Textbook: Verilog® HDL 2nd. Edition
Samir Palnitkar
Prentice-Hall, Inc.

E-mail: kevinsu@yuntech.edu.tw
Chapter 3
Basic Concepts
Outline of Chapter 3

3.1 Lexical Conventions
3.2 Data Types
3.3 System Tasks and Compiler Directives
3.1 Lexical Conventions
3.2 Data Types
3.3 System Tasks and Compiler Directives
3.1 Lexical Conventions

- **Lexical Conventions in Verilog HDL**

  - The Lexical Conventions in Verilog HDL are similar to those in the C Language.
  - Verilog HDL is a **case-sensitive** language.
### 3.1 Lexical Conventions

#### Whitespace

- Blank spaces (\b), tabs (\t), and newline (\n) comprise the whitespace.
- Whitespace is ignored by Verilog except when it separates tokens.
- Whitespace is not ignored in strings.
3.1 Lexical Conventions

- **Comments**
  - Comments can be inserted in the code for readability and documentation.
  - **One-line comments** start with “//”. Verilog skips from that point to the end of line.
  - **Multiple-line comments** start with “/∗” and end with “∗/”.
  - Multiple-line comments cannot be nested.
  - One-line comments can be embedded in the multiple-line comments.
3.1 Lexical Conventions

Example for the Verilog 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 */

Nested Comments
3.1 Lexical Conventions

- **Operators**

  - Three type operators: Unary / Binary : Ternary
  - Unary operators precede the operand.
  - Binary operators appear between two operands.
  - Ternary operators have two separate operators that separate three operands.
3.1 Lexical Conventions

- Example for the Operators

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. b, c, and d are operands
Number Specification

- Two types of numbers in Verilog: Sized and Unsized
Sized Numbers

- Lexical Conventions: `<size>` `<base format>` `<number>`
- `<size>` is written only in the decimal and specifies the number of bits in the number.
- **Base format**: 1. Decimal (`'d` or `'D`) 2. Hexadecimal (`'h` or `'H`) 3. Binary (`'b` or `'B`) 4. Octal (`'o` or `'O`)
- **Number**: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f
- Uppercase letter are legal for number specification.
### Example for the Sized Numbers

4'b1111 // This is a 4-bit binary number

12'habc // This is a 32-bit hexadecimal number

16'd22  // This is a 16-bit decimal number
3.1 Lexical Conventions

- **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 number of bits that is simulator- and machine-specific (must be at least 32).
Example for the Unsized Numbers

23456  // This is a 32-bit decimal number by default
hc3    // This is a 32-bit hexadecimal number
'o21   // This is a 32-bit octal number
3.1 Lexical Conventions

- **X or Z Values**

  - Unknown Values: $X$
  - High Impedance Values: $Z$
  - $X$ or $Z$ sets: 4 bits in Decimal based / 3 bits in Octal based / 1 bit in Binary based.
  - If the MSB of a number is 0, $X$, or $Z$, the number is automatically extended to fill the MSB, especially, with 0, $X$, or $Z$. 
3.1 Lexical Conventions

- Example for X or Z Values

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
3.1 Lexical Conventions

- **Negative Numbers**

  - Negative number 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>`.
  - An optional signed specifier can be added to sign arithmetic.
3.1 Lexical Conventions

Example for Negative Number

-6'd3    // 8-bit negative number stored as 2's complement of 3
-6'sd3   // Used for performing signed integer math
4'd-2    // Illegal specification
3.1 Lexical Conventions

- **Underscore Characters and Question Marks**
  - Underscore characters “_” are allowed anywhere in a number except the first character.
  - Underscore characters improve the readability of number.
  - Question marks “?” is Verilog alternative for Z in the context of numbers.
  - The “?” is to enhance readability in the casex and casez statements.
3.1 Lexical Conventions

- Example for “_” and “??”

12'b1111_0000_1010 // Use of underline characters for readability

4'b10?? // Equivalent of a 4'b10zz
3.1 Lexical Conventions

- **Strings**
  - Strings are sequences of characters that are enclosed by double quotes.
  - The restriction on a string is that it must be contained on a single line.
  - Strings are treated as a sequence of one-byte ASCII values.
3.1 Lexical Conventions

Example for Strings

“Hello Verilog World” // is a string

“a / b” // is a string
Identifiers and Keywords

- Keywords are special identifiers reserved to define the language constructs.
- Keywords are in lowercase.
- A list of all keywords is contained in Appendix C.
- Identifiers are names given to objects so that they can be referenced in the design.
- Identifiers: Alphanumeric Characters, "_", or, "$"
- Identifiers: Case sensitive
- Identifiers: Start with Alphanumeric Characters or "_"
- Identifiers: Cannot start with "$"
3.1 Lexical Conventions

- Example for Identifiers and Keywords

```plaintext
reg value; // reg is a keyword; value is an identifier
input clk: // input is a keyword; clk is an identifier
```
3.1 Lexical Conventions

- Escaped Identifiers

- Escaped Identifiers begin with the backslash (\) character and end with whitespace (space, tab, or, newline)
3.1 Lexical Conventions

- Example for Escaped Identifiers

\a+b+c

\**my_name**
3.1 Lexical Conventions
3.2 Data Types
3.3 System Tasks and Compiler Directives
### 3.2 Data Types

#### Value Set

- **Four Value levels:**

<table>
<thead>
<tr>
<th>Value Level</th>
<th>Condition in Hardware Circuits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Logic zero, false condition</td>
</tr>
<tr>
<td>1</td>
<td>Logic one, true condition</td>
</tr>
<tr>
<td>x</td>
<td>Unknown logic value</td>
</tr>
<tr>
<td>z</td>
<td>High Impedance, Floating state</td>
</tr>
</tbody>
</table>
### 3.2 Data Types

#### Value Set

- Eight Strength Levels: Reference to Appendix A resolve conflicts between drivers of different strength in digital circuits.

<table>
<thead>
<tr>
<th>Strength Level</th>
<th>Type</th>
<th>Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>supply</td>
<td>Driving</td>
<td>strongest</td>
</tr>
<tr>
<td>strong</td>
<td>Driving</td>
<td></td>
</tr>
<tr>
<td>pull</td>
<td>Driving</td>
<td></td>
</tr>
<tr>
<td>large</td>
<td>Storage</td>
<td></td>
</tr>
<tr>
<td>weak</td>
<td>Driving</td>
<td></td>
</tr>
<tr>
<td>medium</td>
<td>Storage</td>
<td></td>
</tr>
<tr>
<td>small</td>
<td>Storage</td>
<td></td>
</tr>
<tr>
<td>highz</td>
<td>High Impedance</td>
<td>weakest</td>
</tr>
</tbody>
</table>
3.2 Data Types

Example for Strength Levels
3.2 Data Types

- **Nets**
  - Nets present connections between hardware elements.
  - Nets: Continuous Driven (Assignment)
  - Nets are declared by keyword `wire`.
  - Nets get the output value of their drivers.
  - If a net has no driver, it gets the value `z`.
  - Example:
3.1 Lexical Conventions

Example for Nets

wire a;    // Declare net a for the above circuit
wire b, c; // Declare two wires b, c for the above circuit
wire d = 1'b0; // Net d is fixed to logic 0 at declaration
3.2 Data Types

More Net Types in Verilog HDL

- wand: Wire AND
- wor: Wire OR
- tri: Tri-State Wire
- triand: Tri-State Wire AND
- trior: Tri-State Wire OR
- trireg: Tri-State Wire with capacitance to store value
3.2 Data Types

- **Registers**
  - Storage data types are declared by the keyword `reg`.
  - Registers represent data store elements (Memory, ...).
  - Registers retain value until another value is placed onto them.
  - “reg” is not a real edge-triggered FF.
  - “reg” do not need a clock signal.
  - Value of `reg` can be changed anytime in a simulation by assigning a new value to the register.
  - The default value for the `reg` data type is `x`. 
Register Example: 3-1

module reg_example;

reg reset; // declare a variable reset which can hold its value

initial // this construct will be discussed later
begin
    reset = 1'b1; // initialize reset to 1 to reset the digital circuit.
    #100 reset = 1'b0; // after 100 time units reset is deasserted.
end

endmodule

Results:
3.2 Data Types

- Signed Registers
  - Registers can also be declared as signed variables.
  - Example for Signed Register Declaration:

    ```
    reg signed [63:0] m; //64 bit signed value
    integer i;          //32 bit signed value
    ```
3.2 Data Types

- **Vectors**
  - "wire" and "reg" data types can be declared as vectors (multiple bit widths).
  - If bit width is not specified, the default is scalar (1-bit).
  - The left number in the squared brackets is always the most significant bits of the vector.
3.2 Data Types

- Example for Vectors

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

MSB
3.2 Data Types

- **Vector Part Selects**

- **Example:**

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

  bus [2:0] //three least bus 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_adder [0:1] //two MSBs of vector address
Variable Vector Part Selects

- The variable vector part select of vectors is allowed in Verilog.
- The parts selects to be put in for loops to select various parts of the vector.
- Lexical Convention:
  - \([<\text{starting\_bit}>+\text{width}]\) //part select increments from starting\_bit
  - \([<\text{starting\_bit}>-\text{width}]\) //part select decrements from starting\_bit
- Starting Bit: Variable
- Bit Width: Constant
### Example for Variable Vector Part Selects

```plaintext
reg [255:0] data1:     // Little endian notation
reg [0:255] data2:     // Big endian notation
reg [7:0] byte

// Using a variable part select, one can choose parts
byte = data1[31-:8]   // starting bit=31, width=8 => data1[31:24]
byte = data2[31-:8]   // starting bit=31, width=8 => data2[24:31]

// The starting bit can also be a variable. The width has to be constant. Therefore, one can use the variable part select in a loop to select all bits of the vector:
for (j=0; j<=31, j=j+1)
    byte = data1[(j*8)+:8];  // Sequence is [7:0], [15:8], ..., [255:248]

// Can initialize a part of the vector
data1 [(byteNum*8)+:8] = 8'b0; // If byteNum=1, clear 8 bits [15:8]
```
3.2 Data Types

- Integers
  - An integer is a general purpose register data type for manipulating quantities.
  - Integers are declared by the keyword `integer`.
  - The default width is the host-machine word size, bit >32 bits.
  - “reg”: Unsigned Quantities
  - “integer”: Signed Quantities
3.2 Data Types

- **Example for Integers**

  //This file does not really do anything.
  //It only demonstrates the declaration of an integer.
  module integer_example;

  integer counter; // general purpose variable used as a counter.
  initial
      counter = -1; // A negative one is stored in the counter
endmodule
3.2 Data Types

- **Real Numbers**
  - Real number constants and real register data types are declared with the keyword `real`.
  - Real number can be specified in decimal (e.g. 3.14) or scientific (e.g. 3e6=3x10^6) notations.
  - The default value of the Real number is 0.
  - When a real value is assigned to an integer, the real number is rounded off to the nearest integer.
Example for Real Numbers

module real_example;

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)
endmodule
3.2 Data Types

- **Time**
  - Verilog simulation is done with respect to simulation time.
  - The “*time*” (keyword) data type is used in Verilog to store simulation time.
  - The width for *time* is implementation-specific but > 64 bits.
  - The system function `$time` in invoked to get the current time.
  - Simulation time is measured in terms of simulation seconds (s).
Example for Time

module time_example;

time save_sim_time; // Define a time variable save_sim_time
initial
    save_sim_time = $time; // Save the current simulation time
endmodule
### Arrays

- Array are allowed in Verilog for `reg`, `integer`, `time`, `real`, `realtime`, and `vector register` data types.
- **Multi-dimensional** arrays can also be declared with any number of dimensions.
- Arrays are accessed by `<array_name>[<subscript>]`
- For multi-dimensional arrays, indexes need to be provided for each dimension.
3.2 Data Types

- Example for Arrays

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 width
integer matrix [4:0][0:255]; //Two dimensional array of integers
reg [63:0] array_4d [15:0][7:0][7:0][255:0]; //Four dimensional array of integers
wire [7:0] w_array2 [5:0]; //Declare an array of 8-bit vector wires
wire w_array1 [7:0][5:0]; //Declare an array of single bit wires
3.2 Data Types

- **Example for Assignments to Arrays**

```c
count[5] = 0; //Reset 5th element of array of count variables
chk_point[100] = 0; //Reset 100th time check point value
port_id[3] = 0; //Reset 3rd element (a 5-bit value) of port_id array

matrix[1][0] = 3359; //Set value of element indexed by [1][0] to 4459
array_4d[0][0][0][0][15:0] = 0; //Clear bits 15:0 of the register
    //accessed by indices [0][0][0][0]

port_id = 0; //Illegal syntax – Attempt to write the entire arrays
matrix[1] = 0; //Illegal syntax – Attempt to write [1][0] … [1][255]
```
Memories

- Memories are modeled in Verilog as a one-dimension arrays of registers.
- Each word can be one or more bits.
- A particular word in memory is obtained by using the address as memory array subscript.
3.2 Data Types

Example for Memories

```vhdl
reg mem1bit [0:1023];  //Memory mem1bit with 1K 1-bit works
reg [7:0] membyte [0:1023];  //Memory membyte with 1K 8-bit words
membyte [511]  //Fetch 1 byte word whose address is 511.
```
3.2 Data Types

- **Parameters**
  - Verilog allows constants to be defined in a module by the keyword `parameter`.
  - Parameters cannot be used as variables.
  - Parameter values for each module instance can be overridden individually at compile time.
  - Parameter types and sizes can also be defined.
3.2 Data Types

Example for Parameters

parameter port_id = 5;  // Defines a constant port_id
parameter cache_line_width = 256;  // Constant defines width of cache line.
parameter signed [15:0] WIDTH;  // Fixed sign and range for parameter width
Definition Parameters and Local Parameters

- Hardcoded numbers should be avoided written the definition parameters (Keyword: `defparam`).
- Parameter value can be changed by using `defparam` statement.
- Verilog local parameters (Keyword: `localparam`) are identical to parameter except that they cannot be modified with `defparam`.
- “localparam” provides protection against inadvertent parameter redefinition.
3.2 Data Types

- Example for “localparam”

Localparam state1 = 4’h0001,
state2 = 4’h0010,
state3 = 4’h0100,
state4 = 4’h1000,

Fixed state machine
3.2 Data Types

- **Strings**
  
  - String can be stored in `reg`.
  - Each character in string takes up 8 bits (byte).
  - 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 the register width is smaller than the string width, Verilog *truncates the leftmost bits* of the string.
  - It is always safe to declare a string that is slightly wider than necessary.
module string_storage;

reg [8*18:1] string_value; //Declare a variable which is 18 bytes wide
initial
    string_value = "Hello Verilog World"; //String can be stored in variable
endmodule
## Special Characters in Strings

<table>
<thead>
<tr>
<th>Escaped Characters</th>
<th>Character Displayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>newline</td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
</tr>
<tr>
<td>%</td>
<td>%</td>
</tr>
<tr>
<td>\</td>
<td>\</td>
</tr>
<tr>
<td>&quot;</td>
<td>“</td>
</tr>
<tr>
<td>\000</td>
<td>Character written in 1-3 octal digits</td>
</tr>
</tbody>
</table>
3.3 System Tasks and Compiler Directives

3.1 Lexical Conventions
3.2 Data Types
3.3 System Tasks and Compiler Directives
3.3 System Tasks and Compiler Directives

- **System Tasks**

  - Verilog provides standard system tasks for certain routine operations.
  - System tasks keyword: `$<keyword>`
  - Operations includes: display on screen, monitoring values of nets, stopping, finishing ... etc.
Displaying Information

- "$display" is the main system task for displaying on the screen.
- Usage: $display (p1, p2, ... , pn);
- A $display inserts a newline at the end of the string by default.
### String Format Specifications

<table>
<thead>
<tr>
<th>Format</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d or %D</td>
<td>Display variable in decimal</td>
</tr>
<tr>
<td>%b or %B</td>
<td>Display variable in binary</td>
</tr>
<tr>
<td>%s or %S</td>
<td>Display string</td>
</tr>
<tr>
<td>%h or %H</td>
<td>Display variable in hex</td>
</tr>
<tr>
<td>%c or %C</td>
<td>Display ASCII character</td>
</tr>
<tr>
<td>%m or %M</td>
<td>Display hierarchical name (no argument required)</td>
</tr>
<tr>
<td>%v or %V</td>
<td>Display strength</td>
</tr>
<tr>
<td>%o or %O</td>
<td>Display variable in octal</td>
</tr>
<tr>
<td>%t or %T</td>
<td>Display in current time format</td>
</tr>
<tr>
<td>%e or %E</td>
<td>Display real number in scientific format (e.g., 3e10)</td>
</tr>
<tr>
<td>%f or %F</td>
<td>Display real number in decimal format (e.g., 2.13)</td>
</tr>
<tr>
<td>%g or %G</td>
<td>Display real number in scientific or decimal, whichever is shorter</td>
</tr>
</tbody>
</table>
module display_example;
    reg [0:40] virtual_addr;
    reg [4:0] port_id;
    reg [3:0] bus;

initial
begin
    // Display the string in quotes
    $display("Hello Verilog World");

    #200;
    // Display value of 41-bit virtual address 1fe0000001c and time 200
    virtual_addr = 41'h1fe0000001c;
    $display("At time %d virtual address is %h", $time, virtual_addr);

    #30;
    // Display value of current simulation time 230
    $display($time);

    // Display value of port_id 5 in binary
    port_id = 5;
    $display("ID of the port is %b", port_id);

    // Display x characters
    // Display value of 4-bit bus 10xx (signal contention) in binary
    bus = 4'b10xx;
    $display("Bus value is %b", bus);

    // Display the hierarchical name of the current instance
    $display("This string is displayed from %m level of hierarchy");

    // Display special characters, newline and %
    $display("This is a \n multi-line string with a %% sign");
end
endmodule
3.3 System Tasks and Compiler Directives

- Simulation Results for $display
3.3 System Tasks and Compiler Directives

- Monitoring Information

- Verilog provides a mechanism (\$monitor) to monitor a signal when its value change.
- Usage: \$monitor (p1, p2, ..., pn)
- “\$monitor” continuously monitors the value of variable or signals.
- Only once monitoring list can be active at a time.
- Switching monitoring on: \$monitoron
- Switching monitoring off: \$monitoroff
module monitor_example;

reg clock, reset;

//setup reset signal
initial
begin
    reset = 1'b1;
    clock = 1'b0;
    #50 reset = 1'b0;
end

//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);
    #400 $finish;
end

//setup clock signal
always #5 clock = ~clock;
endmodule
Simulation Results for Monitoring
Stopping and Finishing

- The task $\texttt{stop}$ is provided to stop during a simulation.
- The $\texttt{stop}$ task puts the simulation in an interactive mode.
- The designer can debug in the interactive mode.
- The $\texttt{finish}$ task terminates the simulation.
Simulation Results for $\texttt{stop}$ and $\texttt{finish}$
Compiler Directives

- Compiler directives are provided in Verilog.
- Syntax: `'<keyword>'`
- "define" directive is used to define text macros in Verilog.
- The Verilog compiler substitutes the text of the macro wherever it encounters a `<macro_name>`.
Example for `define

//Define a text macro that defines word size
//Uses a `WORD_SIZE in the code
`define WORD_SIZE 32

//Define an alias. A $stop will be substituted whenever `S appears
`define S $stop;

//Define a frequently used text string
`define WORD_REG reg [31:0]
//You can then define a 32-bit register as `WORD_REG reg32;
### Compiler Directives: ‘include

- The ‘include directive allows you to include entire contents of a Verilog source file in another Verilog file during compilation.

- The ‘include directive is typically used to include header files, which contain global or commonly used definitions.

- Example:

```verilog
//Include the file header.v, which contains declarations in the
//main Verilog file design.v
`include header.v

...<Verilog code in file design.v>
...