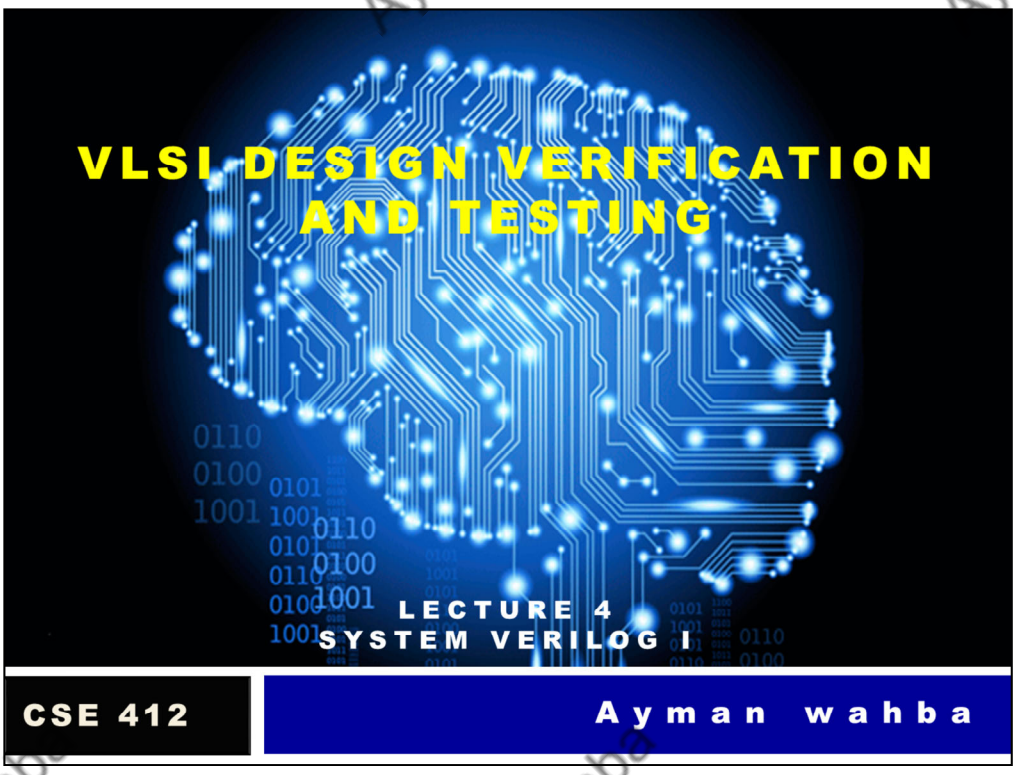

VLSI DESIGN VERIFICATION AND TESTING
LECTURE 4
SYSTEM VERILOG I
CSE 412
Ayman wahba

Why was the system Verilog Created?

2

- In the late 1990s, the Verilog HDL was most widely used for hardware modeling and simulation.
- The first two IEEE standard versions in 1995, and 2001 had only simple constructs for creating tests.
- Companies spent a lot creating their own custom verification tools.
- Accellera initiative created Verilog with verification capabilities.
- IEEE adopted this version as a standard in 2005.

accellera SYSTEMS INITIATIVE

- In the late 1990s, the Verilog Hardware Description Language (HDL) became the most widely used language for describing hardware for simulation and synthesis.
- The first two versions standardized by the IEEE (1364-1995 and 1364- 2001) had only simple constructs for creating tests.
- As design sizes outgrew a productivity crisis happened.
- Companies spent hundreds of man-years creating their own custom verification tools.
- As an initiative, a consortium of EDA companies and users created an entity called Accellera with the purpose to create the next generation of Verilog with verification capabilities.
- This effort led to the adoption of the IEEE standard P1800-2005 for SystemVerilog, IEEE (2005).

Importance of a unified language

3

- In industry engineers are divided into two groups:
  - Design activity
  - Verification activity
- There is a communication bottleneck between the two groups.
- SystemVerilog is both a design language, and a verification language. enabling both the design and verification engineers to work together to fix problems.

- Verification is generally viewed as a fundamentally different activity from design.
- This split has led to dividing engineers into two groups, one specialized in design, and the other specialized in verification, and this created bottlenecks in communication between them.
- SystemVerilog being both a design language, and a verification language at the same time improved the communication bottlenecks, enabling both the design and verification engineers to work together to identify and fix problems.
- SystemVerilog is based on the Verilog constructs that engineers have used for decades.
- SystemVerilog enables designers, who understand the inner workings of their blocks, to write assertions about it. At the same time it allows verification engineers, who may have a broader view, to create assertions between blocks.
- The value of an HVL (Hardware Verification Language) is its ability to create high level, flexible tests.

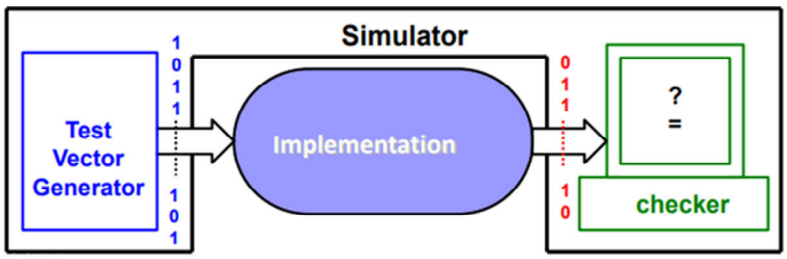
Basic Testbench Functionality

4

- Generate stimulus
- Apply stimulus to the DUT
- Capture the response
- Check for correctness
- Measure progress against the overall verification goals

- The purpose of a testbench is to determine the correctness of the design under test (DUT). This is accomplished by the following steps:
  - Generate stimulus
  - Apply stimulus to the DUT
  - Capture the response
  - Check for correctness
  - Measure progress against the overall verification goals
- Some steps are accomplished automatically by the testbench, while others are manually determined by you. The methodology you choose determines how the preceding steps are carried out.

How does a testbench look like?

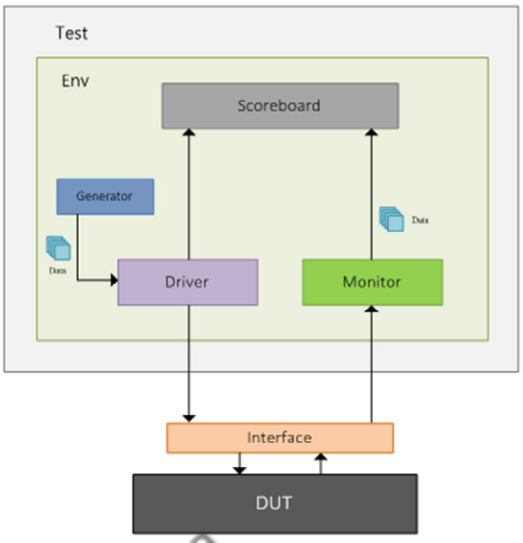


- Generates input stimulus
- Drive the design inputs with the generated stimulus
- Simulate the design to generate the outputs
- Compare the generated outputs with the expected correct outputs
- If a bug is found, modify the design to fix the bug
- Re-execute the previous steps, till the design becomes bug-free

Components of a testbench

6

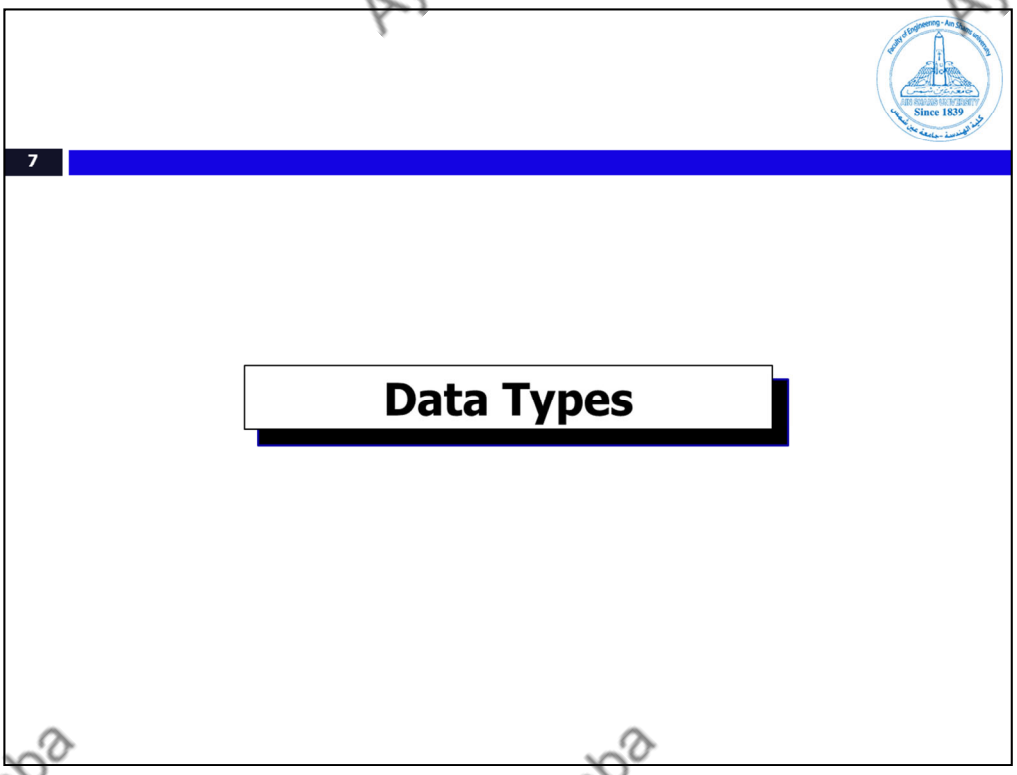
1. Generator
2. Interface
3. Driver
4. Monitor
5. Scoreboard
6. Environment
7. Test



- **Generator:** Generates different input stimulus to test the DUT. These signals can be predetermined or generated randomly to cover various design features.
- **Interface:** Contains design signals that can be driven or monitored. It allows engineers to monitor and debug the behavior of he DUT and the testbench during testing.
- **Driver:** The driver is a component responsible for applying input signals or stimuli to the device under test (DUT). The driver ensures that the DUT receives the appropriate inputs for testing its functionality.
- **Monitor:** The monitor, observes or monitors the outputs of the DUT during the test execution. It captures the output responses produced by the DUT in response to the input stimuli applied by the driver.
- **Scoreboard:** The scoreboard maintains a record of expected values for various signals or outputs based on the input stimuli provided to the device under test (DUT). As the simulation or testing progresses, the scoreboard compares the actual outputs produced by the DUT with the expected values.

  If there is a mismatch between the expected and actual results, the scoreboard flags an error, indicating a potential issue in the hardware design or test environment.

  This helps verification engineers identify and debug problems more efficiently, ensuring the correctness and reliability of the hardware system.
- **Environment:** Contains all the verification components mentioned above
- **Test:** Contains the environment that can be tweaked with different configuration settings
- All these components will be implemented using SystemVerilog.



- Datatypes in Verilog are not sufficient to develop efficient testbenches and testcases. Hence SystemVerilog has extended Verilog by adding more data-types for better encapsulation and compactness.

System Verilog Types

8

	Data type	2-state/4-state	No. of bits	Signed/unsigned	
System Verilog Types	reg	4	>=1	unsigned	Verilog Types
	wire	4	>=1	unsigned	
	integer	4	32	Signed	
	real				
	time				
	realtime				
	logic	4	>=1	unsigned	
	bit	2	>=1	unsigned	
	Byte	2	8	signed	
	Shortint	2	16	signed	
	Int	2	32	signed	
	Longint	2	64	signed	
	shortreal				

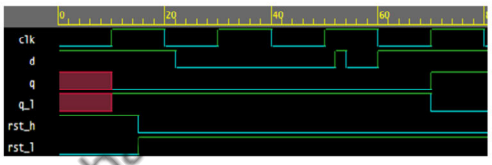
- So what are the 4 states mentioned above ?
- **0:** variable/net is at 0 volts
- **1:** variable/net is at some value > 0.7 volts
- **x: or X:** variable/net has either 0/1 - we just don't know
- **z or Z:** net has high impedance - may be the wire is not connected and is floating.

The "Logic" data type

9

```
//-----------------------------------
module logic_data_type();
 parameter CYCLE = 20;
 logic q, q_l, d, clk, rst_h, rst_l;

 initial begin
  clk = 0; // Procedural assignment
  forever #(CYCLE/2) clk = ~clk;
 end

 initial begin
  d = 1;
  rst_h = 1;
  #15 rst_h = 0;
  #7 d = 0;
  #30 d = 1;
  #2 d = 0;
  #6 d = 1;
 end

 assign rst_l = ~rst_h; // Continuous assignment
 not n1(q_l, q); // q_l is driven by gate
 my_dff d1(q, d, clk, rst_l); // q is driven by module

 initial
 begin
  $dumpfile("uvm.vcd");
  $dumpvars;
  #200 $finish;
 end

endmodule
```

```
//-----------------------------------
module my_dff(q, d, clk, rst_l);

//------------Input Ports--------------------------
input d, clk, rst_l;
//------------Output Ports-------------------------
output q ;
//------------Input ports Data Type----------------
// By rule all the input ports should be wires
wire d,clk,rst_l;
//------------Output Ports Data Type---------------
// Output port can be a storage element (reg) or a wire
reg q;

//------------Code Starts Here---------------------
// Since this f/f is a positive edge trigged one,
// We trigger the below block with respect to positive
// edge of the clock.

always @ (posedge clk)
begin
 if (rst_l == 1'b0) begin
  q <= 1'b0;
 end
 else q <= d;
end // end of Block
endmodule // end of module
```



- The one thing in verilog that always leaves new users scratching their heads is the difference between a *reg* and a *wire*.
- When driving a port, which should you use? How about when you are connecting blocks?
  - reg <= reg or wire (non-blocking assignment)
  - reg = reg or wire (blocking assignment)
  - assign wire = reg or wire (continuous)
- System Verilog improves the classic reg data type so that it can be driven by continuous assignments, gates, and modules, in addition to being a variable. It is given the synonym *logic* so that it does not look like a register declaration.
- A logic type is a 4-state type (1,0,x,z)
- A logic signal can be used anywhere a net is used,
- A logic variable *cannot be driven by multiple structural drivers*, such as when you are modeling a bidirectional bus. In this case, the variable needs to be a net-type such as wire so that System Verilog can resolve the multiple values to determine the final value.
- Names are case sensitive so CYCLE is the same is cycle
- Sometimes the waveform is not displayed due to the popup blocker. (Settings -> Privacy & Security -> Site Settings -> Pop-ups & Redirects -> add edaplayground.com to Allowed to send pop-ups

The "2-State" data type

10

```
1  //---------------------------------
2  module state_data_type();
3
4    bit b;            // 2-state, single-bit
5    bit [31:0] b32;   // 2-state, 32-bit unsigned integer
6
7    byte b8;          // 2-state, 8-bit signed integer
8    shortint s;       // 2-state, 16-bit signed integer
9    int i;            // 2-state, 32-bit signed integer
10   int unsigned ui;  // 2-state, 32-bit unsigned integer
11   longint l;        // 2-state, 64 -bit signed integer
12
13   integer i4;       // 4-state, 32-bit signed integer
14   time t;           // 4-state, 64-bit unsigned integer
15   real r;           // 2-state, double precision floating pt
```

```
16
17 initial begin
18   #20; b <= 1;    b32 <= 32'hA5BFx1z7;   b8 <= -30;
19   s <= 80000;     i <= -5;               ui <= 40;
20   l <= 655168;    i4 <= 32'bxxxx1000zzzz1010;
21   t <= 64'hABCDEF5234123897;             r <= 345.987;
22   #20
23   b <= 0;         b32 <= 32'hABF543B7;   b8 <= 40;
24   s <= 15;        i <= 24;               ui <= 0;
25   l <= 656169;    i4 <= 32'hCxAz;        t <= 500;
26   r <= 45324.5437;
27   #50; b <= 1;
28 end
29
```

	0	10	20	30	40
b					
b32[31:0]	0000_0000		a5bf_0107		abf5_43b7
b8[7:0]	00		e2		28
s[15:0]	0000		3880		000f
i[31:0]	0000_0000		ffff_fffb		0000_0018
ui[31:0]	0000_0000		0000_0028		0000_0000
l[63:0]	0000_0000_0000_0000		0000_0000_0009_ff40		0000_0000_000a_0329
i4	xxxx_xxxx		xxxx_x8za		0000_cxaz
t	xxxx_xxxx_xxxx_xxxx		abcd_ef52_3412_3897		0000_0000_0000_01f4
r	0.000000000000000		345.9870000000000		45324.54370000000

- System Verilog introduces several 2-state data types to improve simulator performance and reduce memory usage compared with variables declared as 4-state types.
- The simplest type is the bit , which is always unsigned. There are four signed 2-state types: *byte*, *shortint*, *int*, and *longint*.

- *byte* versus *logic [7: 0]*:

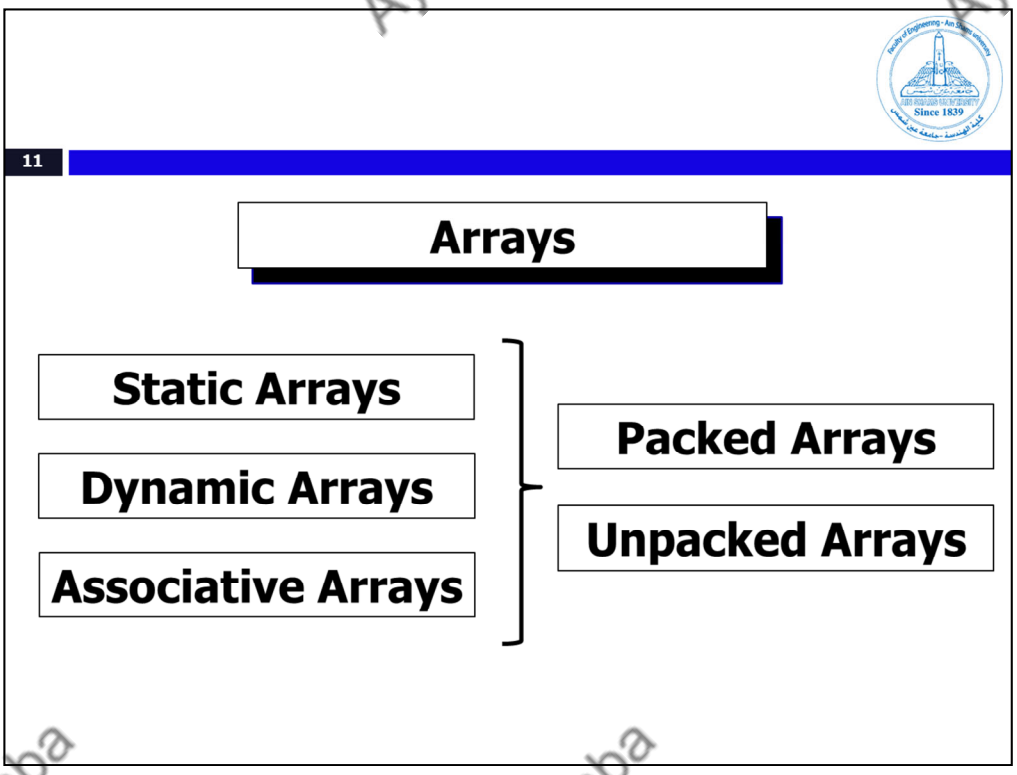
  You may think, why should I write *logic [7: 0]*, while I can simply write *byte* to declare signals?

  You should be careful as these new types *are signed variables*, and so a *byte* variable can only count up to 127, not the 255 as you may think.

  Signed variables can cause unexpected results with randomization, as will be discussed later.

- From the figure note the following:
  - Uninitialized 2-state signals are set to 0, while 4-state signals are set to x.
  - In line 18, b32 is set to $(a5bfx1z7)_{16}$ but in the waveform it is $(a5bf0107)_{16}$ that is because 2-state type can take only the 2 values; 1 or 0.
  - In line 19, **s** is set to $(80{,}000)_{10}$ but **s** is of type *shortint* which is only 16 bits long, so the range of values it can represent is $(-2^{15}$ to $2^{15}-1)$ i.e. (–32768 to 32767), so only the least 16 bits are taken.

    $(80000)_{10} = (1\ 0011\ 1000\ 1000\ 000)_2$ that is why in the waveform we see $(0011\ 1000\ 1000\ 000)_2$ which is $(3880)_{16}$.



- Datatypes in Verilog are not sufficient to develop efficient testbenches and testcases. Hence SystemVerilog has extended Verilog by adding more data-types for better encapsulation and compactness.

Fixed Size Arrays Declaration

12

```
1  module arrays();
2
3    int lo_hi[0:15]; // 16 ints [0]..[15]
4    int c_style[16]; // 16 ints [0]..[15]
5
6    int array2 [0:7][0:3]; // Verbose declaration
7    int array3 [8][4];     // Compact declaration
8
9    initial begin
10     lo_hi[3] = 234;    // Set one array element
11     array2[7][3] = 1; // Set last array element
12   end
13 endmodule
```

- SystemVerilog offers several flavors of arrays beyond the single-dimension, fixed size Verilog-1995 arrays, as many enhancements have been made to these classic arrays.
- Verilog requires that the low and high array limits must be given in the declaration. Since almost all arrays use a low index of 0, SystemVerilog lets you use the shortcut of just giving the array size, which is similar to C's style.
- You can create multidimensional fixed-size arrays by specifying the dimensions after the variable name. Multidimensional arrays were introduced in Verilog-2001, but the compact declaration style is new.
- If your code accidently tries to read from an out-of-bounds address, SystemVerilog will return the default value for the array element type:
  - That just means that an array of 4-state types, such as logic, will return X's.
  - whereas an array of 2-state types, such as int or bit, will return 0.

  This applies also if your address has an X or Z.

The Array Literal '{}

13

```
1  module arrays();
2
3    int ascend[4] = '{0,1,2,3}; // Initialize 4 elements
4    int descend[5];
5
6    initial begin
7      $display("1: ascend: ",ascend);
8      $display("2: descend: ",descend);
9
10     descend = '{4,3,2,1,0};  // Set 5 elements
11     $display("3: descend: ",descend);
12
13     descend[0:2] = '{5,6,7}; // Set first 3 elements {5,6,7,1,0}
14     $display("4: descend: ",descend);
15
16     ascend = '{4{8}};        // Four values of 8 {8,8,8,8}
17     descend = '{default:-1}; // {-1, -1, -1, -1, -1}
18     $display("5: ascend: ",ascend);
19     $display("6: descend",descend);
20   end
21 endmodule
```

```
1: ascend: '{0, 1, 2, 3}
2: descend: '{0, 0, 0, 0, 0}
3: descend: '{4, 3, 2, 1, 0}
4: descend: '{5, 6, 7, 1, 0}
5: ascend: '{8, 8, 8, 8}
6: descend'{-1, -1, -1, -1, -1}
```

- You can initialize an array using an array literal, which is an apostrophe followed by the values in curly braces. '{}
  - You can set some or all elements at once.
  - You are able to replicate values by putting a count before the curly braces.
  - Lastly, you might specify a default value for any element that does not have an explicit value.

Array Operations *for* and *foreach*

14

```
1  module arrays();
2
3    initial begin
4      bit [31:0] src[5], dst[5];
5      for (int i=0; i<$size(src); i++)
6        src[i] = i;
7      foreach (dst[j])          // j is automatically declared
8         dst[j] = src[j] * 2; // dst doubles src values
9
10     foreach (src[j])
11       $display("src[%0d]= %b = %0d   dst[%0d]= %b = %0d", j, src[j],
   src[j], j, dst[j], dst[j]);
12
13   end
14 endmodule
```

```
src[0]= 00000000000000000000000000000000 = 0   dst[0]= 00000000000000000000000000000000 = 0
src[1]= 00000000000000000000000000000001 = 1   dst[1]= 00000000000000000000000000000010 = 2
src[2]= 00000000000000000000000000000010 = 2   dst[2]= 00000000000000000000000000000100 = 4
src[3]= 00000000000000000000000000000011 = 3   dst[3]= 00000000000000000000000000000110 = 6
src[4]= 00000000000000000000000000000100 = 4   dst[4]= 00000000000000000000000000001000 = 8
        VCS Simulation Report
```

- The most common way to manipulate an array is with a *for* or *foreach* loop.
- The SystemVerilog function $size returns the size of the array.
- In the foreach-loop, you specify the array name and *an index* in square brackets, and SystemVerilog automatically steps through all the elements of the array. The index variable is automatically declared for you and is local to the loop.

Array Operations *for* and *foreach* (2-D array)

15

```
1  module arrays();
2    initial begin
3      int two_d[3][5];          // 3 rows and 5 columns
4      foreach (two_d[i,j])
5        two_d[i][j] = 10*i+j;   // initialization
6
7      foreach (two_d[i]) begin // Step through the rows
8        $write("%2d:", i);     // write in a buffer
9        foreach(two_d[,j])      // Step through the columns
10         $write("%3d", two_d[i][j]);
11       $display;              // Display what is in the buffer
12     end
13   end
14 endmodule
```

```
0:  0  1  2  3  4
1: 10 11 12 13 14
2: 20 21 22 23 24
              V C S   S i m u l a t i o n   R e p o r t
```

- Note that using *foreach* for a two-dimensional array, it must be written as follows: array_name[i,j], not array_name[i][j].
- In the normal use of the multi-dimensional array, each index must be written in a separate pair of square brackets. i.e. array_name[i][j].

Array Operations copy and compare

16

```
1  module arrays();
2    initial begin
3      bit [31:0] src[5] = '{0,1,2,3,4},
4                 dst[5] = '{5,4,3,2,1};
5   // Aggregate compare the two arrays
6      if (src==dst)
7        $display("src == dst");
8      else
9        $display("src != dst");
10  // Aggregate copy all src values to dst
11     dst = src;              // dst = src = {0,1,2,3,4}
12  // Change just one element
13     src[0] = 5;             // dst = {5,1,2,3,4}
14  // Are all values equal (no!)
15     $display("src %s dst", (src == dst) ? "==" : "!=");
16  // Use array slice to compare elements 1-4
17     $display("src[1:4] %s dst [1:4]",
18              (src[1:4] == dst[1:4]) ? "==" : "!=");
19   end
20 endmodule
```

```
src != dst
src != dst
src[1:4] == dst [1:4]
```

- You can perform aggregate *compare* and *copy* of arrays without loops. (An aggregate operation works on the entire array as opposed to working on just an individual element.)
- Comparisons are limited to just *equality* and *inequality*.
- In line 15, line 18, The "? :" conditional operator is a mini if-statement. In the above code it is used to choose between two strings.
- The final compare uses an array slice, src[1:4], which creates a temporary array with four elements.
- You cannot perform aggregate *arithmetic operations* such as addition on arrays. Instead, you can use loops. For *logical operations* such as xor, you have to either use a loop or use *packed arrays* as we will see later.

Bit and Array Subscripts together

17

```
1  module arrays();
2    initial begin
3      bit [31:0] src[5] = '{5{5}};
4
5      foreach (src[j])
6        $displayb("src[%0d]= %b",j,src[j]);
7
8      $displayb(src[0],,          // 'b101 or 'd5
9                src[0][0],,       // 'b1
10               src[0][2:1]);     // 'b10
11   end
12 endmodule
```

```
src[0]= 00000000000000000000000000000101
src[1]= 00000000000000000000000000000101
src[2]= 00000000000000000000000000000101
src[3]= 00000000000000000000000000000101
src[4]= 00000000000000000000000000000101
00000000000000000000000000000101 1 10
```

- A common annoyance in Verilog-1995 is that you cannot use array and bit subscripts together.
- Verilog-2001 removes this restriction for fixed-size arrays.
- The above code prints the first array element (binary 101), its lowest bit (1), and the next two higher bits (binary 10).
- Although this change is not new to SystemVerilog, many users may not know about this useful improvement in Verilog-2001
- Remarks:
  - Note that *$displayb* is used to display in binary format.
  - Not that you can use two comas “, ,” in the arguments of the display function to leave a blank space.

Dynamic Arrays

18

```
1  module arrays();
2    int dyn[], d2[];                  // Declare dynamic arrays
3    initial begin
4      dyn = new[5];                   // A: Allocate 5 elements
5      foreach (dyn[j]) dyn[j] = j;    // B: Initialize the elements
6      d2 = dyn;                       // C: Copy a dynamic array
7      d2[0] = 5;                      // D: Modify the copy
8      $display(dyn,d2);               // E: See both values
9      dyn = new[20](dyn);             // F: Allocate 20 ints & copy
10     $display(dyn);
11     dyn = new[40];                  // G: Allocate 40 new ints
12                                     // Old values are lost
13     $display(dyn);
14     $display("Size= %0d",$size(dyn));
15     dyn.delete();                   // H: Delete all elements
16     $display(dyn);
17   end
18 endmodule
```

```
'{0, 1, 2, 3, 4} '{5, 1, 2, 3, 4}
'{0, 1, 2, 3, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
'{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
Size= 40
'{}
        V C S   S i m u l a t i o n   R e p o r t
```

- A fixed-size array, has its size set a priori. What if the size of the array isn't known yet?
- For example, you may want to generate a random number of transactions at the start of simulation. If you stored the transactions in an fixed-size array, it would have to be large enough to hold the maximum number of transactions, but would typically hold fewer transactions, thus wasting memory.
- A dynamic array can be allocated and resized during simulation, so your simulation will consume a minimal amount of memory.
- A dynamic array is declared with empty word square brackets [ ] (see line 2). This means that you do not specify the array size in advance; instead, you give it at run-time.
- The array is initially empty, and so you must call the new[ ] constructor to allocate space, passing in the number of entries in the square brackets.
- When you copy a fixed-size array to a dynamic array, SystemVerilog calls the new[ ] constructor to allocate space, and then copies the values.
  - Line A calls new[5] to allocate 5 array elements.
  - Line B sets the value of each element of the array to its index value.
  - Line C allocates another array and copies the contents of dyn into it.
  - Lines D and E show that the arrays dyn and d2 are separate.
  - Line F allocates 20 new elements, and copies the existing 5 elements of dyn to the beginning of the array. The old 5-element array is deallocated. The result is that dyn points to a 20-element array.
  - Line G allocates 40 elements, but the existing values are not copied. The old 20-element array is deallocated.
  - Finally, line H deletes the dyn array.
  - The $size function returns the size of an array. Dynamic arrays have several built-in routines, such as delete and size.

Associative Arrays

19

```
module arrays();
  // Declaring an associative array
  int scoreboard[string];
  int expected_output;

  initial begin
    // Adding elements to the associative array
    scoreboard["alu"] = 10;
    scoreboard["ecu"] = 50;
    scoreboard["storage"] = 100;
    scoreboard["cpu"] = 150;

    // Accessing elements of the associative array
    expected_output = scoreboard["cpu"]; // expected_out = 150

    // Iterating through the associative array
    foreach(scoreboard[key]) begin
      $display("Key: %s, Value: %d", key, scoreboard[key]);
    end
  end
endmodule
```

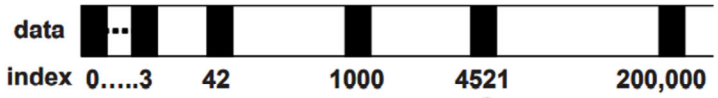
```
Key: alu, Value: 10
Key: cpu, Value: 150
Key: ecu, Value: 50
Key: storage, Value: 100
```

- Unlike a regular array, where the indices are integers, (e.g dyn[3]), associative arrays allow any data type to be used as the index.
- In SystemVerilog, an associative array is a data structure that associates a unique key with a value.
- An associative array is declared with a data type in square brackets, such as *[string]*.
- In the shown example, *scoreboard* is an associative array where strings are used as keys, and integers are used as values. The keys “alu", “ecu", “storage”, and “cpu" are associated with the values 10, 50, 100, and 150 respectively.
- The foreach loop iterates through each key-value pair in the associative array, displaying the key and its corresponding value.

Associative Arrays

20

- Imagine you are modeling a processor that has a multi-gigabyte address range.
- During a typical test, the processor may only touch a few locations containing executable code and data
- Allocating and initializing gigabytes of storage is wasteful.
- Associative arrays is a good solution in these cases.
- This means that while you can address a very large address space, SystemVerilog only allocates memory for an element when you write to it.



- Dynamic arrays are good if you want to occasionally create a big array, but what if you want something really large?
- Perhaps you are modeling a processor that has a multi-gigabyte address range.
- During a typical test, the processor may only touch a few hundred or thousand memory locations, so allocating and initializing gigabytes of storage is wasteful.
- Associative arrays that store entries in a sparse matrix. This means that while you can address a very large address space, SystemVerilog only allocates memory for an element when you write to it.
- In the shown figure, the associative array holds values in locations 0, 3, 42, 1000, 4521, and 200,000. The memory used to store these is far less than would be needed to store a fixed or dynamic array with 200,000 entries.

Associative Arrays (continued 1)

21

```
module arrays();
  initial begin
    bit[39:0] assoc[int];
    int idx = 1;

    // Initialize widely scattered values
    repeat(20) begin
      assoc[idx] = idx*3;
      idx =idx << 1;
    end

    // Step through the array elements using foreach
    foreach(assoc[i])
      $display("assoc[%0d] = %0d", i,assoc[i]);
  end
endmodule
```

```
assoc[1] = 3
assoc[2] = 6
assoc[4] = 12
assoc[8] = 24
assoc[16] = 48
assoc[32] = 96
assoc[64] = 192
assoc[128] = 384
assoc[256] = 768
assoc[512] = 1536
assoc[1024] = 3072
assoc[2048] = 6144
assoc[4096] = 12288
assoc[8192] = 24576
assoc[16384] = 49152
assoc[32768] = 98304
assoc[65536] = 196608
assoc[131072] = 393216
assoc[262144] = 786432
assoc[524288] = 1572864
```

- The shown code shows declaring, initializing, and stepping through an associative array.
- In the above code we have an associative array, “assoc”, with very scattered elements: 1, 2, 4, 8, 16, etc.
- A simple for-loop cannot step through them; you need to use a foreach loop.

Associative Arrays (continued 2)

22

```
module arrays();
  initial begin
    bit[39:0] assoc[int];
    int idx = 1;

    // Initialize widely scattered values
    repeat(20) begin
      assoc[idx] = idx*3;
      idx =idx << 1;
    end

    // Step through the array elements using functions
    if(assoc.first(idx)) begin
      do
        $display("assoc[%0d] = %0d", idx,assoc[idx]);
      while(assoc.next(idx));
    end

    // find an delete theirst element
    $display("The array before deletion has %0d elements",assoc.num);
    assoc.first(idx);
    assoc.delete(idx);
    $display("The array now has %0d elements",assoc.num);
  end
endmodule
```

SV/Verilog

```
assoc[1] = 3
assoc[2] = 6
assoc[4] = 12
assoc[8] = 24
assoc[16] = 48
assoc[32] = 96
assoc[64] = 192
assoc[128] = 384
assoc[256] = 768
assoc[512] = 1536
assoc[1024] = 3072
assoc[2048] = 6144
assoc[4096] = 12288
assoc[8192] = 24576
assoc[16384] = 49152
assoc[32768] = 98304
assoc[65536] = 196608
assoc[131072] = 393216
assoc[262144] = 786432
assoc[524288] = 1572864
The array before deletion has 20 elements
The array now has 19 elements
```

- If you want finer control, you can use the *first* and *next* functions in a *do...while loop*. These functions modify the index argument, and return 0 or 1 depending on whether any elements are left in the array.
- Used methods in the above code: *first(<index>), next(<index>), delete(<index>).*

Associative Arrays (continued 3)

23

```
module arrays();
  int switch[string]; // Asociative rray searchable by a string
  int low, high;

  initial begin
    int i, myfile;
    string s;

    myfile = $fopen("switch.txt","r"); // create a file handler

    while(!$feof(myfile)) begin
      $fscanf(myfile,"%d %s", i, s);
      switch[s] = i; // fill the associative array
    end
    $fclose(myfile);

    // Get the lowest address, default is 0
    low = switch["min_address"];
    // Get the highest address, default = 1000
    if (switch.exists("max_address"))
      high = switch["max_address"];
    else
      high = 1000;
    // Print the file contents
    foreach(switch[j])
      $display("switch[%s] = %0d",j, switch[j]);
  end
endmodule
```

testbench.sv | switch.txt

```
1 51 min_address
2 1730 max_addres
```

```
switch[max_addres] = 1730
switch[min_address] = 51
```

- Associative arrays can also be addressed with a string index, similar to Perl's hash arrays.
- The above code reads a file with strings and builds the associative array so that you can quickly map from a string value to a number.
- Strings are explained later. You can use the function *exists()* to check if an element exists.
- If you try to read an element that has not been written, SystemVerilog returns the default value for the array type, such as 0 for 2- state types, or X for 4-state types.
- Used methods in the above code: *exists(<index>).*

Queues

24

```
module arrays();
  int j = 9;
  int q1[$] = {6,7,5}; // queues don't need '

  initial begin
    $display("Initial: ",q1);
    q1.insert(1,j); //{6,9,7,5}
    $display("After inserting at index 1: ",q1);
    q1.delete(2); //{6,9,5} Delete element# 2
    $display("After deleting element of index 2: ",q1);

    // the following operations are fast
    q1.push_front(12); //{12,6,9,5} Insert at front
    $display("After pushing at front: ",q1);
    j = q1.pop_back; //{12,6,9} j = 5
    $display("After pop from the back: ",q1);
    q1.push_back(8); //{12,6,9,8} Insert at back
    $display("After pushing at the back: ",q1);
    j = q1.pop_front; //{6,9,8} j = 12
    $display("After pop from the front: ",q1);
    q1.delete(); // {} Delete the entire queue
    $display("After deleting the queue: ",q1);
  end
endmodule
```

```
Initial: '{6, 7, 5}
After inserting at index 1: '{6, 9, 7, 5}
After deleting element of index 2: '{6, 9, 5}
After pushing at front: '{12, 6, 9, 5}
After pop from the back: '{12, 6, 9}
After pushing at the back: '{12, 6, 9, 8}
After pop from the front: '{6, 9, 8}
After deleting the queue: '{}
```

- SystemVerilog introduces a new data type, the queue, which combines the best of a linked lists and arrays.
- Like a linked list, you can add or remove elements anywhere in a queue, without the performance hit of a dynamic array that has to allocate a new array and copy the entire contents.
- Like an array, you can directly access any element with an index, without linked list's overhead of stepping through the preceding elements.
- A queue is declared with square brackets containing a dollar sign: [$].
- The code above shows how you can add and remove values from a queue using *methods*.
- Note that queue literals only have curly braces, and are *missing the initial apostrophe* of array literals.
- If you add enough elements that the queue runs out of that extra space, SystemVerilog automatically allocates more. As a result, you can grow and shrink a queue without the performance penalty of a dynamic array, and SystemVerilog keeps track of the free space for you.
- Note that you never call the *new[ ]* constructor for a queue.
- Used methods in the above code: *insert(<index>,<value>), delete(<index>), delete(), push_front(<value>), push_back(<value>), pop_front(<value>), pop_back(<value>),*

Queues (continued)

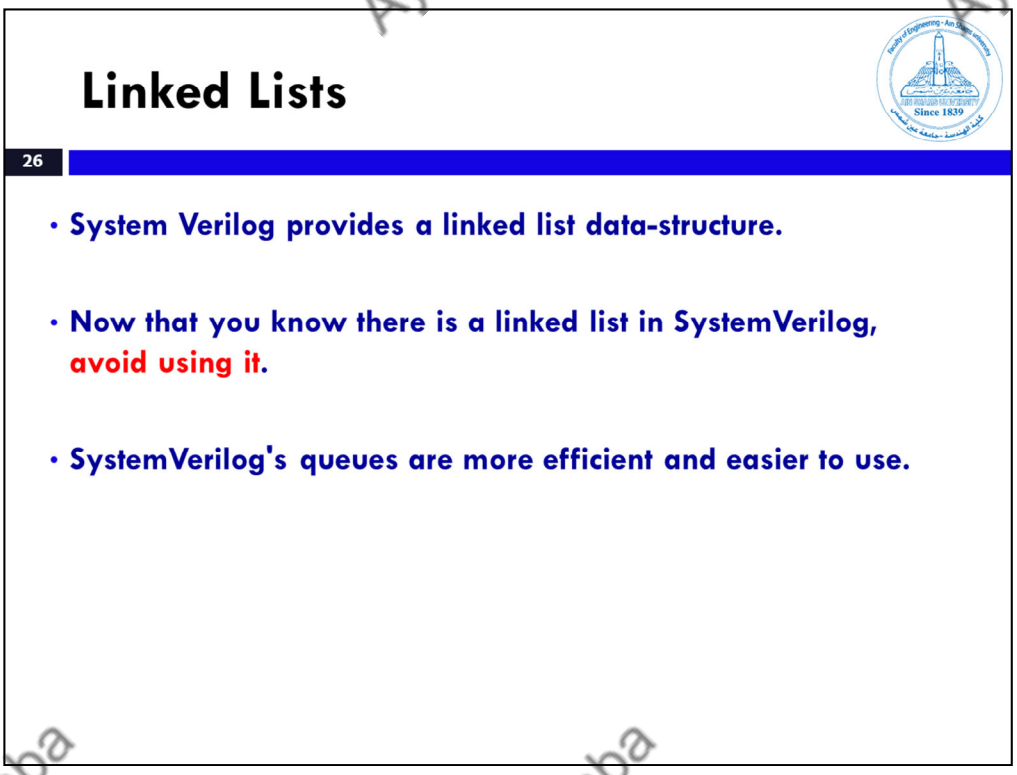
25

```
1  module arrays();
2    int j = 1,
3    q2[$] = {3,4},    // Queue literals do not use '
4    q[$] = {0,2,5};   // {0,2,5}
5    initial begin
6      q = {q[0], j, q[1:$]};     // {0,1,2,5} Insert 1 before 2
7      q = {q[0:2], q2, q[3:$]}; // {0,1,2,3,4,5} Insert queue in q
8      q = {q[0], q[2:$]};        // {0,2,3,4,5} Delete elem. #1
9      // These operations are fast
10     q = {6, q};    // {6,0,2,3,4,5} Insert at front
11     j = q[$];      // j = 5
12     q = q[0:$-1]; // {6,0,2,3,4} pop_back
13     q = {q, 8};    // {6,0,2,3,4,8} Insert at back
14     j = q[0];      // j = 6
15     q = q[1:$];    // {0,2,3,4,8} pop_front
16     q = {};        // {} Delete entire queue
17 end
18 endmodule
```

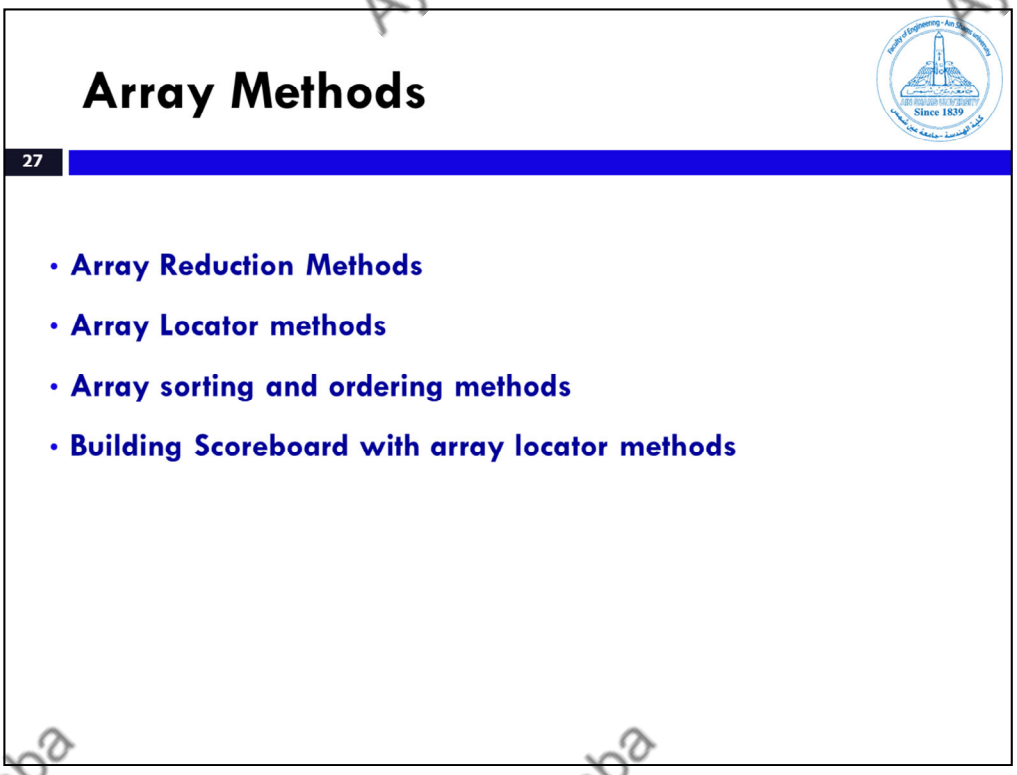
Equivalent to pop_back

Equivalent to pop_front

- You can use concatenation instead of methods.
- As a shortcut, if you put a $ on the left side of a range, such as [$:2], the $ stands for the minimum value, [0:2].
- If you put $ on the right side, as in [1:$], stands for the maximum value, [1:2], in first line of the initial block shown above.
- The queue elements are stored in contiguous locations, and so it is efficient to push and pop elements from the front and back. This takes a fixed amount of time no matter how large the queue.
- Adding and deleting elements in the middle of a queue requires shifting the existing data to make room. The time to do this grows linearly with the size of the queue.
- You can also copy the contents of a fixed or dynamic array into a queue.



- System Verilog provides a linked list data-structure.
- Now that you know there is a linked list in SystemVerilog, avoid using it.
- SystemVerilog's queues are more efficient and easier to use.



- There are many array methods that you can use on any *unpacked* array types: fixed, dynamic, queue, and associative. These routines can be as simple as giving the current array size or as complex as sorting the elements.

Array Reduction Methods

(sum, product, and, or, xor)

28

```
1  module arrays();
2    bit on[10]; // Array of single bits
3    int total;
4    initial begin
5      foreach (on[i])
6        on[i] = i; // on[i] gets 0 or 1
7  // Print the single-bit sum
8      $display("on.sum = %0d", on.sum); // on.sum 1
9  // Compute with 32-bit signed arithimetic
10     $display("int sum=%0d", on.sum with (int'(item)));
11   end
12 endmodule
```

```
on.sum = 1
int sum=5
```

- A basic array reduction method takes an array and reduces it to a single value.
- The most common reduction method is *sum*, which adds together all the values in an array.
- Be careful of SystemVerilog's rules for handling the width of operations. By default, if you add the values of a single-bit array, the result is a single bit.
- However, if you use the proper *with* expression, System Verilog uses 32-bits when adding up the values. The *with* expression is described in full details later.
- Other array reduction methods are *product*, *and*, *or*, and *xor*.

Array Reduction Methods

(sum … with (condition on item))

29

```
1  module arrays();
2    int count, total, d[] = '{9,1,8,3,4,4};
3    initial begin
4      count = d.sum with (int'(item > 7));       // 2: {9, 8}
5      $display(count);
6      total = d.sum with ((item > 7) * item);    // 17= 9+8
7      $display(total);
8      count = d.sum with (int'(item < 8));       // 4: {1, 3, 4, 4}
9      $display(count);
10     total = d.sum with (item < 8 ? item : 0); // 12=1+3+4+4
11     $display(total);
12     count = d.sum with (int'(item == 4));      // 2: {4, 4}
13     $display(count);
14   end
15 endmodule
```

```
2
17
4
12
2
```

- The code above shows various ways to total up a subset of the values in the array.
- When you combine an array reduction such as sum using the with clause, the results may surprise you.
- The sum operator totals the number of times that the expression is true. For the first statement there are two array elements that are greater than 7 (9 and 8) and so count is set to 2.
- The first total compares the item with 7. This relational returns a 1 (true) or 0 (false) and multiplies this with the array. So the sum of {9,0,8,0,0,0} is 17.
- The second total is computed using the ? : conditional operator.

Array Locator Methods

(min, max, unique)

30

```
1  module arrays();
2    int f[6] = '{1,6,2,6,8,6};     // fixed array
3    int d[] = '{2,4,6,8,10};       // dynamic array
4    int q[$] = {1,3,5,7}, tq[$]; // queue
5    initial begin
6      tq = q.min;                  // {1}
7      $display(tq);
8      tq = d.max;                  // {10}
9      $display(tq);
10     tq = f.unique;               // {1,6,2,8}
11     $display(tq);
12   end
13 endmodule
```

```
{1}
{10}
{1, 6, 2, 8}
```

- What is the largest value in an array?
- Does an array contain a certain value?
- The array *locator* methods find data in an unpacked array.
- These methods always return a queue.
- The shown code uses:
  - a fixed-size array, f [6],
  - a dynamic array, d [ ],
  - and a queue, q [$].
- The *min* and *max* functions find the smallest and largest elements in an array.
- Note that they return a queue, not a scalar as you might expect.
- These methods also work for associative arrays.
- The *unique* method returns a queue of the unique values from the array - duplicate values are not included.

Array Locator Methods

(find ... with (condition on item))

31

```
1  module arrays();
2    initial begin
3      int d[] = '{9,1,8,3,4,4}, tq[$];
4  // Find all elements greater than 3
5      tq = d.find with (item > 3); // {9,8,4,4}
6  // Equivalent code
7      tq.delete();
8      foreach (d[i])
9        if (d[i] > 3)
10         tq.push_back(d[i]);
11 // with expression
12     tq = d.find_index with (item > 3); // {0,2,4,5}
13     tq = d.find_first with (item > 99); // {} Ð none found
14     tq = d.find_first_index with (item==8); // {2} d[2]=8
15     tq = d.find_last with (item==4); // {4}
16     tq = d.find_last_index with (item==4); // {5} d[5]=4
17   end
18 endmodule
```

- You could search through an array using a foreach-loop.
- SystemVerilog can do this in one operation with a locator method.
- The *with* expression tells SystemVerilog how to perform the search, as shown above.
- In a with clause, the name *item* is called the *iterator* argument and represents a single element of the array.
- You can specify your own name by putting it in the argument list of the array method. For example:

  tq = d.find_first(x) with (x>4);

- The array locator methods that return an index, such as *find_index*, return a queue of type int, not integer. Your code may not compile if you use the wrong queue type with these statements.

Array Sorting & Ordering

(reverse, sort, rsort, shuffle)

32

```
1  module arrays();
2    initial begin
3      int d[] = '{9,1,8,3,4,4};
4      $display("Original Array: ", d);
5
6      d.reverse(); // reverses the order of the array
7      $display("Reversed Array: ", d);
8
9      d.sort ();  // sort the array
10     $display("Sorted Array: ", d);
11
12     d.rsort (); // reverse sort the array
13     $display("Reverse Ordered Array: ", d);
14
15     d.shuffle();
16     $display("Shuffled Array 1: ", d);
17
18     d.shuffle();
19     $display("Shuffled Array 2: ", d);
20
21   end
22 endmodule
```

```
Original Array: '{9, 1, 8, 3, 4, 4}
Reversed Array: '{4, 4, 3, 8, 1, 9}
Sorted Array: '{1, 3, 4, 4, 8, 9}
Reverse Ordered Array: '{9, 8, 4, 4, 3, 1}
Shuffled Array 1: '{4, 8, 4, 9, 1, 3}
Shuffled Array 2: '{9, 1, 3, 4, 4, 8}
```

- SystemVerilog has several methods for changing the order of elements in an array.
- You can *sort* the elements, *reverse* their order, or *shuffle* the order.
- Notice that these methods change the original array, unlike the array locator methods which create a queue to hold the results.

Building a Scoreboard with Array Locator Methods

33

```
module arrays();
  typedef struct
  {bit[7:0] addr;
   bit[7:0] pr;
   bit[15:0] data;} Packet; // user defined type

  Packet scb[$]; //Scoreboard is defines as a queue of packets.

  function void check_address([7:0] address);
    int intq[$]; //a queue to receive results of locator methods
    intq = scb.find_index() with (item.addr == address);
    case(intq.size())
      0: $display("Addr %0d not found in scoreboars", address);
      1: scb.delete(intq[0]); // delete the found record from the scoreboard
      default: $display("ERROR: Multiple hits for addr %0d", address);
    endcase
  endfunction

  initial begin // fill the scoreboard
    scb.push_front('{100,2,365});
    scb.push_front('{95,1,425});
    scb.push_front('{54,0,177});
    $display(scb);
    check_address(101);
    check_address(95);
    $display(scb);
  end
endmodule
```

```
'{'{addr:'h36, pr:'h0, data:'hb1}, '{addr:'h5f, pr:'h1, data:'h1a9}, '{addr:'h64, pr:'h2, data:'h16d}}
Addr 101 not found in scoreboars
'{'{addr:'h36, pr:'h0, data:'hb1}, '{addr:'h64, pr:'h2, data:'h16d}}
```

- The array locator methods can be used to build a scoreboard.
- Here we define a packet structure, then create a scoreboard made from a queue of these structures.
- Note that in defining struct, we use “;” to separate the different items. Even the last item has a “;” after it inside the curly braces “}”.
- The check_addr () function looks up an address in the scoreboard.
- The find_index () method returns an int queue.
- If the queue is empty (size==0), no match was found.
- If the queue has one member (size== 1), a single match was found, which the check_addr () function deletes.
- If the queue has multiple members (size > 1), there are multiple packets in the scoreboard whose address matches the requested one.