



שפת תכנון חומרה ורילוג - Verilog

Verilog – Basic Concepts

Dr. Avihai Aharon

Objectives

- ❖ **Lexical conventions for operators, comments, whitespace, numbers, strings, and identifiers**
- ❖ **Logic value set and data types such as nets, registers, vectors, numbers, simulation time, arrays, parameters, memories, and strings**
- ❖ **Useful system tasks for displaying and monitoring information, and for stopping and finishing the simulation**
- ❖ **Basic compiler directives (defining macros and including files)**

Lexical Conventions: White-space

- ❖ Blank spaces (**\b**), tabs (**\t**) and new-lines (**\n**) comprise the **white-spaces**
- ❖ **White-space** is ignored by Verilog except when it separate tokens
- ❖ **White-space** is not ignored in string

Lexical Conventions: Comments

- ❖ **Comments** can be inserted in the code for readability and documentation
- ❖ There two ways to write **comments**
 - ✓ A **on-line comment** starts with “//”. Verilog skips from that point to the end of line
 - ✓ A **multiple-line comment** starts with “/*” and ends with “*/”. Multiple-line comments cannot be nested!!!

```
a = b && c;      // This is a one-line comment
```

```
/* This is a multiple-line  
   comment */
```

```
/* This is /* an illegal */ comment */
```

Lexical Conventions: Operators

- ❖ **Operators** are of three types: **unary**, **binary**, and **ternary**
 - ✓ **Unary** operators precede the operand
 - ✓ **Binary** operators appear between two operands
 - ✓ **Ternary** operators have two separate operators that separate three operands

`a = ~ b;` // ~ is a unary operator. b is the operand

`a = b && c;` // && is a binary operator. b and c are operands

`a = b ? c : d;` // ?: is a ternary operator. a, b and c are operands

Lexical Conventions: Number Specification

- ❖ Numbers in Verilog can be **integers** or **reals**.
- ❖ Real numbers can be represented in **decimal** or **scientific** format.
- ❖ There two types of integer number: **sized** and **unsized**.

Lexical Conventions: Sized Numbers

❖ Sized numbers are represented as **<size>'<base format><value>**

- ✓ **<size>** is the size in bits (written only in decimal and specifies the number of bits in the number)
- ✓ **<base format>** - can be b(binary), o(octal), d(decimal) or h(hexadecimal)
- ✓ **<value>** - specified as consecutive digits from **0 - 9**, and **a - f**. Only a subset of these digits is legal for a particular base. Uppercase letters are legal for number specification

4'b1111	// This is a 4-bit binary number
64'hfb01	// This is a 64-bit hexadecimal number
9'o17	// This is a 9-bit octal number
16'd255	// This is a 16-bit decimal number

Lexical Conventions: Unsized Numbers

- ❖ Unsized numbers are represented as '**<base format><value>**'
 - ✓ 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 number of bits that is simulator- or machine-specific (at least 32 bits)
 - ✓ **<base format>** - can be b(binary), o(octal), d(decimal) or h(hexadecimal)
 - ✓ **<value>** - specified as consecutive digits from **0 - 9**, and **a - f**. Only a subset of these digits is legal for a particular base. Uppercase letters are legal for number specification

```
23456    // This is a 32-bit decimal number by default
'b1111   // This is a 32-bit binary number
'hfb01   // This is a 32-bit hexadecimal number
'd255    // This is a 32-bit decimal number
```


Lexical Conventions: X or Z values

- ❖ Verilog has two values for **unknown** and **high impedance** values
 - ✓ An **unknown** value is denoted by “**x**”
 - ✓ An **high impedance** value is denoted by “**z**”
 - ✓ These values are very important for modeling real circuits
 - ✓ An “**x**” or “**z**” sets **four** bits for a number in the **hexadecimal** base, **three** bits for a number in the **octal** base and **one** bit for a number in **binary** base
 - ✓ If the MSB of a number is **0**, **x** or **z**, the number is automatically extended to fill the most significant bits, respectively with **0**, **x** or **z**.

```
12'h13x // 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
```

Lexical Conventions: 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 **<value>**

```
-6'd3    // This is a 6-bit negative number stored as 2's  
          // complement of 3  
4'd-2    // Illegal specification
```

Lexical Conventions: Special Characters & Marks

- ❖ An underscore character “_” is allowed anywhere in a number except the first character
 - ✓ Allowed only to improve readability of numbers and are ignored by Verilog
- ❖ A question mark “?” is the Verilog HDL alternative for “z” in the context of numbers
 - ✓ The “?” is used to enhance readability in the “**casex**” and “**casez**” statements (will be discussed later)

12'b1111_0000_1010	// Use of underline characters for
	// readability of the number
4'b10??	// Equivalent of a 4'b10zz

Lexical Conventions: Strings

- ❖ A string is a sequence of characters that are enclosed by double quotes
 - ✓ A string must be contained on a single line (without a carriage return)
 - ✓ A string cannot be on multiple lines
 - ✓ Strings are treated as a sequence of one-byte ASCII value

"Hello Verilog World!!!"

// Is a string

"a / b = c"

// Is a string

"3x3 + 4x4 = 5x5"

// Is a string

Lexical Conventions: Identifiers & Keywords

- ❖ **Keywords** are special identifiers reserved to define the language constructs
 - ✓ **Keywords** are in lower case
 - ✓ A list of all **keywords** in Verilog is contained in Verilog User Manual (List of Keywords, System Tasks, Compiler Directives)
- ❖ **Identifiers** are names given to objects and they can be referenced in the design (can be up to 1023 characters long)
 - ✓ **Identifiers** are made up of alphanumeric characters, the underscore “_” and the dollar sign “\$” and are case sensitive
 - ✓ **Identifiers** start with an alphabetic character or an underscore “_” and cannot start with a number or a “\$” sign (reserved for system tasks)

reg value;	// reg is a keyword; value is an identifier
input clk;	// input is a keyword; clk is an identifier

Lexical Conventions: Case Sensitivity

- ❖ Verilog is a **case-sensitive** language
- ❖ All Verilog keywords are in lowercase
- ❖ Simulators can be used in **case-insensitive** mode by specifying **-u** command-line option

```
module MUX2_1(out,a,b,sel);  
output out;  
input a,b,sel;  
    not not1(SEL,sel);  
    and and1(a1,a,SEL);  
    and and2(b1,b,sel);  
    or or1(out,a1,b1);  
endmodule
```

Data Types: Value Set

- ❖ Verilog supports four **values** and eight **strengths** to model the functionality of real hardware
 - ✓ The four **value levels** are listed in the table

<i>Value Level</i>	<i>Condition in Hardware Circuits</i>
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown value
z	High impedance, floating state


Data Types: Unknown Logic Values

- ❖ Verilog contains three **unknown logic values** (**x**, **L** and **H**)
 - ✓ **X** represents a complete unknown (value can be logic **1**, **0** or **z**)
 - ✓ **L** represents a partial unknown (value can be logic **0** or **z**, but not **1**)
 - ✓ **H** represents a partial unknown (value can be logic **1** or **z**, but not **0**)
 - ✓ The **X** value is far more commonly used than **L** and **H**

<i>Value Level</i>	<i>Condition in Hardware Circuits</i>
x	complete unknown (logic 1 , 0 or Z)
L	partial unknown (logic 0 or z , but not 1)
H	partial unknown (logic 1 or z , but not 0)

Data Types: Strength Levels

- ❖ Eight **strength levels** are often used to resolve conflicts between drivers of different strengths in digital circuits
 - ✓ **Value levels** “1” and “0” have the following **strengths levels**
 - ✓ The **capacitive** (storage) strengths (**large**, **medium**, **small**) apply only to nets of type **triereg** and to **tran** primitives.

<i>Strength Level</i>	<i>Type</i>	<i>%V formatting</i>	<i>Specification</i>	<i>Degree</i>
7 Supply	Driving	Su0, Su1	Supply0, Supply1	strongest
6 Strong	Driving (default)	St0, St1	Strong0, Strong1	
5 Pull	Driving	Pu0, Pu1	Pull0, Pull1	
4 Large	Capacitive	La0, La1	Large	
3 Weak	Driving	We0, We1	Weak0, Weak1	
2 Medium	Capacitive	Me0, Me1	Medium	
1 Small	Capacitive	Sm0, Sm1	Small	
0 High Z	High impedance	Hi0, Hi1	Highz0, Highz1	weakest

Data Types: Strength Levels

- ❖ If two signals with unequal strengths are driven on a wire, the stronger signal prevails (**strong1** + **weak0** = **strong1**)
- ❖ If two signals of equal strengths are driven on a wire, the result is unknown (**strong1** + **strong0** = **x**)
- ❖ Strength levels are particularly useful for accurate modeling of signal contention, MOS devices, dynamic MOS, and over low-level devices
- ❖ Only **triereg** nets can have storage strengths (large, medium, and small)

Data Types: Nets

- ❖ **Nets** represent connections between hardware elements
- ❖ **Nets** have values continuously driven on them by the outputs of gates or modules that they are connected to
- ❖ For example: net **a** is connected to the output of and gate **g1**. Net **a** will continuously assume the value computed at the output of gate **g1**, which is **b & c**
- ❖ **Nets** are declared primarily with the keyword **wire**
- ❖ Undeclared **nets** are one-bit values of type **wire** by default unless they are declared explicitly as another net-type vectors
- ❖ The terms wire and net are often used interchangeably
- ❖ **Cannot be assigned in an initial or always block**

Data Types: Registers

- ❖ **Registers** represent data storage elements
- ❖ Registers retain value until another value placed into them
- ❖ In Verilog, the term **register** merely means a variable that can hold a value (don't confuse with structural storage elements!!!)
- ❖ A register does not need a driver (unlike a net)
- ❖ Verilog register do not need a clock (as hardware register do)
- ❖ Values of registers can be changed anytime in a simulation by assigning a new value to the register.
Hold their value until explicitly assigned in an initial or always block.
- ❖ Register data types are commonly declared by the keyword **reg**.
- 22❖ A default value for a reg data type is **x**.

Data Types: Registers vs. Nets

reg vs. wire

- Oh no... Don't go there!

- A reg is not necessarily an actual register, but rather a “driving signal”... (huh?)
- This is truly the most ridiculous thing in Verilog...
- But, the compiler will complain, so here is what you have to remember:

1. Inside **always** blocks (both sequential and combinational) only **reg** can be used as LHS.
2. For an **assign** statement, only **wire** can be used as LHS.
3. Inside an **initial** block (Testbench) only **reg** can be used on the LHS.
4. The **output** of an instantiated module can only connect to a **wire**.
5. **Inputs** of a **module** cannot be a **reg**.

```
reg r;  
always @*  
    r = a & b;
```

```
wire w;  
assign w = a & b;
```

```
reg r;  
initial  
begin  
    r = 1'b0;  
    #1  
    r = 1'b1;  
end
```

```
module m1 (out)  
    output out;  
endmodule  
  
reg r;  
m1 m1_instance(.out(r));
```

```
module m2 (in)  
    input in;  
    reg in;  
endmodule
```

Data Types: Vectors

- ❖ **Nets** or **reg** data types can be declared as vectors (multiple bit width). If bit width is not specified, the default is scalar (1-bit)
- ❖ **Net** and **register** vectors declaration

```
<net_type> [range] [delay] <net_name1>, <net_name2>, ... ;  
<reg_type> [range] [delay] <net_name1>, <net_name2>, ... ;
```

```
wire a;                // scalar net variable, default  
wire [7:0] bus;         // 8-bit bus  
wire [31:0] busA, busB, busC; // 3 buses of 32-bits width  
reg clock;             // scalar register, default  
reg [0:40] virtual_addr; // vector register, virtual address  
                        // 41-bits width
```

Data Types: Vectors

- ❖ Vectors can be declared at [*high#* : *low#*] or [*low#* : *high#*]
- ❖ The left number in the squared brackets is always the most significant bit of the vector
- ❖ For the declared vector it is possible to address bits or parts of vectors

```
wire [7:0] bus;  
reg [0:40] virtual_addr;
```

```
bus[2:0];           // three least significant bits of vector bus  
                    // using bus[0:2] is illegal because the significant bit  
                    // should always be on the left of a range specification
```

```
virtual_addr[0:1]; // two most significant bits of vector virtual_addr
```

Data Types: Types of Registers

- ❖ **Integer**, **real** and **time** register data types are supported in Verilog

<i>Register Data Types</i>	<i>Functionality</i>
<i>integer</i>	Signed integer variable, 32-bits wide. Arithmetic operations produce 2's-complement results
<i>real</i>	Signed floating-point variable, double precision
<i>time</i>	Unsigned integer variable, 64-bits wide. (Verilog-XL stores simulation time as a 64-bit value)

Data Types: Integer Registers

- ❖ An **integer** is a general purpose register data type used for manipulating quantities (such as counting) and declared by keyword **integer**
- ❖ The default width for an **integer** is the host-machine word size, which is implementation specific (at least 32 bits)
- ❖ Registers declared as data type **reg** store **unsigned** values
- ❖ Registers declared as data type **integer** store **signed** values

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

Data Types: Real Registers

- ❖ Real number constants and real register data types are declared with the keyword **real** and can be specified in **decimal** or in **scientific** notation
- ❖ Real numbers cannot have a range declaration (default value is “0”)
- ❖ When assigned to an **integer**, the **real** number is rounded off to the nearest **integer**

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

Data Types: Time Registers

- ❖ Verilog simulation is done with respect to **simulation time**
- ❖ A special **time register** data type, that declared with the keyword **time**, is used in Verilog to store **simulation time**
- ❖ The width for **time register** data types is implementation specific but is at least 64 bits
- ❖ The system function **\$time** is invoked to get the current **simulation time** that measured in **simulation seconds** (will be discussed later)

```
time save_sim_time;           // define a time variable called save_sim_time
initial
    save_sim_time = $time; // save the current simulation time
```

Data Types: Arrays

- ❖ **Arrays** are allowed in Verilog for **reg**, **integer**, **time**, and **vector** register data types and not allowed for **real** variables
- ❖ Arrays are accessed by **<array_name> [<subscript>]**
- ❖ Multidimensional **arrays** are not permitted in Verilog

```
integer count[0:7];           // an array of 8 count variables
reg bool[31:0];               // array of 32 one-bit boolean register variables
time chk_point[1:100];       // array of 100 time checkpoint variables
reg [4:0] port_id[0:7];       // array of 8 port_ids; each port_id is 5-bits wide
integer matrix[4:0] [4:0];    // illegal declaration – multidimensional array

count[5]                       // 5th element of array of count variable
chk_point[100]                 // 100th time check point value
port_id[3]                     // 3rd element of port-id array; this is a 5-bit value
```

Data Types: Memories

- ❖ **Memories** are modeled in Verilog simply as an array of registers
- ❖ Each element of the array is known as a **word**. Each word can be one or more bits
- ❖ It is important not to confuse arrays with net or register vectors
 - ❖ vector: n-bits wide single element
 - ❖ array: 1-bit or n-bits wide multiple elements
- ❖ It is important to differentiate between n 1-bit registers and one n-bit register

```
reg mem1bit [0:1023];           // memory mem1bit with 1k 1-bit words
reg [7:0] membyte [0:1023];    // memory membyte with 1k 8-bit words (bytes)

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

Data Types: Parameters

- ❖ Constants in Verilog can be defined in a module by the keyword **parameter** (cannot be used as variables)
- ❖ **Parameter** values for each module instance can be overridden individually at compile time (allows the module instance to be customized)
- ❖ Module definitions may be written in terms of parameters (hard-coded numbers should be avoided)
- ❖ Parameters can be changed at module instantiation or by using the **defparam** statement (will be discussed later)

```
parameter port_id = 5;           // defines a constant port_id  
parameter cache_line_didth = 256; // constant defines width of cache line
```

Data Types: Strings

- ❖ **String** can be stored in **reg** (the width of the register variable must be large enough to hold the string)
- ❖ Each character of the string takes up 8 bits (1 byte)
- ❖ It is always safe to declare a string that is slightly wider than necessary
 - ✓ If the width of the register is greater than the size of the **string**, Verilog fills bits to the left of the **string** with zeros
 - ✓ If register width is smaller than the **string** width, Verilog truncates the left most bits of the **string**

```
reg [8*19:1] string_value;           // declare a variable that is 18 bytes wide
initial
    string_value = "Hello Verilog World";    // string can be stored in variable
```

Data Types: String Special Characters

- ❖ **Special characters** serve a special purpose in displaying **strings**, such as newline, tabs and displaying argument values
- ❖ **Special characters** can be displayed in **strings** only then they are preceded by **escape** characters

<i>Escaped Characters</i>	<i>Character Displayed</i>
\n	newline
\t	tab
%%	%
\\	\
\"	"
\ooo	Character written in 1-3 octal digits

Special Language Tokens

System Tasks & Functions

- ❖ The “\$” sign denotes Verilog *system tasks and functions*

\$<task_identifier>

Delay Specification

- ❖ The pound “#” sign character denotes the *delay specification* (for procedural statements and gates instances)

#<delay_value>

System Tasks

- ❖ Verilog provides standard **system tasks** to do certain routine operations
- ❖ All **system tasks** appear in the form **\$<keyword>**
- ❖ We will discuss only the most useful **system tasks**
 - ✓ Reading simulation time
 - ✓ Accessing time information
 - ✓ Displaying on the screen
 - ✓ Monitoring values of nets
 - ✓ Stopping or finishing simulations
 - ✓ Dumping signal values

System Tasks: Reading Simulation Time

- ❖ **\$time**, **\$realtime**, and **\$stime** functions return the current simulation time
- ❖ Each of these functions returns value that is scaled to the time unit of the module that invoked it (will be discussed later)
- ❖ **\$time** returns time as a 64-bit integer
- ❖ **\$stime** returns time as a 32-bit integer
- ❖ **\$realtime** returns time as a real number

System Tasks: Accessing Time Information

- ❖ **\$timeformat** system task and **%t** formatter can be used to globally control how time values are displayed
- ❖ Usage: **\$timeformat(<unit>,<precision>,<suffix>,<min_width>);**
 - ✓ **<unit>** - Integer between 0 (sec) and -15 (fsec)
(indicating the time scale)
 - ✓ **<precision>** - Number of decimal digits to display
 - ✓ **<suffix>** - String to display after time value
 - ✓ **<min_width>** - Minimum fields width used for display
- ❖ When using multiple **`timescale** compiler directives, displayed values are scaled to the smallest precision

System Tasks: Printing Time Information

```
// Printing time information example
// The time display will be similar to: <180.00ns >
...
$display(" time \t realtime \t stime \t in1 \t o1 ");
$timeformat(-9, 2, "ns", 10);
$monitor("%d \t %t \t %d \t %b \t %b", $time, $realtime, $stime, in1, o1);
...
```



time	realtime	stime	in1	o1
0	0.00ns	0	0	x
10	9.50ns	10	0	1
15	14.50ns	15	1	1
20	19.50ns	20	1	0

System Tasks: Displaying Information

- ❖ **\$display** is the main system task for displaying values of variables or strings or expressions (one of the most useful Verilog tasks)
- ❖ **\$display** task usage: **\$display**(p1, p2, p3, ..., pn);
 - ✓ p1, p2, ..., pn can be quoted strings, variables or expressions
- ❖ **\$display** without any arguments produces a **newline**
- ❖ **\$display** support different bases (default is decimal)
 - ✓ **\$display**(p1, p2, p3, ..., pn);
 - ✓ **\$displayb**(p1, p2, p3, ..., pn);
 - ✓ **\$displayo**(p1, p2, p3, ..., pn);
 - ✓ **\$displayh**(p1, p2, p3, ..., pn);
- ❖ Strings can be formatted by using the format specifications

System Tasks: \$display Format Specifications

<i>Format</i>	<i>Display</i>
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hexadecimal
%c or %C	Display ASCII character
%m or %M	Display hierarchical name (no argument needed)
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format (e.g., 3e10)
%g or %G	Display real number in decimal format (e.g., 2.13)
%f or %F	real number in scientific or decimal (whichever is shorter)

System Tasks: \$write and \$strobe Tasks

- ❖ **\$write** and **\$strobe** are identical to **\$display** except following:
 - ✓ **\$write** does not print a new-line character
 - ✓ **\$strobe** the argument evaluation is delayed just prior to the advance of simulation time (print steady-state values of the signals)

```
$display ($time, "%b \t %h \t %d \t %o", signal1,signal2,signal3,signal4);  
$display ($time, "%b \t", signal1, "%h \t", signal2, "%d \t", signal3, "%o", signal4);  
$write ($time, "%b \t %h \t %d \t %o \n", signal1,signal2,signal3,signal4);  
$strobe ($time, "%b \t %h \t %d \t %o", signal1,signal2,signal3,signal4);
```

- ❖ Both **\$write** and **\$strobe** support multiple bases

\$writeb	\$strobeb
\$writeo	\$strobo
\$writeh	\$strobeh

System Tasks: \$display Task Examples

```
$display("Hello Verilog World!!!");           // display the string in quotes
```

```
--- Hello Verilog World!!!
```

```
$display($time);                             // display the current simulation time
```

```
--- 230
```

```
// display value of 41-bit virtual address and simulation time
```

```
$display("At time %d virtual address is %h", $time, virtual_addr);
```

```
--- At time 230 virtual address is 1fe000001c
```

```
// display value of 5-bit port_id in binary
```

```
$display("ID of the port is %b", port_id);
```

```
--- ID of the port is 00101
```

```
// display the hierarchical name of instance p1 instantiated under the highest-level module
```

```
// called top. No arguments is required. This is a useful feature
```

```
$display("This string is displayed from %m level of hierarchy");
```

```
--- This string is displayed from top.p1 level of hierarchy
```

System Tasks: Monitoring Information

- ❖ **\$monitor** is the main system task provides a mechanism to monitor a signal when its value changes
- ❖ **\$monitor** task usage: **\$monitor**(p1, p2, p3, ..., pn);
 - ✓ p1, p2, ..., pn can be quoted strings, variables or expressions (format similar to **\$display** task)
- ❖ **\$monitor** continuously monitors the values of the variables or signals specified in the parameter list and displays all parameters in the list whenever the value of any one variable or signal changes
- ❖ Only one monitoring list can be active at a time (the last **\$monitor** statement will be the active statement, the earlier **\$monitor** statement will be overridden)
- ❖ **\$monitor** supports different default bases (**\$monitorb**, **\$monitro**, **\$monitorh**)

System Tasks: \$monitor Tasks Control

- ❖ Monitoring is turning on by default at the beginning of the simulation and can be controlled during the simulation.

```
// Monitor time and value of the signals clock and reset.  
// Clock toggles every 5 time units and reset goes down at 10 time units  
initial begin  
    $monitor($time, " Value of signals clock = %b reset = %b", clock, reset);  
end
```

Partial output of the monitor statements:

```
--- 0 Value of signals clock = 0 reset = 1  
--- 5 Value of signals clock = 1 reset = 1  
--- 10 Value of signals clock = 0 reset = 0
```

- ❖ Two tasks are used to switch monitoring on and off
 - ❖ The **\$monitoron** task enables monitoring during a simulation
 - ❖ The **\$monitroff** task disables monitoring during a simulation

System Tasks: Stopping Simulation

- ❖ The task **\$stop** is provided to stop during simulation
- ❖ **\$stop** task usage: **\$stop;**
- ❖ **\$stop** task puts the simulation in an interactive mode
- ❖ **\$stop** task is used whenever the designer wants to suspend the simulation and examine the values of signals in the design from the interactive mode

```
// Stop at time 100 in the simulation and examine the results
initial // to be explained later (time = 0)
begin
    clock = 0;
    reset = 1;
    #100 $stop; // This will suspend the simulation at time = 100
end
```

System Tasks: Finishing Simulation

- ❖ The **\$finish** task terminates the simulation
- ❖ **\$finish** task usage: **\$finish;**

```
// Stop at time 100 in the simulation and examine the results
initial // to be explained later (time = 0)
begin
    clock = 0;
    reset = 1;
    #900 $finish; // This will terminated the simulation at
time = 900
end
```

System Tasks: File Output

- ❖ **\$fopen** task opens a file and returns a Multi-Channel Descriptor (MCD)
 - ✓ The MCD is a 32-bit unsigned integer uniquely associated with the file
 - ✓ MCD will equal “0” if the file cannot be opened for writing
 - ✓ If the file successfully opened – one bit in the MCD will be set
 - ✓ The MCD should be thought of as a set of 32 flags, where each flag represents a single output channel
 - ✓ The least significant bit (bit 0) of a MCD always refers to the standard output (the log file and the screen)

System Tasks: \$fopen & \$fclose

- ❖ Display system tasks that begin with “\$f” direct their output to whichever file or files are associated with the MCD
 - ✓ These tasks (\$fdisplay, \$fwrite, \$fmonitor, and \$fstrobe) accept the same parameters as the tasks they are based upon, except the first parameter that must be an MCD
- ❖ \$fclose closes the channel specified in the multi-channel

```
integer MCD1;  
    MCD1 = $fopen("<name_of_file>");  
    $fdisplay(MCD1, p1, p2, p3, ..., pn);  
    $fwrite(MCD1, p1, p2, p3, ..., pn);  
    $fstrobe(MCD1, p1, p2, p3, ..., pn);  
    $fmonitor(MCD1, p1, p2, p3, ..., pn);  
    ...  
    $fclose(MCD1);
```

System Tasks: \$fopen example

```
initial
begin
    cpu_chan = $fopen("cpu.dat"); if(!cpu_chan) $finish;
    alu_chan = $fopen("alu.dat"); if(!alu_chan) $finish;
    // channel to both cpu.dat and alu.dat files
    messages = cpu_chan | alu_chan;
    // channel to: cpu.dat and alu.dat files, standard out and verilog.log
    broadcast = 1 | messages;
end

always @(posedge clock) // print the following to alu.dat every positive edge of
                        // clock signal
    $fdisplay(alu_chan, "acc= %h f=%h a=%h b=%h", acc, f, a, b);

// at every reset print a message to:
// cpu.dat and alu.dat files, standard out and verilog.log file
always @(negedge reset)
    $fdisplay(broadcast, "System reset at time %d", $time);
```

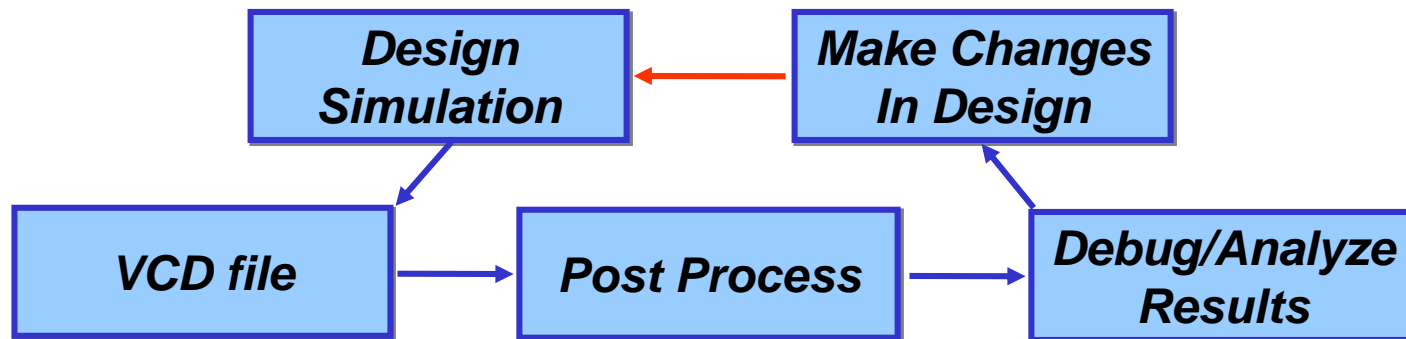

System Tasks: Random Number Generation

- ❖ The system task **\$random** is used for generating a random number
- ❖ **\$random** task usage: **\$random**; or **\$random(<seed>)**;
 - ✓ The value **<seed>** is optional and used to ensure the same random number sequence each time the test is run
- ❖ The task **\$random** returns a 32-bit random number. All bits, bit-selects, or part-selects of the 32-bit random number can be used

```
...  
integer r_seed;  
reg [31:0] addr;  
...  
initial r_seed = 2; // arbitrary define the initial seed as 2  
...  
// generates random addresses  
always @(posedge clock) addr = $random(r_seed);  
...
```

System Tasks: Value Changed Dump (VCD) File

- ❖ A **Value Change Dump (VCD)** is an ASCII file that contains information about simulation time, scope and signal definition, and signal value changes in the simulation run
 - ✓ All signals or a selected set of signals in a design can be written to a **VCD file** during simulation
- ❖ Post-processing tools (Magellan, VirSim, Signalscan, Simvision, Debussy) can take the **VCD file** as input and visually display hierarchical information, signal values, and signal waveforms



System Tasks: VCD File Control

❖ System tasks are provided

- ✓ **\$dumpvars** - for selecting module instances or signals to dump
\$dumpvars(<levels>, <module1/signal1>, <module1/signal1>, ...);
- ✓ **\$dumpfile(<dumpfile_name>);** - defining a name of VCD file
- ✓ **\$dumplimit(<dumpfile_size>);** - setting maximum size of VCD file
- ✓ **\$dumpon** - starting the dump process
- ✓ **\$dumpoff** - stopping the dump process
- ✓ **\$dumpall** - generating checkpoints
- ✓ **\$dumpflush** - emptying the operating system's dump file buffer and ensuring that all the data in that buffer is stored in the dump file



System Tasks: VCD File Control Example

```
initial
begin
    $dumpfile("mydump.vcd"); // simulation info dumped to mydump.vcd file
    $dumplimit(25000000);      // max. size of the dump file will not exceed 25Mbytes
    $dumpall;                  // create a checkpoint; dump current value of all VCD variables
    $dumpvars;                 // no arguments, dump all signals in the design
    $dumpvars(1, top);         // dump all variables in module instance top.
                                // Number 1 indicates levels of hierarchy. Dump one hierarchy
                                // level below top, i.e., dump variables in top, but not signals in
                                // modules instantiated by top
    $dumpvars(2, top.m1);       // dump up to 2 levels of hierarchy below top.m1
    $dumpvars(0, top.m2);       // number "0" means dump the entire hierarchy below
    top.m2

    #50000 $dumpoff;           // stop the dump process first 50,000 time units from the start
    #50000 $dumpon; // start the dump process after 50,000 time units
    #50000 $dumpoff;           // stop the dump process after 50,000 time units
end
```

Compiler Directives

- ❖ **Compiler directives** are provided in Verilog
- ❖ **Compiler directives** cause the Verilog compiler to take special action
- ❖ All **compiler directives** are defined by using the ``<keyword>` construct
- ❖ We deal with the following most useful **compiler directives**
 - ``define`
 - ``include`
 - ``ifdef`
 - ``timescale`
 - ``resetall`

Compiler Directives: Text Substitution

- ❖ The **`define** directive is used to define text macros in Verilog
- ❖ Text substitution usage: **`define <macro_name> <macro_text>**
- ❖ The Verilog compiler substitutes the text of the macro (**<macro_text>**) wherever it encounters a **`<macro_name>**
- ❖ To remove the definition of the macro use **`undef <macro_name>**

```
// define a text macro that defines default word size; used as `WORD_SIZE in the code
```

```
`define WORD_SIZE 32
```

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

```
`define S $stop
```

```
// define a frequently used text string
```

```
`define WORD_REG reg [31:0]
```

Compiler Directives: Text Inclusion

- ❖ The **`include** allows you to include entire contents of a Verilog source file in another Verilog file during compilation
- ❖ Text substitution usage: **`include** “<included_file>”
- ❖ **`include** directive is typically used to include header files, global or commonly used definitions (text macros), and tasks (without encapsulating repeated code)

```
// include the header.v, delays.v and global.v files which contains header,  
// delays and global text macros declaration in the main verilog file design.v  
`include “header.v”  
`include “timing_settings/delays.v”  
`include “../global_parameters/global.v”  
...  
<Verilog code in file design.v>  
...
```

Compiler Directives: Conditional Compilation

- ❖ Conditional compilation can be accomplished by using compiler directives ``ifdef`, ``else` and ``endif`
- ❖ The ``ifdef` can appear anywhere in the design.
- ❖ Statements, blocks, declarations, and other compiler directives can be conditionally compiled using ``ifdef`, ``else` and ``endif` compiler directives
- ❖ The ``else` statement is optional (a maximum of one ``else` statement can accompany the ``ifdef`)
- ❖ An ``ifdef` is always closed by ``endif`
- ❖ A boolean expression is not allowed with the ``ifdef` statement

Compiler Directives: `ifdef Statement Usage

```
`ifdef TEST // compile module test only if text macro
            // TEST is defined
    module test;
        ...
        ...
    endmodule
`else // compile the module stimulus as default
    module stimulus;
        ...
        ...
    endmodule
`endif // completion of `ifdef statement
```

Compiler Directives: Time Scales

- ❖ Verilog HDL allows the reference time unit for modules to be specified with the **`timescale** compiler directives
- ❖ Time scales usage: **`timescale <time_unit> / <time_precision>**
 - ✓ The **<time_unit>** specifies the unit of measurement for times and delays
 - ✓ The **<time_precision>** specifies the precision to which the delays are rounded off during simulation
 - ✓ The **time_precision** must be at least as precise as the **time_unit**
 - ✓ Only 1, 10, and 100 are valid integers for specifying **time_unit** and **time_precision**
 - ✓ The valid unit strings are **s**(second), **ms**(milisecond), **ns**(nanosecond), **us**(microsecond), **ps**(picosecond), and **fs**(femtosecond)
 - ✓ Any combination of these is allowed

Compiler Directives: Time Scales (cont.)

- ❖ The **`timescale** compiler directive must appear outside a module boundary
- ❖ Keep precision as close in scale to the time units as is practical
 - ✓ The simulation speed is greatly reduced if there is a large difference between the **time_units** and the **time_precision**, because the **time-wheel** advances by the multiples of the precision
 - ✓ At a **`timescale 1s/1ps**, to advance 1 second, the time-wheel scans its queues 10^{12} times versus a **`timescale 1s/1ms**, where it only scans the queues 10^3 times
- ❖ The smallest precision of all the timescale directives determines the time unit of the simulation

Compiler Directives: `timescale Usage Example

```
// Define a timescale for the module dummy1
// Reference time_unit is 100 nanoseconds and precision is 1 nanosecond
`timescale 100ns / 1ns
module dummy1;
reg toggle;
initial
    toggle = 1'b0;    // Initialize toggle
// Flip the toggle register every 5 time units
// In this module 5 time units = 500 ns = 0.5us
always #5
    begin
        toggle = ~toggle;
        $display("%d ,In %m toggle = %b ", $time, toggle);
    end
endmodule
```



Compiler Directives: `timescale Usage Example

```
// Define a timescale for the module dummy2
// Reference time_unit is 1 microseconds and precision is 10
// nanoseconds
`timescale 1us / 10ns
module dummy2;
reg toggle;
initial
    toggle = 1'b0;      // Initialize toggle
// Flip the toggle register every 5 time units
// In this module 5 time units = 5us = 5000ns
always #5
    begin
        toggle = ~toggle;
        $display("%d ,In %m toggle = %b ", $time, toggle);
    end
endmodule
```



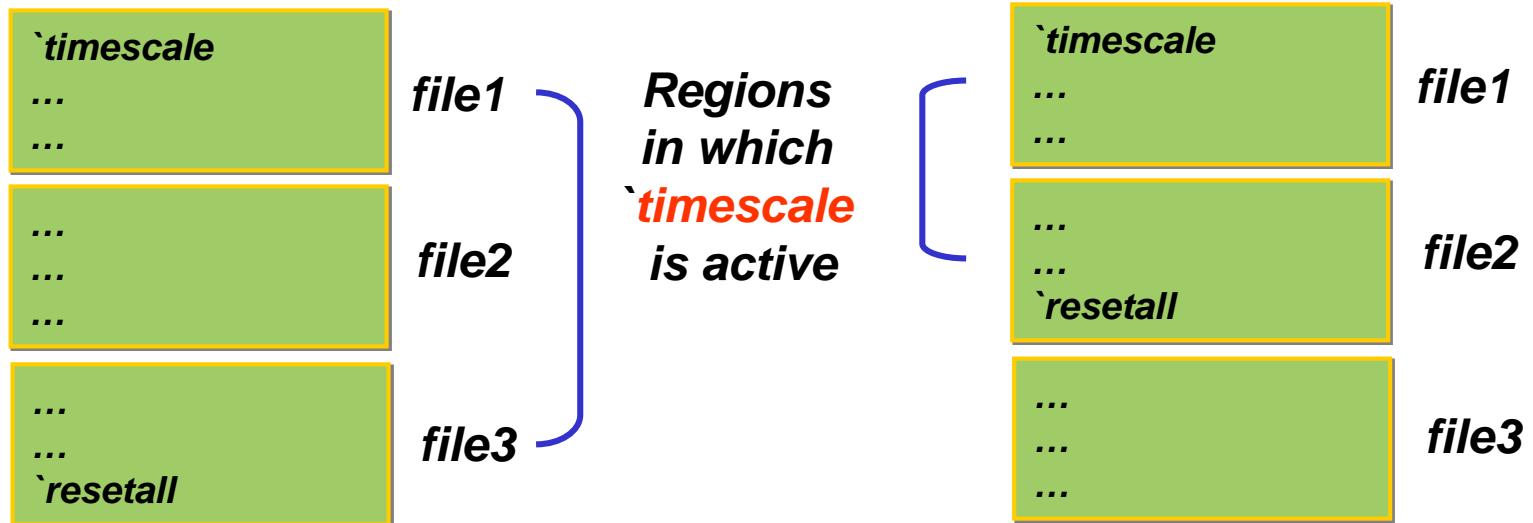
Compiler Directives: `timescale Usage Example

- ❖ The **\$display** statement in dummy2 executes once for every ten in **\$display** statements dummy1

```
5 , In dummy1 toggle = 1
10 , In dummy1 toggle = 0
15 , In dummy1 toggle = 1
20 , In dummy1 toggle = 0
25 , In dummy1 toggle = 1
30 , In dummy1 toggle = 0
35 , In dummy1 toggle = 1
40 , In dummy1 toggle = 0
45 , In dummy1 toggle = 1
→ 5 , In dummy2 toggle = 1
50 , In dummy1 toggle = 0
55 , In dummy1 toggle = 1
```

Compiler Directives: Reset Directives

- ❖ The **`resetall** compiler directive resets all the compiler directives to their default values (only if there is a default value) that are active when it encountered during compilation
 - ✓ The **`resetall** compiler directive does not affect text macros (defines).
 - ✓ To remove the definition of the macro, use **`undef** compiler directive;



Verilog Basic Concept Summary

- ❖ **Verilog** is similar in syntax to the **C programming language**. Hardware designers with previous C programming experience will find Verilog easy to learn
- ❖ Lexical conventions for **operators**, **comments**, **whitespaces**, **numbers**, **strings**, and **identifiers** were discussed .
- ❖ Various **data types** are available in Verilog. There are four **logic values**, each with different **strength levels**. Available data types include **nets**, **registers**, **vectors**, **numbers**, **simulation time**, **arrays**, **memories**, **parameters**, and **strings**. Data types represent actual hardware elements very closely
- ❖ Verilog provide useful **system tasks** to do functions like displaying, monitoring, suspending, finishing a simulation and more...
- ❖ Verilog provide useful **compiler directives**