Digital Logic Design and HDL Coding

Verilog® HDL
Basic Concepts

Today Program Agenda

- Verilog Lexical Conventions
- Verilog Data Types
- Verilog System Tasks and Compiler Directives

Learning Objectives

- Understand verilog lexical conventions for operators, comments, whitespace, numbers, strings, and identifiers and its use in verilog HDL programming.
- Learn about digital logic value set and various usage of various data types such as nets, registers, vectors, numbers, simulation time, arrays, parameters, memories, and strings.
- Learn about various system tasks used for displaying and monitoring information, and for stopping and finishing the simulation.
- Understand basic compiler directives to define macros and include files.

Verilog lexical conventions

Lexical conventions used in verilog are similar to C programming Language.A verilog program consists of a stream of Tokens.Some of the lexical tokens like space, tab, newline and comments are used to improve verilog code readability. Lexical token used in verilog are:

- White Space
- Comment
- Numbers
- Strings
- Identifiers
- Keywords

White Space

White space is used to describe the characters programmer use to space out verilog code to make it more readable. It consists of

- Blank space (\b)
- Tab (\t)
- Newline (\n)

Whitespace is ignored in Verilog except

- In strings
- when separating tokens

Use of white Space Example

Bad Coding Style: No use of White space in this example. Never write code like this because it lacks readability and style.

```
module addbit(a,b,ci,sum,co);
input a,b,ci;output sum co;

wire a,b,ci,sum,co;endmodule
```

White Space example continued

Good Code : Nice way to write code/Use to Space, tab and new line Improves code readability

//Code for an adder

```
module addbit (   a,
                  b,
                  ci,
                  sum,
                  co
                  );

//Input signals
input a;
input b;
input ci;

//output signals
output sum;
output co;

//code for addbit module

endmodule
```

Comments

There are two types of comments in verilog. A one-line comment starts with the two characters // and end with a carriage return. A multiple-line comment starts with /* and ends with */. Multiple-line comments cannot be nested. However, one-line comments can be embedded in multiple-line comments.

```
a = b & c; // This is a one-line comment
/* This is a multiple line
      comment */
/* This is /* an illegal */ comment */

/* This is //a legal comment */
```

Number Specification

Constant numbers can be specified in decimal, hexadecimal, octal, or binary format. There are two types of number specification in Verilog: sized and unsized. Upper letters are legal in number specifications. Legal base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O).

- Sized numbers
- Unsized numbers
- Unknown(x) and high-impedance (z) values
- Negative numbers

Sized Numbers

Sized numbers syntax: ***<size> '<base format> <number>***

<size> number of bits (in decimal)

<number> is the number in radix <base>

<base> : d or D for decimal (radix 10)

b or B for binary (radix 2)

o or O for octal (radix 8)

h or H for hexadecimal (radix 16)

Sized numbers example

Example:

```
4'b1100  // This is a 4-bit binary number

12'hdba  // This is a 12-bit hexadecimal number

16'd252  // This is a 16-bit decimal number.

 5 ’D 3    // is a 5-bit decimal number
```

Unsized numbers

Numbers that are specified without a <base format> specification are decimal numbers by default. Numbers that are written without a <size> specification have a default of at least 32(depends on verilog compiler and machine-specific)
Examples:

```
23232 // This is a 32-bit decimal number by default

'h3a    // This is a 32-bit hexa decimal number

'o215  // This is a 32-bit octal number

4af     // is illegal (hexadecimal format requires 'h)
```

X or Z values

Verilog uses two symbols for unknown and high impedance values. For modeling digital circuits these values are very important. An unknown value is denoted by an x(4 bits in hex, 3 bits in octal, 1 bit in binary). A high impedance value is denoted by z(4 bits in hex, 3 bits in octal, 1 bit in binary).

Example: 12'h12x //This is a 12-bit hex number;4 least significant bits unknown

6'hx // This is a 6-bit hex number

32'bz // This is a 32-bit high impedance number

X or Z values continued

When the <size> is bigger than the specified value of number the most significant bit is extended to match the <size>.

- Filled with x if the specified MSB is x
- Filled with z if the specified MSB is z
- Zero-extended otherwise

Example1: 6’hA //is stored as 001010

16’bz // is stored as zzzzzzzzzzzzzzzz

8’bx // is stored as xxxxxxxx

X or Z values continued

Example:

```
reg [11:0] a; //variable a is declared as reg type and 12 bit

initial

    begin

        a = 'h x;       // yields xxx

        a = 'h 3x;      // yields 03x

        a = 'h 0x;      // yields 00x

    end
```

Negative Numbers

Negative numbers can be specified by putting a minus sign before the **<size>** for a constant number. Size constants are always positive. It is illegal to have a minus sign between **<base format>** and **<number>**.2’s complement is used to store the value.

Example:

```
-8'd69   // 8-bit negative number stored as 2's complement of 69(10111011)

-8'D3   // Stored as 1111_1101[2’s complement of 3(0000_0011)]

4'd-2    // Illegal specification
```

Underscore character and question marks

An underscore character ‘_’ is allowed anywhere in a number except the first character and are ignored by Verilog.

- Use ‘_’ to improve readability

  Example: 12’b1111_0000_1010

- ‘?’ is the same as ‘z’ (only regarding numbers)

  Example: 4’b10?? // the same as 4’b10zz

Strings

A string is a sequence of characters that are enclosed by double quotes(“ “) and must be contained on a single line. It cannot be on multiple lines. Strings are treated as a sequence of one-byte ASCII values.

Example:

"Hello Verilog World" // is a string with 19 Bytes

“c / d” // is a string with 5 Bytes

"This is a orginal string" // is a string with 24 Bytes

String Variable Declaration in verilog

In verilog a variable is declared to store a string, and declare a register large enough to hold the maximum number of characters the variable will hold. Note that no extra bits are required to hold a termination character; Verilog does not store a string termination character. For example, to store the string “Hello world!” requires a register 8*12, or 96 bits wide.

reg [8*12:1] stringvar; // stringvar is a variable of type reg can hold 96 bits.

```
initial
begin
stringvar = ”Hello world!”;
end
```

Special characters can be displayed in strings only when they are preceded by escape characters.

Escape Characters	Character displayed
\n	newline
\t	tab
\%%	%
\\	\
\”	“

Identifiers and keywords

Identifiers are names used to give an object, such as a register or a function or a module, a name so it can be referenced in verilog code.Identifiers consists of alphanumeric characters, ‘_’, and ‘$.

- Should must start with an alphabetic character(a-z A-Z) or(‘_’)
- Identifiers may contain alphabetic characters, numeric characters, the underscore, and the dollar sign (**a-z A-Z 0-9 _ $)**
- Can not with a digit or a $ sign (The $ sign as the first character is reserved for system tasks)
- Identifiers can be up to 1024 characters long.

Identifiers and keywords continued

Keywords are the type of identifiers reserved by Verilog

Example: reg count; // reg is a keyword; count is an identifier

input clk_in; // input is a keyword, clk_in is an identifier

Legal Identifiers name: wx456 // legal because starts with alphabetic character

_ak523 // legal because starts with _

Ilegal Identifiers name: 22a87b // Illegal because start with a number

$abxxx // Illegal because $ is reserved for system task

Escaped Identifiers

- Escaped identifiers Start with backslash (\ )
- End with whitespace (space, tab, newline)
- Terminate escaped identifiers with white space, otherwise characters that follow the identifier are considered as part of it.

Example1: \x+y-z // x+y-z is the identifier

my_name // **my_name** is the identifier

Note: Escaped identifiers should not be used under normal circumstances. Normally used for translators from other hardware description languages where special characters may be allowed in identifiers

Escaped Identifiers continued

```
// There must be whitespace after the string which uses escape character

        module \1dff ( q,    // Q output

                       \q~ , // Q_out output

                       d,    // D input

                       cl$k, // CLOCK input

                       \reset* // Reset input

                       );

        endmodule
```

Data Types in Verilog

Data types in Verilog are divided into nets and registers.

- Value set and strengths
- Nets and Registers
- Vectors
- Integer, Real, and Time Register Data Types
- Arrays
- Memories
- Parameters
- Strings

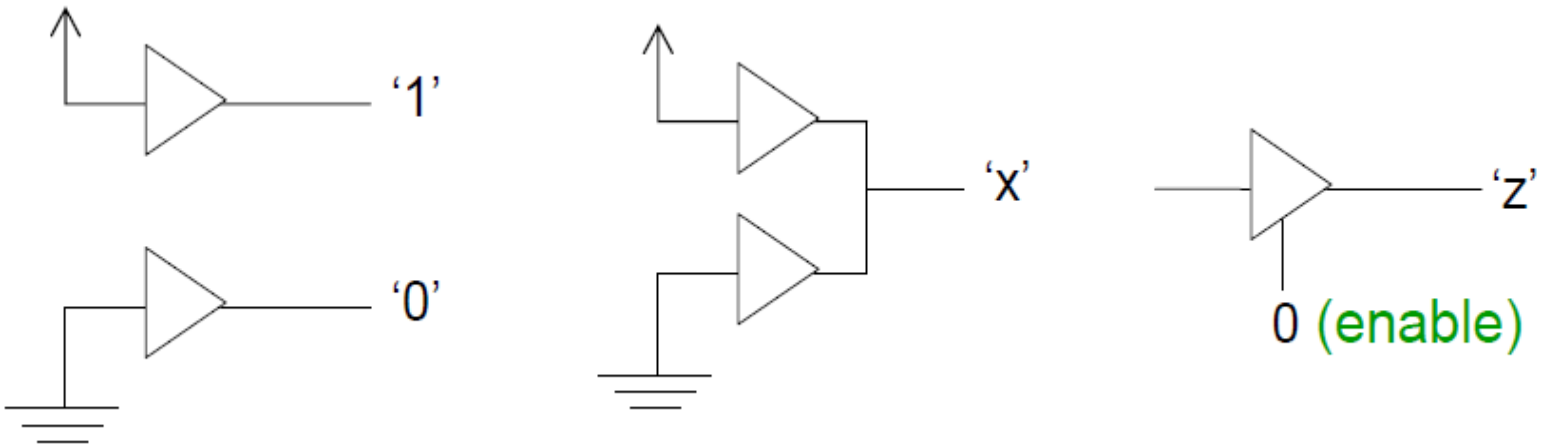
Value Set and Strength

Verilog supports four values and eight strengths to model the functionality of real hardware. The 4 Value levels are:

Value Level	Condition in Hardware Circuit
0	Logic 0, false condition
1	Logic 1, true condition
x	Unknown logic value
z	High impedance, floating state

Value Set and Strength continued

Driver output showing logic Values



Value Set and Strength continued

The Verilog HDL provides for accurate modeling of signal contention, bidirectional pass gates, resistive MOS devices, dynamic MOS, and charge storage circuits by allowing scalar net signal values to have a full range of unknown values and different levels of strength or combinations of levels of strength. Strength level basically used to accurately model

- Signal contention
- MOS devices
- Dynamic MOS
- Charge storage circuits

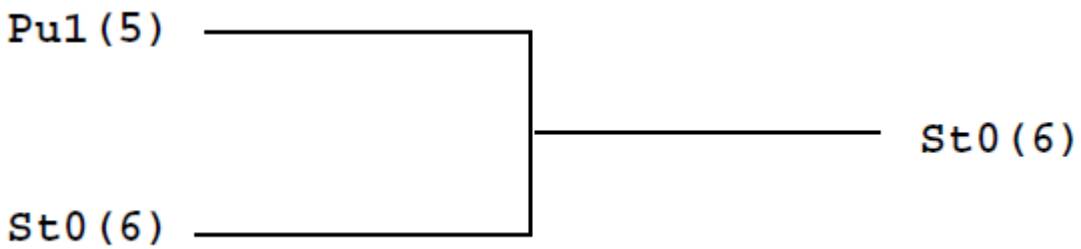
The 8 strength levels to model digital circuits are shown in table(next slide)

Value Set and Strength continued

Strength Name	Type	Strength Level
supply	Driving	7(Strongest)
strong	Driving	6
pull	Driving	5
large	Storage	4
weak	Driving	3
medium	Storage	2
small	Storage	1
hignz	High impedance	0(weakest)

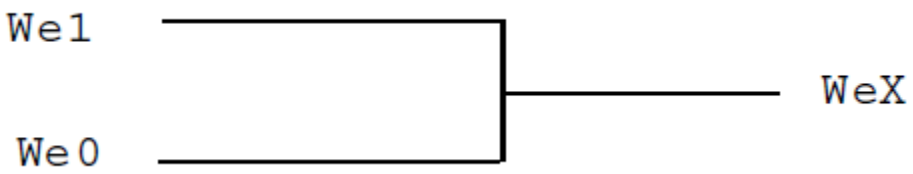
Value Set and Strength continued

Example: If two signals of unequal strength combine in a wired net configuration, the stronger signal is the result. Example below shows the combination of a pull 1 and a strong 0, results in a strong 0, which is the stronger of the two signals.



Value Set and Strength continued

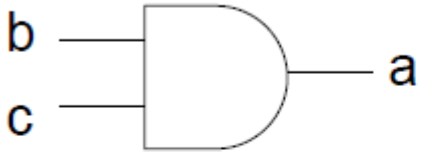
Example: When two signals of equal strength and opposite value combine, the result has a value of x. Example below shows combination of a weak signal with a value of 1 and a weak signal with a value of 0 yielding a signal with weak strength and a value of x.



Nets

Nets are declared with verilog keyword **wire**.Default value is z(for trireg it is x), and 1-bit(scalar), unless declared as vector(multi-bit).

It represent connections between hardware elements



which is continuously driven on

```
wire   a;     // declare net a
wire   b, c;  // declare two wires b, c
```

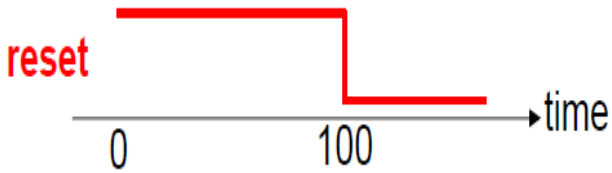
```
wire   d = 1'b0;   // net d is fixed to logic value 0
```

Registers

Registers represent data storage elements in verilog. Registers retain value until next assignment.In Verilog, the term register merely means a variable that can hold a value(this is not a hardware register or flipflop). Keyword: **reg ,** Default value: x

Example: reg reset; //declare a variable reset that can hold its value

```
initial
    begin
        reset = 1'b1;   //
        #100 reset=1'b0; //after 100 time units reset is deasserted
    end
```



Registers continued

Registers can also be declared as signed variables. Such registers can be used for signed arithmetic.

Signed Register Declaration:

```
reg signed [63:0]  n;     // 64 bit signed value

integer i;                // 32 bit signed value
```

Vectors

Net and register data types can be declared as vectors (multiple bit widths).If bit width is not declared default scalar(1-bit).

Syntax: **wire** [msb# : lsb#] data_id; //for net variable

reg [msb# : lsb#] data_id; //for register variable

Example:

```
wire a;          // scalar net variable, default
wire [7:0] bus;  // 8-bit bus
reg clock; // Scalar register default
reg [0:39] addr; // Vector register 40 bit wide
```

Vector part select

verilog allows access to bits or parts of a vector.

Example:

```
busA[7]  // bit # 7 of vector busA

bus[2:0]  // three least-significant bits of bus

addr[0:1] // two most-significant bits addr
```

Integer , Real, and Time Register Data Types

Integer: Keyword: **integer.**

- Very similar to a vector of reg
  - integer variables are signed numbers
  - reg vectors are unsigned numbers
- Bit width: implementation-dependent (at least 32-bits)

  Designer can also specify a width: integer [7:0] tmp;

Examples:

```
integer count; // general purpose variable used as a count
initial
    count = -1; // A negative one is stored in the count
```

Integer , Real, and Time Register Data Types

Real: Keyword: **real**

Values:

- Default value: 0
- Decimal notation: 15.34
- Scientific notation: 3e6 (=3x10 power 6)
- Cannot have range declaration

When a real value is assigned to an integer, the real number is rounded off to the nearest integer, shown in next slide example.

Integer , Real, and Time Register Data Types

Example:

```
real theta;
initial   begin
          theta=4e10; // theta is assigned in scientific notation
         theta=2.13; // theta is assigned a value 2.13
         end
integer i; // Define an integer i
initial
 i = theta; // i gets the value 2 (rounded value of 2.13)
```

Integer , Real, and Time Register Data Types

Time: Used to store values of simulation time

- Keyword: **time**
- Bit width: implementation-dependent (at least 64)
- $time system function gives current simulation time

Example:

```
time save_current_simtime; // a time variable save_current_simtime

initial

save_current_simtime = $time; /* $time is invoked to get the
                                 current simulation time */
```

Arrays

- Only one-dimensional arrays supported by verilog.
- Allowed for reg, integer, time and vector register data types

Syntax: **<data_type> <var_name>[start_idx : end_idx];**

Examples: integer count[0:15]; // An array of 16 count variables

reg count[31:0]; // array of 32 one bit count register variable

reg [4:0] port_id[0:7]; // array of 8 port and each port 5 bit

integer matrix[4:0][4:0]; //illegal multidimensional array not supported

memories

In digital simulation, one often needs to model register files, RAM, and ROM. Memory is modeled in Verilog simply as a one dimensional array of registers.

- Memory = array of registers in Verilog
- Word = an element of the array(can be 1 or more bits)

Examples: reg memory_bit[0:1023]; //memory_bit with 1K 1-bit word

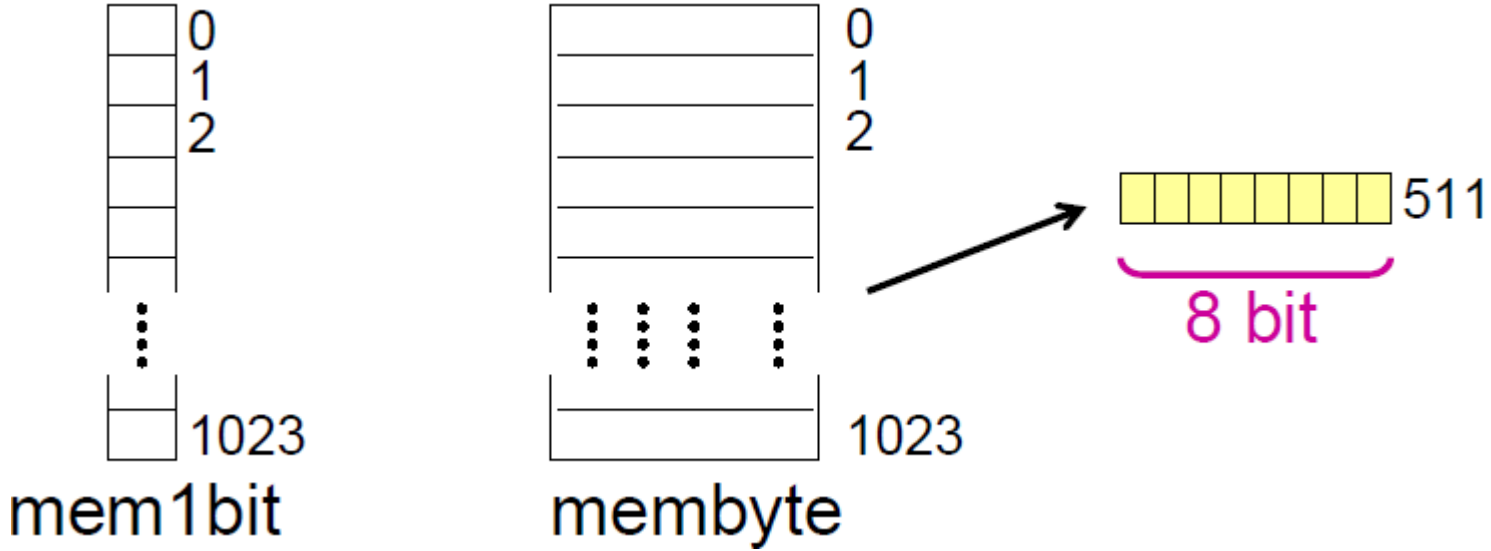
reg [15:0] memword[0:1023]; // memword with 1K 16-bit word

memories continued

reg mem1bit [0:1023]; // mem1bit with 1K 1-bit words

reg [7:0] membyte [0:1023]; // membyte with 1K 8-bit words

membyte[511] // fetches 1 byte word whose address is 511



Parameters

Verilog allows constants to be defined in a module by the keyword **parameter**. Hardcoded numbers should be avoided, instead use parameters to define module definition. Parameters values can be changed at module instantiation or by using the defparam statement.Module behavior can be altered simply by changing the value of a parameter.

Syntax: **parameter <const_id>=<value>;**

- Gives flexibility, allows to customize the module

Example: parameter fifo_depth =16; //defines fifo_depth of 16

parameter bus_width=8; //define bus_width as 8-bit

wire [bus_width-1:0] bus; // a wire variable bus uses bus_width = 8 declared using parameter

Parameters continued

Example://Parameter Override using defparam

```
module mem_ram_sync( clk, rst, read_rq, write_rq, rw_address, write_data, read_data);

parameter WIDTH = 8; parameter DEPTH = 64;

input clk; input   rst; input   read_rq; input     write_rq;

rw_address; input[WIDTH-1:0]      write_data; output[WIDTH-1:0]      read_data;

reg[WIDTH-1:0]          read_data;

//Actual code for memory

endmodule
```

Parameters continued

// Test Bench module **showing parameter override using defparam**

```
module memory_tb();

reg clk,rst; reg    read_rq; reg       write_rq; reg[5:0]rw_address; reg[31:0]write_data; wire[31:0]
read_data; reg[6:0]q_cnt;

//Actual test bench code

//Overriding using defparam

defparam u_dut_ram.WIDTH = 32;

defparam u_dut_ram.DEPTH = 64
```

```
// module instantiation

mem_ram_sync      u_dut_ram      ( .clk(clk), .rst
(rst),.read_rq(read_rq),.write_rq(write_rq),          .
rw_address(rw_address), .write_data(write_data), .
read_data(read_data));
```

Parameters continued

Example: // Passing more than one parameter

```
module ram_sp_sr_sw (clk , address , data , cs , we , oe);

parameter DATA_WIDTH = 8 ;

parameter ADDR_WIDTH = 8 ;

parameter RAM_DEPTH = 1 << ADDR_WIDTH;

// Actual code of RAM here

endmodule
```

multiple parameter values must be passed in the order they are declared in the sub module.

```
//ram controller test bench code
module ram_controller ();

        // Controller Code

//passing 3 parameter during module instantiation

ram_sp_sr_sw #(16,8,256) ram(clk,address, data,cs,we,oe);

endmodule
```

System Tasks and Compiler Directives

Verilog provides standard system tasks for displaying, monitoring of nets, stopping and finishing simulation. All system tasks appear in the form **$<keyword>**.

displaying on the screen	$display
monitoring values of nets	$monitor
stopping the simulation	$stop
finishing the simulation	$finish

System task usage

System task	Usage
$display	$display(p1, p2, p3,....., pn);
$monitor	$monitor(p1, p2, p3,....., pn);
$stop	$stop;
$finish	$finish;

system task display format

Format	Display
%d or %D	variable in decimal
%b or %B	variable in binary
%h or %H	variable in hex
%t or %T	in current time format
%s or %S	string
%c or %C	ASCII character
%m or %M	hierarchical name

$display system task

$display is the main system task for displaying values of variables or strings or expressions. This is one of the most useful tasks in Verilog.

Usage: $display(p1, p2, p3,....., pn);

p1, p2, p3,..., pn can be quoted strings or variables or expressions. A $display inserts a newline at the end of the string by default. A $display without any arguments produces a newline.

Example: $display($time); // Display value of current simulation time 230

Simulation output: 230

$display more examples

```
module arithmetic_operators();

        initial

         begin

        $display (" 5 + 10 = %d", 5 + 10);

        $display (" 5 - 10 = %d", 5 - 10);

        $display (" 10 - 5 = %d", 10 - 5);

        $display (" 10 * 5 = %d", 10 * 5); #10 $finish;  end

endmodule
```

Simulation output displayed using $display system task:

```
5 + 10 = 15

5 - 10 = -5

10 - 5 = 5

10 * 5 = 50
```

use of special character in $display

Special Characters: //Display special characters, newline and %

$display("This is a \n multiline string with a %% sign");

Simulation output: This is a

multiline string with a % sign

reg [0:40] virtual_addr;

$display(“At time %d virtual address is %h”, $time, virtual_addr);

Output: At time 200 virtual address is 1fe000001c

$monitor system task

$monitor: monitors a signal when its value changes.

Syntax: **$monitor(p1, p2, p3, …, pn);**

p1,…, pn can be quoted string, variable, or signal names

- Continuously monitors the values of the specified variables or signals, and displays the entire list whenever any of them changes.
- $monitor needs to be invoked only once (unlike $display)
- Only one $monitor (the latest one) can be active at any time
  - $monitoroff to temporarily turn off monitoring
  - $monitoron to turn monitoring on again

$monitor system task continued

$monitor Examples: initial begin

$monitor($time, “Value of signals clock=%b, reset=%b”, clock, reset); end

Simulation Output:

0 value of signals clock=0, reset=0

5 value of signals clock=1, reset=1

10 value of signals clock=0, reset=1

```
reg clock, reset; /*variable clock and reset
                    declared as reg*/
initial
        begin
          clock = 1’b0; reset =1’b0;
          #5   clock = 1’b1; reset =1’b1;
          #10  clock = 1’b0; reset =1’b1;
        end
```

$monitor another example

```
module initial_begin_end();

reg clk,reset,enable,data;

initial begin

$monitor("%g clk=%b reset=%b enable=%b data=%b",

            $time, clk, reset, enable, data);

#1 clk = 0;  #10 reset = 0;

#5 enable = 0; #3 data = 0; #1 $finish; end

endmodule
```

Simulation output:

0 clk=x reset=x enable=x data=x

1 clk=0 reset=x enable=x data=x

11 clk=0 reset=0 enable=x data=x

16 clk=0 reset=0 enable=0 data=x

19 clk=0 reset=0 enable=0 data=0

$stop and $finish system task

$stop: stops simulation

- Simulation enters interactive mode when reaching a $stop system task
- Most useful for debugging

$finish: terminates simulation

Examples: initial begin

```
clock=0;  reset=1;  #100 $stop; // This will suspend the simulation at time = 100

#1000 $finish;                  // This will terminate the simulation at time = 1100
```

end

compiler directives

General syntax: **`<keyword>**

`define: similar to #define in C, used to define macros

`<macro_name> to use the macro defined by `define

Examples:

`define WORD_SIZE 32 //macro WORD_SIZE defined using `define compiler directive

`define S $stop // define an alias. A $stop will be substituted wherever 'S appears

reg [`WORD_SIZE-1:0] reg32; // use of macro WORD_SIZE in register variable reg32

compiler directive example

```
// Examples of `define text macros

`define MY_NUMBER 5 //macro MY_NUMBER

`define MY_STRING "Hello world!" //macro MY_STRING

module test;   reg [7:0] a, b;

initial

begin

$display(`MY_NUMBER);   $display(`MY_STRING);

endmodule
```

Simulation output:

5

Hello world!

compiler directive

`include

The `include directive allows you to include entire contents of a Verilog source file in another Verilog file during compilation. This directive is typically used to include header files, which typically contain global definitions.

split into 2 files

```
`define A 16 // macro A
`define B 8  // macro B
`define C 32 // macro C

module KM ( inA, inB, outC);
input [`A-1:0] inA;
input [`B-1:0] inB;
output [`C-1:0] outC;
……
endmodule
```

```
//header file def.v contains macros

`define A 16 // bit width for inA

`define B 8 // bit width for inB

`define C 32 // bit width for outC
```

```
`include def.v // header file included using `include

module KM ( inA, inB, outC);

input [`A-1:0] inA; input [`B-1:0] inB; output [`C-1:0] outC;

…… endmodule
```