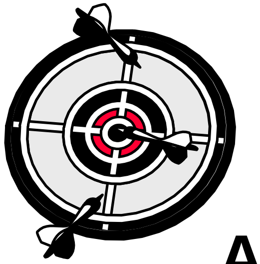
Agenda

DAY 1

1	The Device Under Test (DUT)
2	SystemVerilog Verification Environment
3	SystemVerilog Language Basics - 1
4	SystemVerilog Language Basics - 2

Unit Objectives



After completing this unit, you should be able to:

- Define the structure of a SystemVerilog(SV) program
- Declare variables and understand scope of variables in a SV program
- Define and use arrays in a SV program

SystemVerilog Testbench Code Structure

- **Test code is embedded inside `program` block**
  - `program` is instantiated in the top-level harness file

```
// root global variables
// `include files
program [automatic] name(interface);
  // `include files
  // program global variables
   initial begin
   // local variables
   // top-level test code
   end
   task task_name(…);
   // local variables
   // code
   endtask
endprogram
```

From Lab 1:

```
program automatic test ( … );
  initial begin
    $vcdpluson;
    reset();
  end
  task reset();
```

```
module router_test_top;
  router_io top_io(SystemClock);
  test t(top_io);
  router dut(.reset_n(top_io.reset_n),
             .clock (top_io.clock),
             .…     (top_io.…));
  …;
endmodule
```

SystemVerilog Lexical Convention

- **Same as Verilog**
  - Case sensitive identifiers (names)
  - White spaces are ignored except within strings
  - Comments:
    - // …
    - /* … */ (Do not nest! As in /* /* */ */)
  - Number format:

```
<size>'<base><number>
'b (binary)       :[01xXzZ]
'd (decimal)      :[0123456789]
'o (octal)        :[01234567xXzZ]
'h (hexadecimal)      :[0123456789abcdefABCDEFxXzZ]
```

    - Can be padded with '_' (underscore) for readability:

```
16'b_1100_1011_1010_0010
32'h_beef_cafe
```

2-State (0|1) Data Types (1/3)

bit [*msb*:*lsb*] *var_name* [=*initial_value*];

- **Better compiler optimizations get better performance**
- **Variable initialized to ’0 if initial_value is not specified**
  - ’0 is unsized literal (See note)
- **Assigned 0 for x or z value assignments**
  - Sized as specified
  - Defaults to unsigned

```
bit flag;
bit[15:0] sample, temp = 16’hdeed;
bit[7:0] a = 8’b1;       // 8’b0000_0001
bit[7:0] b = ’b1; // 8’b0000_0001
bit[7:0] c = ’1;  // 8’b1111_1111
bit[31:0] signed ref_data = -155;
```

2-State (0|1) Data Types (2/3)

```
2-state-type variable_name =initial_value];
```

- **Sized integral 2-state data types:**
  - `byte` - 8-bit signed data type
  - `shortint` - 16-bit signed data type
  - `int` - 32-bit signed data type
  - `longint` - 64-bit signed data type

```
shortint temp = 256;
int sample, ref_data = -9876;
longint a, b;
longint unsigned testdata;
```

2-State (0|1) Data Types (2/3)

`2-state-type` ***variable_name =initial_value*];**

- **Real 2-state data types:**
  - `real` – Equivalent to **double in C**
  - `shortreal` – Equivalent to **float in C**
  - `realtime`
    - 64-bit real variable for use with `$realtime`
    - Can be used interchangeably with real variables

```
real alpha = 100.3, cov_result;
realtime t64;
#100 t64 = $realtime;
cov_result = $get_coverage();
if (cov_result == 100.0) ...;
```

4-State (0|1|X|Z) Data Types (1/2)

```
reg | logic [msb:lsb] variable_name [=initial_value];
```

- **DUT variables need to be 4-state to emulate correct hardware behavior in simulation**
  - `reg` and `logic` are synonyms
  - Used to drive/store DUT interface signals in testbench
  - Variable initialized to `'x` if initial_value is not specified
    - `'x` is unsized literal
  - Can be used in continuous assignment (single driver only), unlike Verilog
  - Can be used as outputs of modules
  - Defaults to unsigned

```
logic[15:0] sample = '1, ref_data = 'x;
assign sample = rtr_io.cb.dout;
```

4-State Data Types (2/2)

- **Sized 4-state data types:**

```
integer variable_name [=initial_value];
```

  - 32-bit signed data type

```
time variable_name [=initial_value];
```

  - 64-bit unsigned data type

```
integer a = -100, b;
time current_time;
b = -a;
current_time = $time;
if (current_time >= 100ms) ...;
```

String Data Type

`string variable_name [=initial_value];`

- Defaults to empty string “”
- Can be created with `$sformatf()` system function
- Built-in operators and methods:
  - `==`, `!=`, `compare()` and `icompare()`
  - `itoa()`, `atoi()`, `atohex()`, `toupper()`, `tolower()`, etc.
  - `len()`, `getc()`, `putc()`, `substr()` (See LRM for more)

```
string name, s = “Now is the time”;
for (int i=0; i<4; i++) begin
  name = $sformatf(“string%0d”, i);
  $display(“%s, upper: %s”, name, name.toupper());
end
s.putc(s.len()-1, s.getc(5)); // change e to s
$display(s.substr(s.len()-4, s.len()-1));
```

Enumerated Data Types

```
typedef enum [data_type] {named constants} enumtype;
```

Type creation

- **Create enumerated data types:**
  - Data type defaults to `int`

```
enumtype var_name [=initial_value];
```

Variable creation

- **Create enum variables:**
  - Variable initialized to ’0 if initial_value is not specified
  - enum variables can be displayed as ASCII with name() function

```
typedef enum {IDLE=1, TEST, START} state_e;
state_e curr, next = IDLE;
$display("curr = %0d, next = %s", curr, next.name());
$display("next = %p", next);//or use %p for ASCII
```

What will be displayed on screen?

Data Arrays (1/4)

- **Fixed-size Arrays:**

  ***type array_name*[*size*] [=*initial_value*];**

  - Out-of-bounds write ignored
  - Out-of-bounds read returns ’0 for 2-state, ’x for 4-state arrays
  - Multiple dimensions are supported

```
integer numbers[5];          // array of 5 integers, indexed 0 – 4
int b[2] = '{3,7}; // ( b[0] = 3, b[1] = 7)
int c[2][3] = '{{3,7,1},{5,1,9}};
byte d[7][2] = '{default:-1}; // all elements set = -1
int a[2][3] = c;     // array copy – types must be same
for(int i=0; i<$dimensions(a), i++;)
  $display($size(a, i+1));          // 2  3  32
```

Returns size of particular dimension

Returns number of dimensions

Data Arrays (2/4)

- **Dynamic Arrays:**

  ***type array_name*[] [=*initial_value*];**

  - Array size allocated at runtime with constructor
  - Out-of-bounds write ignored
  - Out-of-bounds read returns ’0 for 2state, ’x for 4state arrays
  - Multiple dimensions supported

```
logic[7:0] ID[], array1[] = new[16];
logic[7:0] data_array[], mdim[][];
ID = new[100]; // allocate memory
data_array = new[ID.size()] (ID); // copy
data_array = ID;                  // copy
ID = new[ID.size() * 2] (ID);     // double the size
data_array.delete();              // de-allocate memory
```

Returns size of array

Data Arrays (3/4)

- **Queues:**

  ```
  type array_name[$[:bound]] [=initial_value];
  ```

  - Array memory allocated and de-allocated at runtime with:
    - `push_back()`, `push_front()`, `insert()`
    - `pop_back()`, `pop_front()`, or `delete()`
  - Can not be allocated with `new[]`
    - `bit[7:0] ID[$] = new[16];` // **Compilation error!**
  - Index `0` refers to lower (first) index in queue
  - Index `$` refers to upper (last) index in queue
  - Out-of-bounds write ignored
  - Out-of-bounds read returns '0 for 2state, 'x for 4state arrays (for single or rightmost dimension)
  - Can be operated on as an array, FIFO or stack
  - Multiple dimensions supported

Queue Manipulation Examples

```
int j = 2;
int q[$] = {0,1,3,6}; // note no '
int b[$] = {4,5};     // note no '
q.insert(2, j);       // {0,1,2,3,6}
q.insert(4, b);       // {0,1,2,3,4,5,6}
q.delete(1);          // {0,2,3,4,5,6}
q.push_front(7);      // {7,0,2,3,4,5,6}
j = q.pop_back();     // {7,0,2,3,4,5}   j = 6
q.push_back(8);       // {7,0,2,3,4,5,8}
$display(q.size());   // 7
$display("%p", q);    // '{7,0,2,3,4,5,8}
q.delete();           // delete all elements
$display(q.size());   // 0
```

Data Arrays (4/4)

- **Associative Arrays:**

  type array_name*[*index_type*]*; **// indexed by specified type**

  - Index type can be any numerical, string or class type
  - Dynamically allocated and de-allocated

```
integer ID_array[int];
ID_array[71] = 99;            // allocate memory
ID_array.delete(71);          // de-allocate one element
ID_array.delete();            // de-allocate all elements
```

  Array can be traversed with:

  - **first(), next(), prev(), last()**

  - Number of allocated elements can be determined with **num()**
  - Existence of a valid index can be determined with **exists()**
  - Out-of-bounds read returns ’0 for 2state, ’x for 4state arrays
  - Multiple dimensions supported

Associative Array Examples

```
byte opcode[string], t[int], a[int]; int index;

opcode["ADD"] = -8;          // create index "ADD" memory
for (int i=0; i<10; i++)
  t[1<<i] = i;               // create 10 array elements

a = t;                       // array copy

$display("num of elements in t is: %0d", t.num());

//process each element
if (t.first(index)) begin // locate first valid index
  $display("t[%0d] = %0d", index, t[index]);
  while(t.next(index))       // locate next valid index
    $display("t[%0d] = %0d", index, t[index]);
end
//better to use foreach shown on next slide
```

Array Loop Support

- **Array support**
  - Supports all array types
  - Loop: `foreach`
  - Reduction operators

```
int data[] = '{1,2,3,4,5,6,7}, qd[$][];
qd.push_back(data);
foreach(data[i]) begin
   $display("data[%0d] = %0d", i, data[i]);
end
//foreach(qd[i,j]) // to loop through 2-dimensional array
$display("sum of array content = %0d", data.sum());
$display("product value is = %0d",     data.product());
$display("and'ed value is = %0d",      data.and());
$display("or'ed value is = %0d",       data.or());
$display("xor'ed value is = %0d",      data.xor());
```

Array Methods (1/4)

```
function array_type[$] array.find()
  with (expression)
```

- Finds all the elements satisfying the **`with`** expression
- Matching elements are returned as a queue

```
function int_or_index_type[$] array.find_index()
  with (expression)
```

- Finds all the indices satisfying the **`with`** expression
- Matching indices are returned as a queue

- **`item` references the array elements during search**
- **Empty queue is returned when match fails**

Array Methods (2/4)

- **Example: `find()` and `find_index()`**

```
program automatic test;
 bit[7:0] SQ_array[$] = {2, 1, 8, 3, 5};
 bit[7:0] SQ[$];
 int idx[$];

 initial begin
   SQ  = SQ_array.find() with ( item > 3 );
   // SQ[$] contains 8, 5  - item is default iterator variable

   idx = SQ_array.find_index(addr) with ( addr > 3 );
  // idx[$] contains 2, 4 – addr: user defined iterator variable
 end
endprogram: test
```

Array Methods (3/4)

function ***array_type*****[$]** ***array*****.find_first() with ([*expr*]|1)**

- First element satisfying the `with` expression is returned
  - If `with` expression is 1, first element is returned
- First matching element is returned in `array_type[0]`

function int_or_index_type[$] *array*.find_first_index() with ([*expr*]|1)

- First index satisfying the `with` expression is returned
  - If `with` expression is 1, first index is returned
- First matching index is returned in `int_or_index_type[0]`

- **`with` is mandatory for both methods**
  - `item` in expression references array element during search
- **Empty queue is returned when match fails**

Array Methods (4/4)

- **Example:** `find_first()` **and** `find_first_index()`

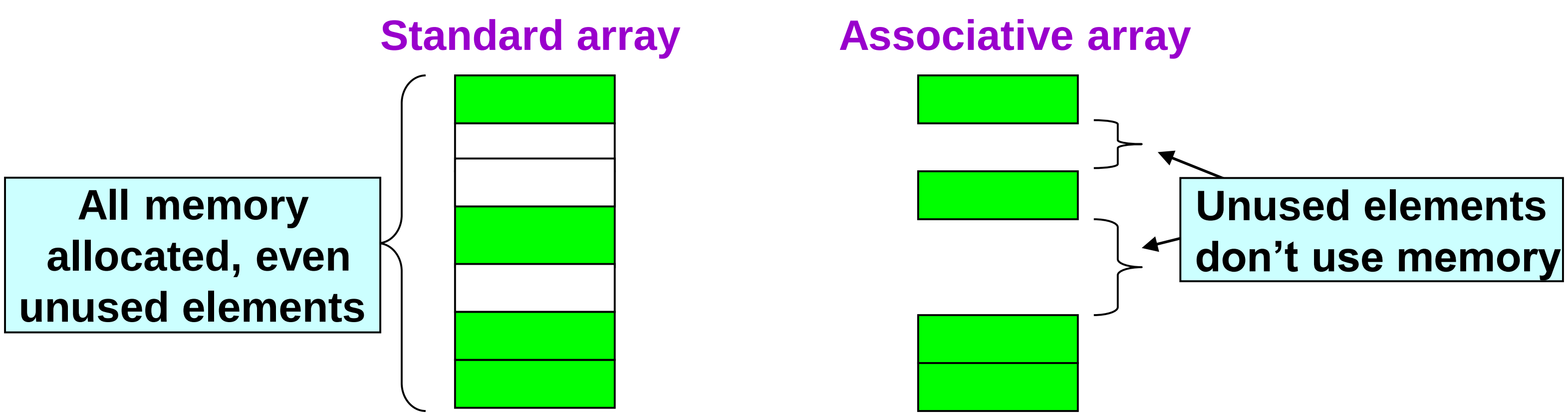
```
program automatic test;
 int array[] = new[5];
 int idx[$], val[$], dyn_2d[][], mixed_2d[$][];

 initial begin
 foreach(array[i])
   array[i] = 4 - i;
   val = array.find_first() with ( item > 3 );
              // val[0] == 4
   idx = array.find_first_index() with ( item < 0 );
              // idx == {};
 end
endprogram: test
```

More array methods available - check LRM

Array Summary

Type	Memory	Index	`Example` (speed)
Fixed Size (Multi-dimension)	Allocated at compile-time Unchangeable afterwards	Numerical	`int addr[5];` (fast)
Dynamic (Multi-dimension)	Allocated at run-time changeable at run-time	Numerical	`logic flags[];` (fast)
Queue (Multi-dimension)	Push-Pop/copy at run-time to change size	Numerical	`int in_use[$];` (fast)
Associative (Multi-dimension)	Write at run-time to allocate memory	Typed*	`state d[string];` (moderate)

* The index of associative arrays should always be typed



Quiz Time

Quiz 1

- **Is the code below legal? Will it compile?**

```
program automatic test;
bit [31:0] count;
logic [31:0] Count = `x;

 initial begin
   count = Count;
  $display("Count = %0x count = %0d", Count, count);
 end
endprogram: test
```

- **What type is type `logic` a synonym of? What does the `x initialize *`Count`* to?**
- **What will the program display? Why is value of *`count`* different from *`Count`*?**

Quiz 2

- **Define three types of arrays**
  - Fixed array of size 1024
  - Dynamic array of size 1024
  - Associative array with an **`int`** type index
- **Write to three locations in each array**
  - 0, 500, 1023
- **How many elements has each of these allocated after the write operation?**
  - Fixed array –
  - Dynamic array –
  - Associative array uses –

Unit Objectives Review

Having completed this unit, you should be able to:

- **Define the structure of a SystemVerilog(SV) program**
- **Declare variables and understand scope of variables in a SV program**
- **Define and use arrays in a SV program**

Appendix

Unpacked Array Performance

Advanced SystemVerilog Constructs

- Packed Array
- Struct
- Union

Streaming Operators: Pack/Unpack

Unpacked Array Performance

Array Performance

Array Type	Application
Fixed-Size	Use in RTL for FIFO, Memory, Buffer. Use when size of array is known and fixed for duration of simulation. Gives best performance.
Dynamic	Use this whenever you need random read/write access to any element of the variable sized array. Gives very good performance.
Queue	Use for stack, CAM applications, Scoreboard queues. Gives good performance.
Associated	Very useful for sparse memory applications. Use when creating hash tables. Moderate performance.

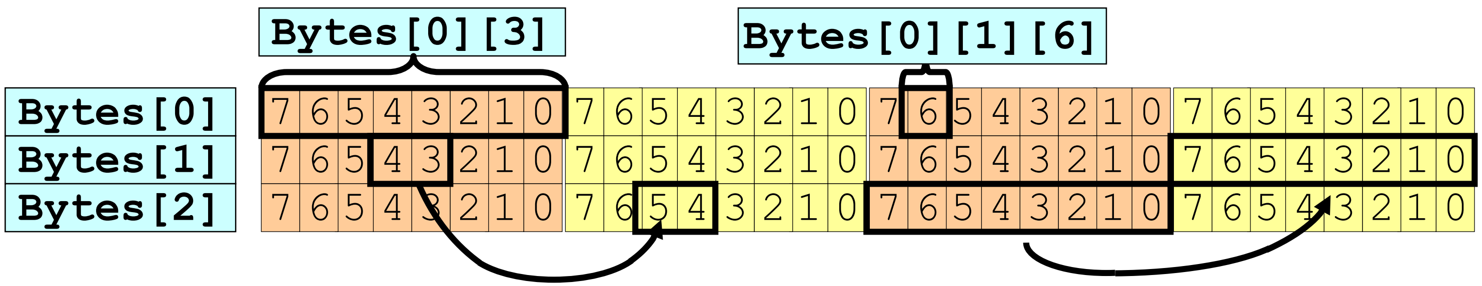
Packed Array

Packed Array

- **Defines a packed array structure**

type [msb:lsb] [msb:lsb] name [constant];

```
bit [3:0][7:0] Bytes [3];    // 3 entries of packed 4 bytes
```




```
Bytes[2][2][5:4] = Bytes[1][3][4:3];
```

```
Bytes[1][0] = Bytes[2][1];
```

```
Bytes[2] = 32'hbeef_deed;
```

Bytes[2]	1	0	1	1	1	1	1	0	1	1	1	0	1	1	1	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1	0	1

Array Querying System Functions

- **$dimensions (*array_name*)**
  - Returns the # of dimensions in the array
- **$left (*array_name*, *dimension*)**
  - Returns MSB of specified dimension
- **$right (*array_name*, *dimension*)**
  - Returns LSB of specified dimension
- **$low (*array_name*, *dimension*)**
  - Returns the min of **`$left`** and **`$right`**
- **$high (*array_name*, *dimension*)**
  - Returns the max of **`$left`** and **`$right`**
- **$increment (*array_name*, *dimension*)**
  - returns 1 if: **`$left`** is >= **`$right`**,
  - returns –1 if: **`$left`** is < **`$right`**.
- **$size (*array_name*, *dimension*)**
  - Returns the total # of elements in the specified dimension **`($high - $low`** +1)

dimension numbers

$$\texttt{bit}\ \overbrace{[3:0]}^{3}\overbrace{[7:0]}^{4}\ \texttt{Bytes}\ \overbrace{[0:2]}^{1}\overbrace{[0:5]}^{2};$$

Array Querying System Functions Examples

```
int c[2], a[2][2];
bit[31:0] b[0:2][0:1]={{3,7},{5,1},{0,4}};
$display($dimensions(a));
for (int i=1; i<=$dimensions(a); i++) begin
  $display("a dimension %0d size is %0d", i, $size(a, i));
  $display("a dimension %0d left is %0d", i, $left(a, i));
  $display("a dimension %0d right is %0d", i, $right(a, i));
  $display("a dimension %0d low is %0d", i, $low(a, i));
  $display("a dimension %0d high is %0d", i, $high(a, i));
  $display("a dimension %0d increment is %0d", i, signed'($increment(a, i)));
end
$display($dimensions(b));
for (int i=1; i<=$dimensions(b); i++) begin
  $display("b dimension %0d size is %0d", i, $size(b, i));
  $display("b dimension %0d left is %0d", i, $left(b, i));
  $display("b dimension %0d right is %0d", i, $right(b, i));
  $display("b dimension %0d low is %0d", i, $low(b, i));
  $display("b dimension %0d high is %0d", i, $high(b, i));
  $display("b dimension %0d increment is %0d", i, signed'($increment(b, i)));
end
```

See note section below for print out

struct

`struct` - Data Structure

- **Defines a wrapper for a set of variables**
  - Similar to C **`struct`** or VHDL **`record`**
  - Integral variables can be attributed for randomization using **`rand/randc`**

```
typedef struct {
  data_type variable0;
  data_type variable1;
} struct_type [, …];
```

Example:

```
typdef struct {
  rand int my_int;
  real my_real;
} my_struct;
my_struct var0, var1;
var0 = { 32, 100.2 };
var1 = { default:0 };
var1.my_int = var0.my_int;
```

Can not randomize reals

Both fields set to 0

union

`union` - Data Union

- **Overloading variable definition similar to C `union`**
  - Only **`packed`** unions supported in VCS (as of 2014.12)
    - All members must be of same size unless **`tagged`*** (See Note)

```
typedef union packed {
  data_type variable0;
  data_type variable1;
} union_type;
```

Example:

```
typdef union packed {
  int my_int;
  int my_val;
} my_union;
my_union var0, var1;
var0.my_int = 32;
var1.my_val = 100;
var1.my_int = var0.my_int;
```

All members must have same size

Can only access one field in unions

```
union packed tagged {
  data_type0 variable0;
  data_type1 variable1;
} union_variable;
```

Example:

```
union packed tagged {
  int my_i;
  bit my_r;
} my_var0, my_var1;
my_var0.my_i = 32;
my_var1.my_r = '1;
my_var1.my_i = my_var0.my_i;
```

tagged union members may have different size

VCS can not randomize unions

Streaming Operators: Pack/Unpack

Streaming Operators: Pack/Unpack(1/3)

- **The streaming operators perform packing and unpacking of data into a sequence of bits in a user-specified order.**
  - “>>“ operator streams data from left to right
  - “<<“ operator streams data from right to left
- **Data is packed when operators are used on the RHS of an assignment**

```
bit[31:0] s; bit[7:0] a,b,c;
s = {<< {a,b,c}};
```

- **Data is unpacked when operators are used on the LHS of an assignment**

```
{<< {a,b,c}} = s; // Unpack s into a,b,c
```

Streaming Operators: Pack/Unpack(2/3)

- **Stream can be sized**
  - e.g. `dbl_wrd = { << byte {w1, w2} };`
    - Stream in byte-sized slices
    - All braces are required
  - Allows any arbitrary order / organization
    - bit-reversed, little-endian, byte-swapped, nibbles, etc…

```
bit[15:0] l = 16'h_abcd, m = 16'h_cafe;
bit[31:0] lm_pack;
lm_pack = {<< byte{l,m}}; // pack byte-size chunks
$display("%h",lm_pack);   // displays "fecacdab"
{>>{l,m}} = lm_pack;      // unpack
$display("l=%h m=%h",l,m);// displays "l=feca m=cdab
```

Streaming Operators: Pack/Unpack(3/3)

- **Streaming operators can be used to pack complex data structures into arrays**
  - structs and unions
  - objects – discussed in later section

```
typedef struct {

  int hdr, len;
  byte data[];
  byte crc;

} Packet; //unpacked
```

```
Packet p;

byte q[$];   // queue of bytes
```

```
q = {<< byte {p.hdr, p.len, p.data, p.crc}}; //pack

//unpack
{<< byte {p.hdr, p.len, p.data with [0+:p.len], p.crc}} = q;
```