default 64))*

User defined `bins`

```
rand bit [7:0] a;
bit expr = 1;
bit  ok = 1;
covergroup cg_a @(smpl);
   cp_a : coverpoint a iff ( expr )
 {
   bins arange      = { [0:63] };
   bins vals        = { 64, [67:127] };

   bins mid_a[ ]    = { [128:147] };
   bins distr[10]   = { [149:169] };

   wildcard bins wb = {8'b0101zx?1};

   bins seq         = ( 255 => 0, 0 => 99);
   bins upper       = { [150:$] } iff ok;
 }
endgroup
```

Coverpoint guard expression

Single bins tracking ranges

Dynamic Array of bins, 1 per value (20)

Explicit array of 10 bins, Since specified range == 21 values are evenly distributed among available bins (2 per) with extra one going in last bin

Wildcard (casex rules)

Per-bin guard expression

Range with $ (upper limit of a)

Excluded bins

- **default** - catch the values that do not lie within the defined value bins
- **default sequence** - catch transitions not included in the defined sequence bins
- **ignore_bins** are filtered from coverage <u>including</u> values included in other bins
- **illegal_bins** are same as **ignore_bins** but they trigger a runtime error message
  - NOTE: You may NOT combine default with illegal/ignore

```
bit [7:0] a;
bit expr = 1;
bit ok = 0;
covergroup cg_a @(smpl);
cp_a : coverpoint a iff ( expr ) {
    bins some_range = { [0:65] };
    bins chk_66 = {66};
    ignore_bins  ib_67 = {67};
    illegal_bins ib_66 = {66};
    illegal_bins ib3 = ( 4, 5 => 6 ) iff !ok;
    bins oops_val  = default;
    bins oops_seq  = default sequence;
      }
endgroup
```

Excluded from coverage

ib_66 will exclude chk_66 from coverage

Exclude AND trigger a run-time error

Per-bin guard expression

Excluded from coverage

Excluded `bins` - output

```
# ** Warning: After enforcing the illegal and ignore values, the values list associated with scalar bin 'chk_66' in Coverpoint 'cp_a' of
Covergroup instance '/cover_illegal/check_illegal::cg_a' has converged to empty list. The bin will be taken out of coverage calculation.

run -all
# ** Error: Illegal transition bin got covered at value='b00000110. The bin counter for the illegal bin
'/cover_illegal/check_illegal::cg_a.cp_a.ib3' is 1.
#    Time: 7 ns  Iteration: 0  Instance: /cover_illegal/check_illegal
# ** Error: Illegal range bin value='b01000010 got covered. The bin counter for the bin '/cover_illegal/check_illegal::cg_a.cp_a.ib_66' is
1.
#    Time: 67 ns  Iteration: 0  Instance: /cover_illegal/check_illegal
# Break at cover_illegal.sv line 40

fcover report -r /*
# COVERGROUP COVERAGE:
# ---------------------------------------------------------------------------------
# Covergroup                                                Metric         Goal/ Status
#                                                                            At Least
# ---------------------------------------------------------------------------------
#  TYPE /cover_illegal/check_illegal/#cg_a#              100.0%        100 Covered
#     Coverpoint #cg_a#::cp_a                            100.0%        100 Covered
#         illegal_bin ib_66                                              1 Occurred
#         illegal_bin ib3                                                1 Occurred
#         ignore_bin ib67                                                1 Occurred
#         bin some_range                                    65           1 Covered
#         bin chk_66                                         0           1 ZERO
#         default bin oops_val                                         188 Occurred
#         default bin oops_seq                                         253 Occurred
#
# TOTAL COVERGROUP COVERAGE: 100.0%  COVERGROUP TYPES: 1
#
#
```

```
for (int i = 0; i<256; i++) begin
   a = i;
```

There are 255 transitions in the for loop:
1 – illegal_bins ib3 (transition 5 => 6)
1 – illegal_bins ib_66 (filters transition 65=>66 from oops_seq)
253 – oops_seq default sequences

There are 256 values for a in the for loop:
65 – bins some_range
1 – illegal_bins ib_66
1 – ignore_bins ib67
1 – illegal_bins ib3 (filters 5=>6 from some_range)
188 – default bins oops_val

Transitions

- The only way to track transitions is in user defined bins (no auto bins)
- The syntax for specifying transition sequences is a subset of the SVA sequence syntax

```
12 => 13 => 14              // Simple sequence of values

1, 4 => 7, 8                // Set of transitions.
    ( i.e. 1 => 7, 1 => 8, 4 => 7, 4 => 8 )
```

Discussed in SVA section later

```
val1[*4]                    // Repeat val1 4 times
    ( i.e.  val1 => val1 => val1 => val1 )

3[*2:4]                     // Set of repetition sequences
    ( i.e.  3 => 3 , 3 => 3 => 3, 3 => 3 => 3 => 3 )

2[->3]                      // goto repetition (not necessarily consecutive)
    ( i.e.   …=>2=>...=>2=>...=>2 )

2[=3]                       // Not-necessarily-consecutive repeat
    ( i.e.   …=>2=>...=>2=>...=>2=>... )

default sequence            // "other" values, ignored for coverage
```

Transitions Example fsm_cover - 1

```
module fsm_cover( input clk, rst,
                  input_sig_1,
                  input_sig_2,
                  output reg a, b);

enum bit[1:0] { S0 = 2'h0,
                S1 = 2'h1,
                S2 = 2'h2 }
                state, next_state;

always @ (posedge clk)
  if (rst)   // Fully synchronous reset
    state <= #1 S0;
  else
    state <= #1 next_state;
```

```
always_comb
    case (state)
      S0:  begin
             b = 0;
             if(input_sig_1 || input_sig_2 )
               a = 1;
            else  a = 0;
             if(input_sig_1 == 1)
               next_state = S1;
             else
               next_state = S0;
            end
      S1:
        begin
          b = 1; a = 0;
          if(input_sig_2 == 1)
             next_state = S2;
          else
             next_state = S0;
        end
      S2:
        begin
          b = 0; a = 0; next_state = S0;
        end
      default:
        begin
           a = 1'bx; b = 1'bx;
           next_state = S0;
        end
    endcase
```

Example fsm_cover - 2

```
covergroup cfsm  @(negedge clk);
  type_option.comment = "Coverage of FSM";
  type_option.strobe = 1;
  stat : coverpoint state
     {
       option.at_least = 1;
       bins valid    = {S0,S1,S2};
       bins S0_S0    = (S0 => S0);
       bins S0_S1    = (S0 => S1);
       bins S1_S0    = (S1 => S0);
       bins S1_S2    = (S1 => S2);
       bins S2_S0    = (S2 => S0);
       illegal_bins ib = {2'b11};
       bins oops        = default;
       bins oops_seq    =
                 default sequence;
     }
endgroup
cfsm C0 = new();
endmodule
```

```
# COVERGROUP COVERAGE:
# ----------------------------------------------------------------------------
# Covergroup                          Metric     Goal/ Status
#                                                At Least
# ----------------------------------------------------------------------------
#  TYPE /test_fsm/u1/cfsm             83.3%      100 Uncovered
#    Coverpoint cfsm::stat            83.3%      100 Uncovered
#       illegal_bin ib                 0            ZERO
#       bin valid                     12          1 Covered
#       bin S0_S0                      7          1 Covered
#       bin S0_S1                      2          1 Covered
#       bin S1_S0                      0          1 ZERO
#       bin S1_S2                      1          1 Covered
#       bin S2_S0                      1          1 Covered
#       default bin oops               0            ZERO
#       default bin oops_seq           1            Occurred
#
# TOTAL COVERGROUP COVERAGE: 83.3%  COVERGROUP TYPES: 1
```

cross

- The **cross** construct specifies cross coverage between one or more crosspoints or variables

```
[label :] cross <coverpoint list> [ iff ( <expr> )]
  {
     bins name = binsof (binname) op binsof (binname) op …  [ iff (expr) ];
     bins name = binsof (binname)  intersect { value | [ range] } [ iff (expr) ];
     ignore_bins name = binsof (binname) …;
     illegal_bins  name = binsof (binname) …;
}
```

- If **cross** specifies a variable, a **coverpoint** for that variable is implied
- Expressions cannot be used directly in a **cross** but may be described in a new **coverpoint** which is made part of the **cross**
- **binsof** yields the bins of it's expression (optional **intersect** { } for more refinement)
- **intersect** { } can specify a range of values or a single value
- Supported operators (op) are: **!**, **&&**, **||**
- **illegal_bins**, **ignore_bins**, **iff** etc are allowed as in coverpoints

Cross coverage

- A coverpoint typically defines coverage for a single variable or expression.
  - Answers questions like:
    - Have I entered every state of the FSM?
    - Have I tested all operating modes?
- Sometimes we need to study two or more coverpoints simultaneously.
  - Answers questions above PLUS :
    - Have I entered every state WHILE in each mode?

```
typedef enum {s1, s2, s3} sts;
sts state;
typedef enum {a, b, c} mds;
mds mode;

covergroup cg @(posedge clk);
   cp_state   : coverpoint state;
   cp_mode    : coverpoint mode;
   cs_modstat: cross cp_state, cp_mode;
endgroup
```

s1	s2	s3

cp_state bins

s1,a	s2,a	s3,a
s1,b	s2,b	s3,b
s1,c	s2,c	s3.c

a
b
c

cp_mode bins

Simple Auto `cross` Coverage

```
module simple_cross;
event smpl;

class scross;
  typedef enum {a,e,i,o,u} vowels;
  rand vowels v;
  bit inactive = 1;
  rand bit[2:0] cnt;
  covergroup cg @(smpl);
      a: cross cnt, v iff (inactive);
  endgroup
  function new();
    cg = new();
  endfunction
endclass

scross sc = new();
initial
  for (int i=0; i<100; i++) begin
    void'(sc.randomize());
    -> smpl;
  end
endmodule
```

cg crosses 2 variables: one 3-bit, the other 5 element

Implicit coverpoint cg::cnt has 8 bins (cnt coverpoint)
cg::v has 5 bins (# of vowels)

Auto cross points (and bins) is 8x5 = 40
Total of 53 bins generated

Implies coverpoints for v, cnt

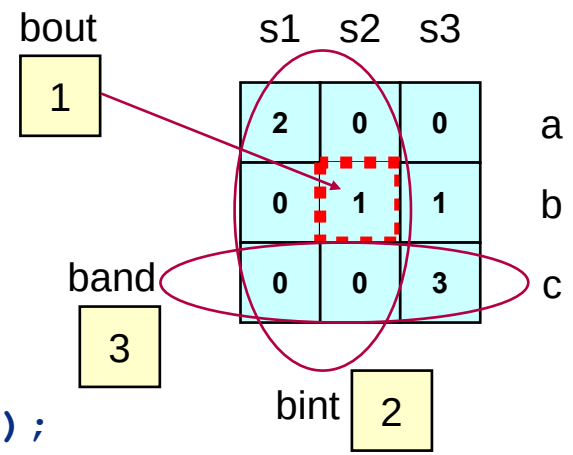
```
# COVERGROUP COVERAGE:
# ----------------------------------------------------------------------------
# Covergroup                                  Metric      Goal/ Status
#                                             At Least
# ----------------------------------------------------------------------------
#  TYPE /simple_cross/scross/#cg#             96.7%       100 Uncovered
#    Coverpoint #cg#::cnt                     100.0%      100 Covered
#      bin auto[0]                            12            1 Covered
 . . .
#    Coverpoint #cg#::v                       100.0%      100 Covered
#      bin auto[a]                            21            1 Covered
 . . .
#    Cross #cg#::a                            90.0%       100 Uncovered
#      bin <auto[0],auto[a]>                  1             1 Covered
. . .
#      bin <auto[7],auto[e]>                  0             1 ZERO
 . . .
#
# TOTAL COVERGROUP COVERAGE: 96.7%  COVERGROUP TYPES: 1

.... Rest of report not shown ....
```

Selective `cross` example

```
typedef enum {s1, s2, s3} sts;
sts state;
typedef enum {a, b, c} mds;
mds mode;
```

```
# ------------------------------------------------------------------
# Covergroup                          Metric      Goal/ Status
#                                                At Least
# ------------------------------------------------------------------
#  TYPE /cross_ex/cgb                100.0%        100 Covered
#     Coverpoint cgb::cp_state       100.0%        100 Covered
#         bin s1                          2          1 Covered
#         bin s2                          1          1 Covered
#         bin s3                          4          1 Covered
#     Coverpoint cgb::cp_mode        100.0%        100 Covered
#         bin a                           2          1 Covered
#         bin b                           2          1 Covered
#         bin c                           3          1 Covered
#     Cross cgb::cs_modstat          100.0%        100 Covered
#         ignore_bin bout                 1              Occurred
#         bin band                        3          1 Covered
#         bin bint                        2          1 Covered
```

```
covergroup cgb() @(posedge clk);
  cp_state: coverpoint state{
            bins s1 = {s1};
            bins s2 = {s2};
            bins s3 = {s3};
  }
  cp_mode: coverpoint mode{
            bins a = {a};
            bins b = {b};
            bins c = {c};
  }
  cs_modstat: cross cp_state, cp_mode{
    bins band = binsof (cp_state) && binsof (cp_mode.c);
    bins bint  = binsof (cp_state) intersect {[s1:s2]};
    ignore_bins bout = binsof (cp_state.s2) && binsof (cp_mode.b);
  }
endgroup
```



Coverage: Predefined Methods

- A number of built-in methods allow interrogation of the coverage database

```
void sample()                    - sample the covergroup
real get_coverage()              - cumulative coverage number (all instances) [0-100]
                                 - is a static method across cg's or instances

                                   <cg_name>::get_coverage()
                                   <inst>.get_coverage()

real get_inst_coverage()         - cumulative coverage number [0-100]

                                   <inst>.get_inst_coverage()
void set_inst_name(string)
void start()                     - start collecting coverage
void stop()                      - halt collecting coverage information
```

- A number of built-in system_tasks and functions are also defined

```
$set_coverage_db_name(name) - name the coverage db file written at end of simulation
$load_coverage_db(name)     - load cumulative coverage info from a file
$get_coverage()             - returns overall coverage of all covergroups  real:[0-100]
```

NOTE
Coverage is internally calculated as a "real" and reported as a score 0-100

Coverage: `option`

- A wide assortment of options can be selected at each `covergroup, coverpoint, cross,` etc.
- `option` (apply to most levels: covergroup, coverpoint, cross): (default)

`weight=`*`number`*	- statistical weight within level	(1)
`goal=`*`number`*	- target coverage (instance)	(100)
`name=`*`string`*	- specifies covergroup instance name if unspecified, name is auto generated	("")
`comment=`*`string`*	- unique text appears in reports	
`at_least=`*`number`*	- minimum # of hits per bin	(1)
`detect_overlap=`*`boolean`*	- If set, a warning is given when overlap between 2 bins of a coverpoint	(0)
`auto_bin_max=`*`number`*	- max # of auto-bins created	(64)
`cross_num_print_missing =`*`number`*	- # of not covered cross-product bins	(0)
`per_instance=`*`boolean`*	- If true also track per covergroup instance	(0)

Coverage: `type_option`

- `type_option` - applies to `covergroup,coverpoint or cross` by type (i.e. across all instances):

`weight=`*`constant_number`*	- statistical weight within database (default = 1)
`goal=`*`constant_number`*	- target goal for covergroup (default = 100)
`comment=`*`string_literal`*	- unique text appears in reports (default = "")
`strobe=`*`boolean`*	- If true, sample once per clk-event in the postponed region (default = 0)

Some settings exist as type_options AND options. What does this mean?

Think of type_option as a default setting (for the type). Then, the option setting is useful to override the default on an instance basis. However, for this to work, the “per_instance” option must be set to true AND we need some way to set per-instance options… for example by passing arguments to the covergroup constructor.

Clocking Event

- The optional clocking event says when coverpoints should be sampled
- If omitted, the user must sample procedurally (`.sample` method)
  - Sampling is performed when the clocking event triggers.
  - Non-synchronous sources can yield multiple-trigger events which may mess-up coverage
  - `.strobe type_option` works like `$strobe` in Verilog
    - Triggers in "postponed" step
    - Ensures that sampling happens only once per timestep

```
covergroup cg @(sig);
  type_option.comment = "your_name_here";
  type_option.strobe = 1;
          a : coverpoint vara;   // expression allowed
          b:  coverpoint varb;
          c:  cross a, vard;     // 1 var & 1 coverpoint
endgroup

cg cg1 = new();
```

Sequence, Method Coverage Sampling

- Sampling can be triggered by the endpoint of an SVA sequence…

```
sequence smpl; (@(posedge clk) a ##1 b [->4] ##1 c;) endsequence

covergroup cga @(smpl);
  . . .
endgroup

cga cga1 = new();
```

- …or at the begin/end of a method, task or function call…

```
covergroup cgb @@(begin doit);
  . . .
endgroup

cgb cgb1 = new();
```

Could also be:
@@(end doit)

```
task doit();
  . . .
endtask

initial
  forever #1000 doit;
```

Procedural Coverage Sampling

- If a clocking event is omitted, sampling can be performed by calling the built-in **`sample()`** method

```
covergroup net_mp3;
  type_option.comment = "Coverage model for network MP3 player";
     Mp3 : coverpoint mp3_data; // expression allowed
     Junk: coverpoint noise;
  Traffic: cross ether_pkts, Mp3;
endgroup

net_mp3 mp3_1 = new();

always @(posedge clk)
   if (i_want_2_cover)
       mp3_1.sample();
```

Functional Coverage Report - GUI

- **In QuestaSim™ there are 2 ways to determine functional coverage achieved at the current simulation time:**

  From the GUI:

  - **Coverage metrics are reported directly in 2 places**

Covergroups tab of Analysis pane

Analysis

Name	Coverage	Goal	% of Goal	Status
/top/test				
TYPE router_cvg	70.8%	90	78.7%	
Cvp router_cvg::srce	100.0%	90	111.1%	
Cvp router_cvg::dest	100.0%	90	111.1%	
Cross router_cvg::#...	12.5%	90	13.9%	

Assertions | Cover Directives | Covergroups

Covergroup Coverage: 70.8% | Recursive Mode

Workspace Pane

Workspace

Instance	Design unit	Design unit type	Visibility	Total coverage	% Goal total
top	top(fast)	Module	+acc=	35.4%	78.7%
defs	defs(fast)	VlPackage	+acc=<fu...		

Library | sim | Files | Memories

Functional Coverage Report - CLI

- From the CLI:
  - On the QuestaSim™ command line type the following
    <vsim> fcover report –r /*

```
# COVERGROUP COVERAGE:
# ----------------------------------------------------------------------
# Covergroup                                  Metric      Goal/ Status
#                                                           At Least
# ----------------------------------------------------------------------
#  TYPE /top/test/router_cvg                   70.8%          90 Uncovered
#     Coverpoint router_cvg::srce             100.0%          90 Covered
#         bin auto[0]                              6           1 Covered
#         bin auto[1]                              6           1 Covered
#         bin auto[2]                              3           1 Covered

…. Rest of report not shown ….
```

Key

Covered, Uncovered: Indicates whether or not “at least” has been met.

Metric: Say a coverpoint has 10 bins. 9 have met their “at least” setting, 1 has not…
The metric for this would be 90%

Lab – Functional Coverage: Introduction

- This lab is a further development of the router testbench you worked with in the OOP and Random lab exercises
- Since Functional Coverage is aimed at the testbench and not the design it should not come as a surprise that in this exercise you will add a covergroup to the Scoreboard class itself.
- Why?:
  - The Scoreboard is an ideal place to track & report continuing coverage
  - The scoreboard can easily cause sampling to occur for the Covergroup
    - event or sample method

Lab – Functional Coverage: Instructions -1

- Lab directory: **Router/coverage**
- Overview:
  - During this lab, you will complete the router testbench by extending the base Scoreboard class to implement functional coverage.
  - 100% coverage will be required for the test to complete.

- Edit the file test_router.sv
  - From the base class **`base_scoreboard`**, declare a derived class **`scoreboard`**
  - Add text after class **`base_scoreboard`** (look for "class scoreboard")
  - **`scoreboard`** class characteristics:
    - A covergroup called **`router_cvg`**
      - sample based on a named event called **`smpl`**
      - Add coverpoints on **`srce`** and **`dest`** properties
      - Add cross coverage between **`srce`** & **`dest`**
      - Minimum # of hits per bin is 2
      - Maximum # of auto bins is 256

Lab – Functional Coverage: Instructions -2

- Continue editing the **`scoreboard`** class in the file test_router.sv
  - A custom constructor
    - Same 2 arguments as parent class (2 mailbox handles)
    - Remember to initialize class properties correctly
    - Instantiate router_cvg
  - Override base class task run2 (see next slide)
    - Call **`$get_coverage()`** and report the value returned
    - Add code to trigger the **`router_cvg`** covergroup after each successful compare (trigger event **`smpl`**)
    - Detect when coverage is achieved (100) and stop simulation
  - Compile and run by typing:
    make
    -or- make gui

HINT

Not seeing your coverage value reported ?
Simulation running forever ?

Perhaps it is the base class **run2()** which is executing.

Lab – Functional Coverage: Instructions -3

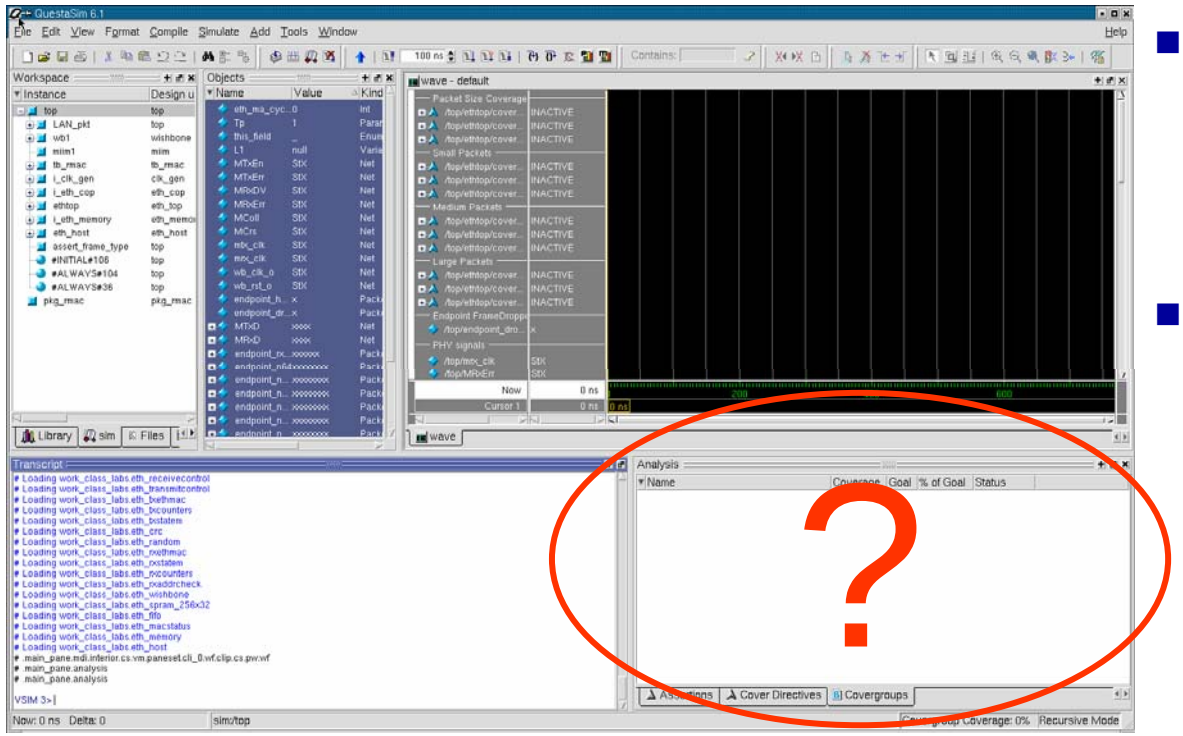
```
class base_scoreboard;
 mailbox #(Packet) stim_mb, mon_mb;
 Packet check [ int ];
 int s_pkt_cnt, m_pkt_cnt, pkt_mismatch;
 Packet s_pkt, m_pkt;
 int errors, run_for_n_packets,;
 string name;
 bit[2:0] srce, dest;
 event smpl;

task automatic run2();
   while(1) begin
     mon_mb.get(m_pkt);
     ++m_pkt_cnt;
     if (check.exists(m_pkt.pkt_id))
       case( m_pkt.compare(check[m_pkt.pkt_id]) )
         0: begin
              $display("Compare error",,m_pkt.pkt_id,, check[m_pkt.pkt_id].pkt_id);
              pkt_mismatch++; $stop;
              if(`TRACE_ON)
                s_pkt.display; check[s_pkt.pkt_id].display;
            end
         1: begin
              check.delete(m_pkt.pkt_id);
              srce = s_pkt.srce;
              dest = s_pkt.dest;
            end
       endcase
     else  check[m_pkt.pkt_id] = m_pkt;
     report;
    end
```

Named event used to trigger coverage group

task to override in derived class

Sol

Lab – Functional Coverage: QuestaSim



- If the Analysis window is not already open, do so now:

  View >Coverage>Covergroups

- Select the test_router_sv unit in the sim tab of the workspace

- **QUESTION:** Why is the Covergroups pane blank, even though we know there are covergroups defined in the code?

  2 reasons: First, covergroups are created at runtime. Second, you must be in the "top" workspace

Calculating Coverage

Reference Page

- The coverage of a coverage group, Cg , is the weighted average of the coverage of all items defined in the coverage group, and it is computed by the following formulae

$$C_g = \frac{\sum_i W_i * C_i}{\sum_i W_i}$$

- *i is the set of coverage items (cover-points and crosses) in the covergroup*
- *W_i is the weight associated with item i.*
- *C_i is the coverage of item i.*

- The coverage of each item, Ci, is a measure of how much the item has been covered, and its computation depends on the type of coverage item: coverpoint or cross

Coverpoint

$$C_i \text{ (user-defined bins)} = \frac{\text{\# of bins that met goals}}{\text{Total \# of bins}}$$

$$C_i \text{ (auto-defined bins)} = \frac{\text{\# of bins that met goals}}{\text{MIN (auto\_bin\_max, } 2^{\mathbf{N}}\text{)}}$$

Cross

$$C_i = \frac{\text{\# of bins that met goals}}{B_c \mathbf{+} \text{ \# user-bins } \mathbf{-} \text{ \# user-excluded bins}}$$

$$B_c = (\prod_j B_j) - B_b$$

- *B_c is the number of auto cross bins*
- *B_j is the number of bins in the jth coverpoint being crossed*
- *B_b is the number of cross products in all user-defined cross-bins*

SVA

Assertions – Immediate / Concurrent

- **Immediate**
  - Simulation use primarily
  - Execute under simulator control inside a procedural block

Imm. assertion is triggered in procedural code/time

```
always @( posedge clk )
   traf_light : assert ( green && !red && !yellow && go );
```

Failure triggers a default message

- **Concurrent**
  - Usable by other tools as well (e.g. formal verification)
  - Clocked/sampled paradigm

Conc. assertions may be triggered in various ways (including procedural code), but time is spec'd internally and may include sequential checks over time

```
property traf_light;
  ( @ ( posedge clk ) ( green && !red && !yellow && go );
endproperty

do_traf_prop : assert property ( traf_light );
```

Immediate Assertions

- Tested when the **assert** statement is executed in procedural code

```
time t;
always @( posedge clk )
    if ( state == REQ )
       req_assert : assert ( req1 || req2 )  $display( "%m OK" );
                 else
                    begin
                       t = $time;
                       #5 $display( "%m failed at time %0t",t );
                    end
```

*Notice **if-else** style… where **if** is implicit. **else** is also optional in this context*

*Notice optional **req_assert** label… … and the use of %m to implicitly insert that label in messages*

NOTE
assert statements resolve X and Z expression values much like if-else statements… they will fail on 0, X or Z

Concurrent Assertions

In this section

- Basics
- Boolean Expressions
- Sequences
- Properties
- Verification Directives

Concurrent Assertions

- Concurrent **`assert`** statements describe behavior over time.
  - clock-based (clock may be user-defined and avoid glitches!!!)
  - Structured for simplicity, flexibility & reuse

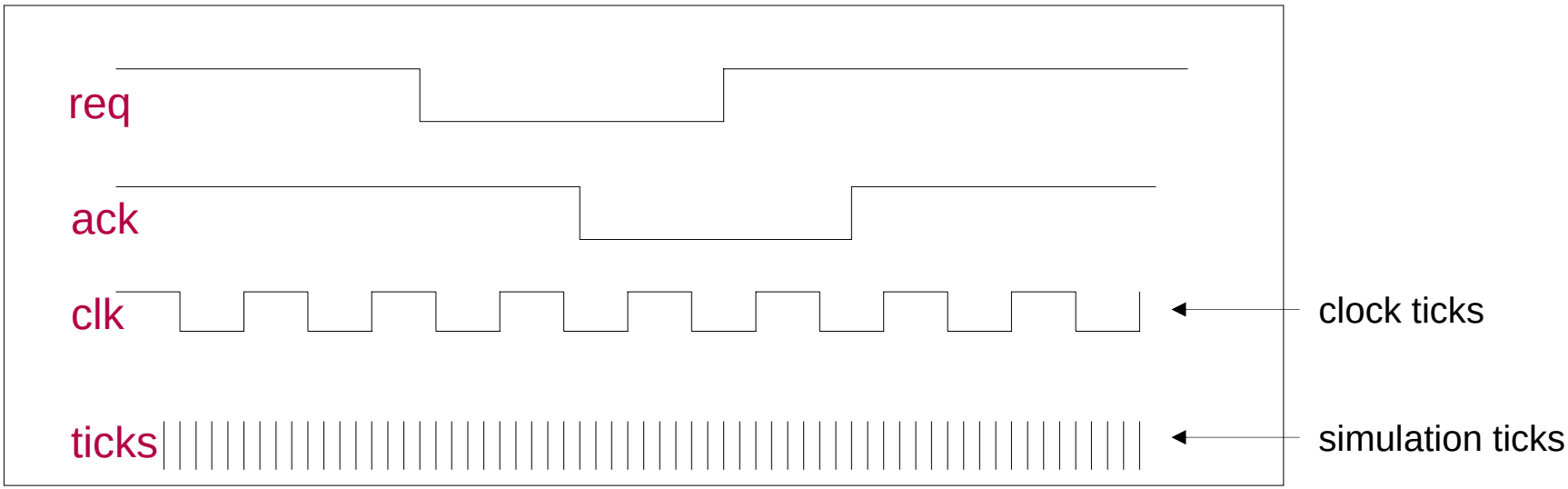
4	**Verification Statements**	`assert , cover, bind, assume`
3	**Property Declarations**	`property , disable iff, not, |->, |=>`
2	**Sequence Regular Expressions**	`and, or, intersect, <time_ shift>, *<repeat>, within, throughout, $past, $stable, $rose, $fell, sequence`
1	**Boolean Expressions**	`<Verilog expr, excl. +=, ++, -- etc.>, System functions: $countones, $inset, $isunknown etc. ended, matched`

Clock Tick

- The timing model employed in a concurrent assertion specification is based on clock ticks
- A clock tick is an atomic moment in time
  - Spans no duration of time.
- A clock ticks only once at any simulation time
  - The sampled values of a variable (in Preponed region) are used in the evaluation of the assertion (in Observed region)

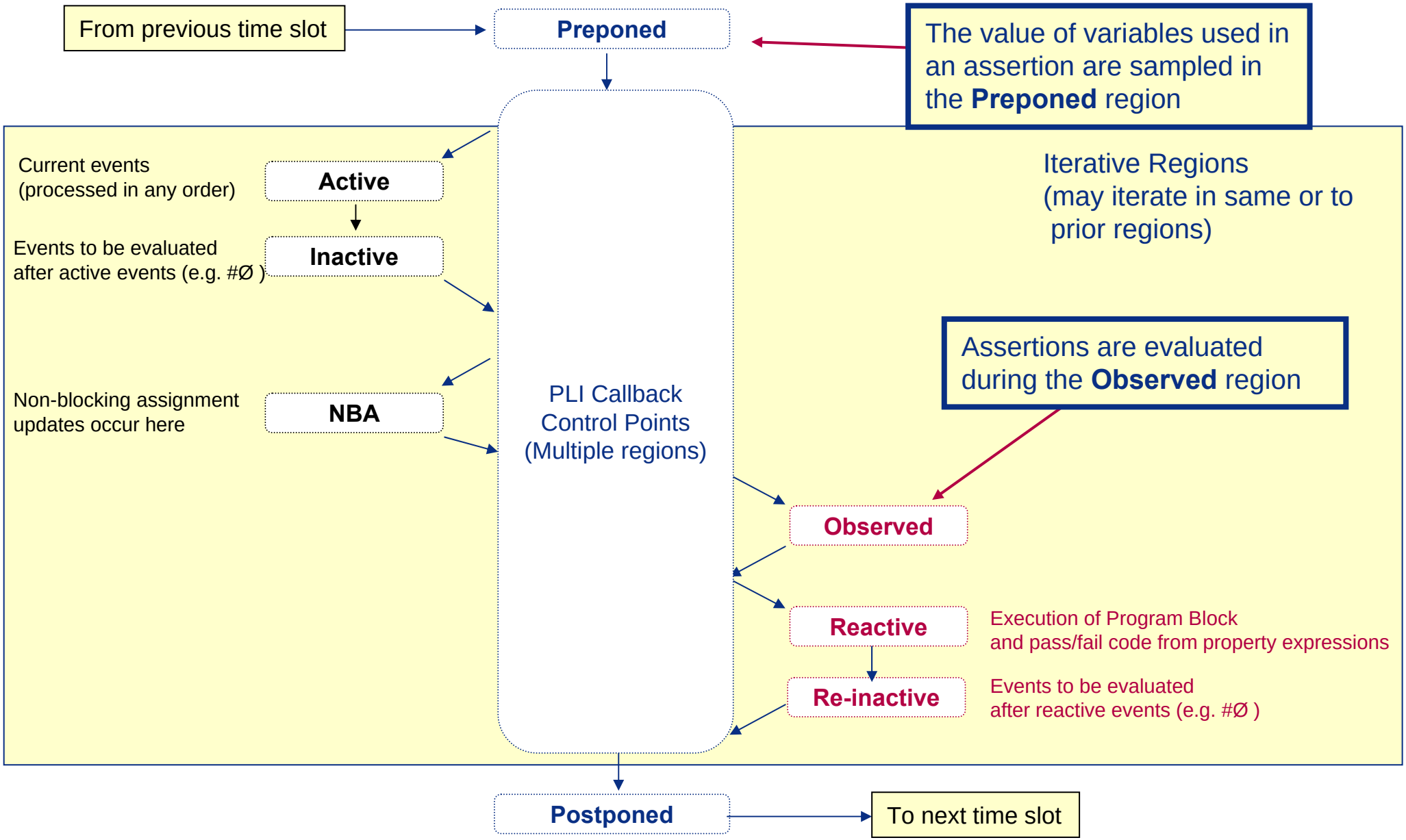
Concurrent Assertion Basics

- Consider two signals, req/ack that handshake together



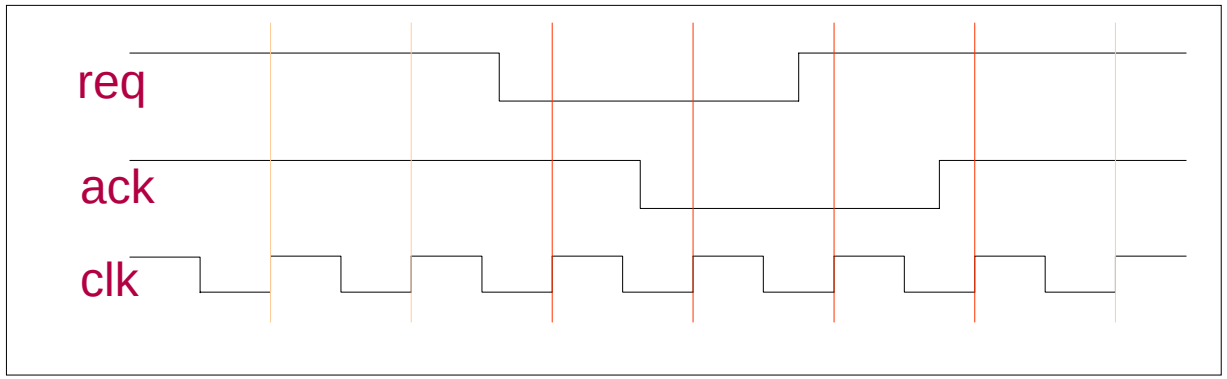
- Signals change over time and interlock to implement handshake
- Traditional Verification requires a model to “handshake” with the DUT to spot:
  - Logic errors
  - Sequence errors
  - Time errors
- Concurrent Assertions describe the sequence of changes in signals over time
- Introduces the concept of a clock to ‘sample’ signal changes and capture the sequence

Assertions & SV Time Slot Regions

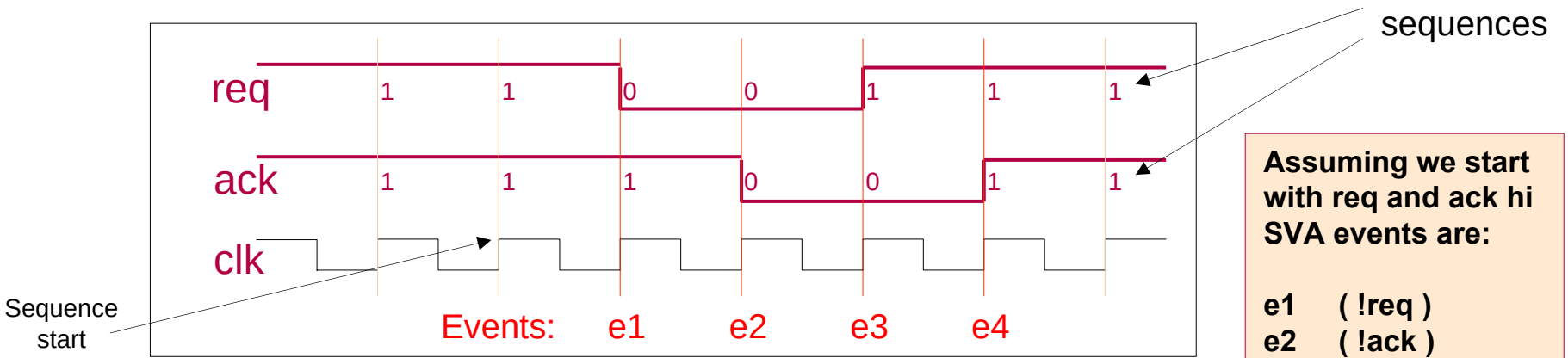


Verilog 95 Regions — **PLI Regions** (Verilog 2001+) — **SystemVerilog Regions**

Concurrent Assertion Basics 2



- If we sample both signals on the posedge of clk…
- The waveforms effectively become:



Assuming we start with req and ack hi SVA events are:

e1 (!req)
e2 (!ack)
e3 (req)
e4 (ack)

Remember, this diagram shows what the signals look like from the point of view of the sampling clock

Boolean Expressions

1 Boolean Expressions

- Basic building blocks of assertions
  - Evaluate sampled values of variables used
  - 0, X or Z interpret as false
  - Excludes certain types:
    - `time`, `shortreal`, `real`, `realtime`
    - `string`, `event`, `chandle`, `class`
    - Associative/dynamic arrays
  - Variables used must be `static`
  - Excludes these operators:
    - C-assignments (`+=`, `-=`, `>>=`, etc)
    - Bump operators (`i++`, `i--`, `++i`, etc)

Sequences

2 Sequence Regular Expressions

- A list of SV boolean expressions in linear order of increasing time
- Boolean test for whether a signal matches a given sequence or not
- Assumes an appropriate sampling clock and start/end times
- If all samples in the sequence match the simulation result then the assertion *matches*
  - Otherwise it is said to fail

Sequence Delay Operator

- Represents a sequential delay of cycles

Delay:

```
##N              // represents a sequential delay of N cycles

a ##N b         // a is true on current tick, b will be true on Nth tick after a
a ##1 b         // a is true on current tick, b will be true on next tick
a ##0 b         // a is true on current tick, so is b ☺ (Overlapping!)
a ##1 b ##1 c   // a is true, b true on next tick, c true on following tick
```

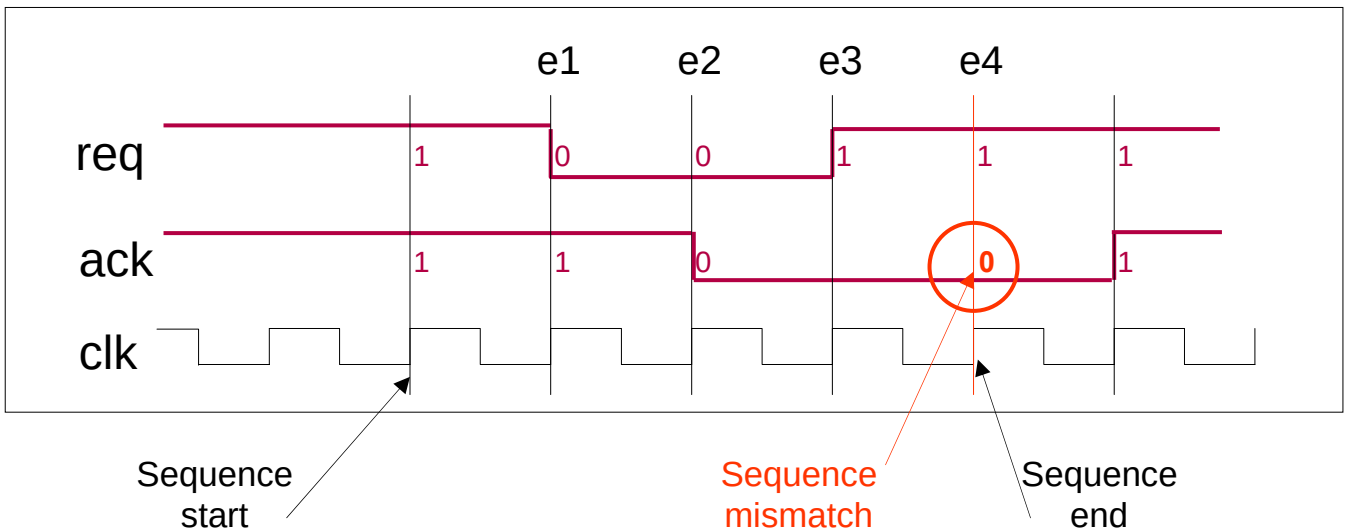
Sequence Example

- Using the sequence delay operator we can specify values over time

```
(req && ack) ##1 !req ##1 !ack ##1 req ##1 (req && ack)
```

Assuming we start with req and ack hi SVA events are:

e1 (!req)
e2 (!ack)
e3 (req)
e4 (ack)



Question:

Although this assertion successfully detects the error shown, why is it a poor coding style?

Answer: Because this assertion targets 2 signals, both signals should be specified throughout.

Sequence Block

- Sequence blocks identify and encapsulate a sequence definition

```
sequence s1;
   @ (posedge clk)
         a ##1 b ##1 c;              // s1 evaluates on each successive edge of clk
endsequence

sequence s2 ;                        // sequence
  (!frame && (data==data_bus)) ##1 (c_be[0:3] == en);
endsequence
```

Notice no clock is defined. This may be inherited from a higher hierarchical statement like property or assert (More on this later)

```
sequence s3;
   start_sig ##1 s2  ##1 end_sig;  // sequence as sub-expression
endsequence

Where:  s3  - same as -

    start_sig ##1 (!frame && (data == data_bus)) ##1 (c_be[0:3] == en)  ##1 end_sig;
```

Property Block

3 Property Declarations

- Property blocks describe behavior in a design
  - Gives a name for reference
  - Allows a range of tools to test/check/cover/etc. that behavior
- By themselves, properties do nothing
  - Must appear in `assert` or `cover` (more later)
- Result of a property evaluation is either true or false

May specify the sampling clock for the property

```
property p1;
   @ (posedge clk)
   (req && ack) ##1 !req ##1 !ack ##1 req ##1 (req && ack);
endproperty
```

Sequence described in property

Implication: `|->` `|=>`

- Using the implication (`|->`, `|=>`) operators you can specify a prerequisite sequence that implies another sequence
  - Typically this reduces failures that you expect and wish to ignore

```
<antecedent seq_expr> |->/|=> ( <consequent seq_expr> );
```

- Think of it this way:
  - If the antecedent matches, the consequent must too.
  - If the antecedent fails, the consequent is not tested and a true result is forced
    - Such forced results are called “vacuous” and are usually filtered out.

- Two forms of the implication operator are supported:
  - Overlapping form:

    ```
    ( a ##1 b ##1 c ) |-> ( d ##1 e );
    ```

    - If a/b/c matches, then d is evaluated on ***THAT*** tick

  - Non-overlapping form:

    ```
    ( a ##1 b ##1 c ) |=> ( d ##1 e );
    ```

    - If a/b/c matches, then d is evaluated on the ***NEXT*** tick

Verification Directives

4 Verification Statement

- A property (or sequence) by itself does nothing
  - It must appear within a verification statement to be evaluated
- **assert** verification directive
  - *[always]* **assert property**
    - *Enforces a property as "checker"*
- **assert** verification directive can appear in modules, interfaces, programs and clocking domains

An 'edge' in antecedent is more efficient (fewer false triggers)

```
property p1;
   ((req && ack) ##1 !req) |-> ack  ##1 (!req && !ack)
                                    ##1 ( req && !ack)
                                    ##1 ( req &&  ack);
endproperty

assert_p1:  assert property (p1)
                 begin $info("%m OK"); end
                 else begin $error("%m Failed"); end
```

ACTION BLOCK

Severity System Tasks

- Several system tasks provide control over severity of a failing assertion.
- These tasks all follow `$display` symantics
  - `$fatal()` - *run-time fatal, terminate simulation*
  - `$error()` - *run-time error, simulation continues*
  - `$warning()` - *run-time warning, varies by tool*
  - `$info()` - *no severity implied, just informative*
  - `$display()` - *like $info*
- By default, an assertion (with no severity task) that fails, triggers `$error`

```
modelsim.ini has SVA settings
; IgnoreSVAInfo = 0          default value is set to ignore (=1)
; IgnoreSVAWarning = 0       default value is set to ignore (=1)
; IgnoreSVAError = 1         default value is set to enable (=0)
; IgnoreSVAFatal = 1         default value is set to enable (=0)
```

SVA - Concurrent Example

```
module sva_ex;
logic [2:0] cnt;
logic clk;

initial
  begin
    clk = 0; cnt = 0;
    forever #20 clk = !clk;
  end

initial
  begin
    wait(cnt == 2) cnt = 4;
    #240 $stop;
  end

always @(posedge clk)
  cnt <= #1 cnt +1;
```

Error insertion!!!

Verification directives

Named behavior of the design

Boolean expressions in linear order over time

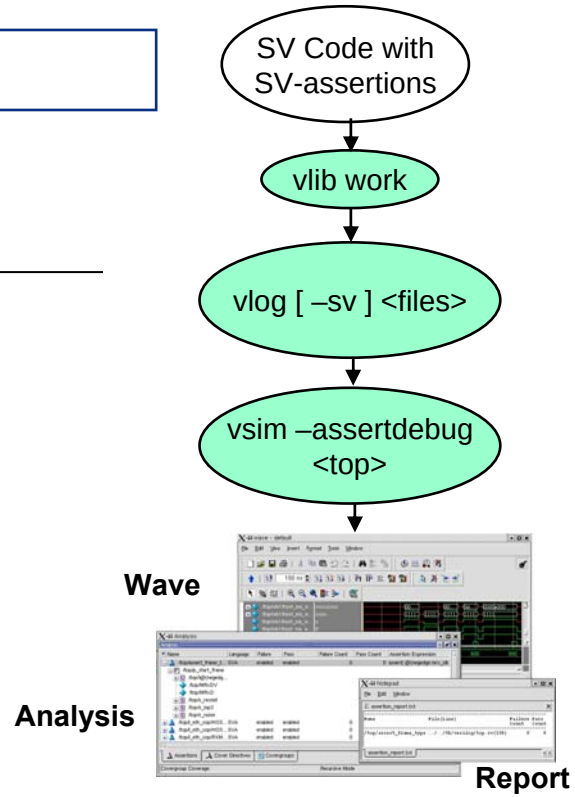
```
sequence s_count3;
   (cnt == 3'h1) ##1 (cnt == 3'h2)
                 ##1 (cnt == 3'h3);
endsequence


property p_count3;
  @(posedge clk) (cnt == 3'h0) |=> s_count3;
endproperty

assert_count3: assert property (p_count3);
cover_count3: cover property (p_count3);

endmodule
```

Sampling clock

NOTE: The counter will NOT increment correctly the first time it advances past 2!

Intro to SVA- Compile/simulate

Lab directory: **sva_q/sva_intro**



Do: <**unix**> vlib work
<**vsim**> vlog +acc=a sva_ex.sv
<**vsim**> vsim –assertdebug sva_ex
<**vsim**> run –all

The QuestaSim™ simulation environment will load

In the main QuestaSim window:

1. select View / Coverage / Assertions
   (*This opens the Analysis pane to display your assertion activity)*

2. in Workspace pane, right-click on **sva_ex** design unit and select Add / Add to Wave
   (*This opens and populates the wave viewer pane)*

3. If desired, undock the Wave pane from the Questasim™ window by clicking the undock icon

Intro to SVA - Questasim™ and assertions

Click here to expand the assertion to show the clock and signals involved

Assertion assert_count3 failed (time 100ns here)

Assertion Passed
Assertion Failed

A green line indicates assertion is active…

A blue line indicates assertion is inactive…

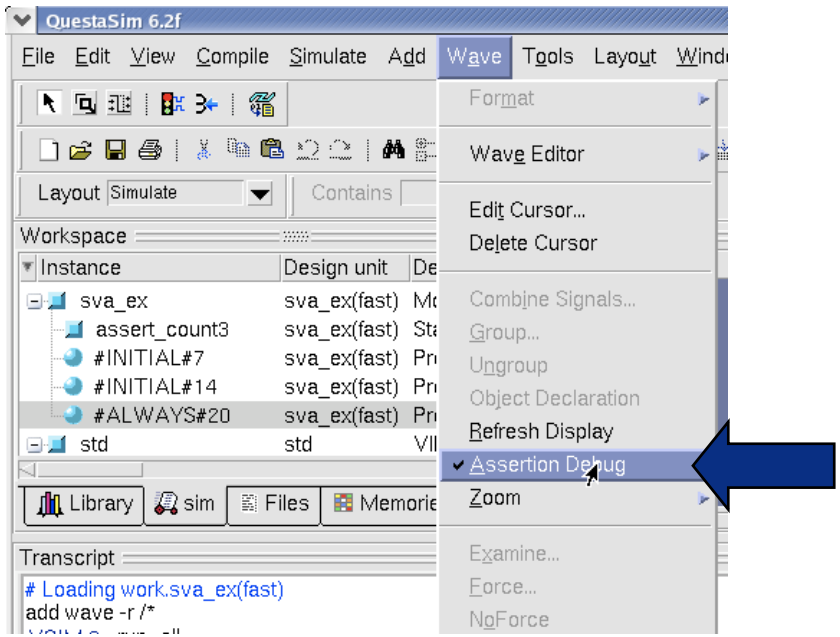
Question

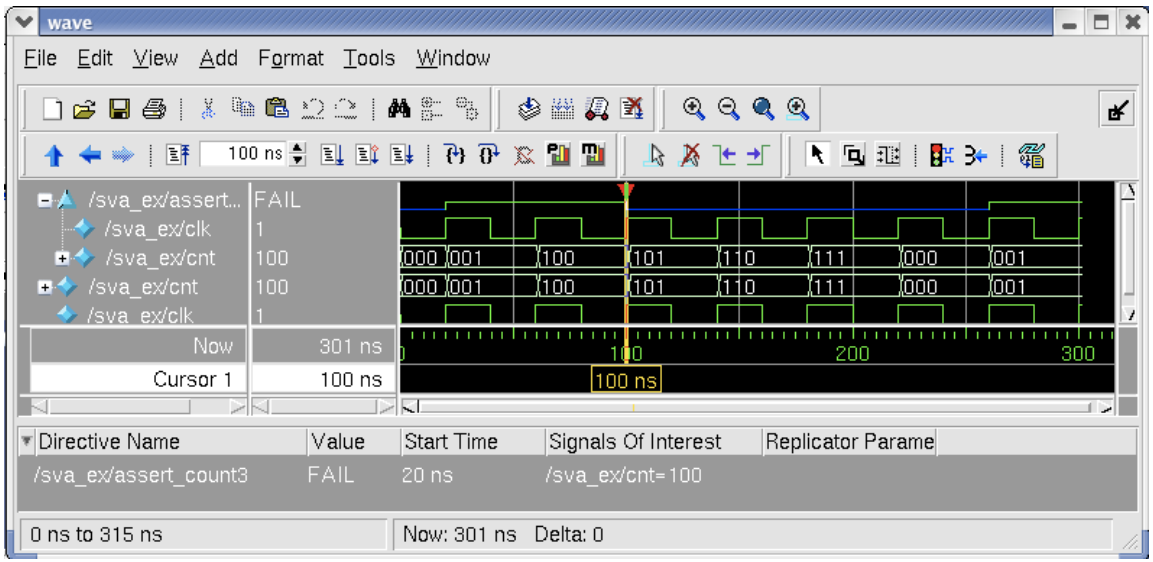
Why does the assertion fail at 100ns when the counter error (1 to 4) happens at time 60ns?

Intro to SVA - Assertion Debug -1

Remember the –assertdebug switch passed to vsim? Here's what it does:

4. select Wave / Assertion Debug (**View / Assertion Debug** if Wave Pane is undocked)
   (*This opens the Assertion Debug pane within the Wave window)*



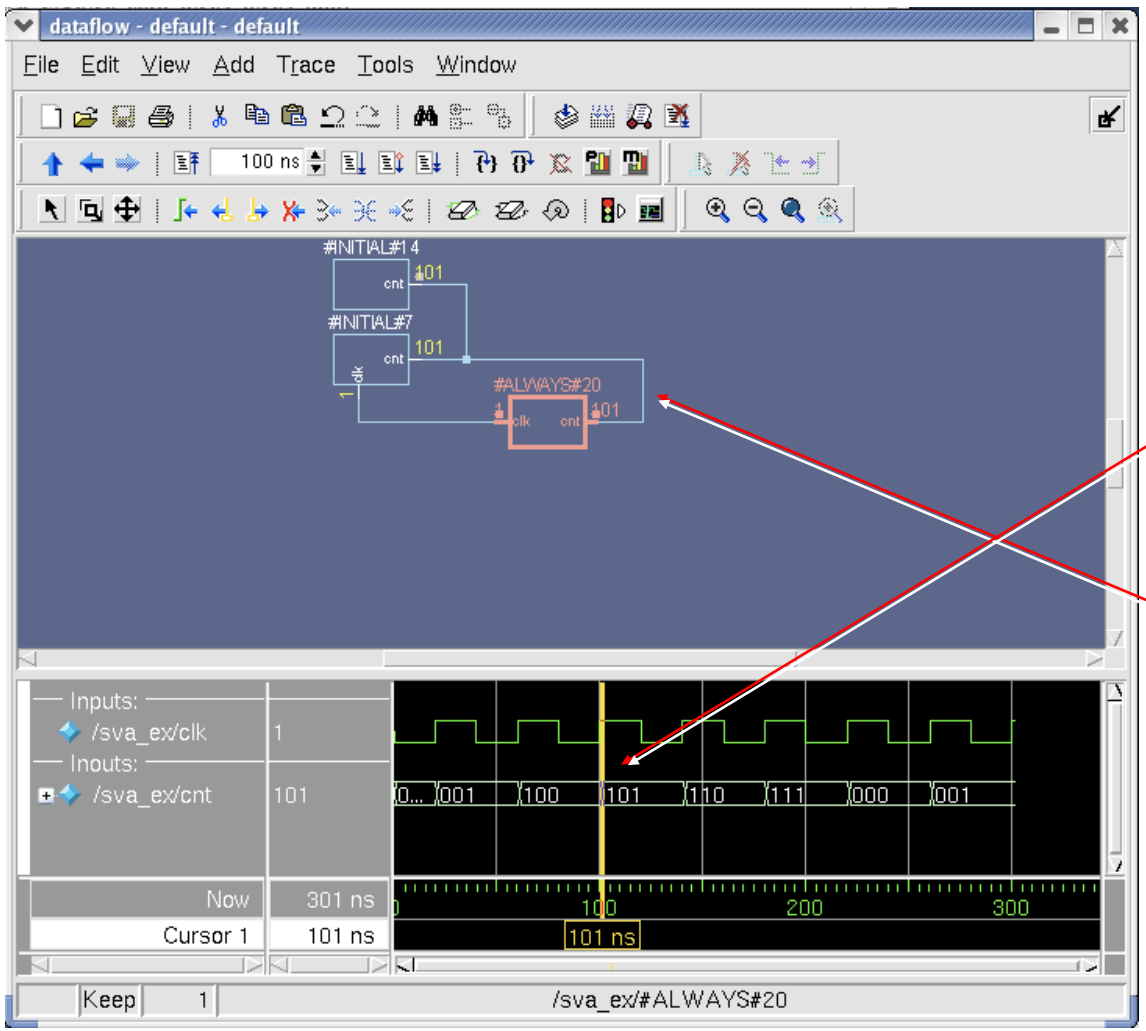


Assertion Debug Pane showing additional failure info

Intro to SVA - Assertion Debug -2

There is another way to get more information on assertion failures:

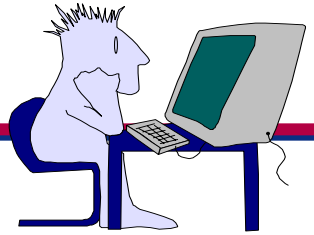
5. In Wave, double-click on the red triangle that indicates an assertion failure



The Dataflow Pane now shows waveform data for that assertion… and clicking on a signal transition on that waveform...

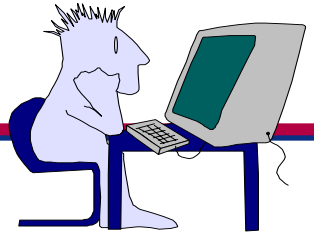
will show the dataflow diagram for that signal or transition

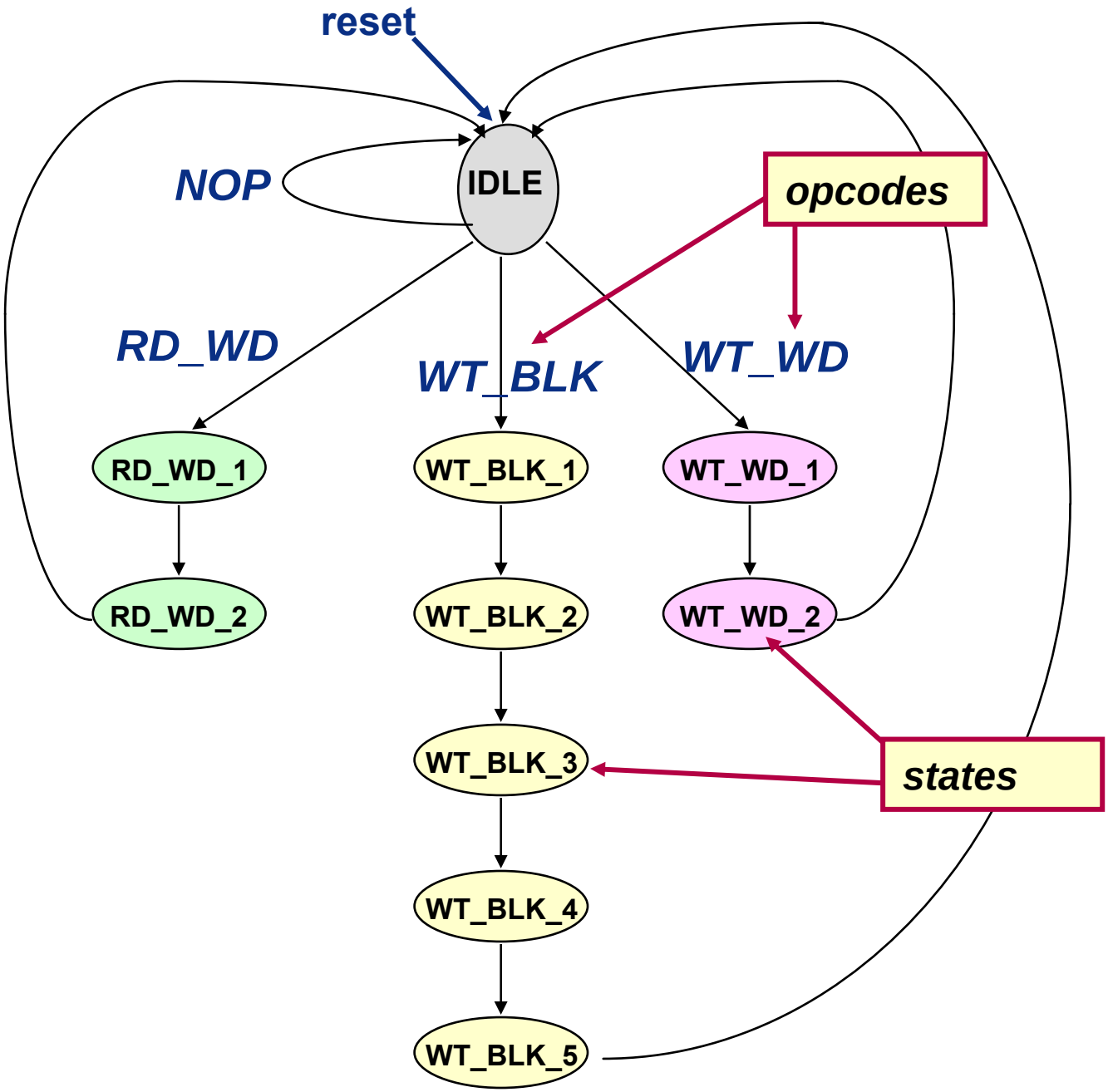
Lab – Simple Assertions: Overview



- Lab directory: **sva_q/fsm**
- Overview:
  - Write simple assertions to verify the state flow of a state machine
- Edit the file sva_container.sv
  - Add assertions to module **`sva_container`** to verify that the state machine transitions from state to state correctly
    - Add sequences, properties and assert statements as needed
    - State diagram is on the next slide
  - The module **`sva_container`** is "bound" to the sm module such that all the signals you need are visible inside this module
    - All needed signals are inputs to this module
    - We will talk more about binding and how it works later
  - The enumeration of the state values are in a package in the file types_pkg.sv
- Run "make" or "make gui" to compile and run
  - You should get no assertion failures
- Run "make bad" to verify your assertions catch errors

Lab – Simple Assertions: State Transitions





Sol

Sequences

In this section

- Sequence Blocks – more
- Sequence Operators
- Sequence Methods
- Relating Sequences

Sequence Block with Arguments

- Sequence blocks may have formal arguments that substitute for
  - Identifiers, Expressions, Event control expressions
  - Upper delay range or repetition range if the actual argument is $
- Arguments may typed or untyped

```
sequence s1;
   @ (posedge clk)
         a ##1 b ##1 c;              // s1 evaluates on each successive edge of clk
endsequence

sequence s2 (data,en);               // sequence with name AND arguments
   (!frame && (data ==data_bus)) ##1 (c_be[0:3] == en);
endsequence

sequence s3;
   start_sig ##1 s2(a,b)  ##1 end_sig;     // sequence as sub-expression
endsequence

Where:  s3
         - same as -
         start_sig ##1 (!frame && (a == data_bus)) ##1 (c_be[0:3] == b)  ##1 end_sig;
```

Sequence Operators

- Available sequence operators (in order of precedence):

`[*N]  /  [*M:N]`	- consecutive repetition operator
`[=N]  /  [=M:N]`	- non-consecutive repetition
`[->N]  /  [->M:N]`	- goto repetition (non-consecutive, exact)
`and`	- all sequences expected to match, end times may differ
`intersect`	- all sequences expected to match, end times are the SAME
`or`	- 1 or more sequences expected to match
`throughout`	- expression expected to match throughout a sequence
`within`	- containment of a sequence expression
`##`	- sequence delay specifier

Range

Range:

```
a ##[3:5] b  // a is true on current tick, b will be true 3-5 ticks from current tick
a ##[3:$] b  // a is true on current tick, b will be true 3 or more ticks from now
```

$ represents a non-zero and finite number

SVA Coding Style Tips

1. Avoid very large delimiters (`a##[1:1023]`) because they may consume cpu time and resources. Use signals instead.

2. Open-ended delay ranges can be a problem since they cannot fail. At end of simulation, if they haven't matched… not a fail either… Consider adding a "guard" assertion around them to define a timeout period (see intersect operator in next section)

Consecutive Repetition `[*N]`

- **Consecutive Repetition:**

```
a [*N]                    // repeat a , N times consecutively
```

- **Examples:**

```
a ##1 b ##1 b ##1 b ##1 c   // long-winded sequence expression
a ##1 b [*3] ##1 c          // same… but using efficient *N syntax
```

- **Between each repetition of the sequential expression is an implicit `##1`**

```
b [*3]          - same -   b ##1 b ##1 b        // b true over 3 ticks total
(a ##2 b)[*3]   - same -   (a ##2 b) ##1 (a ##2 b) ##1 (a ##2 b )
```

Consecutive Repetition [*N:M]

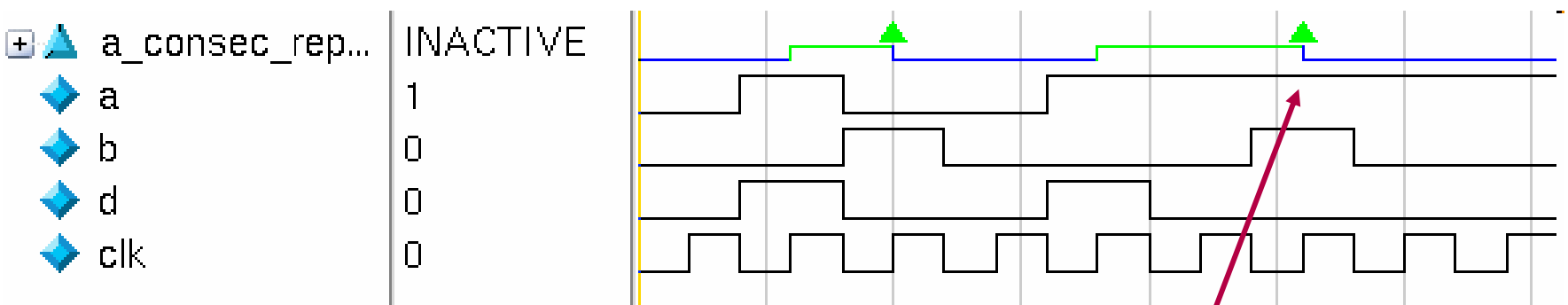
- **Consecutive Range Repetition:**

```
a [*N:M]        // repeat a at least N
                // and as many as M times consecutively
```

- **Examples:**

```
property p_consec_repetition;
 @(posedge clk) d |-> a [*1:2] ##1 b;
endproperty
```

same as: (a ##1 b)



or same as: (a ##1 a ##1 b)

Code in examples_sva/consec_repetition.sv

Consecutive Repetition `[*N:$]`

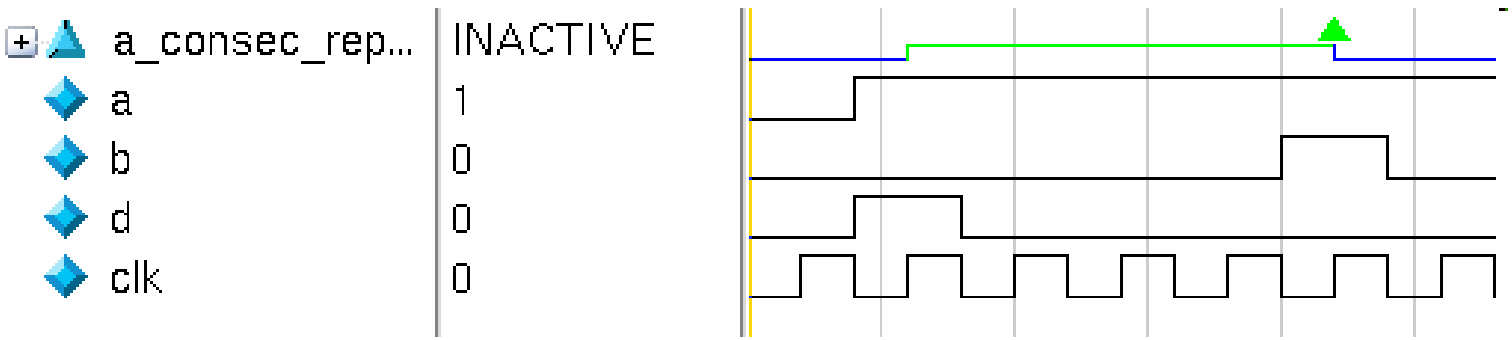
- **Consecutive Range Repetition with $:**

```
a [*N:$]
```
*// repeat **a** an unknown number of times but at least **N** times*

- **Example:**

```
property p_consec_repetition;
 @(posedge clk) d |-> a [*1:$] ##1 b;
endproperty
```

Succeeds!



Code in examples_sva/consec_repetition.sv

Consecutive Repetition `[*0:M]`

- **Consecutive Range Repetition:**

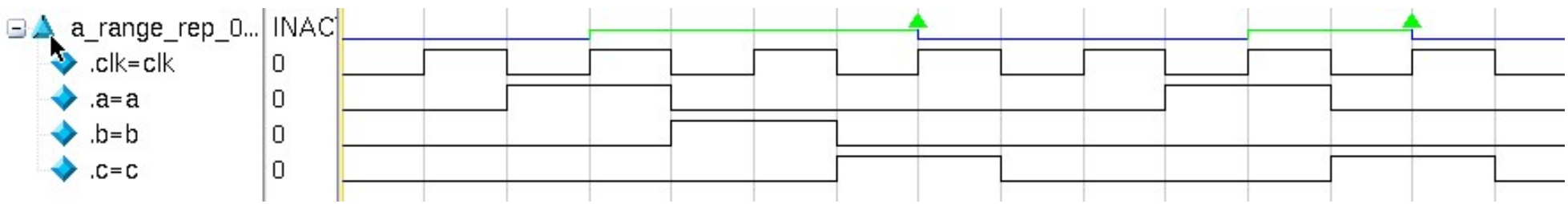
```
a [*0:M]        // repeat a at least 0 times
                // but no more than M times consecutively
```

- **Example:**

```
property p_range_rep_0_exb;
  @(posedge clk) a |-> (a ##1 b [*0:1] ##1 c);
endproperty
```

same as: (a ##1 b ##1 c)

or same as: (a ##1 c)

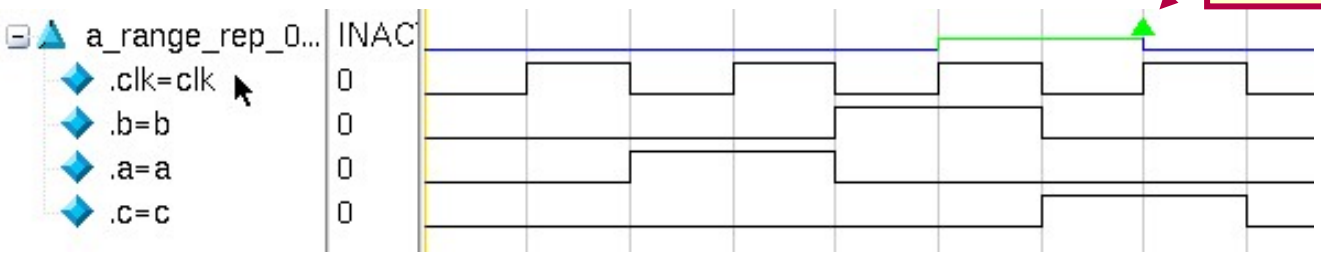


Consecutive Repetition [*0:M] - 2

- **Examples:**

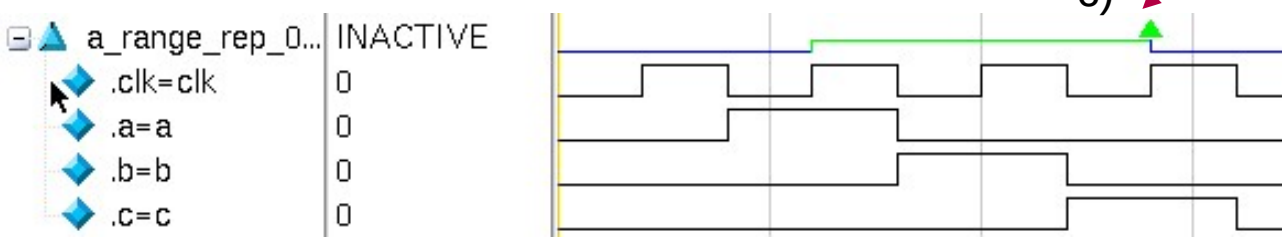
```
property p_range_rep_0_exa;
  @(posedge clk) b |-> (a[*0:1] ##1 b ##1 c);
endproperty
```

same as: (b ##1 c)




```
property p_range_rep_0_exa;
  @(posedge clk) a |-> (a[*0:1] ##1 b ##1 c);
endproperty
```

or same as: (a ##1 b ##1 c)



Code in examples_sva/range_repetition_0.sv

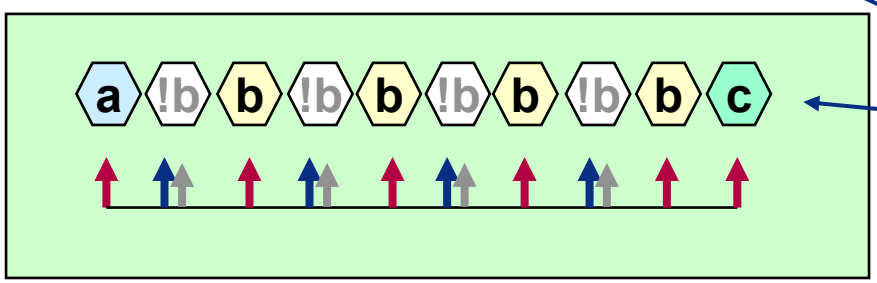
Goto Repetition `[->N]`

- **Non-consecutive repetition: (repeats a boolean expression)**

```
b [->N]                    // goto the N'th repeat of b
```

- **Example:**

```
a ##1 b [->4] ##1 c        // a followed by exactly
                           // 4 not-necessarily-consecutive b's,
                           // with last b followed next tick by c
```



VERY important:
c is expected to happen 1 tick after final **b**

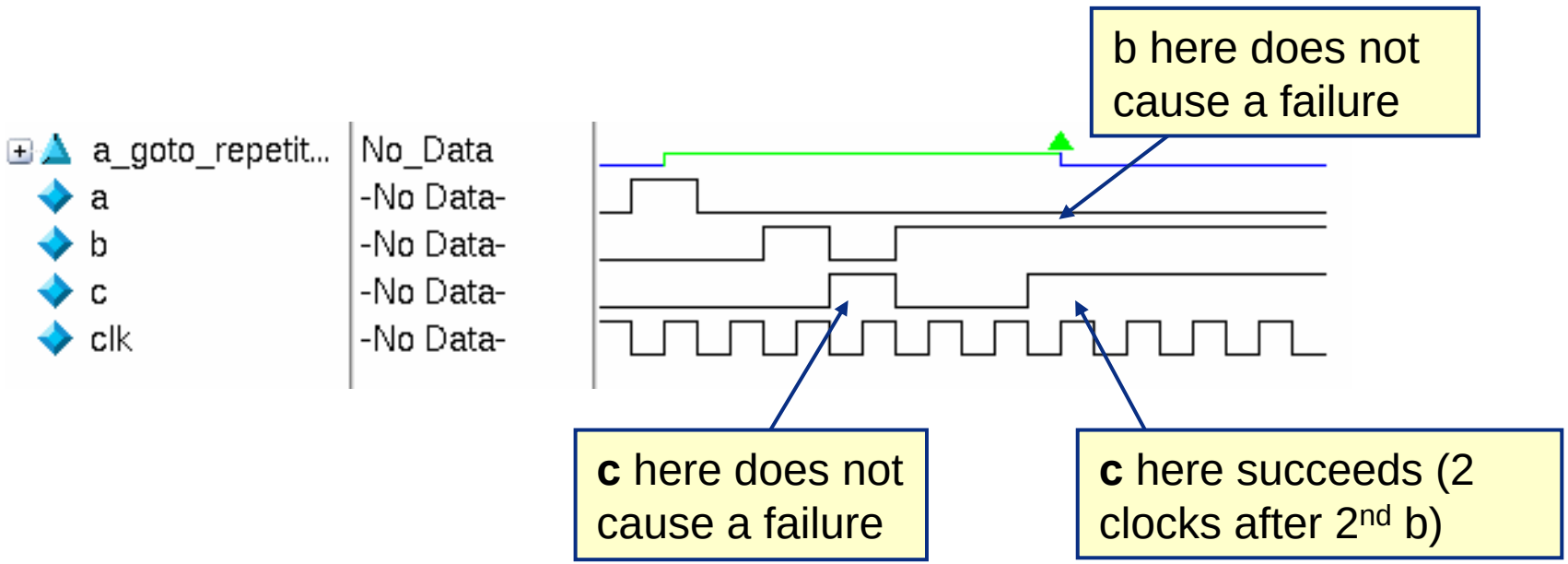
- **There may be any number (including zero) of ticks where b is false before and between but not after the 4 repetitions of b**

Goto Repetition - Observations

- **Given**

  `a ##1 b [->2] ##2 c` // ***b*** *followed* *2 ticks later* *by* ***c***

- **What if there is a c before the 2nd b?**
  or perhaps there is a 3rd b before c?



Goto Repetition `[->N:M]`

- **Non-consecutive repetition: (repeats a boolean expression)**

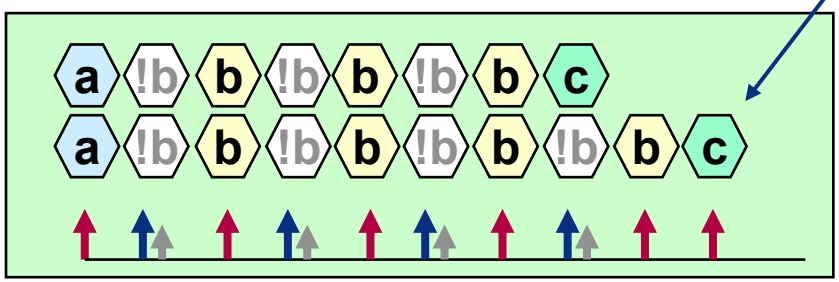
```
a ##1 b [->N:M] ##1 c
```

*// **a** followed by at least **N**,*
*// at most M **b**'s before **c***
*// NOTE the last **b** is followed <u>next tick</u> by **c***

- **Example:**

```
a ##1 b [->3:4] ##1 c;
```

VERY important: c is expected to happen 1 tick after final **b**



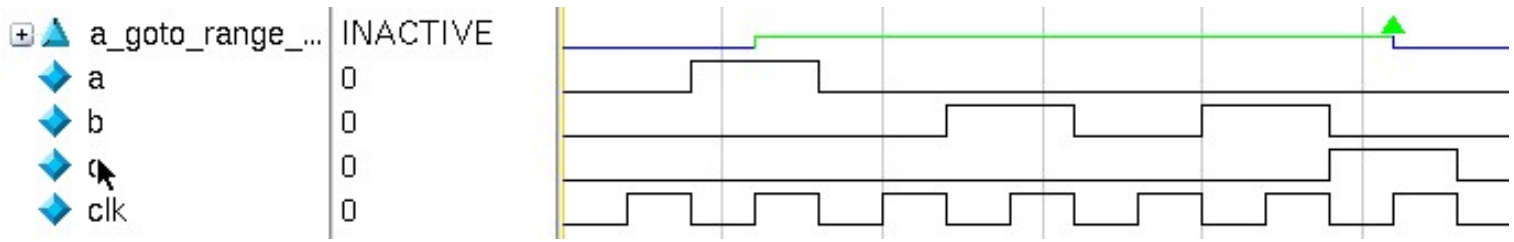
- **There may be any number (including zero) of ticks where b is false before and between but not after the 3:4 repetitions of b**

Goto Repetition `[->N:M]` Examples

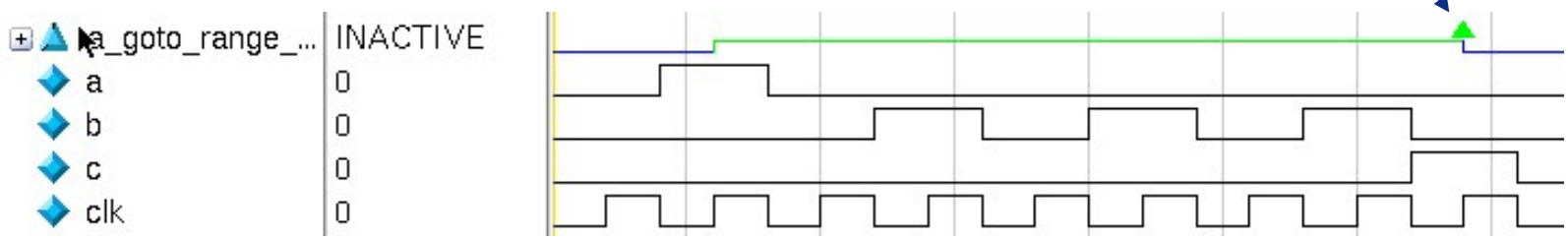
- **Given**

```
property p_goto_range_rep;
  @(posedge clk) a |->  b [->2:3] ##1 c  ;
endproperty
```

c here succeeds
(1 clock after 2nd b)



c here succeeds
(1 clock after 3rd b)



Non-Consecutive Repetition [=N]

- **Non-consecutive repetition:**

```
b [=N] ##1 c
```

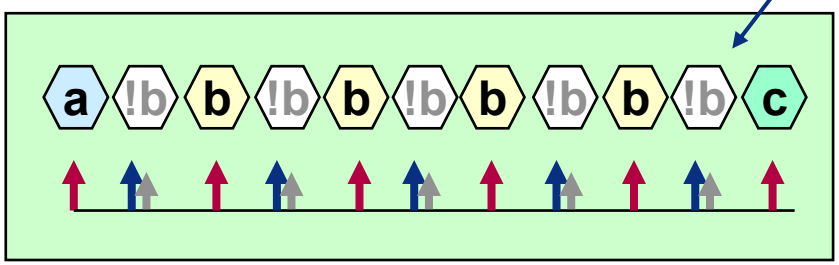
*// repeat **b, N** times (not necessarily consecutive)*
*// followed sometime by **c***

- **Example:**

```
a ##1 b[=4] ##1 c
```

*// **a**, then exactly 4 repeats of **b**, then **c***

VERY important
c does not have to follow immediately after the last **b**



Non-Consecutive Repetition [=N:M]

- **Non-consecutive repetition (range):**

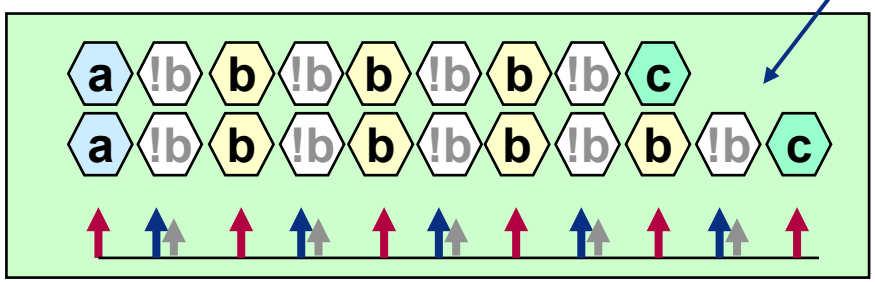
```
a ##1 b [=N:M] ##1 c
```

*// **a** followed by at least **N***
*// but no more than **M** repeats of **b***
*// followed sometime by **c***

- **Example:**

```
a ##1 b[=3:4] ##1 c
```

VERY important
c does not have to follow immediately after the last **b**



- **There may be any number (including zero) of ticks where b is false before, between and after the 3:4 repetitions of b**

Handy System Functions

- Functions for use in boolean expressions

`$onehot (<expression>)`	returns true if only one bit of the expression is high
`$onehot0 (<expression>)`	returns true if at most one bit of the expression is high
`$isunknown (<expression>)`	returns true if any bit of the expression is ‘x’ or ‘z’
`$countones (<expression>)`	returns the # of 1's in a bit-vector

Value Change Functions

SVA Coding Style Tip

V-C functions have an overhead … consider whether a simple boolean level isn't enough

- These 3 functions detect transitions between 2 consecutive ticks

  `$rose ( expression [, clocking_event])`
  - True if LSB of expression went lo-hi

  `$fell ( expression [, clocking_event] )`
  - True if LSB of expression went hi-lo

  `$stable ( expression [, clocking_event] )`
  - True if no change between the previous sample of expression and current sample

```
sequence s1;
  $rose(a) ##1 $stable(b && c) ##1 $fell(d) ;
endsequence
```

- Rising edge on LSB of `a`, followed by `b` & `c` not changing, followed by a falling edge on LSB of `d`

Use Outside of Concurrent Assertions

- Value change functions may be used in SystemVerilog code outside of concurrent assertions

```
always @(posedge clk)
  reg1 <= a & $rose(b);
```

  - The clocking event (posedge clk) is applied to `$rose`
  - `$rose` is true whenever the sampled value of b changed to 1 from its sampled value at the previous tick of the clocking event.

Relating Sequences Together

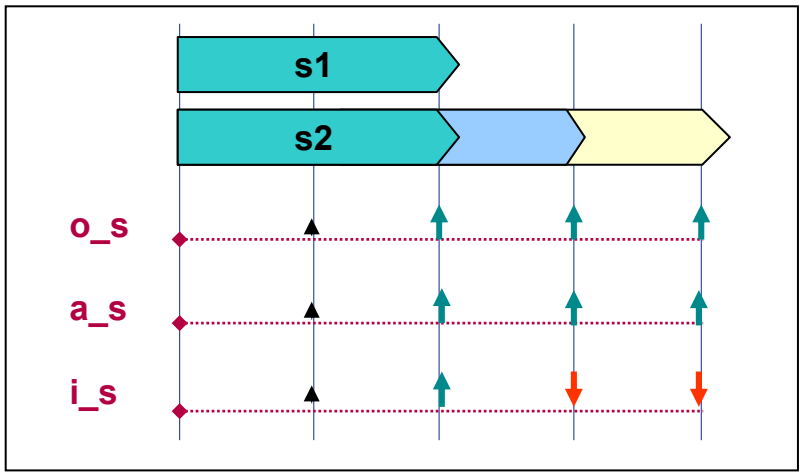
- We have already seen how sequences can be assembled hierarchically:

```
sequence s1 (a,b);
  ( a == var ) ##1 (var2[0:3] >= b);
endsequence

sequence s2;
   sig ##1 s1(x,y)  ##1 !sig;
endsequence
```

- There are three other ways to logically relate 2 or more sequences:
  - *Sequences with same - or - different start-times:*
    - `S1 or S2` *// at least one sub-sequence matches*
  - *Sequences with same start-time:*
    - `S1 intersect S2` *// all sub-sequences match simultaneously*
    - `S1 and S2` *// all sub-sequences match but not necessarily*
      // simultaneously

Sequence Expressions: `and / or / intersect`

```
sequence s1;
  a ##1 b  ##1 c;
endsequence

sequence s2;
  d ##[1:3] e  ##1 f;
endsequence
```



Examples:

```
sequence o_s;
  s1 or s2;          // o_s matches if at least one sub-sequence matches
endsequence

sequence a_s;        // both expressions must match
  s1 and s2;         //   (first to match waits for the other)
endsequence          // a_s  endtime is later of s1 or s2

sequence i_s;        // both expressions must match
  s1 intersect s2;   //   (both sequences must end at same time)
endsequence          // Here: i_s  matches only if  e occurs
                     // 1 cycle after d
```

Sequences MUST have same start time

Condition Over a Sequence - throughout

- Often, some property is expected to hold during a sequence -or- perhaps NOT hold during a sequence

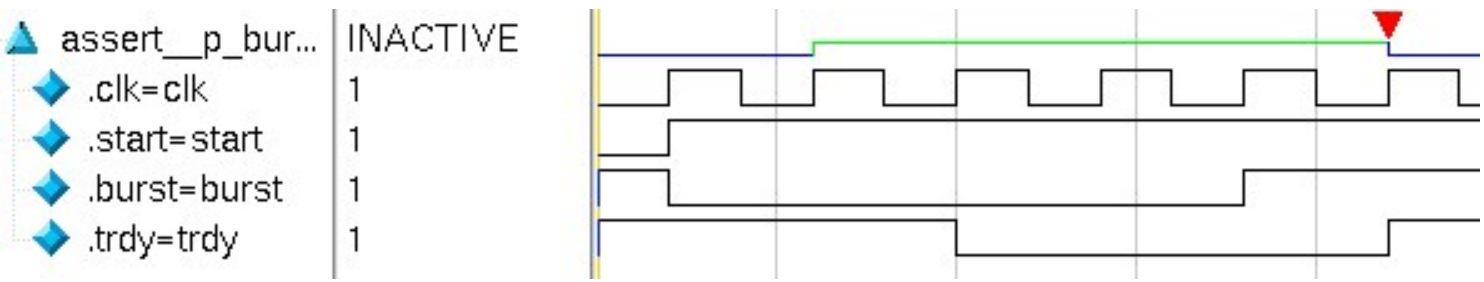
```
req  throughout ( gnt [->1] )
```

VERY useful technique

  - `req` must stay true until the first time `gnt` is sampled true

```
sequence burst_rule;
   @(posedge clk)
       $fell (burst) ##0
       (!burst) throughout (##2 (!trdy [*3]));
endsequence
```

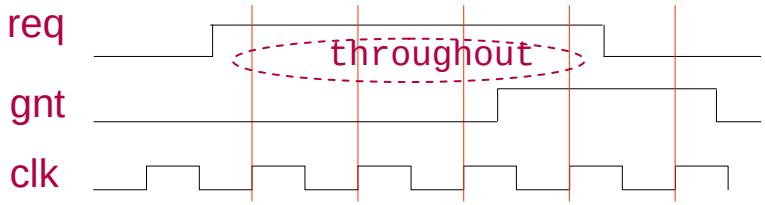
Failure because burst was not low throughout



- Here, when `burst` goes true (low), it is expected to stay low for the next 2 ticks and also for the following 3 clock ticks, during which `trdy` is also to stay low.

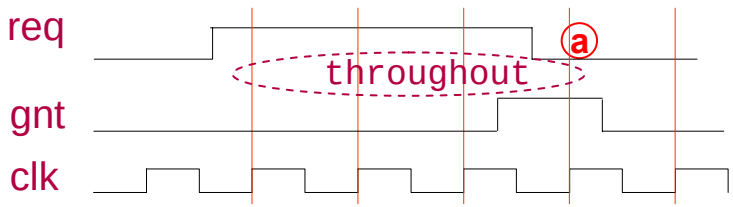
Caution ! - throughout

- The throughout sequence must be carefully described




```
req  throughout ( gnt [->1] )
```

  - Works great because req doesn't deactivate until 1 tick after gnt goes active
  - But what if the specification permits req to drop asynchronously with gnt?



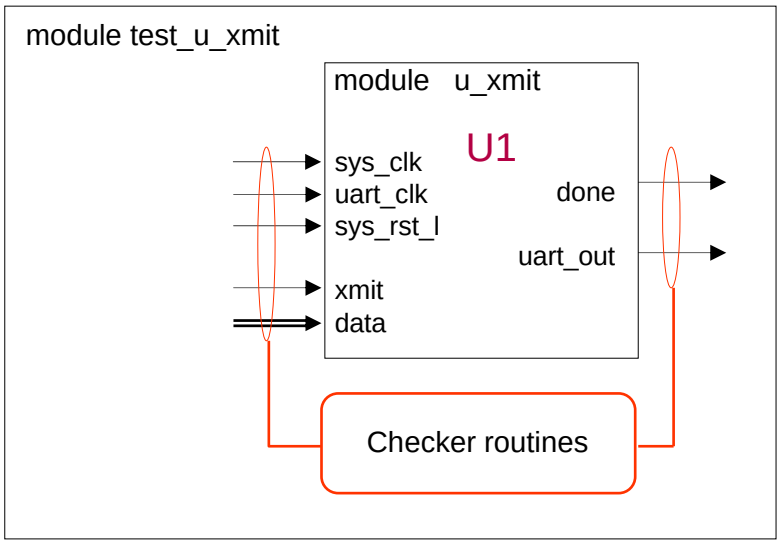
  - At sample point a (the last of the throughout clause), req is already false so the assertion fails…
  - In this case it might be easier to say:

```
req [*1:$] ##1 gnt
```

Warning: This may never fail if simulation ends before gnt

UART Lab : UART Transmitter

- For the next few labs we'll be developing assertions to verify an RTL UART transmitter module. Design hierarchy and pin-names are described below:



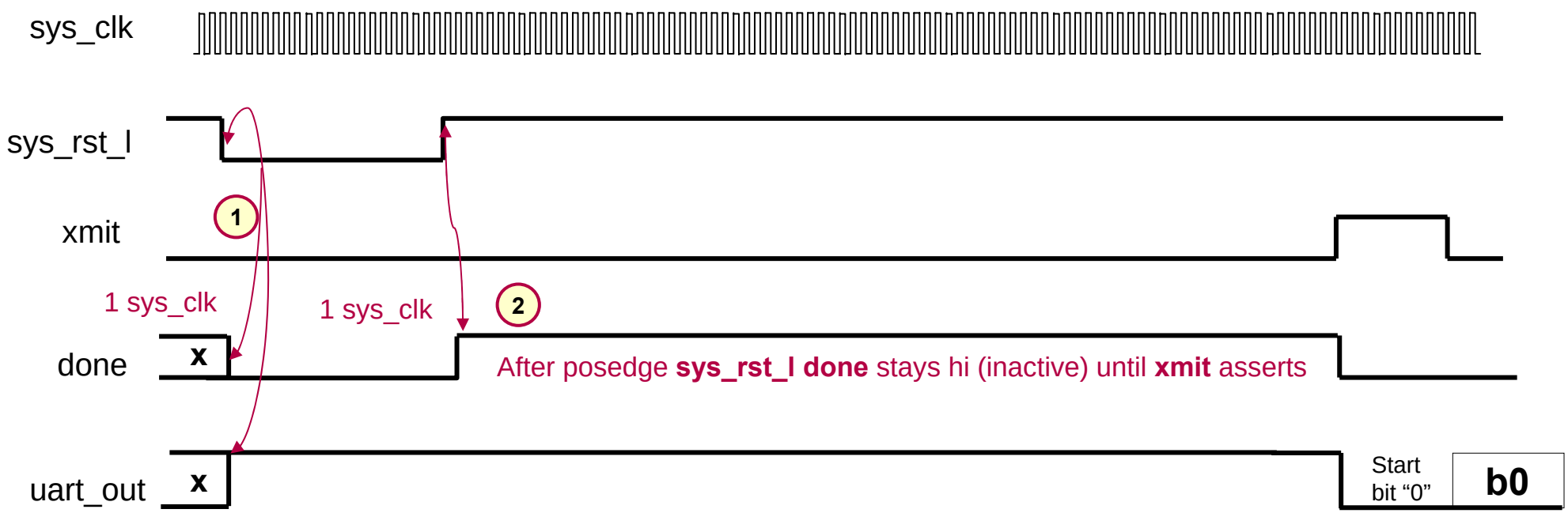
Signals	
sys_clk	**System clock**
uart_clk	**Serial clock (sys_clk ÷16)**
sys_rst_l	**Reset (active low)**
xmit	**Load data and start transmission**
data	**Byte of data to transmit**
done	**Done flag (low during transmission**
uart_out	**Serial bitstream out**

- In each lab you will be given "specification" waveforms on which you will base the assertions you write

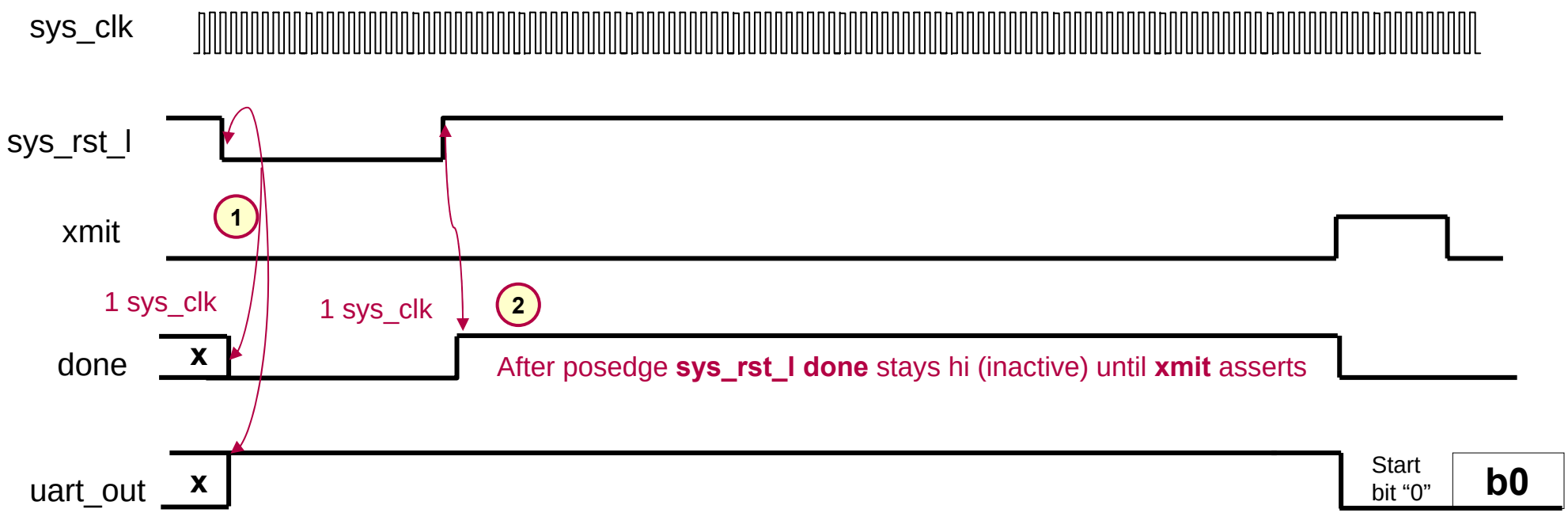
Worked Example: Reset Assertions

- To begin, let's look at a worked example involving some UART outputs and what they are expected to do at reset time

Timing diagram



Worked Example: Description



- From the waveforms, we can derive these 2 sequences:
  1. "s_rst_sigs"
     assert uart_out(hi) and done(lo) 1 sys_clk after sys_rst_l(negedge)
  2. "s_rst_done"
     assert done(hi) and xmit(lo) 1 sys_clk after sys_rst_l
     goes inactive(posedge) and remain so until xmit(posedge)
  3. Also, let's combine those 2 sequences into another "parent" sequence:
  4. "s_rst_pair"
     assert that "s_rst_sigs" and "s_rst_done" both match.

Worked Example: Property

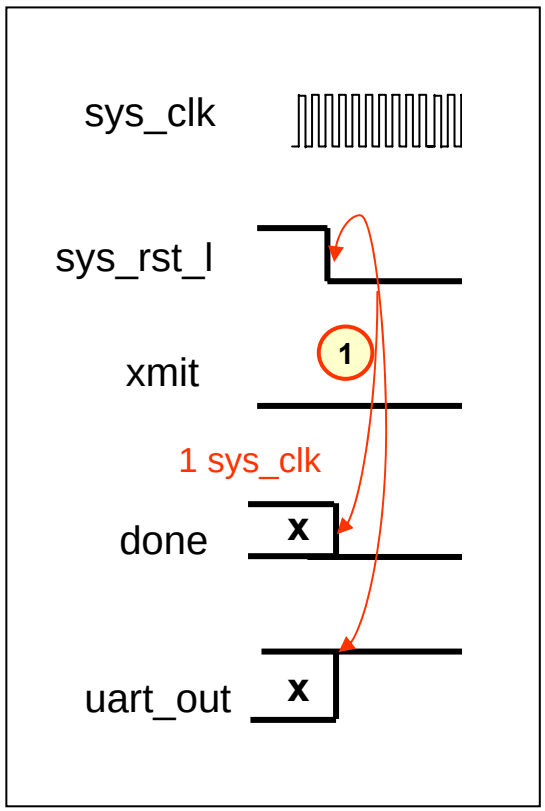
- Since this first example involves writing three simple sequences, we're going to assume the following property and assert statements (discussed shortly):

```
property p_post_rst;
    @(posedge sys_clk) ($fell(sys_rst_l)) |-> s_rst_pair;
endproperty

assert_post_rst: assert property (p_post_rst)
                        else $display("%m : device did not reset");
```

- This code asserts a property (p_post_rst), which uses our "parent" sequence (s_rst_pair) to specify the behavior of the UART signals done and uart_out at reset.
- For now, we'll place the assertions inside the testbench test_u_xmit.sv:
- So, let's write the 3 missing sequences…

Worked Example: s_rst_sigs

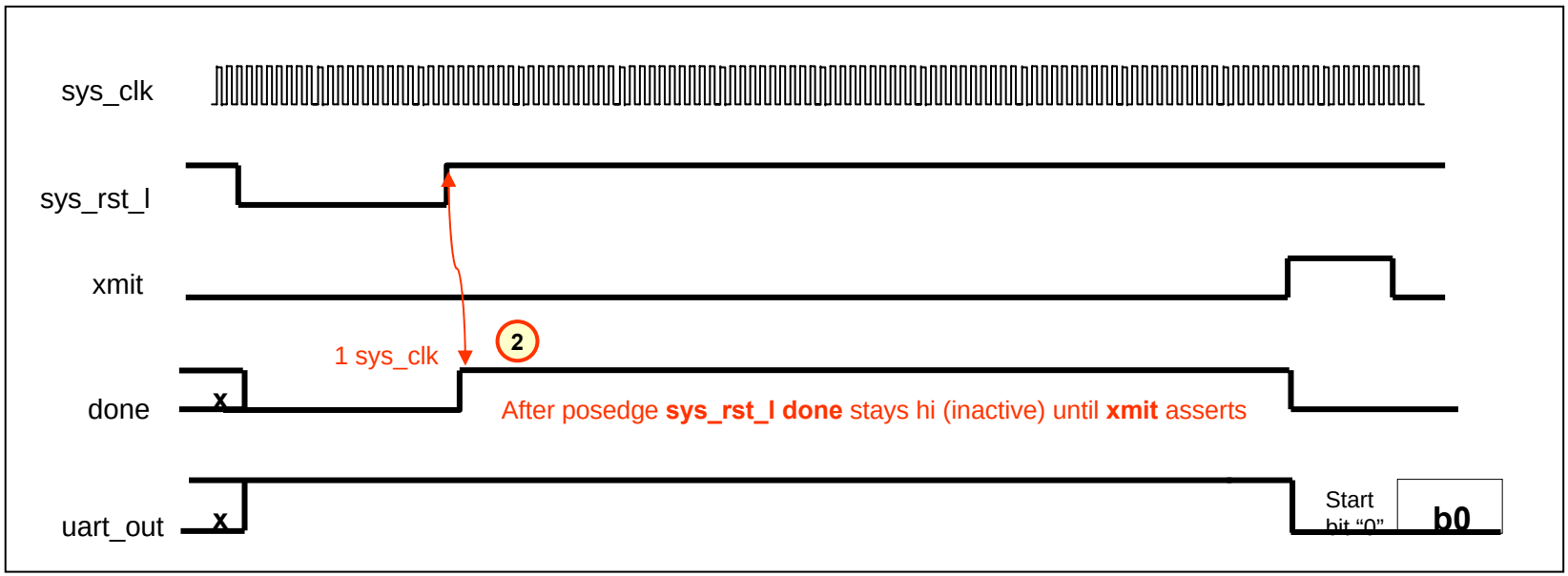
- First, we know that the property "triggers" at `$fell ( sys_rst_l )` so that will be the assumed startpoint of our 3 sequences
- Second, the spec says that one `sys_clk` cycle later done and `uart_out` respond to the active reset




```
sequence s_rst_sigs;
   ##1 (uart_out && !done);
endsequence
```

Worked Example: s_rst_done

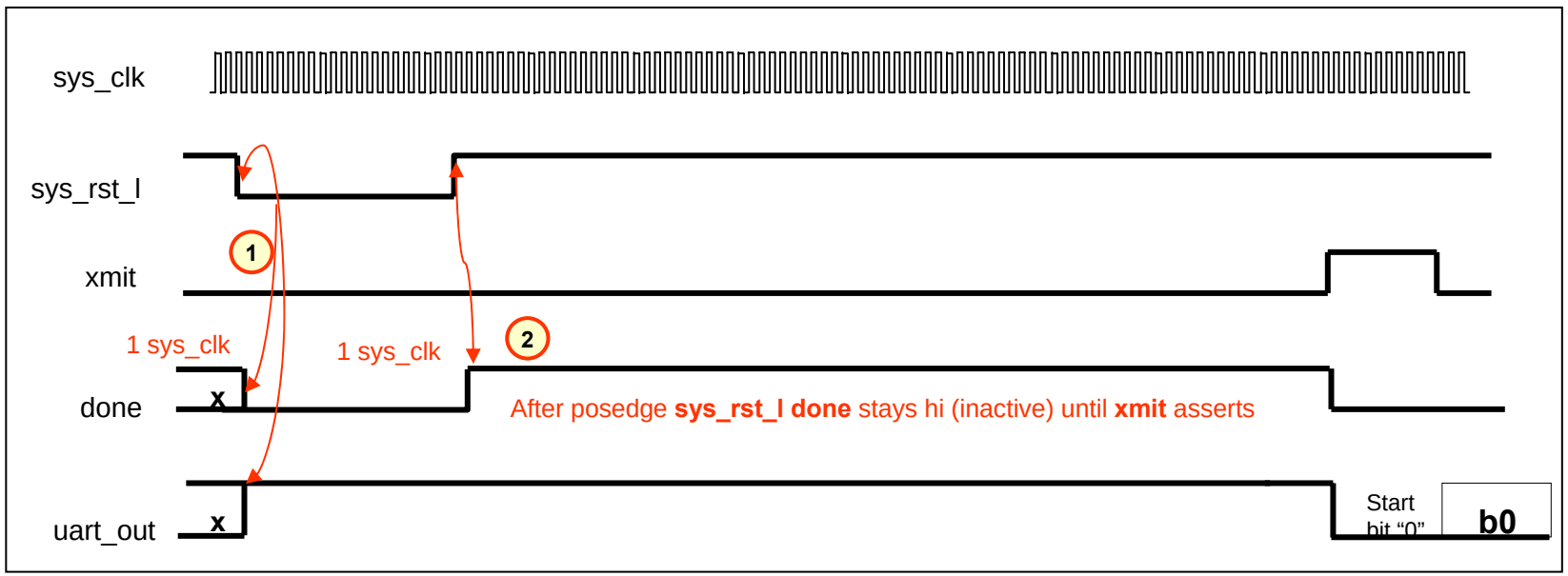
- Again, we know that the property "triggers" at **`negedge sys_rst_l`**
- However we are interested in what happens at the following **`posedge sys_rst_l`**, so we need to allow for an indefinite length **reset** pulse.
- Also, we can't predict when **`xmit`** will rise, so we need to continually assert the behavior of **`done`** between **`posedge sys_rst_l`** and **`xmit`** going true.




```
sequence s_rst_done;
  (sys_rst_l)[->1] ##1 (done throughout ((xmit)[->1]));
endsequence
```

Worked Example: s_rst_pair

- Both sub-sequences start at the same time but with different end (match / fail) times
- So, we use the **and** expression to relate the two sub-sequences



Q: Why not use intersect ?

```
sequence s_rst_pair;
  s_rst_sigs and s_rst_done ;
endsequence
```

A: Because intersect implies that both sequences have the same start and completion times

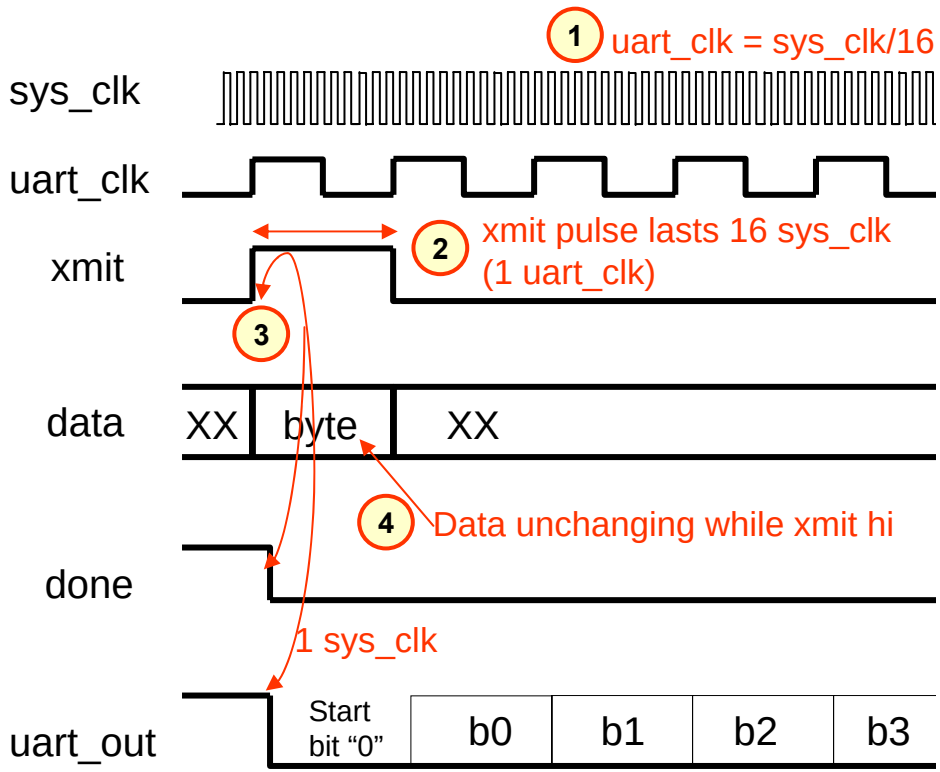
Worked example: Complete

test_u_xmit.sv

```
sequence s_rst_sigs;
   ##1 (uart_out && !done);
endsequence

sequence s_rst_done;
 (sys_rst_l)[->1] ##1 (done throughout ((xmit)[->1]));
endsequence

sequence s_rst_pair;
  s_rst_done and s_rst_sigs;
endsequence

property p_post_rst;
  @(posedge sys_clk) ($fell(sys_rst_l)) |-> s_rst_pair;
endproperty

assert_post_rst: assert property ( p_post_rst )
                 else
                    $display("%m : device did not reset fully");
```

Lab – UART Sequences: waveform

Timing diagram



Signals	
sys_clk	System clock
uart_clk	Serial clock (sys_clk ÷ 16)
sys_rst_l	Reset (active low)
xmit	Load data and start transmission
data	Byte of data to transmit
done	Done flag (low during transmission
uart_out	Serial bitstream out

Lab – UART Sequences : Instructions - 2

- Working directory: sva_q/sva_uart_xmit
- Examine the design (file: test_u_xmit.sv) and find the following code:

```
LAB STARTS HERE
property p_uart_sys16;
  @(posedge sys_clk) $rose(uart_clk) && !sys_rst_l |-> s_uart_sys16;
endproperty

property p_xmit_hi16;
  @(posedge sys_clk) $rose(xmit) |-> s_xmit_hi16;
endproperty

property p_xmit_done;
  @(posedge sys_clk) $rose(xmit) |-> s_xmit_done;
endproperty

property p_xmit_nc_data;
  @(posedge sys_clk) ($rose(xmit)) |=> s_xmit_nc_data;
endproperty
```

Q: Why is this here ?

A: For efficiency, if we didn't qualify this check it would happen throughout simulation, very wasteful of resources…

Lab – UART Sequences : Instructions - 2

test_u_xmit.sv

```
//XXXXXXXXXXXXXXXXXXXX  LAB   XXXXXXXXXXXXXXXXXXXX
//XXXXXXXXXXXXXXXXXXXX STARTS XXXXXXXXXXXXXXXXXXXX
//XXXXXXXXXXXXXXXXXXXX  HERE  XXXXXXXXXXXXXXXXXXXX

    WRITE YOUR SEQUENCES HERE

//XXXXXXXXXXXXXXXXXXXX END OF LAB  XXXXXXXXXXXXXXXXXXXX
//XXXXXXXXXXXXXXXXXXXX END OF LAB  XXXXXXXXXXXXXXXXXXXX
//XXXXXXXXXXXXXXXXXXXX END OF LAB  XXXXXXXXXXXXXXXXXXXX



. . .
```

- Edit the design (file: **test_u_xmit.sv**)
  - At the indicated area of the file (nowhere else!!), write these 4 sequences.
    - **s_uart_sys16**: Verify uart_clk == sys_clk / 16
    - **s_xmit_hi16**: xmit stays hi for 16 sys_clk cycles
    - **s_xmit_done**: One sys_clk after xmit (assert, hi), we expect to see done(de-assert, lo) and uart_out(lo)
    - **s_xmit_nc_data**: While xmit is asserted(hi) data value is unchanging
- Compile & run your simulation

```
vlog   <files>       +acc=a
vsim  <top_of_hier> -voptargs=+acc -assertdebug
```

Sol

Sequence Expressions

In this section

Expressions
Procedural controls

Sequence Expressions

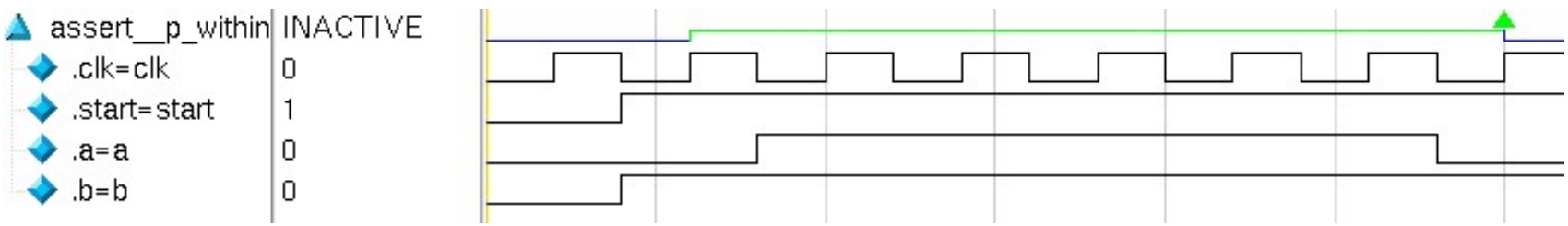
- There are several more ways to create logical expressions over sequences
- **Sequence Expressions**
  - `(seq1) within (seq2)` - *contain seq1 within seq2*
  - `seq.ended` - *detect end of seq within another sequence*
- **Expressions within procedural code**
  - `@ clocked_seq` - *Block code thread until clocked_seq matches*
  - `clocked_seq.triggered` - *Instantly returns true if clocked_seq has matched in the current timestep, false otherwise*

Sequential Expressions - `within`

- **within** allows containment of one sequence within another
  - The first expression may begin when or after the second expression begins
  - The first expression must match before or when the second expression matches

```
sequence s_within;
  a[*5] within ( $rose ( b) ##1 b[*6] );
endsequence
```

  - Here, **a** must be high for 5 consecutive cycles contained within the second expression

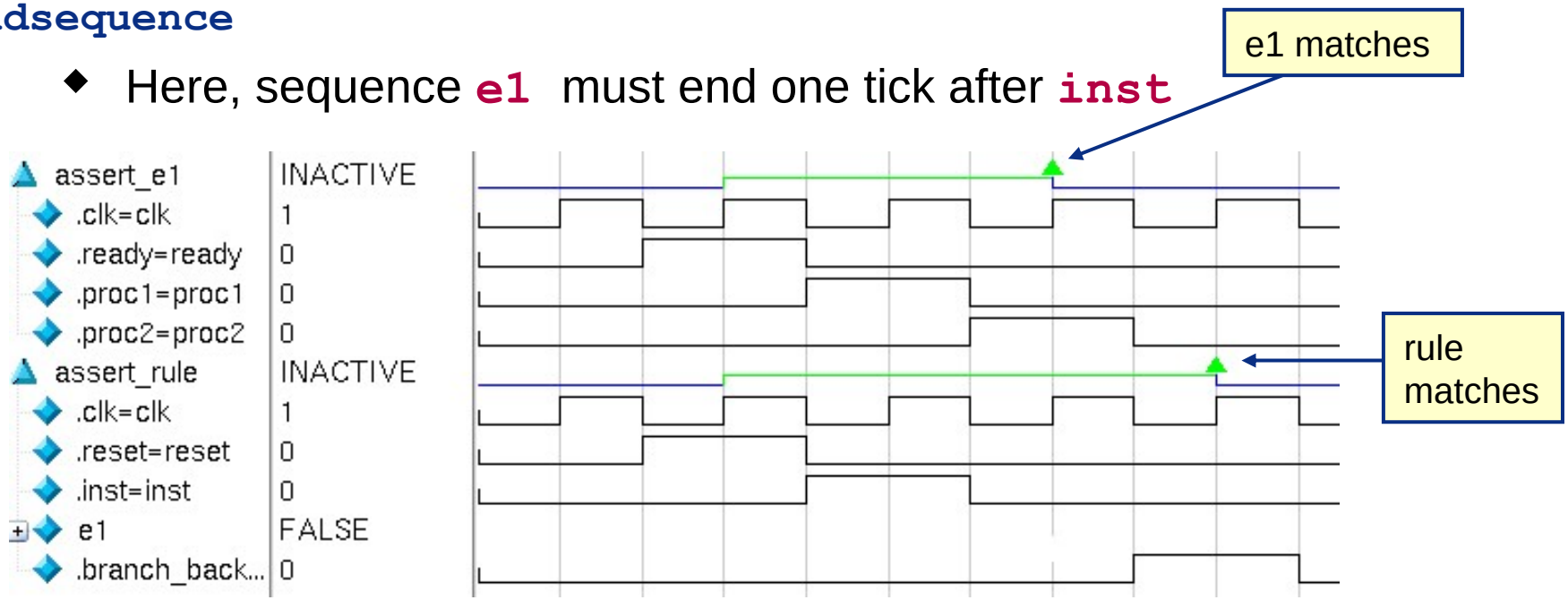


Sequential Expressions - ended

- `.ended` allows detection that a sequence expression has ended (matched)
  - It is a method on a sequence
  - The referenced sequence must specify its reference clock
    - It may be asserted independently but if not, assertion is implied by **.ended**

```
sequence e1;
   @(posedge clk) $rose(ready) ##1 proc1 ##1 proc2;
endsequence

sequence rule;
  reset ##1 inst ##1 e1.ended ##1 branch_back;
endsequence
```

- Here, sequence `e1` must end one tick after `inst`



Procedural Sequence Controls

- There are two ways to detect the endpoint of a sequence from within sequential code
  - In both cases a verification directive (assert property) is implied for the sequence.

Given:

```
sequence abc;
   @(posedge clk) a ##1 b ##1 c;
endsequence

// edge-triggered
always @ (abc)
   $display( "abc succeeded" );
```

```
sequence def;
   @(negedge clk) d ##[2:5] e ##1 f;
endsequence

// level sensitive
initial
   wait( abc.triggered || def.triggered )
   $display( "Either abc or def succeeded" );
```

@ unblocks the always block each time the sequence endpoint is reached i.e. each time it matches

***.triggered** indicates the sequence has reached its endpoint in the current timestep. It is set in the Observe Region and remains true for the rest of the time step*

- **NOTE**:
  - Both controls imply an instantiation of the sequence (i.e. assert property(seq))
  - Only clocked sequences are supported
  - Local sequences only (Hierarchical references are not supported)

Procedural Control Example router_assertions.sv

Router/coverage/router_assertions.sv

```
module router_assertions(input clk,
             input bit[7:0] valid, stream );
import defs::*;

sequence s_pass_thru_0 ;
  @(posedge clk)
  $rose(valid[0]) ##1 !stream[0]
   ##1 !stream[0] ##1 !stream[0];
endsequence

sequence s_pass_thru_1 ;
  @(posedge clk)
  $rose(valid[1]) ##1 !stream[1]
   ##1 !stream[1] ##1  stream[1];
endsequence

// 2 - 6 not shown

sequence s_pass_thru_7 ;
  @(posedge clk)
  $rose(valid[7]) ##1  stream[7]
   ##1  stream[7] ##1  stream[7];
endsequence

always @( s_pass_thru_0) begin
   if (`TRACE_ON) $display("Passthru 0 ");
      passthru[ 0] = 1;
end

always @( s_pass_thru_1) begin
   if (`TRACE_ON) $display("Passthru 1 ");
   passthru[ 1] = 1;
end

// 2 - 6 not shown

always @( s_pass_thru_7) begin
   if (`TRACE_ON) $display("Passthru 7 ");
   passthru[ 7] = 1;
end

endmodule
```

Property Blocks

In this section

- Operators

Property Block - more

- Properties have
  - Special operators:
    - `disable iff, not, |->, |=>`
    - Also support for: `and, or, if…else`
  - Optional arguments

Optional arguments

Like an asynchronous reset for the property

```
property p1(a,b,c,d);
   @ (posedge clk) disable iff (reset)
         (a) |-> not ( b ##[2:3] c ##1 d );
endproperty
```

implication

Reversal: sequence must NEVER evaluate true

    - Here, when a is true the b/c/d expr. is forbidden

Property Block 2

- Properties are often built from sequences (though NOT vice-versa)
- Properties can appear in modules, interfaces, programs, clocking domains

SVA Coding Style Tips
- Always use disable iff to disable assertions during reset
- Be cautious, during reset some assertions must still be verified (reset control logic?)

MAC Property example

```
property p_valid_states;
  @(negedge MRxClk) disable iff(Reset)
     $onehot( {StateIdle, StatePreamble,
               StateSFD, StateData0, StateData1, StateDrop } );
endproperty

rx_statesm_valid : assert property( p_valid_states );
```

not Operator

- The keyword not
  - May be before the declared sequence or sequential expression
  - May be before the antecedent
  - Makes the property true, if it otherwise would be false
  - Makes the property false if it otherwise would be true

```
sequence s1;
   a ##1 b;
endsequence

property p1_not;
  @(posedge clk) not s1;
endproperty
```

p1_not is true if a is never true,
or if it is, one clock tick later b is never true

p2_not is true if the implication antecedent is never true,
or if it is, the consequent sequential expression succeeds

```
property p2_not;
  @(posedge clk) not (c && d)
      |-> not a ##1 b;
endproperty
```

or, and Operators

- **or** - property evaluates to true if, and only if, at least one of property expressions (s1, s2) evaluates to true

```
sequence s1;          sequence s2;
  a ##1 b;              b ##1 c;
endsequence           endsequence

property p_or;
  @ (posedge clk) t1 |->  s1 or s2 ;
endproperty
```

- **and** - property evaluates to true if, and only if, both property expressions evaluate to true

```
property p_and;
  @ (posedge clk) t2 |-> ( (a ##1 b) and (c |=> d) ) ;
endproperty
```

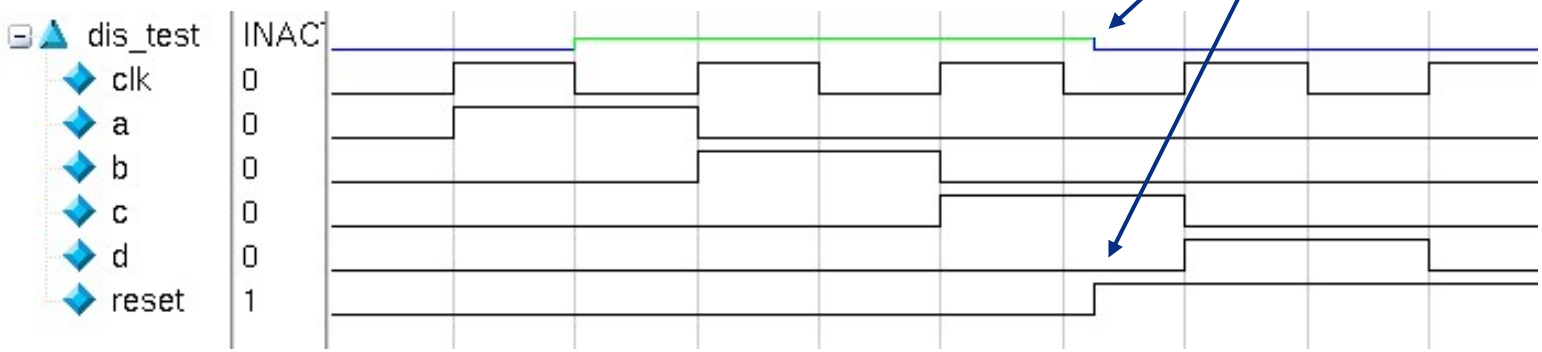
*If t2, then sequence (a##1b) must match **and** if c then we must see d 1 tick later.*

disable iff Clause

- Called the reset expression
- The reset expression is tested independently for different evaluation attempts of the property
- Enables the use of asynchronous or preemptive resets
  - The values of variables used in the reset expression are those in the current simulation cycle, i.e., not sampled
  - If the reset expression evaluates true then the evaluation attempt of the property is aborted immediately
    - Neither a pass nor a failure is reported

```
property p1;
  @(negedge clk) disable iff(reset)
    a |=> (b ##1 c ##1 d);
endproperty
```

Note that reset goes active asynchronously to the clk and the property goes inactive immediately



Past Value Function: `$past()`

- Returns the value of a signal from a previous clock tick

```
$past( expression1 [, number_of_ticks] [, expression2] [, clocking_event])
```

- **expression1** is required
- **number_of_ticks**
  - Optional number of ticks into the past to look (defaults to 1)
- **expression2**
  - Optional gating expression to the **clocking_event**

```
sequence s_rd_3;
   ##3 (out_busData == mem[$past( busAddr,3 )]);
endsequence

property p_rd_3;
  @(posedge clk) read |-> s_rd_3;
endproperty
```

code in file: examples_sva/past_1.sv

SVA Coding Style Tips

1. **Try to use ##N to avoid possibility of referencing an undefined value ($past (x, N))**
   property p1; @ (posedge clk) (a) |-> ##**2** (b == $past (b, **2**));

2. **$past has overhead recording/logging values over time… Rule: Keep # ticks < 100**

$past() Outside of Concurrent Assertions

- **$past** can be used in any SystemVerilog expression

```
always @(posedge clk)
   reg1 <= a & $past(b);
```

  - **$past** is evaluated in the current occurrence of posedge clk
  - Returns the value of b sampled at the previous occurrence of posedge clk

```
always @(posedge clk)
   if (enable) q <= d;

always @(posedge clk)
   assert property (done |=> (out == $past(q, 2,enable)) );
```

  - The sampling of **q** for evaluating **$past** is based on the clocking expression:

```
posedge clk iff enable
```

Local data values

In this section

- Data use in a sequence or property

Data-use Within a Sequence

- Sequences are very useful for verifying correct control signal behavior
  - But data must be functionally verified too
- SV allows local variables to be defined/assigned in a sequence or property
- The variable can be assigned at the end point of any subsequence
  - Place the subsequence, comma separated from the sampling assignment, in parentheses

```
sequence aaa;
  a ##1 b[->1] ##1 c[*2];
endsequence
```

```
sequence aaa;
byte x;
 a ##1 (b[->1], x = e) ##1 c[*2];
endsequence
```

Data-use Within a Sequence - 2

- The local variable may be reassigned later in the sequence

```
sequence aaa;
byte x;
    a ##1 ( b[->1], x = e )  ##1 ( c[*2 ], x = x + 1 );
endsequence
```

- For every attempt, a new copy of the variable is created for the sequence (i.e. local variables are automatic)
- All SV types are supported
- The variable value can be tested like any other SystemVerilog variable
- Hierarchical references to a local variable are not allowed
- It also allows pure/automatic functions

Data-use Within a Sequence - 3

- An example

Sampling clause

```
property pd_rd_3;
  logic [7:0] temp;
  @(posedge clk) (read, temp = mem[busAddr])
                          |-> ##3 temp == out_busData;
endproperty
```

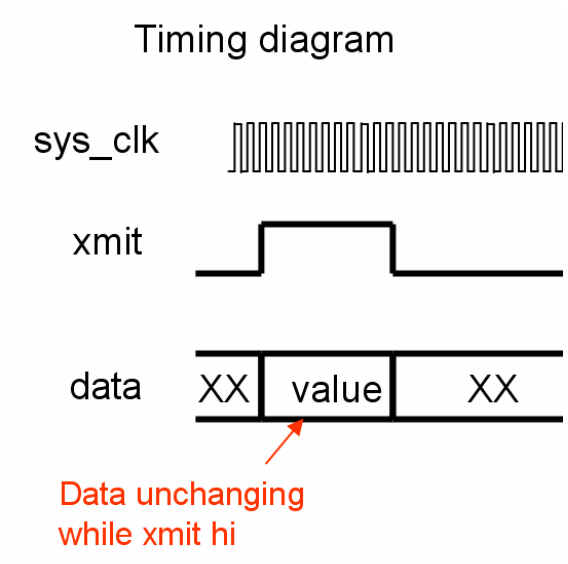
- When read is true, temp samples the memory (indexed by busAddr)
- The sequence matches 3 cycles later, if temp equals out_busData

SVA Coding Style Tip

Formal Verification tools can't handle local variables because it's impossible to produce equiv. RTL code.

Example: Automatic Data Use

- Let's consider this simple transmit protocol. While xmit is high, data must be stable, unchanging. There are at least two ways to check this:




```
sequence s_xmit_nc_data;
   $stable(data) throughout xmit[*15];
endsequence

property p_xmit_nc_data;
  @(posedge sys_clk) $rose(xmit) |=> s_xmit_nc_data;
endproperty
```

1

Next, let's try it with a local variable within the property

We need a variable update clause to sample data at the start of a write ($rose(xmit))

Lastly, check the sampled value against data repeatedly until end of the xmit pulse

```
property p_xmit_nc_data;
   bit[7:0] x_byte;
   @(posedge sys_clk) ($rose(xmit), x_byte = data) |=>
            (data == x_byte) throughout xmit[*15];
endproperty
```

2

SVA Coding Style

1 $stable, like all value change functions is useful but somewhat inefficient…

2 Uses a local variable which is more efficient, but Formal Verification tools can't handle local variables

Method Calls Within a Sequence

- At the end of a successful match of a sequence
  - Tasks, task methods, void functions, void function methods, and system tasks may be called
  - The calls appear in the comma separated list that follows the sequence inside of parenthesis (like local variable assignment syntax)

Called on every successful match

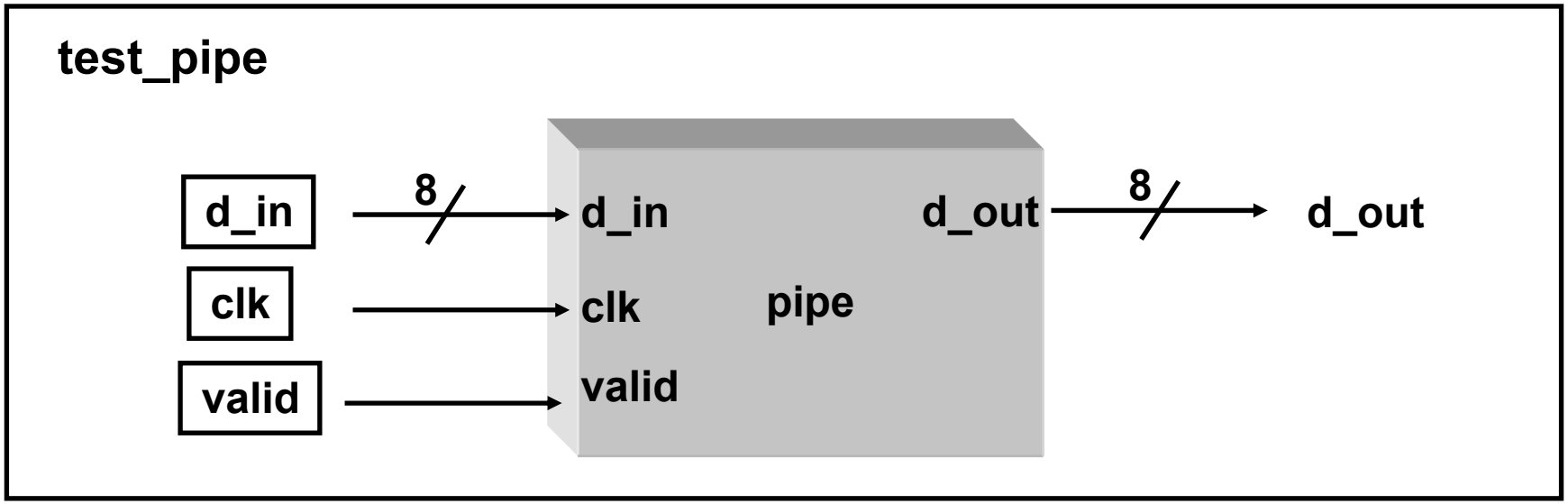
```
sequence xyz;
   @(posedge clk)   (  x ##1 y ##1 z , $display("sequence xyz matched!" )   );
endsequence
```

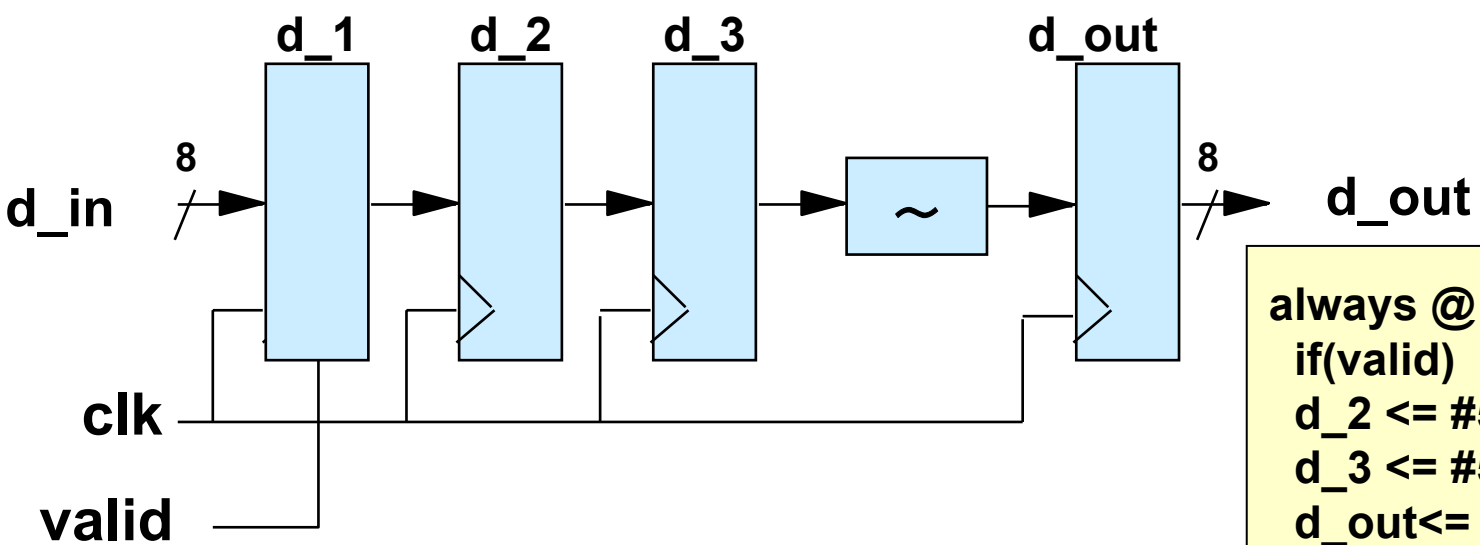
Must have enclosing parentheses

Lab – Pipe Assertions : Instructions - 1

- Working directory: **`sva_q/sva_pipe`**
- Objective: Write assertions using **`$past`** and automatic data-use
- Instructions:
  - Modify the file test_pipe.sv
    - Enter your code after the comment that reads: YOUR CODE HERE
  - Write a property **`p_pipe`** (and associated sequence(s) if desired) using **`$past`** to verify the output of the 4-stage pipeline module is correct
  - Write a property **`p_pipe_2`** (and associated sequence(s) if desired) using automatic data use to also verify the output of the 4-stage pipeline module is correct
  - Compile and run
    - Verify your assertions by running with bad_pipe.sv

Lab – Pipe Assertions : Instructions - 2






```
always @ (posedge clk) begin
  if(valid)  d_1<= #5 d_in;
  d_2 <= #5 d_1;
  d_3 <= #5 d_2;
  d_out<= #5 ~d_3;
end
```

Sol

Verification Directives

In this section

Verification Directives – more
Cover
Bind

Verification Directives - More

- A property (or sequence) by itself does nothing
  - It must appear within a verification statement to be evaluated
- Three types of verification statement
  - `[always] assert property` - *enforces a property as "checker"*
  - `[always] cover property` - *tracks metrics (# attempts, # match, # fail, etc)*
  - `[always] assume property` - *Used by FORMAL VERIF. Tools only*
- Properties can appear in modules, interfaces, programs or clocking domains

```
property p1(a,b,c);
    @ (posedge clk) a |-> ( b ##1 c );
endproperty

assert_p1:  assert property (p1(rst,in1,in2))
                begin $info("%m OK"); end
                 else begin $error("%m Failed"); end
```

ACTION BLOCK

Embedded Assertion Statements

- Assert / cover statements may be embedded in procedural code
- This aids maintenance AND captures designer intent…executable documentation

embedded assertion

```
sequence s1;
   ( req && !gnt) [*0:5] ##1 gnt && req ##1 !req;
endsequence

always @( posedge clk)
   if ( !reset ) do_reset;
   else if ( mode )
      …
          if (!arb)
            st <= REQ2;
`ifdef SVA

  PA: assert property (s1);

`endif
```

Enabling condition is always current with design changes, etc.

GOTCHA: Design flaws may be masked

Must be maintained by hand

BUT: implies double-blind checking

equivalent concurrent assertion

```
property p1;
   @( posedge clk ) ( reset && mode && !arb ) |-> s1;
endproperty

ap1: assert property(p1);
```

SVA Coding Style Tips:

- Use embedded assertions for properties that are implementation-specific, rather than ones that are based on the design spec. Specification assertions should be concurrent.
- Enclose embedded assertions in ‘ifdef statements (not all design tools may accept them)

Cover Directive

- Like **assert**, the **cover** statement is a verification directive which identifies properties or sequences of interest
  - Where they differ is that **cover** simply identifies properties to be tracked for coverage statistic purposes
- NOT enforced as behavioral checks.
- In response to a cover directive, the simulator will gather and report metrics on the specified sequences or properties, including:
  - # of times attempted
  - # of passes
  - # of failures

```
property p1(a,b,c);
    disable iff (a)  not @clk ( b ##1 c );
endproperty

cover_c1:  cover property (p1(rst,in1,in2))
                $display("Property p1 succeeded");
```

Optional statement triggered every time:
- a property succeeds
- a sequence matches

Controlling Assertions

- **$assertoff** [*levels [, list_of_modules_or_assertions]*]
  - Stop checking all specified assertions (until subsequent **$asserton**)
  - Assertion checks already in progress are unaffected
- **$asserton** [*levels [, list_of_modules_or_assertions]*]
  - Re-enable checking of specified assertions.
- **$assertkill** [*levels [, list_of_modules_or_assertions]*]
  - Same as **$assertoff** except assertions in progress are aborted too.

Optional levels argument specifies how many hierarchical levels the command should affect

SVA Coding Style Tip:

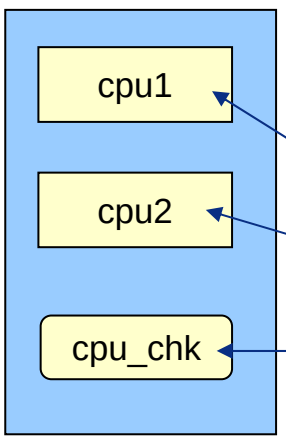
These are more efficient than adding **disable iff (**reset**)** to every property but may be more difficult to use in complex test environments e.g. with multiple concurrent test processes

Bind Directive

- As we saw, embedded assertions have advantages/disadvantages
- Another approach is to keep verification code separate from the design and use the **bind** directive to associate them.
  - **bind** can connect a module, interface or program instance (with checkers?) to a module / instance by implicitly instantiating the checker within the target.
  - **bind** may appear in a module, an interface or a compilation unit scope

target **assertions container**

bind { <module> <module_instance> } { <module> <program> <interface> } <instance_name> (signals, ..);




```
module cpu_chk (input clk, a,b,c,d);
  property p1; @ (posedge clk) a |-> ##1 (b !== (c ^ d));
  endproperty
  cpu_p1: assert property (p1);
endmodule

bind  cpu  cpu_chk  CHK1( clk, enable, d_out, d_in, val );
```

Bind Directive Coding Style

- Since the assertion container is implicitly defined within the target block, it can bind to any target variable

```
module my_design (input bit clk, b);
 bit[2:0] d;
 always @ (posedge clk)
   d <= b;
endmodule
```

```
module bind_ex;
 my_if my_if_inst();
 my_design m1(.clk(my_if_inst.clk),

 .b(my_if_inst.b) );
 my_design m2(.clk(my_if_inst.clk),

 .b(my_if_inst.b) );
 bind my_design my_svas MS (.clk, .d);
endmodule
```

```
interface my_if();
 bit clk = 0;
 logic b = 0;
 always #20 clk = !clk;
 always @(negedge clk) b++;
endinterface
```

```
module my_svas (input clk, input [2:0]  d);
 property p1;
   @ (posedge clk)
     d==2 |-> ##1 d==3 ##1 d==4;
 endproperty
 a1: assert property(p1)
                $display ("%m p1 passed");
        else $display("%m p1 failed");
endmodule
```

```
output:
# bind_ex.m2.MS.a1 p1 passed
# bind_ex.m1.MS.a1 p1 passed
```

SVA Coding Style Tips:
When verifying design blocks it is usually better to capture assertions in a separate module/interface and use a bind statement to connect to the design block. This equates to black-box testing and avoids Verif. Engineers modifying design modules.

Bind Directive Example - sm

```
import types_pkg::*;

module sva_container (
  input state_values state,
  input opcodes opcode,
  input clk
  );

property p_rd_wd;
  @(posedge clk)
  state==IDLE && opcode == RD_WD |=>
  state == RD_WD_1 ##1
  state == RD_WD_2 ##1
  state == IDLE;
endproperty

assert_p_rd_wd1: assert
property(p_rd_wd)
            else $display ("p_rd_wd
error");

// other properties/assert not shown
endmodule
```

```
module test_sm;
// test bench testing code not shown
sm_seq  sm_seq0( .into, .outof(out_wire),
                 .rst, .clk, .mem(dat),
                 .addr, .rd_, .wr_);

beh_sram sram_0(.clk, .dat, .addr,
                .rd_, .wr_);

bind sm sva_container sva_1 (.*) ;

endmodule

module sm_seq;
. . .
endmodule

module sm;
. . .
endmodule
```

Instance of ***sva_container*** module is
test_sm.sva_1

That is *not* at the same hierarchical level as the ***sm*** module which is:
test_sm.sm_seq0.sm_0

But that is OK!

Clock Specification

- The sampling clock of a property/sequence may be specified in several ways

1. Specified within the property (clk1 below)

```
property p1; @(posedge clk1) a ##2 b; endproperty
ap1: assert property (p1);
```

2. Inherited from a sub-sequence (clk2 below)

```
sequence s1;  @(posedge clk2) a ##2 b; endsequence
property p1; not s1; endproperty
ap1: assert property (p1);
```

3. Inherited from an embedding procedural block (clk3 below)

```
always @(posedge clk3) assert property ( not ( a ##2 b ));
```

Multi-clock Support

- Most systems have more than a single clock
- SV assertions allow for this by supporting the use of multiple clocks, even within a single property/assert
  - The concatenation operator (##1) is used:

```
sequence m_clk_ex;
   @(posedge clk0)  a  ##1  @(posedge clk1) b;
endsequence
```

- Here (above code), assuming **a** matches on **clk0**, **b** is checked on the next edge of **clk1**
- Implication is supported but ONLY the non-overlapping form **|=>**

```
property m_clk_ex2;
   @(posedge clk0)  a  ##1  @(posedge clk1) b
            |=> @(posedge clk2) c;
endproperty

property  m_clk_ex3;
   m_clk_ex |=> m_clk_ex2;
endproperty
```

Multi-clock Sequences - 2

- Only the ##1 operator may be used to concatenate multiply clocked sequences:

illegal!

```
@(posedge clk0)  seq1     ##0   @(posedge clk1) seq2
@(posedge clk0)  seq3     ##2   @(posedge clk1) seq4
@(posedge clk0)  seq5 intersect @(posedge clk1) seq6
```

- Multi-clock sequences/properties:
  - must have well-defined start/end times, clock transitions
  - subsequences must not admit empty matches

```
@(posedge clk0) seq1 ##1 @(posedge clk1) seq2
                                  |=> @(posedge clk2)seq3;
```

  - Here, seq1, seq2 & seq3 must not allow empty matches like: sig3[*0:1]

Multi-clock Sequences and `ended` method

- Can be applied to detect the end point of a multi-clocked sequence
- Can also be applied to detect the end point of a sequence from within a multi-clocked sequence
- The ending clock of the sequence instance to which ended is applied
  - Must be the same as the clock where the application of method ended appears

```
sequence s1;
  @(posedge clk2) a ##1 @(posedge clk) b;
endsequence

sequence s2;
  @(posedge clk)  c ##1 s1.ended ##1 d;
endsequence

property p_ended;
  @ (posedge clk) t1 |-> s2;
endproperty
```

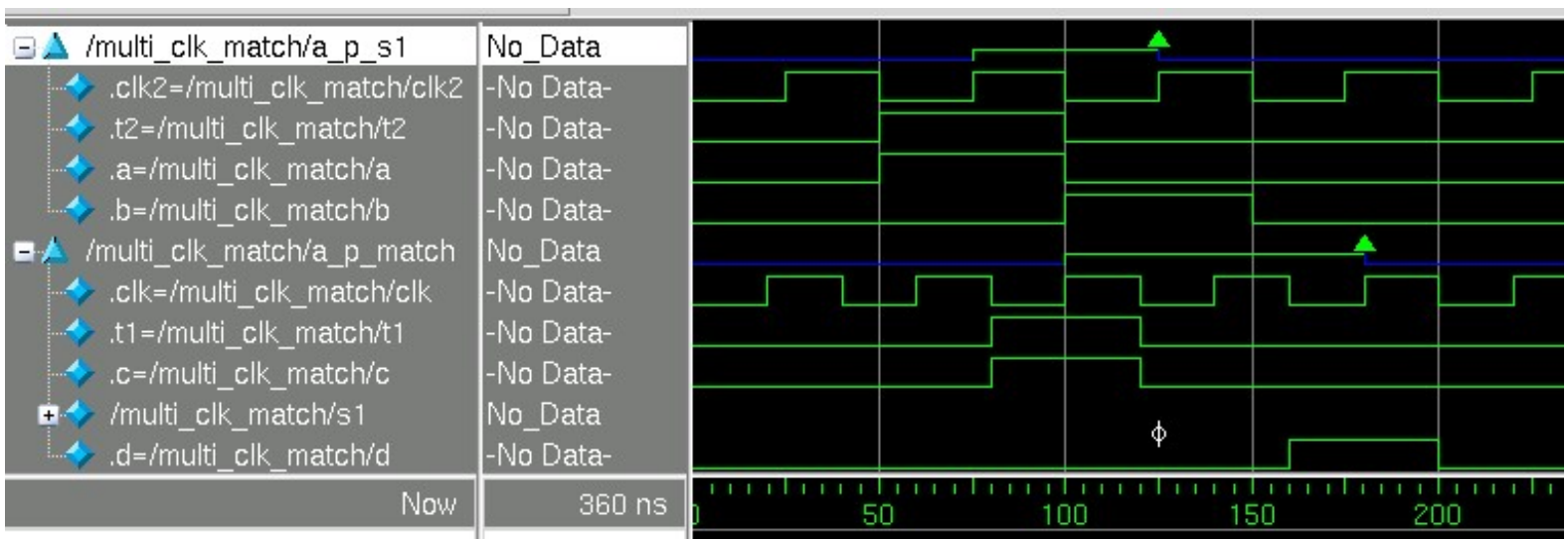
must be same clock

Multi-clock and `matched` method

- To detect the endpoint of a sequence running on a different clock
  - Use the **matched** method - like the **ended** method but for multiple clocks.

```
sequence s1;
  @(posedge clk2) a ##1 b;
endsequence
property p_match;
  @ (posedge clk) t1 |-> s2;
endproperty
```

```
sequence s2;
  @(posedge clk)  c ##1
s1.matched [->1] ##1 d;
endsequence
```



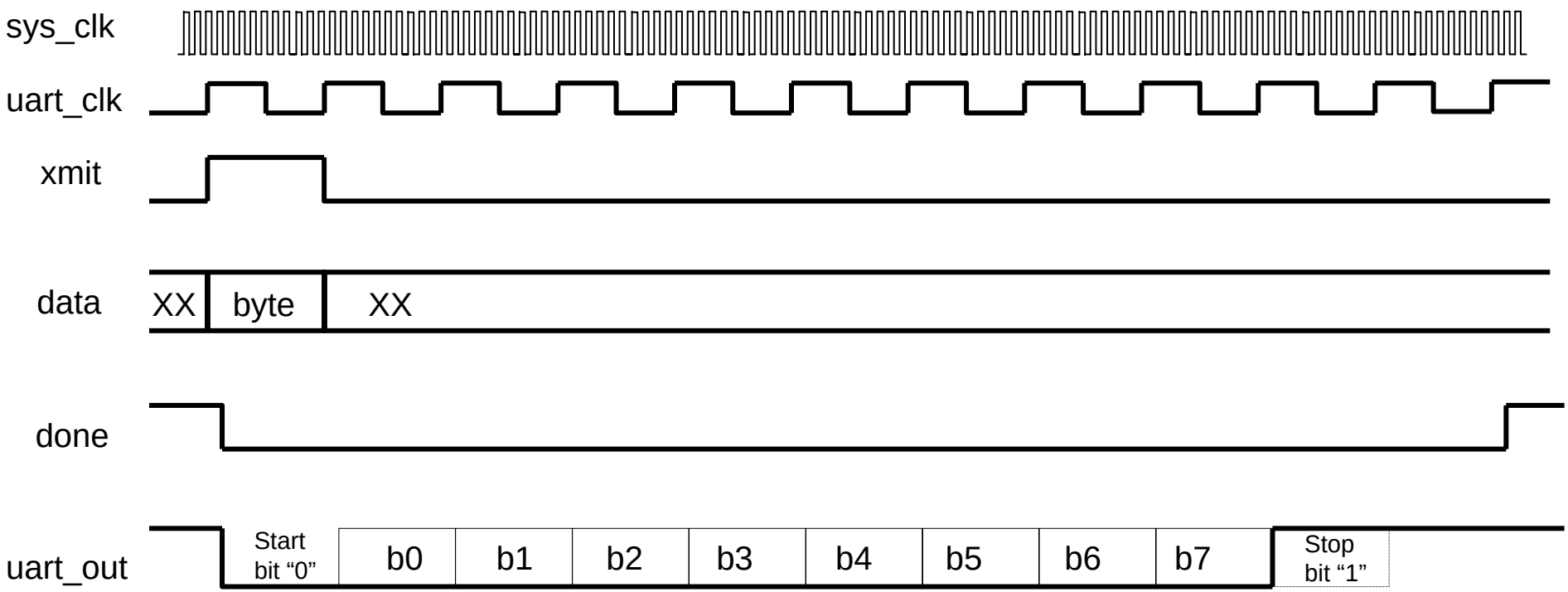
sequence s1 must match sometime after c and before d

Code in examples_sva/multi_clk_match.sv

UART Lab: Byte Transmit Waveform

Signals	
sys_clk	System clock
uart_clk	Serial clock (sys_clk ÷ 16)
sys_rst_l	Reset (active low)
xmit	Load data and start transmission
data	Byte of data to transmit
done	Done flag (low during transmission
uart_out	Serial bitstream out

Timing diagram



6 Data transmits LSB first, with 1 start-bit, 1 stop-bit

Lab – Bind: Instructions

- Lab directory: **sva_q/sva_uart_xmit**
- Instructions:
  - Edit the design (file: **test_u_xmit.sv**)
  - Code up the following assertions:
  - "**p_val_bit_stream**" – Verify the correct serial bit stream is transmitted for each byte loaded
    - Remember, you are responsible for any sequences, properties and assert statements necessary
  - Next, move **all** assertions in the testbench to a new module (**my_assertions**) and bind that module to the U1 instance of **u_xmit**
    - Consider using an implicit port connection to save typing
  - Verify your assertion in simulation by inserting errors.

Sol

Sample Solutions

SystemVerilog

Sample Solution: sparse_mem.sv

```
module sparse_mem();
  typedef enum {
    FALSE, TRUE
  } boolean;

  boolean big_mem [bit[31:0]];

  int unsigned cnt = ($random() % 25);
  bit[31:0] index;
  int unsigned total_indexes;

  initial begin
    for(int i = 0; i< cnt; i++) begin
      index = $random();  big_mem[index] = TRUE;
    end
    $display("big_mem has %0d entries", big_mem.num());
    if( big_mem.first(index) )
      $display("the smallest index is %0d", index);
    if (big_mem.last(index) )
      $display("the largest index is %0d", index);
    $display("Here are the addresses:");
    if (big_mem.first(index) )
      do
        $display(index);
      while (big_mem.next(index));
  end
endmodule
```

Back

Sample Solution mbox [Part 1]: types.sv, mbox.sv

```
package types;
  typedef struct {
  int pid;
} packet;
endpackage
```

```
module mbox;
import types::*;

mailbox #(packet) exp_mbox = new();
mailbox #(packet) act_mbox = new();

int error_cnt = 0;

task stimulus();
  packet stim_pkt;
  for (int i = 0; i < 256; i++) begin
    stim_pkt.pid = i;
    $display("Sending pkt: ",i);

    // Write stim_pkt to both mailboxes here
    act_mbox.put(stim_pkt);
    exp_mbox.put(stim_pkt);
  end
endtask
```

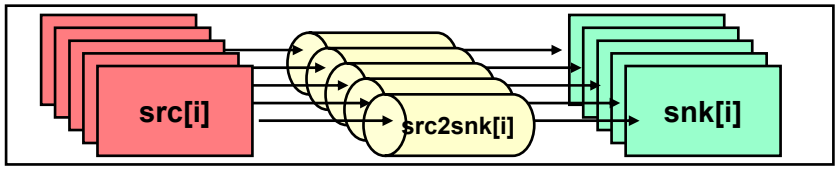
```
task checker();
  packet exp_pkt, act_pkt;
  while(1) begin
    exp_mbox.get(exp_pkt);
    act_mbox.get(act_pkt);
    if (compare(exp_pkt,act_pkt) == 0)
     error_cnt++;
    if(act_pkt.pid == 255) break;
  end
  $display("Finished: Had %0d errors",error_cnt);
  $stop;
endtask

initial begin
 fork
   stimulus();
   checker();
 join_none
 #1;
 end

endmodule
```

Sample Solution mbox (Part 2): types.sv, mbox.sv

```
function bit compare (packet exp_pkt, act_pkt);

//  if(act_pkt.pid == 22) act_pkt.pid = 44; // uncomment to inject error

  if( (exp_pkt == act_pkt) ) begin
     $display("Compared packet: ",exp_pkt.pid);
     return 1;
     end
  else begin
     $display("ERROR: pid mismatch in packet # %0d",exp_pkt.pid);
     return 0;
     end
endfunction
```

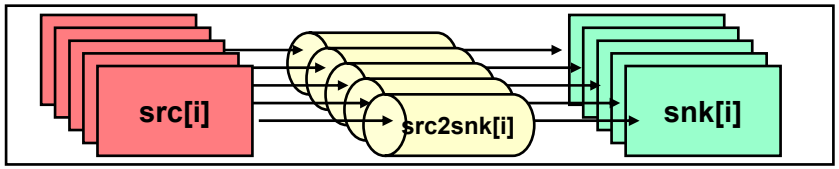
Back

Sample Solution Classes & Mboxes: array_handles.sv




```
initial
  begin
    snk     = new[5];          // create array of 5 sinks
    src     = new[5];          // create array of 5 generators
    src2snk = new[5];                    // create array of 5 mailboxes
    for (int i=0; i<5; i++) begin
       src[i]         = new(i);   // create each generator
       snk[i]         = new(i);   // create each sink
       src2snk[i] = new();        // create each mbox
       src[i].out_chan = src2snk[i];     // map snk to src via mailboxes
       snk[i].in_chan  = src2snk[i];
    end
    start_run();   // start the run tasks
  end
  task start_run();
    for (int i=0; i<5; i++) begin
      automatic int j = i; // required!
      fork
         src[j].run(); // start generators
         snk[j].run(); // start sinks
      join_none
    end
  endtask
endmodule
```

Sample Solution Classes & Mboxes: array_handles_smarter.sv




```
initial
  begin
    snk     = new[5];          // create array of 5 sinks
    src     = new[5];          // create array of 5 generators
    src2snk = new[5];          // create array of 5 mailboxes
    for (int i=0; i<5; i++) begin
       src2snk[i] = new();                 // create each mbox
       src[i]     = new(src2snk[i],i);     // create each generator and pass in its mb handle
       snk[i]     = new(src2snk[i],i);     // create each sink and pass in its mb handle
    end
    start_run();   // start the run tasks
  end

 task start_run();
    for (int i=0; i<5; i++) begin
      automatic int j = i; //required!
      fork
         src[j].run(); // start generators
         snk[j].run(); // start sinks
      join_none
    end
  endtask
endmodule
```

```
class source;
  mailbox #(Packet) out_chan; // null handle
  Packet pkt_to_send;
  int id;

  function new( mailbox #(Packet) mb, int i );
    out_chan = mb; // specify an external mailbox to drive
    id = i;
  endfunction

  task run();
   . . .
  endtask

endclass : source
```

Back

Sample Solution – V-Interface [Part 1]: router_if.sv

```
interface router_if (input bit clock);
  logic  rst ;
  logic [7:0] valid ;
  logic [7:0] stream ;
  logic [7:0] streamo ;
  logic [7:0] busy ;
  logic [7:0] valido ;

endinterface: router_if
```

```
module top;
  import defs::*;

  parameter simulation_cycle = 100 ;
  reg  SystemClock ;

  router_if top_if(SystemClock);
  test_router test(top_if);
  router dut(
    .rst ( top_if.rst ),
    .clk ( top_if.clock ),
    .valid ( top_if.valid ),
    .stream ( top_if.stream ),
    .streamo ( top_if.streamo ),
    .busy ( top_if.busy ),
    .valido ( top_if.valido )
  );

  bind dut router_assertions RA (.*);
...
```

Sample Solution – V-Interface [Part 2]: router_if.sv

```
class driver;
  virtual router_if  r_if;
  . . .
  mailbox #(Packet) mb;

  function new (int id, virtual router_if r_if,
   mailbox #(Packet) mb);
    this.r_if = r_if;
    this.id = id;
    this.mb = mb;
  endfunction
  . . .
```

```
module test_router(router_if r_if);

test_env t_env;
. . .

initial begin
  reset();
  pt_mode = 1; // set to pass thru mode to start
  t_env = new(r_if); //create test env
  t_env.run;          // start things running
end  . . .
```

```
class test_env;
 virtual router_if  r_if;
  . . .

 function new (virtual router_if routr);

  r_if = routr;
  . . .
 for ( int id = 0; id < `ROUTER_SIZE; id++)
     begin
       s2d_mb[id] = new(10);
       s[id] = new(.id(id), .mb(s2d_mb[id]),
    .log_mb(log_stim) );
       d[id] = new(.id(id), .mb(s2d_mb[id]), .r_if(r_if));
       m[id] = new(.id(id), .log_mb(log_mon), .r_if(r_if));
     end
 endfunction
```

```
class monitor;
  virtual router_if  r_if;
  . . .
  mailbox #(Packet) log_mb;

  function new (  int id,
                  virtual router_if r_if,
                  mailbox #(Packet) log_mb );
    this.id = id;
    this.r_if = r_if;
    this.log_mb = log_mb;
  endfunction
  . . .
```

Back

Sample Solution - Interface: router_if.sv

```
interface router_if (input bit clock);
  logic  rst ;
  logic [7:0] valid ;
  logic [7:0] stream ;
  logic [7:0] streamo ;
  logic [7:0] busy ;
  logic [7:0] valido ;

endinterface: router_if
```

```
module top;
  import defs::*;

  parameter simulation_cycle = 100 ;
  reg  SystemClock ;

  router_if top_if(SystemClock);
  test_router test(top_if);
  router dut(
    .rst ( top_if.rst ),
    .clk ( top_if.clock ),
    .valid ( top_if.valid ),
    .stream ( top_if.stream ),
    .streamo ( top_if.streamo ),
    .busy ( top_if.busy ),
    .valido ( top_if.valido )
  );

  bind dut router_assertions RA (.*);
...
```

Back

Sample Solution - OOP: defs.sv

```
class Packet extends BasePacket;
  function new(bit[7:0] p_id = 1);
    pkt_id = p_id;
  endfunction

  function bit compare(Packet to );
    if(to == null) begin
      $display("***No Target Compare Object!!!***");
      return(0);
    end
    if (payload  != to.payload) begin
        $display("***Mismatching Payload!!!***");
        return(0);
      end
    return(1);
  endfunction
endclass
```

Back

Sample Solution - polymorph: Derived types

```
class Pkt_type_1  extends
   Packet;

  function void gen_crc();
      crc = payload.sum();
  endfunction

  function bit check_crc();
    if (crc == payload.sum())
       return(1);
    else
      return(0);
  endfunction

endclass
```

```
class Pkt_type_2  extends Packet;

  function void gen_crc();
      crc = payload.product();
  endfunction

  virtual function bit check_crc();
    if (payload.product() == crc)
      return(1);
    else
      return(0);
  endfunction

endclass
```

Sample Solution - polymorph: source

```
class source;
  mailbox #(Packet) out_chan; // null handle
  int t1_num_packets, t2_num_packets;

  function new(int t1_n_pkts, t2_n_pkts);
    t1_num_packets = t1_n_pkts;
    t2_num_packets = t2_n_pkts;
  endfunction

 task run();
   Packet pkt_to_send; // base class handle
   Pkt_type_1 t1_pkt; //derived class handle
   Pkt_type_2 t2_pkt; //derived class handle
   fork
     for(int i = 0; i < t1_num_packets; i++) begin
       t1_pkt = new();  // create derived object
       send_pkt(t1_pkt, i);  // init & send packet
     end
     for(int i = 0; i < t2_num_packets; i++) begin
       t2_pkt = new();  // create derived object
       send_pkt(t2_pkt, i);  // init & send packet
     end
    join
 endtask

 task automatic send_pkt(input Packet pkt_to_send, int i);
     pkt_to_send.init_pkt(i);  // initialize packet
     out_chan.put(pkt_to_send); // write out packet
     $display("source: Sent packet, id = %0d", pkt_to_send.pkt_id);
 endtask
endclass  // source
```

Sample Solution - polymorph: sink

```
class sink;
  mailbox #(Packet) in_chan; // null handle
  Packet r_pkts[int];
  int num_pkts;

  function new(int n_pkts);
    num_pkts = n_pkts;
  endfunction

  task run();
   Packet rcvd_pkt;
   for(int i = 0; i<num_pkts; i++) begin
     in_chan.get(rcvd_pkt);  // get Packet object
     if (rcvd_pkt.check_crc()) begin  // check crc
       $display("sink: Received a good packet, id =
   %0d",rcvd_pkt.pkt_id);
       r_pkts[rcvd_pkt.pkt_id] = rcvd_pkt;  //put in array
     end
     else
       $display("ERROR: Received a BAD packet, id =
   %0d",rcvd_pkt.pkt_id);
   end
  endtask
endclass // sink
```

Back

Sample Solution - Random: packetClass.sv

```
class BasePacket;
  string name;
  bit[3:0] srce;
  bit[7:0] pkt_id;
  rand bit[3:0] dest;
  rand reg[7:0] payload[ ];

  constraint payload_sz {
    payload.size() inside { [1:4] };
  }

  constraint valid {
    dest inside { [0:7] };
  }  . . .
endclass
```

```
class Packet extends BasePacket;

 constraint p_thru {
      (passthru!=8'hff) -> dest == srce;
}

   . . .
endclass
```

Sample Solution - Random: stimulus

```
 class stimulus;
...
 task automatic run();
  static int pkts_generated;
  pkts_generated = 0;
  $display("Building random payload for port %0d...",id) ;
  while(1) begin
      pkt2send = new(packet_id++);
      pkt2send.srce = id;
      if (pkt2send.randomize()) begin
         mb.put(pkt2send);
         log_mb.put(pkt2send);
           pkts_generated++;
      end
      else begin
   $display("Port: %0d  Failed to Randomize Packet (pkt2snd). Aborting test.", id);
   $finish;
      end
   end
```

Sample Solution - Random: stimulus

```
class base_scoreboard;
...
 virtual task automatic run1();
    while(1) begin
      stim_mb.get(s_pkt);
      ++s_pkt_cnt;
      check[s_pkt.pkt_id] = s_pkt;
      if(passthru == 8'hff && pt_mode == 1) begin
        pt_mode = 0;
        $display("\n*********************************");
        $display("Pass Thru achieved!!");
        $display("*********************************\n");
        $stop;
      end
      report;
    end
 endtask
```

Back

Sample Solution Coverage: test_router.sv (1)

```
class scoreboard extends base_scoreboard;
shortreal current_coverage;
  covergroup cov1 @(smpl);
    option.at_least = 2;
    option.auto_bin_max = 256;
      s: coverpoint srce;
      d: coverpoint dest;
      cross s, d;
  endgroup

  function new ( mailbox #(Packet) stim_mb = null,
                 mailbox #(Packet) mon_mb = null   );
    cov1 = new();
    this.stim_mb = stim_mb;
    this.mon_mb = mon_mb;
    this.run_for_n_packets = run_for_n_packets;
 endfunction : new
```

Sample Solution Coverage: test_router.sv (2)

```
task automatic run2();
  while(1) begin
    this.mon_mb.get(m_pkt);
    ++m_pkt_cnt;
    if (check.exists(m_pkt.pkt_id))
      case( m_pkt.compare(check[m_pkt.pkt_id]) )
        0: begin
             $display("Compare error",,m_pkt.pkt_id,,
                      check[m_pkt.pkt_id].pkt_id);
             pkt_mismatch++; $stop;
             if(`TRACE_ON)
               s_pkt.display; check[s_pkt.pkt_id].display;
           end
        1: begin
             check.delete(m_pkt.pkt_id);
             srce = s_pkt.srce;
             dest = s_pkt.dest;
             -> smpl;
             current_coverage = $get_coverage();
             $display("Coverage = %f%% ",current_coverage);
           end
      endcase
```

Sample Solution Coverage: test_router.sv (3)

```
 else begin
        check[m_pkt.pkt_id] = m_pkt;
     end
     if(current_coverage == 100) begin
       $display("\n*********************************");
       $display("    Coverage goal met: 100 !!!   ");
       $display("*********************************");
       report;
       $display("*********************************");
       $display("*********************************\n");
   $stop;
     end
     report;
    end
 endtask


endclass : scoreboard
```

Back

Sample Solution Simple Assertions: Properties

```
property p_nop;
  @(posedge clk) state==IDLE && opcode == NOP |=> state == IDLE;
endproperty

property p_wt_wd;
  @(posedge clk) state==IDLE && opcode == WT_WD |=>
  state == WT_WD_1 ##1 state == WT_WD_2 ##1 state == IDLE;
endproperty

property p_wt_blk;
  @(posedge clk) state==IDLE && opcode == WT_BLK |=>
  state == WT_BLK_1 ##1 state == WT_BLK_2 ##1 state == WT_BLK_3 ##1
  state == WT_BLK_4 ##1 state == WT_BLK_5 ##1 state == IDLE;
endproperty

property p_rd_wd;
  @(posedge clk) state==IDLE && opcode == RD_WD |=>
  state == RD_WD_1 ##1 state == RD_WD_2 ##1  state == IDLE;
endproperty
```

Sample Solution Simple Assertions: assert

```
assert_p_nop1: assert property(p_nop)
                    else $display ("p_nop error");

assert_p_wt_wd1: assert property(p_wt_wd)
                    else $display ("p_wt_wd error");

assert_p_wt_blk1: assert property(p_wt_blk)
                    else $display ("p_wt_blk error");

assert_p_rd_wd1: assert property(p_rd_wd)
                    else $display ("p_rd_wd error");
```

Back

Sample Solution : UART sequences

```
sequence s_uart_sys16;
  uart_clk[*8] ##1 !uart_clk[*8] ##1 uart_clk;
endsequence

sequence s_xmit_hi16;
  @(posedge sys_clk) xmit[*16] ##1 !xmit;
endsequence

sequence s_xmit_done;
// ##1 (!done) && !uart_out;
  ##1 $fell(done) && $fell(uart_out);
endsequence

sequence s_xmit_nc_data;
   $stable(data) [*1:$] ##1 !xmit;
endsequence
```

What is the difference between these 2 alternate solutions?

Back

Sample Solution : Pipe Assertions

```
property p_pipe;
  @(posedge clk) $past(valid,4) |->
                  (~d_out === $past(d_in, 4));
endproperty


property p_pipe_2;
  logic [7:0] temp;
  @(posedge clk) (valid,temp = d_in) |->
                  ##4 (d_out === ~temp);
endproperty


a_pipe:  assert property (p_pipe);
a_pipe2: assert property (p_pipe_2);
```

Back

Sample Solution : bind

```
module my_assertions( input sys_clk, uart_clk, sys_rst_l,
                            uart_out, xmit, done,
                      input [7:0] data );

. . .
. . .

property p_val_bit_stream;
  logic [7:0] cmp;
  @(posedge uart_clk) ($rose(xmit), cmp = data) |->
    !uart_out ##1 (uart_out == cmp[0], cmp = cmp>>1)[*8]
    ##1 uart_out;
endproperty


. . .
. . .

// assertions

assert_val_bit_stream:
     assert property (p_val_bit_stream)
         else $display("%m : uart_out bitstream incorrect");

. . .
endmodule
```

```
module test_u_xmit;


. . .
. . .
. . .

u_xmit U1(  .*  );


bind U1 my_assertions A1 ( .* );


endmodule
```

Back

Appendix 1

DPI

In this section

Calling C code from SystemVerilog
Calling SystemVerilog from C code
Passing data and parameters

Introduction to DPI

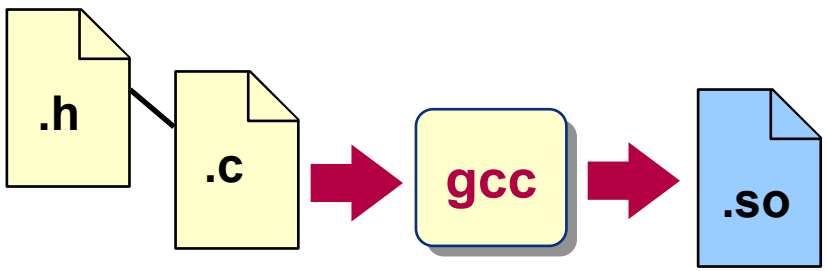
- SystemVerilog adds a new interface to foreign languages called DPI
  - DPI-C is the C and C++ language interface
- DPI-C makes it much easier to integrate C code with SystemVerilog code.
  - Can connect existing C/C++ code to SystemVerilog without the overhead of PLI or VPI
- From SystemVerilog you can:
  - Call C functions, passing arguments and returning data, acting like SystemVerilog functions.
  - Call C functions that do not return data, acting like SystemVerilog tasks.
- From C, you can:
  - Call SystemVerilog tasks and functions
- When passing data across the boundary, DPI automatically converts SystemVerilog-specific types to C types and vice-versa.

Overview: Calling C Code from SV

- To make C functions visible and callable from SystemVerilog, you must:
  - *import* their declarations into SystemVerilog

```
import "DPI-C" task c_func = a_task();
```

  - Compile the C code into a shared library




```
<unix> gcc -Ipath_to_questasim_home/include -shared -g -o my_lib.so my_c_file.c
```

  - Specify the library on the command line when starting the simulator

```
<os> vsim -sv_lib my_lib top_module
```

Import Declarations

- Import declarations are very similar to regular SystemVerilog task and function declarations.
  - They can occur anywhere that SystemVerilog task or function definitions are legal.
- Import declarations provide a "local" task or function name with a foreign implementation
  - Calling a foreign task or function is syntactically identical to calling a "native" SystemVerilog task or function

- function import syntax:

If the C code accesses any SV data that is not a parameter, or calls exported tasks/functions, then you must specify the `context` keyword

import "DPI-C" [**context**|**pure**] [c_name **=**] **function** data_type fname **(**[params]**);**

A function whose outputs depends only on it's inputs (no side effects) may be declared `pure`. This may allow optimizations and faster execution

We'll deal with data types & parameters a little later

- task import syntax:

```
import "DPI-C" [context] [c_name = ] task tname ([params]);
```

Import Declarations Example

import_ex1.sv

```
module import_ex1();
  int num;
  import "DPI-C" task init_rand();
  import "DPI-C" function int Crand();
  initial
  begin
    init_rand();
    $display( Crand() );
  end
endmodule
```

my_lib.c

```
#include <stdlib.h>
#include <dpiheader.h>

int init_rand() {
  printf("Init Rand\n");
  srand(12345);
  return(0);
}

int Crand() {
  printf("Generating random\n");
  return rand();
}
```

```
<os> vlib work
<os> vlog -dpiheader dpiheader.h import_ex1.sv
<os> gcc -Ipath_to_questasim_home/include -shared -g -o my_lib.so my_lib.c
<os> vlog import_ex1.sv
<os> vsim -c -sv_lib my_lib import_ex1
<vsim> run
Init Rand
Generating random
383100999
<vsim>
```

Overview: Calling SV Code from C

- To make SystemVerilog functions and tasks visible and callable from C, you must
  - *export* their declarations from SystemVerilog

```
export "DPI-C" task a_task();
```

  - When you compile the SV code, you can tell the compiler to generate a C header file with the declarations that you need:

```
<unix> vlog top_module.sv -dpiheader dpi_types.h
```

  - Compile the C code into a shared library and load it as before

```
<unix> gcc -I$path_to_questasim_home/include -shared -g -o my_lib.so my_c_file.c

<unix> vsim -sv_lib my_lib top_module
```

Export Declarations

- Export declarations are very similar to regular SystemVerilog task and function declarations.
  - The export declaration and the actual SystemVerilog definition can occur in any order.
- All exported functions and tasks are "context"
  - Instance-specific data is available
- Exported tasks cannot be called from C functions that are imported as functions
  - Upholds the normal SystemVerilog restriction on calling tasks from functions

Function export syntax:

```
export "DPI-C" [c_name = ] function data_type fname ([params]);
```

Task export syntax:

```
export "DPI-C" [c_name = ] task tname ([params]);
```

Data types are coming up next!

Export Declarations Example:

export_ex1.sv

```
module export_ex1();
  export "DPI-C" function SVrand();
  export "DPI-C" task init_rand();
  import "DPI-C" context task my_C_task();

  task init_rand();
  $display("Desired seed 12345");
  endtask

  function int SVrand();
     return $urandom();
  endfunction

  initial
    my_C_task();

endmodule
```

Must be `context` because it calls exported stuff

my_lib.c

```
#include "dpi_types.h"

int my_C_task() {
  int num;
  printf("Starting C task\n");
  init_rand();
  num = SVrand();
  printf ("Got %d from SV\n",num);
  return 0;
}
```

`dpi_types.h` will be created by the SystemVerilog compiler

```
<os> vlib work
<os> vlog export_ex1.sv -dpiheader dpi_types.h
<os> gcc -Ipath_to_questasim_home/include -shared -g -o my_lib.so my_lib.c
<os> vsim -c -sv_lib my_lib export_ex1
<vsim> run
Starting C task
Got 1238934 from SV
<vsim>
```

Passing Data Across the Boundary

- Standard C has very simple data types (int, char, float, etc.)
  - Cannot represent arbitrary precision integers, meta-states, etc.
- DPI defines several new C data types that correspond to SystemVerilog types
- When declaring C functions, use the appropriate data type to match what is expected in SystemVerilog
  - SystemVerilog-compatible types are the *only* data types that can be transferred in either direction.
- Function return types are restricted to these values:
  - void, byte, shortint, int, longint, real, shortreal, chandle, string
  - Packed bit arrays up to 32 bits (and equivalent types)
  - Scalar values of type bit and logic

Passing Parameters

- Parameter types can be:
  - `void, byte, shortint, int, longint, real, shortreal, chandle, string`
  - Scalar values of type bit and logic
  - Packed one-dimensional arrays of bit and logic
  - Enumerated types
  - User-defined types built from the above types with:
    - `struct`
    - `unpacked array`
    - `typedef`
- Output parameters of functions and tasks are represented are passed by reference (i.e. a C pointer to the appropriate type)

Matching Basic SV and C types

SystemVerilog Type	C Type
byte	char
shortint	short int
int	int
longint	long long
real	double
shortreal	float
chandle	void *
string	char *
bit	svBit
logic	svLogic

The values of these types use predefined constant names

SV logic values
sv_0
sv_1
sv_z
sv_x

NOTE: For more complicated data types such as vectors and arrays, see the SystemVerilog P1800 LRM, Annex P

Passing Data Example

data_ex1.sv

```
module data_ex1 ();
import "DPI-C" context function void print_logic ( input logic logic_in );

logic A = 0;

initial
begin
  print_logic(A);
  A = 1;
  print_logic(A);
  A = 1'bX;
  print_logic(A);
  A = 1'bZ;
  print_logic(A);
end
endmodule
```

```
#include "dpi_types.h"
void print_logic(svLogic logic_in)
{
  switch (logic_in)
  {
    case sv_0: printf("Just received a value of logic 0.\n");
      break;
    case sv_1: printf("Just received a value of logic 1.\n");
      break;
    case sv_z: printf("Just received a value of logic Z.\n");
      break;
    case sv_x: printf("Just received a value of logic X.\n");
      break;
  }
}
```

```
$ vlib work
$ vlog data_ex1.sv -dpiheader dpi_types.h
$ gcc -I$path_to_questasim_home/include -shared -g -o my_lib.so my_lib.c
$ vsim -c -sv_lib my_lib data_ex1
vsim> run
Just received a value of logic 0.
Just received a value of logic 1.
Just received a value of logic X.
Just received a value of logic Z.
vsim>
```

Passing Data Example

data_ex2.sv

```
module data_ex2 ();
export "DPI-C" function Double ();
import "DPI-C" context task doit(input int val);

function void Double(input int num_in, output int num_out);
begin
  num_out = 2 * num_in;
end
endfunction

initial
begin
doit(2);
doit(5);
end
endmodule
```

```
#include "dpi_types.h"

int doit( const int val)
{
  int result;
  Double(val,&result)
  printf ("Got value %d from Double\n",result);
  return 0;
}
```

2nd argument is an output, so it is passed as an int *

```
$ vlib work
$ vlog data_ex2.sv -dpiheader dpi_types.h
$ gcc -Ipath_to_questasim_home/include -shared -g -o my_lib.so my_lib.c
$ vsim -c -sv_lib my_lib data_ex2
vsim> run
Got value 4 from Double
Got value 10 from Double
vsim>
```