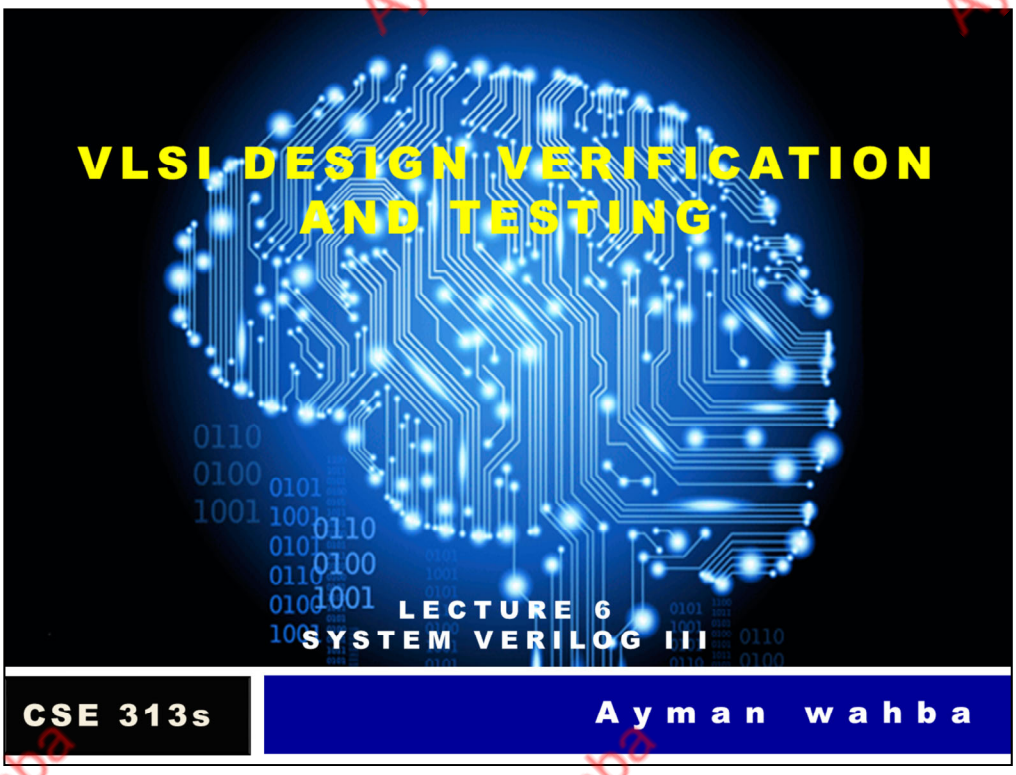

VLSI DESIGN VERIFICATION AND TESTING
LECTURE 6
SYSTEM VERILOG III
CSE 313s
Ayman wahba

Procedural Statements

2

- **SystemVerilog adopts many operators and statements from C and C++.**
  - **You can declare a loop variable inside a for-loop.**
  - **You can use ++ and - - operators in both pre- and post-forms.**
  - **If you have a label on a begin or fork statement, you can put the same label on the matching end or join statement.**

- As you verify your design. you need to write a great deal of code.
- System Verilog has a lot of constructs to make the language look more like C.
- SystemVerilog adopts many operators and statements from C and C++.
  - You can declare a loop variable inside a for-loop that then restricts the scope of the loop variable and can prevent some coding bugs.
  - The auto-increment + + and auto-decrement - - operators are available in both pre- and post-forms.
  - If you have a label on a begin or fork statement, you can put the same label on the matching end or join statement. This makes it easier to match the start and finish of a block.
  - You can also put a label on other SystemVerilog end statements such as endmodule, endtask, endfunction, and others that you will learn in this course.

Procedural Statements

3

```
1  module test();
2    initial
3    begin : example
4      integer array[10], sum, j;
5      // Declare i in for statement
6      for (int i=0; i<10; i++)      // Increment i
7        array [i] = i;
8      // Add up values in the array
9      sum = array [9] ;
10     j=8;
11     do                            // do ... while loop
12       sum += array[j];            // Accumulate
13     while (j --) ;                // Test if j=0
14     $display ("Sum =%4d", sum); // %4d - specify width
15   end : example                   // End label
16 endmodule
```

```
Sum =   45
```

Procedural Statements

(continue and break)

4

```
1  module test();
2    initial begin
3      bit [127: 0] cmd;
4      int file, c;
5      file = $fopen("command.txt", "r");
6      while (!$feof(file)) begin
7        $fscanf (file, "%s", cmd);
8        case (cmd)
9          "": continue;     //Blank line - skip to loop end
10         "done": break;    // Done - leave loop
11       endcase             // case (cmd)
12       $display("%s",cmd);
13     end
14     $fclose (file) ;
15   end
16 endmodule
```

command.txt

```
1  add
2  subtract
3
4  multiply
5  done
6  store
7  jump
```

```
add
subtract
multiply
```

- There are new statements that help with loops.
- First, if you are in a loop, but want to skip over the rest of the statements and do the next iteration, use *continue*.
- If you want to leave the loop immediately, use *break*.
- The loop in the shown code reads commands from a file using the file I/O system tasks. If the command is just a blank line, the code does a continue, skipping any further processing of the command. If the command is "done," the code does a break to terminate the loop.

Tasks versus Functions

5

Functions	Tasks
Have only input arguments	May have input and output arguments
They return a value, and the value must be used, as in an assignment statement	They don't return a value, but the can change the value of arguments.
Can't consume time. It can't have ➢ a delay, #100, ➢ nor a blocking statement such as @(posedge clock) or wait (ready) ➢ nor call a task	Can consume time

- In SystemVerilog, if you want to call a function and ignore its return value, cast the result to void, as follows:

```
void' ($fscanf (file, "%d", i));
```

- Functions have only input arguments, and the function returns a value, and the value must be used, as in an assignment statement. Sometimes the function is void, i.e. it returns no value.
- Tasks may have input and output arguments. The task itself does not return a value, but the value of its arguments may change.
- The most important difference is that a task can consume time, whereas a function cannot.
  - A function cannot have a delay, #100, a blocking statement such as @ (posedge clock) or wait (ready), or call a task.
  - SystemVerilog relaxes this rule a little in that a function can call a task, but only in a thread spawned with the *fork ... join_none* statement, which is described later.
  - Hint: If you have a SystemVerilog task that does not consume time, you should make it a void function, which is a function that does not return a value. Now it can be called from any task or function.
- In SystemVerilog, if you want to call a function and ignore its return value, cast the result to void, as follows:

```
void' ($fscanf (file, "%d", i));
```

Tasks versus Functions

(continued)

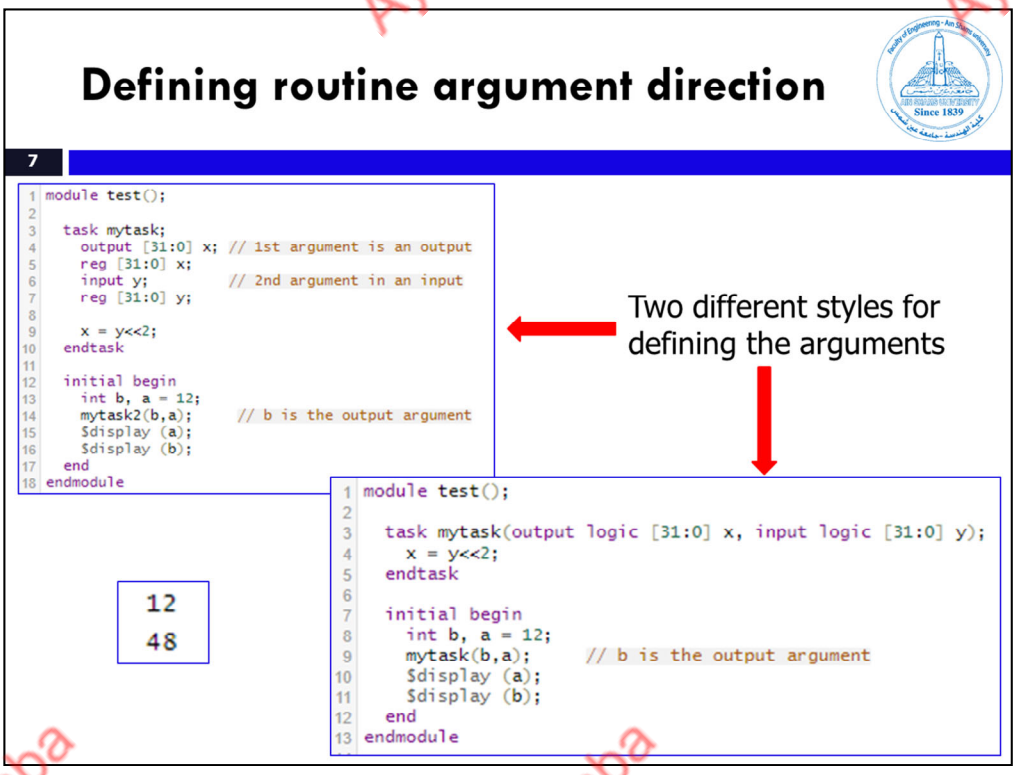
6

```
1  module arrays();
2
3    task multiple_lines;
4      $display("First line");
5      $display("Second line");
6    endtask : multiple_lines
7
8    initial begin
9      multiple_lines;
10   end
11
12 endmodule
```

```
First line
Second line
```

- In Systemverilog, using *begin* and *end* for tasks and functions is optional. Tasks have *endtask*, and functions have *endfunction*.
- The *task* / *endtask* and *function* / *endfunction* keywords are enough to define the routine boundaries.

Defining routine argument direction



- SystemVerilog allows you to declare task and function arguments more cleanly and with less repetition.
- The first task requires you to declare some arguments twice: once for the direction. and once for the type.
- In the second task, we use the less verbose C-style.
- You can take even more shortcuts with declaring routine arguments. The direction and type default to "input logic".

  task T3(a, b, output bit [15:0] u, v);

  - The arguments *a* and *b* are input logic, 1-bit wide.
  - The arguments u and v are 16-bit output bit types.
  - Now that you know this, don't depend on the defaults, as your code will be infested with subtle and hard to find bugs. Always declare the type and direction for every routine argument.

Advanced Argument Types

8

```
1  module test();
2    function void print_checksum (const ref bit [31:0] a[]);
3      // a is a dynamic array of bit [31:0]
4      bit [31:0] checksum = 0;
5      for (int i=0; i<a.size(); i++)
6        checksum ^= a[i];      // XOR of the array elements
7      $display("The array checksum is %0d", checksum);
8    endfunction
9
10   initial begin
11     bit [31:0] w[]={21,17,7};
12     print_checksum(w);
13   end
14 endmodule
```

```
  10101
⊕ 10001
⊕ 00111
--------
  00011
```

```
The array checksum is 3
          V C S   S i m u l a t i o n   R e p o r t
```

- In System Verilog, you can specify that an argument is passed by reference, rather than copying its value.
- This argument type, *ref*, has several benefits over input, output, and inout.
  - You can pass an array into a routine. System Verilog allows you to pass array arguments without the ref direction, but the array is copied onto the stack, which is an expensive operation.
  - The second benefit of ref arguments is that a task can modify a variable and is instantly seen by the calling function.
- The above code shows the *const* modifier. As a result, the contents of the array cannot be modified. If you try to change the contents. the compiler prints an error.

Default Value of an Argument

9

```
1  module test();
2    function void print_checksum(ref bit [31:0] a[],
3                                 input bit [31:0] low = 0,
4                                 input int high = -1);
5      bit [31:0] checksum;
6      checksum = 0;
7      if (high == -1 || high >= a.size())
8        high = a.size()-1;
9      for (int i=low; i<=high; i++)
10       checksum += a[i];
11     $display ("The array checksum is %0d", checksum);
12   endfunction
13
14   initial begin
15     bit [31:0] w[] = {21,17,7,50,10};
16     print_checksum (w);      // Checksum a[0:size()-1]
17     print_checksum (w,2,4); // Checksum a[2:4]
18     print_checksum (w,1);    // Start at 1
19     print_checksum(w,,2);    // Checksum a [0:2]
20     //print_checksum();      // Compile error: a has no default
21   end
22 endmodule
```

```
The array checksum is 105
The array checksum is 67
The array checksum is 84
The array checksum is 45
```

- As your testbench grows in sophistication, you may want to add additional controls to your code but not break existing code.
- For the function in the previous slide, you might want to print a checksum of just the middle values of the array. However, you don't want to go back and rewrite every call to add extra arguments.
- In SystemVerilog you can specify a default value that is used if you leave out an argument in the call.
- The code shown here, adds *low* and *high* arguments to the print_checksum function so that you can print a checksum of a range of values.

Passing Arguments by Name

10

```
1  module test();
2    task many(input int a=1, b=2, c=3, d=4);
3      $display("%0d %0d %0d %0d", a, b, c, d);
4    endtask
5    initial begin          // a b c d
6      many(6, 7, 8, 9); // 6 7 8 9 Specify all values
7      many();           // 1 2 3 4 Use defaults
8      many (.c(5));     // 1 2 5 4 Only specify c
9      many (,6,.d(8));  // 1 6 3 8 Mix styles
10   end
11 endmodule
```

```
6 7 8 9
1 2 3 4
1 2 5 4
1 6 3 8
          V C S   S i m u l a t i o n   R e p o r t
```

- If you have a task or function with many arguments, some with default values, and you only want to set a few of those arguments, you can specify a subset by specifying the name of the argument, as in the shown code.

Common Coding Errors

11

- **Argument type is *sticky* with respect to the previous argument.**
- **The default type for the *first argument* is a single-bit input**

```
task sticky(int a, b);
```

```
task sticky(ref int array [50] , int a, b);
```

```
task sticky(ref int array [50] , input int a, b);
```

- The most common coding mistake that you are likely to make with a routine is forgetting that the argument type is *sticky* with respect to the previous argument.
- The default type for the first argument is a single-bit input.
- `task sticky(int a, b);` The two arguments are input integers.
- As you are writing the task, you may realize that you need access to an array, and so you add a new array argument, and use the ref type so that it does not have to be copied.
- Your routine header now looks like this:

  `task sticky(ref int array [50] , int a, b); // What direction are these?`

  What argument types are a and b? They take the direction of the previous argument: ref. Using ref for a simple variable such as an int is not usually needed, but you would not get even a warning from the compiler, and thus would not realize that you were using the wrong direction.
- If any argument to your routine is something other than the default input type, specify the direction for all arguments as follows.

  `task sticky(ref int array [50] , input int a, b); // Be explicit`

Returning from a Routine

12

```
1  module test();
2    task load_array(int len, ref int array[]);
3      if (len <= 0) begin
4        $display ("Bad len");
5        return;
6      end
7  // Code for the rest of the task
8      for(int i=0; i<len; i++) begin
9        array[i] = i*i;
10       $display("array[%0d] = %0d", i,array[i]);
11     end
12   endtask
13
14   initial begin
15     int a[];
16     a = new[5];
17     load_array(0,a);
18     $display("------------");
19     load_array(5,a);
20   end
21 endmodule
```

```
Bad len
------------
array[0] = 0
array[1] = 1
array[2] = 4
array[3] = 9
array[4] = 16
```

- Verilog had a primitive way to end a routine; after you executed the last statement in a routine, it returned to the calling code.
- System Verilog adds the *return statement* to make it easier for you to control the flow in your routines.
- The task in the shown code needs to return early because of error checking. Otherwise, it would have to put the rest of the task in an else clause. which would cause more indentation and be more difficult to read.

Returning an array from a function

13

```
1  module test();
2    typedef int fixed_array5 [5]; // define a type
3    fixed_array5 f5;              // declare a variable
4
5    function fixed_array5 init(int start); // consumes memory
6    foreach(init[i])
7       init[i] = i + start;
8    endfunction
9
10   initial begin
11     f5 = init(5) ; // two memory spaces are consumed
12     foreach(f5[i])
13       $display("f5[%0d] = %0d", i, f5[i]);
14   end
15 endmodule
```

```
1  module test();
2    function void init(ref int f[5], input int start);
3      foreach (f[i])
4        f[i] = i + start;
5    endfunction
6
7    int fa[5];     // only one memory space is consumed
8    initial begin
9      init (fa,5) ;
10     foreach (fa[i])
11       $display (" fa [%0d] = %0d", i, fa[i]);
12   end
13 endmodule
```

```
f5[0] = 5
f5[1] = 6
f5[2] = 7
f5[3] = 8
f5[4] = 9
```

- In SystemVerilog, a function can return an array, using several techniques.
- *The first way* is to define a type for the array. and then use that in the function declaration. (first code)

  One problem with the preceding code is that the function *init* creates an array, which is copied into the array f5. If the array was large, this could be a large performance problem.
- *An alternative way* is to pass the array by reference. The easiest way is to pass the array into the function as a *ref* argument, as shown in the second code.
- *The last way* for a function to return an array is to wrap the array inside a class, and return a handle to an object (will be explained later).

Time Values

14

```
1  module timing;
2    initial begin
3      #1 $display("%0t", $realtime);
4      #2 $display("%0t", $realtime);
5      #0.2 $display("%0t", $realtime);
6      #0.4 $display("%0t", $realtime);
7      #0.5 $display("%0t", $realtime);
8    end
9  endmodule
```

```
1
3
3
3
4
```

Rounded to 1 delay

Rounded to 0 delay

- **When not otherwise specified, the default time unit is one second.**

- SystemVerilog has several constructs to allow you to unambiguously specify time values in your system.
- When you want to specify a delay in your code you write #1, for example. It means a delay of one time unit. But what is the time unit ? If not otherwise specified, the time unit is one second. So #1 means delay for one second. Any fractions will be rounded, so #0.4 will be considered as 0 delay, but #0.6 will be considered as 1 second delay.
- But what if you want to specify the timing more precisely ?

Time Values

(timeunit & timeprecision)

15

```
1  module timing;
2
3    timeunit 1 ns;          // default time unit
4    timeprecision 1 ps;     // minimum delay you can use &
5                            // used for $display by default
6
7    initial begin
8      #1 $display("%0t", $realtime); // 1ns i.e. 1000ps
9      #2 $display("%0t", $realtime); // 2ns i.e. 2000ps
10     #0.2 $display("%0t", $realtime); // 0.2ns i.e. 200 ps
11     #0.367 $display("%0t", $realtime); //0.367ns i.e. 367ps
12     #0.0004 $display("%0t", $realtime); //0.0004ns i.e. 0.4ps
13   end
14 endmodule
```

```
1000
3000
3200
3567
3567
```

- *timeunit* declaration overwrites the default time unit.
- *timeprecision* declaration specifies the minimum time that you can express, and also how time is displayed with $display.

- You can overwrite the default time unit using *timeunit* declaration.
- You can also specify the minimum time that you can express, instead of being always one time unit. For example if you specify a time unit of 1 ns, and you want to use fractions of the nano second, you can use the *timeprecision* declaration.
- *timeprecision* can be the same as the *timeunit* or a smaller unit. It can also be 10^n of the smaller unit, provided that it keeps smaller than the *timeunit.*
- For example, *timeunit* may be 1 ns, and *timeprecision* is 1 ps, 10ps, 100ps, or 1000ps, but not 10000ps.

- The value of *timeprecision* is also used while displaying time values using $display and %t. The value displayed is expressed using the value of *timeprecision*.
- This is shown in the above example. 1ns is displayed as 1000, as the displaying is expressed using the value specified by *timepresision* which is 1ps.
- The question now is, can we display using other units different than the value of *timeprecision*? The answer is yes.

Time Values
($timeformat)

16

```
1  module timing;
2
3    timeunit 1 ns;
4    timeprecision 1 ps;
5
6    initial begin
7      $timeformat(-9,5,"ns",12);
8      #2 $display("x %t x", $realtime);
9      #0.2 $display("x %t x", $realtime);
10     #0.367 $display("x %t x", $realtime);
11     #0.0004 $display("x %t x", $realtime);
12   end
13 endmodule
```

```
x     2.00000ns x
x     2.20000ns x
x     2.56700ns x
x     2.56700ns x
```

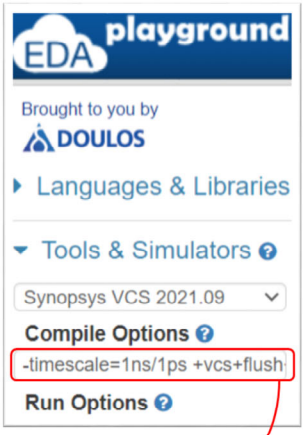
- The question now is, can we display using other units different than the value of *timeprecision*? The answer is yes using *$timeformat.*
- Syntax: $timeformat(<code>,<precision>,<suffix string>,<minimum width>)
  - <code>: (0: s, -3: ms, -6: µs, -9: ns, -12: ps, -15: fs)
  - <precision>: represents the number of fractional digits
  - <suffix_string>: a string written after thenumerical time value
  - <minimum width>: the minimum space on which the value is written.

Time Values
(‘timescale)

17

```
1  `timescale 1ns/1ps
2
3  module timing;
4
5  //  timeunit 1 ns;
6  //  timeprecision 1 ps;
7
8    initial begin
9      $timeformat(-9,5,"ns",12);
10     #2 $display("x %t x", $realtime);
11     #0.2 $display("x %t x", $realtime);
12     #0.367 $display("x %t x", $realtime);
13     #0.0004 $display("x %t x", $realtime);
14   end
15 endmodule
```

```
x     2.00000ns x
x     2.20000ns x
x     2.56700ns x
x     2.56700ns x
```



When the timing values are defined in both the command line, and the Verilog code, the values in the Verilog code overwrite the other ones.

- Instead of writing timeunit, and timeprecision explicitly, you can alternatively use the *'timescale unit/precision* compiler directive.
- Another alternative way is to write “-timescale=1ns/1ps” in the command line that calls the simulator.