**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **02-06-2020** | **Name:** | **Karthik J** |
| **Course:** | **DSDV** | **USN:** | **4AL16EC030** |
| **Topic:** | Verilog HDL Basics | **Semester & Section:** | **8TH A** |
| **GitHub Repository:** | Karthik-J |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| Verilog is a HARDWARE DESCRIPTION LANGUAGE (HDL). It is a language used for describing a digital system like a network switch or a microprocessor or a memory or a flip−flop. It means, by using a HDL we can describe any digital hardware at any level. Designs, which are described in HDL are independent of technology, very easy for designing and debugging, and are normally more useful than schematics, particularly for large circuits.  Verilog supports a design at many levels of abstraction. The major three are −   * Behavioral level * Register-transfer level * Gate level  Behavioral level This level describes a system by concurrent algorithms (Behavioural). Every algorithm is sequential, which means it consists of a set of instructions that are executed one by one. Functions, tasks and blocks are the main elements. There is no regard to the structural realization of the design. Register−Transfer Level Designs using the Register−Transfer Level specify the characteristics of a circuit using operations and the transfer of data between the registers. Modern definition of an RTL code is "Any code that is synthesizable is called RTL code". Gate Level Within the logical level, the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`). The usable operations are predefined logic primitives (basic gates). Gate level modelling may not be a right idea for logic design. Gate level code is generated using tools like synthesis tools and his netlist is used for gate level simulation and for backend. Lexical Tokens Verilog language source text files are a stream of lexical tokens. A token consists of one or more characters, and each single character is in exactly one token.  The basic lexical tokens used by the Verilog HDL are similar to those in C Programming Language. Verilog is case sensitive. All the key words are in lower case. White Space White spaces can contain characters for spaces, tabs, new-lines and form feeds. These characters are ignored except when they serve to separate tokens.  White space characters are Blank space, Tabs, Carriage returns, New line, and Form feeds. Comments There are two forms to represent the comments   * 1) Single line comments begin with the token // and end with carriage return.   Ex.: //this is single line syntax   * 2) Multiline comments begins with the token /\* and end with token \*/   Ex.: /\* this is multiline Syntax\*/ Numbers You can specify a number in binary, octal, decimal or hexadecimal format. Negative numbers are represented in 2’s compliment numbers. Verilog allows integers, real numbers and signed & unsigned numbers.  The syntax is given by − <size> <radix> <value>  Size or unsized number can be defined in <Size> and <radix> defines whether it is binary, octal, hexadecimal or decimal. Identifiers Identifier is the name used to define the object, such as a function, module or register. Identifiers should begin with an alphabetical characters or underscore characters. Ex. A\_Z, a\_z,\_  Identifiers are a combination of alphabetic, numeric, underscore and $ characters. They can be up to 1024 characters long. Operators Operators are special characters used to put conditions or to operate the variables. There are one, two and sometimes three characters used to perform operations on variables.  Ex. >, +, ~, &! =. Verilog Keywords Words that have special meaning in Verilog are called the Verilog keywords. For example, assign, case, while, wire, reg, and, or, nand, and module. They should not be used as identifiers. Verilog keywords also include compiler directives, and system tasks and functions. Gate Level Modelling Verilog has built-in primitives like logic gates, transmission gates and switches. These are rarely used for design work but they are used in post synthesis world for modelling of ASIC/FPGA cells.  Gate level modelling exhibits two properties −  **Drive strength** − The strength of the output gates is defined by drive strength. The output is strongest if there is a direct connection to the source. The strength decreases if the connection is via a conducting transistor and least when connected via a pull-up/down resistive. The drive strength is usually not specified, in which case the strengths defaults to strong1 and strong0.  **Delays** − If delays are not specified, then the gates do not have propagation delays; if two delays are specified, then first one represents the rise delay and the second one, fall delay; if only one delay is specified, then both, rise and fall are equal. Delays can be ignored in synthesis. Gate Primitives The basic logic gates using one output and many inputs are used in Verilog. GATE uses one of the keywords - and, nand, or, nor, xor, xnor for use in Verilog for N number of inputs and 1 output.  Example:  Module gate()  Wire ot0;  Wire ot1;  Wire ot2;    Reg in0,in1,in2,in3;  Not U1(ot0,in0);  Xor U2(ot1,in1,in2,in3);  And U3(ot2, in2,in3,in0) Transmission Gate Primitives Transmission gate primitives include both, buffers and inverters. They have single input and one or more outputs. In the gate instantiation syntax shown below, GATE stands for either the keyword buf or NOT gate.  Example: Not, buf, bufif0, bufif1, notif0, notif1  Not – n outout inverter  Buf – n output buffer  Bufifo – tristate buffer, active low enable  Bufif1 – tristate buffer, active high enable  Notifo – tristate inverter, active low enable  Notif1 – tristate inverter, active high enable  Example:  Module gate()  Wire out0;  Wire out1;    Reg in0,in1;  Not U1(out0,in0);  Buf U2(out0,in0); Data TypesValue Set Verilog consists of, mainly, four basic values. All Verilog data types, which are used in Verilog store these values −  0 (logic zero, or false condition)  1 (logic one, or true condition)  x (unknown logic value)  z (high impedance state)  use of x and z is very limited for synthesis. Wire A wire is used to represent a physical wire in a circuit and it is used for connection of gates or modules. The value of a wire can only be read and not assigned in a function or block. A wire cannot store value but is always driven by a continuous assignment statement or by connecting wire to output of a gate/module. Other specific types of wires are −  **Wand (wired-AND)** − here value of Wand is dependent on logical AND of all the device drivers connected to it.  **Wor (wired-OR)** − here value of a Wor is dependent on logical OR of all the device drivers connected to it.  **Tri (three-state)** − here all drivers connected to a tri must be z, except only one (which determines value of tri).  Example:  Wire [msb:lsb] wire\_variable\_list;  Wirec // simple wire  Wand d;    Assign d = a; // value of d is the logical AND of  Assign d = b; // a and b  Wire [9:0] A; // a cable (vector) of 10 wires.    Wand [msb:lsb] wand\_variable\_list;  Wor [msb:lsb] wor\_variable\_list;  Tri [msb:lsb] tri\_variable\_list; Register A reg (register) is a data object, which is holding the value from one procedural assignment to next one and are used only in different functions and procedural blocks. A reg is a simple Verilog, variable-type register and can’t imply a physical register. In multi-bit registers, the data is stored in the form of unsigned numbers and sign extension is not used.  Example −  reg c; // single 1-bit register variable  reg [5:0] gem; // a 6-bit vector;  reg [6:0] d, e; // two 7-bit variables Input, Output, Inout These keywords are used to declare input, output and bidirectional ports of a task or module. Here input and inout ports, which are of wire type and output port is configured to be of wire, reg, wand, wor or tri type. Always, default is wire type.  **Example**  Module sample(a, c, b, d);  Input c; // An input where wire is used.  Output a, b; // Two outputs where wire is used.  Output [2:0] d; /\* A three-bit output. One must declare type in a separate statement. \*/  reg [1:0] a; // The above ‘a’ port is for declaration in reg. Integer Integers are used in general-purpose variables. They are used mainly in loops-indicies, constants, and parameters. They are of ‘reg’ type data type. They store data as signed numbers whereas explicitly declared reg types store them as an unsigned data. If the integer is not defined at the time of compiling, then the default size would be 32 bits.  If an integer holds a constant, the synthesizer adjusts them to the minimum width needed at the time of compilation.  **Example**  Integer c; // single 32-bit integer  Assign a = 63; // 63 defaults to a 7-bit variable. Supply0, Supply1 Supply0 define wires tied to logic 0 (ground) and supply1 define wires tied to logic 1 (power).  **Example**  supply0 logic\_0\_wires;  supply0 gnd1; // equivalent to a wire assigned as 0  supply1 logic\_1\_wires;  supply1 c, s; Time Time is a 64-bit quantity that can be used in conjunction with the $time system task to hold simulation time. Time is not supported for synthesis and hence is used only for simulation purposes.  **Example**  time time\_variable\_list;  time c;  c = $time; //c = current simulation time Parameter A parameter is defining a constant which can be set when you use a module, which allows customization of module during the instantiation process.  Example  Parameter add = 3’b010, sub = 2’b11;  Parameter n = 3;  Parameter [2:0] param2 = 3’b110;  reg [n-1:0] jam; /\* A 3-bit register with length of n or above. \*/  always @(z)  y = {{(add - sub){z}};  if (z)  begin  state = param2[1];  else  state = param2[2];  end OperatorsArithmetic Operators These operators is perform arithmetic operations. The + and −are used as either unary (x) or binary (z−y) operators.  The Operators which are included in arithmetic operation are −  + (addition), −(subtraction), \* (multiplication), / (division), % (modulus)  **Example** −  parameter v = 5;  reg[3:0] b, d, h, i, count;  h = b + d;  i = d - v;  cnt = (cnt +1)%16; //Can count 0 thru 15. Relational Operators These operators compare two operands and return the result in a single bit, 1 or 0.  Wire and reg variables are positive. Thus (−3’d001) = = 3’d111 and (−3b001)>3b110.  The Operators which are included in relational operation are −   * == (equal to) * != (not equal to) * > (greater than) * >= (greater than or equal to) * < (less than) * <= (less than or equal to)   **Example**  if (z = = y) c = 1;  else c = 0; // Compare in 2’s compliment; d>b  reg [3:0] d,b;  if (d[3]= = b[3]) d[2:0] > b[2:0];  else b[3];  Equivalent Statement  e = (z == y); Bit-wise Operators Bit-wise operators which are doing a bit-by-bit comparison between two operands.  The Operators which are included in Bit wise operation are −   * & (bitwise AND) * | (bitwiseOR) * ~ (bitwise NOT) * ^ (bitwise XOR) * ~^ or ^~(bitwise XNOR)   **Example**  module and2 (d, b, c);  input [1:0] d, b;  output [1:0] c;  assign c = d & b;  end module Logical Operators Logical operators are bit-wise operators and are used only for single-bit operands. They return a single bit value, 0 or 1. They can work on integers or group of bits, expressions and treat all non-zero values as 1. Logical operators are generally, used in conditional statements since they work with expressions.  The operators which are included in Logical operation are −   * ! (logical NOT) * && (logical AND) * || (logical OR)   **Example**  wire[7:0] a, b, c; // a, b and c are multibit variables.  reg x;  if ((a == b) && (c)) x = 1; //x = 1 if a equals b, and c is nonzero.  else x = !a; // x =0 if a is anything but zero. Reduction Operators Reduction operators are the unary form of the bitwise operators and operate on all the bits of an operand vector. These also return a single-bit value.  The operators which are included in Reduction operation are −   * & (reduction AND) * | (reduction OR) * ~& (reduction NAND) * ~| (reduction NOR) * ^ (reduction XOR) * ~^ or ^~(reduction XNOR)   **Example**  Module chk\_zero (x, z);  Input [2:0] x;  Output z;  Assign z = & x; // Reduction AND  End module Shift Operators Shift operators, which are shifting the first operand by the number of bits specified by second operand in the syntax. Vacant positions are filled with zeros for both directions, left and right shifts (There is no use sign extension).  The Operators which are included in Shift operation are −   * << (shift left) * >> (shift right)   **Example**  Assign z = c << 3; /\* z = c shifted left 3 bits;  Vacant positions are filled with 0’s \*/ Concatenation Operator The concatenation operator combines two or more operands to form a larger vector.  The operator included in Concatenation operation is − { }(concatenation)  **Example**  wire [1:0] a, h; wire [2:0] x; wire [3;0] y, Z;  assign x = {1’b0, a}; // x[2] = 0, x[1] = a[1], x[0] = a[0]  assign b = {a, h}; /\* b[3] = a[1], b[2] = a[0], b[1] = h[1],  b[0] = h[0] \*/  assign {cout, b} = x + Z; // Concatenation of a result Replication Operator The replication operator are making multiple copies of an item.  The operator used in Replication operation is − {n{item}} (n fold replication of an item)  **Example**  Wire [1:0] a, f; wire [4:0] x;  Assign x = {2{1’f0}, a}; // Equivalent to x = {0,0,a }  Assign y = {2{a}, 3{f}}; //Equivalent to y = {a,a,f,f}  For synthesis, Synopsis did not like a zero replication.  For example:-  Parameter l = 5, k = 5;  Assign x = {(l-k){a}} Conditional Operator Conditional operator synthesizes to a multiplexer. It is the same kind as is used in C/C++ and evaluates one of the two expressions based on the condition.  The operator used in Conditional operation is −  (Condition) ? (Result if condition true) −  (result if condition false)  **Example**  Assign x = (g) ? a : b;  Assign x = (inc = = 2) ? x+1 : x-1;  /\* if (inc), x = x+1, else x = x-1 \*/ OperandsLiterals Literals are constant-valued operands that are used in Verilog expressions. The two commonly used Verilog literals are −   * **String** − A string literal operand is a one-dimensional array of characters, which are enclosed in double quotes (" "). * **Numeric** − A constant number operand is specified in binary, octal, decimal or hexadecimal Number.   **Example**  n − integer representing number of bits  F − one of four possible base formats −  b for binary, o for octal, d for decimal, h for hexadecimal.  “time is” // string literal  267 // 32-bit decimal number  2’b01 // 2-bit binary  20’hB36F // 20-bit hexadecimal number  ‘062 // 32-bit octal number Wires, Regs, and Parameters Wires, regs and parameters are the data types used as operands in Verilog expressions. Bit-Selection “x[2]” and Part-Selection “x[4:2]” Bit-selects and part-selects are used to select one bit and a multiple bits, respectively, from a wire, reg or parameter vector with the use of square brackets “[ ]”. Bit-selects and part-selects are also used as operands in expressions in the same way that their main data objects are used.  **Example**  reg [7:0] x, y;  reg [3:0] z;  reg a;  a = x[7] & y[7]; // bit-selects  z = x[7:4] + y[3:0]; // part-selects Function Calls In the Function calls, the return value of a function is used directly in an expression without the need of first assigning it to a register or wire. It just place the function call as one of the type of operands.it is needful to make sure you are knowing the bit width of the return value of function call.  Example  Assign x = y & z & chk\_yz(z, y); // chk\_yz is a function  . . ./\* Definition of the function \*/  Function chk\_yz; // function definition  Input z,y;  chk\_yz = y^z;  End function ModulesModule Declaration In Verilog, A module is the principal design entity. This indicates the name and port list (arguments). The next few lines which specifies the input/output type (input, output or inout) and width of the each port. The default port width is only 1 bit. The port variables must be declared by wire, wand,. . ., reg. The default port variable is wire. Normally, inputs are wire because their data is latched outside the module. Outputs are of reg type if their signals are stored inside.  **Example**  module sub\_add(add, in1, in2, out);  input add; // defaults to wire  input [7:0] in1, in2; wire in1, in2;  output [7:0] out; reg out;  ... statements ...  End module Continuous Assignment The continuous assignment in a Module is used for assigning a value on to a wire, which is the normal assignment used at outside of always or initial blocks. This assignment is done with an explicit assign statement or to assign a value to a wire during its declaration. Continuous assignment are continuously executed at the time of simulation. The order of assign statements does not affect it. If you do any change in any of the right-hand-side inputs signal it will change a left-hand-side output signal.  **Example**  Wire [1:0] x = 2’y01; // assigned on declaration  Assign y = c | d; // using assign statement  Assign d = a & b;  /\* the order of the assign statements does not matter. \*/ Module Instantiations Module declarations are templates for creating actual objects. Modules are instantiated inside other modules, and each instantiation is creating a single object from that template. The exception is the top-level module which is its own instantiation. The module’s ports must to be matched to those which are defined in the template. It is specified −   * **By name**, using a dot “.template port name (name of wire connected to port)”. Or * **By position**, placing the ports in the same place in the port lists of both of the template and the instance.   **Example**  MODULE DEFINITION  Module and4 (x, y, z);  Input [3:0] x, y;  Output [3:0] z;  Assign z = x | y;  End module |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Date:** | | **02-06-2020** | **Name:** | **Karthik J** |  |
| **Course:** | | [Programming with Python: Hands-on Introduction for Beginners](https://www.udemy.com/course/python-programming-beginners/) | **USN:** | **4AL16EC030** |  |
| **Topic:** | |  | **Semester & Section:** | **8th A** |  |
|  | **AFTERNOON SESSION DETAILS** | | | | |
|  | **Image of session** | | | | |
|  | Dictionary A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values. Example Create and print a dictionary:  thisdict = {   "brand": "Ford",   "model": "Mustang",   "year": 1964 } print(thisdict) Python - Tuples A tuple is an immutable sequence of Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.  Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also.  For example −  tup1 = ('physics', 'chemistry', 1997, 2000);  tup2 = (1, 2, 3, 4, 5 );  tup3 = "a", "b", "c", "d"; Accessing Values in Tuples To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index.  For example −  tup1 = ('physics', 'chemistry', 1997, 2000);  tup2 = (1, 2, 3, 4, 5, 6, 7 );  print "tup1[0]: ", tup1[0];  print "tup2[1:5]: ", tup2[1:5]; Updating Tuples Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples  Example  tup1 = (12, 34.56);  tup2 = ('abc', 'xyz');  # Following action is not valid for tuples  # tup1[0] = 100;  # So, let's create a new tuple as follows  tup3 = tup1 + tup2;  print tup3; Delete Tuple Elements Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.  To explicitly remove an entire tuple, just use the **del** statement.  example −  tup = ('physics', 'chemistry', 1997, 2000);  print tup;  del tup;  print "After deleting tup : ";  print tup;  This produces the following result. Note an exception raised, this is because after **del tup** tuple does not exist any-more − Basic Tuples Operations Tuples respond to the + and \* operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string. Indexing, Slicing, and Matrixes Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input −  L = ('spam', 'Spam', 'SPAM!') | | | | |