Assignment - 2
Verilog Modelling Techniques

Department of Electronics and Communication Engineering
Indian Institute of Technology, Roorkee

ECN 104 Digital Logic Design

Contents

1 Introduction

Verilog supports a wide variety of modelling techniques. Different modelling techniques allow writing hardware description at different levels of abstraction, starting from the switch level modelling (PMOS/NMOS) and all the way up to behavioural modelling (algorithmic description). Each one of them has their own benefits and use cases. In this assignment we will discuss three of them, namely Gate-level modelling, structural modelling and behavioural modelling.

2 Gate-level modelling

Gate-level modelling in Verilog is used to describe a circuit only using logic gates. This approach is used to describe critical parts of a design, like adders and multipliers. Using gate-level implementation allows greater control over the design than other techniques. Gate-level modelling is only used for small scale design. Due to its complexity other modelling techniques are commonly used to abstract gate-level implementations.

2.1 Gate Primitives in Verilog

Verilog support following gates:

- AND
- NAND
- OR
- NOR
- XOR
- XNOR
- NOT

Gates in Verilog are available as primitives and can be instantiated similar to modules. Listing 1 is an example gate level circuit.

Listing 1: Example module using Gate-level modelling

```
/* Example using gate-level modelling, gate is a built-in
 * primitive in Verilog. Gates are instantiated in a way
 * similar to modules. Gates can be of single input or
 * multiple input
 */
module gateLevelExample(input1, input2, input3, result);
   input input1, input2, input3;
   output [8:0] result;

   // Single input gates
   not  g0(result[0], input1);           // = ~input1

   // Two input gates
   and  g1(result[1], input1, input2);   // =   input1 & input2
   nand g2(result[2], input1, input2);   // = ~(input1 & input2)
   or   g3(result[3], input1, input2);   // =   input1 | input2
   nor  g4(result[4], input1, input2);   // = ~(input1 | input2 )
   xor  g5(result[5], input1, input2);   // =   input1 ^ input2
   xnor g6(result[6], input1, input2);   // = ~(input1 ^ input2)

   // Gates with more than two input
   xnor g7(result[7], input1, input2, input3);
   and  g8(result[8], input1, input2, input3, input1);
endmodule // gateLevelExample
```

Verilog also supports instantiating gates without a instance name demonstrated in Listing 2.

Listing 2: Instantiating unnamed gates

```
/*
 * Gates in Verilog can be instantiated without a name,
 * such instantiation in Verilog is legal.
 */
module unnamedGate(input1, input2, result);
   input input1, input2;
   output result;

   and(result, input1, input2); // Unnamed gate
endmodule // unnamedGate
```

2.2 Delay specification

All the circuits we have studied so far have no delay associated with them, these are called 0-delay circuits. Real circuits however, always have a delay between their input and output. Verilog allows modelling of delays at various level of abstraction using delay statements.

Syntax for specifying a delay is:

```
<gate_primitive> #(<delay>) <inst_name>(...ports...)
```

For example:

```
/* 2-input AND gate with 1 time unit delay */
and #(1) a1(result, input1, input2);
```

To specify unit of delay, we'll write a compiler directive. This is typically written at the begining of the description.

```
`timescale 1ns/1ps
```

Here, `1ns` is the unit time while `1ps` is the time resolution which will be explained in the assignment. A complete example using delay statements is given in Listing 3.

Listing 3: Example usage of delays statement to specify propagation delay of logic gates.

```
1  /* Compiler directive to specify time unit, which will be
2   * used for assigning propogation delays of logic gates.
3   */
4  `timescale 1ns/1ps
5
6  module delayExample(input1, input2, result);
7     input input1, input2;
8     output result;
9
10    // Specify a NAND gate having propogation delay of 2ns
11    nand #(2) nd1(result, input1, input2);
12 endmodule // delayExample
```

Info 1 - Why do gates have delay?

Each gate has some stray capacitance and the interconnect is resistive, these work together as an RC circuit. When the value on the interconnect changes, the capacitor has to either charge or discharge, and since the circuit is an RC circuit - it will take some finite amount of time for this change to happen and hence the delay.

Problem 1

[Weightage: 20%]

NOT gate is one of the primitives available with Verilog. NOT gate is a type of buffer and is also known as inverting buffer. Buffers in digital circuit are often used to isolate two parts of a circuit, but this isolation can also be added with a certain behaviour like inversion of signal or delaying signal to synchronize with other parts.

1. Write a Verilog module which acts as an inverting buffer (that is, a NOT gate), you have to use verilog primitive `not` for the implementation.
2. Another kind of buffer often used in digital circuit is the 'non-inverting buffer'. As the name suggests, the non-inverting buffer doesn't invert the input unlike the inverting buffer. One of the common use case of this buffer is to translate logic levels between two circuits. Level translation is used whenever two parts of a circuit are powered with different Vdd (different supply voltages). Now that you know what a buffer is, write a Verilog module to implement it using the inverting buffers you designed in Problem 1.1.
3. NOT gates are also used in ring oscillators, which is a kind of an oscillator, often used to generate clock signals. Ring oscillator is an oscillator which looks like a ring, and contains a chain of odd number of NOT gates. Ring oscillator's frequency depends on the delay of the NOT gate and number of gates used. A simple 3 NOT gate ring oscillator is shown in Fig. 1. Let the propagation delay of each of the gate in Fig. 1 be t_d, intuitively one can say that the delay between change of values at any point is $3t_d$. Use this information and calculate the time period of an n NOT gate ring oscillator, where n is an odd number.
4. Using the expression from the Problem 1.3, write the description of a module for a 5 NOT gate ring oscillator. You have to design the ring oscillator such that the time period of the generated wave will be 30ns, assume that each of the NOT gate has a propagation delay of 5ns.

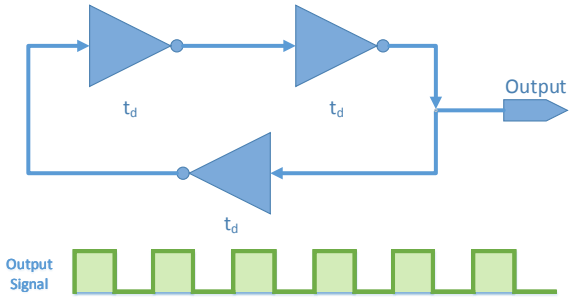

Figure 1: Simulation result for Listing 1, output changes 15ns after the change in input.

5. Will a single not gate ring oscillator work? Why or why not?

Hint - 1

Use behavioural modelling for Problem 1.4, this would allow you to assign initial state of the output and thus preventing the gates from entering state 'X'.

---

3 Data-flow Modelling

Data flow modelling is a higher level of abstraction than the gate level modelling we just studied. Describing a circuit using data-flow modelling does not require knowledge of gates level circuit, thus it is easier than gate-level modelling when description of large scale circuits are written. All the examples from Assignment 1 uses data flow modelling.

3.1 Continuous Assignment

Continuous assignment in Verilog is used for data flow modelling. These assignment starts with `assign` keyword. Continuous assignment drives value into a net (`wire`). Following example describes use of continuous assignment:

Common Pitfall - 1

Since Verilog is concurrent language unlike programming languages such as C,C++ or Java, all the continuous assignments are evaluated at the same time.

Listing 4: Example usage of continuous assignment.

```
1  /*
2   * Following module uses continuous assignement to model
3   * an AND gate
4   */
5  module continuousAssignment(input1, input2, result);
6     input input1, input2;
7     output result;
8
9     /* Use continuous assignment to set result
10     * This is equalent to:
11     *     and (result, input1, input2);
12     */
13    assign result = input1 & input2;
14 endmodule // continuousAssignment
```

Continuous assigment in Verilog can also be done implicitly, which is assigning value on declaration of a net (`wire`). Implicit delcaration of Verilog is:

```
wire new_wire = input1 & input2;
```

3.2 Assignment Delays

Similar to gate-level modelling, Verilog allows specifying delays in assignment to model real circuits. Assignment delay specify the delay between the change of LHS and RHS of a continuous assignment. Listing 5 shows example usage of assignment delay while Fig. 2 shows simulation result of Listing 5.

Listing 5: Using assignment delay in Verilog.

```
1  `timescale 1ns/1ps
2
3  module assignmentDelay(input1, input2, result);
4     input input1, input2;
5     output result;
6
7     // Defines a circuit which will have output value
8     // input1 | input2 15ns after changing input
9     assign #15 result = input1 | input2;
10 endmodule // assignmentDelay
```

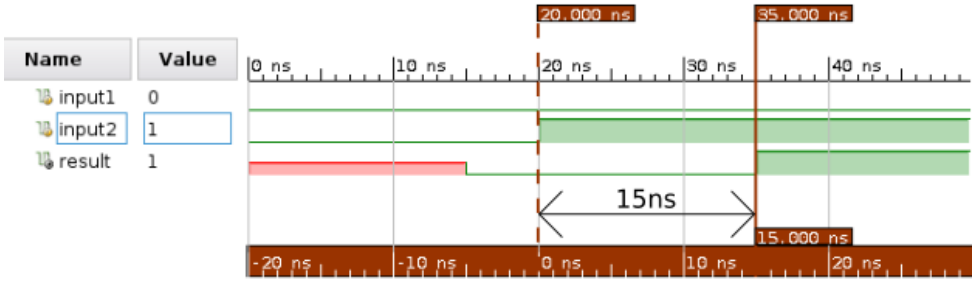

Figure 2: Simulation result for Listing 5, output changes 15ns after the change in input.

Problem 2

[Weightage: 10%]

Multiplexer is a digital element which is used to select a single signal from a group of signals. Block diagram of a multiplexer is shown in Fig. 3, the input in_1 and in_2 are multiplexed and the output is decided using the input c. If the input c is 0 then output $= in_1$ else output $= in_2$.

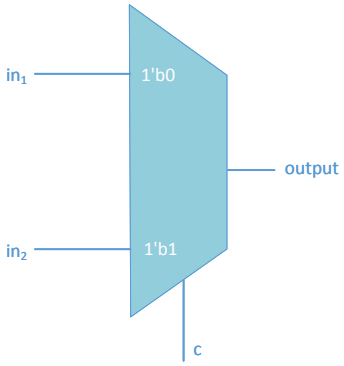

Figure 3

1. Write the hardware description of a 2-to-1 multiplexer using Verilog. You have to implement this using only data-flow modelling.
2. Now suppose you want to model a multiplexer you just purchased from the market, which has a propagation delay of 2ns. Modify the module from part 1 such that the behaviour of your description matches the one you bought.

Table 1: Module attributes for problem 2

Atrribute	Name	Size (in bits)
Module Name	multiplexer	_
First Input	in1	1
Second Input	in2	1
Result	result	1
Select Signal	select	1

Hint - 2

Verilog also supports the ternary operator, which has same behaviour as they have in programming languages.

---

4 Behavioral Modelling

Behavioral modelling is an even higher level of modelling where circuit description is written as its behaviour, this algorithmic representation of a circuit abstracts the details of gate-level and data flow modelling. Behavioral modelling resembles more to programming languages such as C than it does to circuit description.

4.1 `reg` Element

`reg` element is used to represent abstract storage device in Verilog. `reg`s can be used to store information (single bit or of arbitrary length using array, arrays are similar to vectors). We'll get back to usage of `reg` and will differentiate it with `wire` after studying procedural blocks.

4.2 Procedural Blocks

Earlier we have read about continuous assignment which allows us to drive value to a net whenever the driver changes. This kind of assignment allows description of only combinational circuit. To describe sequential circuits Verilog provides procedural blocks. These blocks are used to drive values to variables (`reg`) only if the condition is met.

4.3 `initial` Block

`initial` blocks in Verilog are used to specify initial values of all the storage elements. When simulation starts simulator doesn't know what values are to be assigned to storage elements. Initial block begins with `initial begin` and ends with an `end`. Initial blocks gets executed only once when the simulation is started.

Common Pitfall - 2

Only a `reg` can be assigned values inside an `initial` block. This is because unlike `wire` which are used for connection, `reg` stores information and this information is unknown to the simulator at t=0. Structure of an initial block is given in Listing 6.

Listing 6: Structure of an initial block.

```
...
  initial begin
      ...
      // Specify initial condition
      ...
  end
...
```

4.4 `always@` block

`always@` block in Verilog are used for describing an event which should happen only under certain conditions, such as change in value of one of the elements.

Basic structure of an `always@` block is given in Listing 7.

Listing 7: Structure of an always@ block.

```
...
  always @( ...condition... ) begin
      ...
      // Description of conditional event
      ...
  end
...
```

Listing 8 shows a module which changes output only on the positive edge of input `clk`.

Listing 8: Synchronous logic which changes value of result only at the positive edge of clk.

```
/*
 Synchronous module which changes output only on positive
 edge of clk.
 */
module alwaysExample(clk, input1, input2, result);
   input clk, input1, input2;
   output result;

   // Declare output as reg, this will be explained later
   reg result;

   // Here condition is *positive edge of clk*, this implies
   // following event only takes place when a positive edge
   // of clk arrives
   always @(posedge clk) begin
       // Assign value to reg result using procedural assignment
       result = input1 & input2;
   end
endmodule // alwaysExample
```

4.5 Blocking and Non-Blocking Assignment

4.5.1 Blocking Assignments

Blocking assingments are assignments which block the simulation while their value is being calculated. This means that the execution flow will stop at this statement until it is executed. Blocking assignment in Verilog is done using `=`. Listing 9 demonstrates the functioning of blocking assignment.

Listing 9: Swapping bytes using blocking assignment.

```
// Following example is for swapping bytes
// input is of size 16 bits (2 bytes)
// This example does not work due to the use of blocking
// assignment.

// This example is taken from:
// http://www.sutherland-hdl.com/papers/1996-CUG-presentation_nonblocking_assigns.pdf

module swapBytes;
   ...
   reg [15:0] temp;
   ...

```

```
14      // Can you guess the reason why this block won't swap
15      // bytes?
16      always @(...some_condition...) begin
17          temp[15:8] = temp[7:0];
18          temp[7:0] = temp[15:8];
19      end
20  endmodule // swapBytes
```

In the above example (Listing 9) statement 18 doesn't gets executed until statement 17 is executed, this is due to the use of blocking assignment.

4.5.2 Non-blocking Assignments

In Listing 9 we saw that blocking assignments cannot be used to swap bytes, this is where non-blocking assignments will come to use. Non-blocking assignments are evaluated in two steps first all the RHS values are calculated at the begining of the procedural block, and then the value is assigned to LHS when the execution reaches particular statement.

Listing 10 shows how non-blocking assignments can be used for swapping bytes.

Listing 10: Swapping bytes using non-blocking assignment it works!

```
1   // Following example is for swapping bytes
2   // using non-blocking assigment, this module
3   // unlike last example uses non-blocking assignment
4   // this allows swapping bytes correctly.
5
6   // This example is taken from:
7   // http://www.sutherland-hdl.com/papers/1996-CUG-presentation_nonblocking_assigns.pdf
8
9   module swapBytes;
10      ...
11      reg [15:0] temp;
12      ...
13
14      // This one works!
15      always @(...some_condition...) begin
16          temp[15:8] <= temp[7:0];
17          temp[7:0] <= temp[15:8];
18      end
19  endmodule // swapBytes
```

Common Pitfall - 3

It is a good HDL practice to not mix different assigments in a single procedural block, doing so will make your code difficult to read and a potential source of bugs.

5 Differences between `wire` and `reg` and where to use what

5.1 Legal use of `wire`

Wires in Verilog are used to connect two elements. They can be assigned a value or a value can be read from them. They, however, cannot store this value, to read something off of a wire you'll have to drive them either with constant, other wires or regs.

- `wire`s are allowed only in continuous assignments (page 4).
- A `wire` cannot be assigned a value inside a procedural block.
- A `wire` can be used to assign a value to a `reg` or a `wire`.
- `wire`s can be used for I/O connections of a module instance.

5.2 Legal use of `reg`

`reg`s in Verilog are storage elements. They, however, do not represent physical registers. Once synthesized they can be represented by a physical register, RAM or ROM.

- `reg`s cannot be assigned a value using continuous assigment.
- A `reg` can only be assigned a value in a procedural block.
- A `reg` can be used to assign a value to a `reg` or a `wire`.

5.3 Places where both `wire` and `reg` are allowed

- Both can appear on the right hand side of an `assign` statement or can be used inside a procedural block (`initial`/`always`) block to set value of a `reg`.
- Can be used as inputs to a module.

Problem 3

[Weightage: 35%]

A Turing machine is a theoretical machine which is used as an abstraction to real computational machines, Turing machine was proposed by Alan Turing in 1936. The basis of invention of turing machine was Alan's interest in the philosophy of computation. An interesting thing about the Turing machine is that any algorithm which is impossible to implement on Turing machine has never worked on any other real machine. Programming languages such as Java, C, C++ etc. are said to be turing complete This implies that anything that can be computed by a turing machine can be computed by these languages.

Basics of Turing machine

Turing machine consist of an infinitely long tape which has certain shapes (or characters) on it, this tape is traversed by a head which reads the character directly under it and takes appropriate movement decision. An example of such machine is shown in Fig. 4.

The video at `https://www.youtube.com/watch?v=mPec64RUCsk` (9 mins : Lecture 12 - Turing Machines (Part 1/10)) will help you understand the Turing machine a bit better.

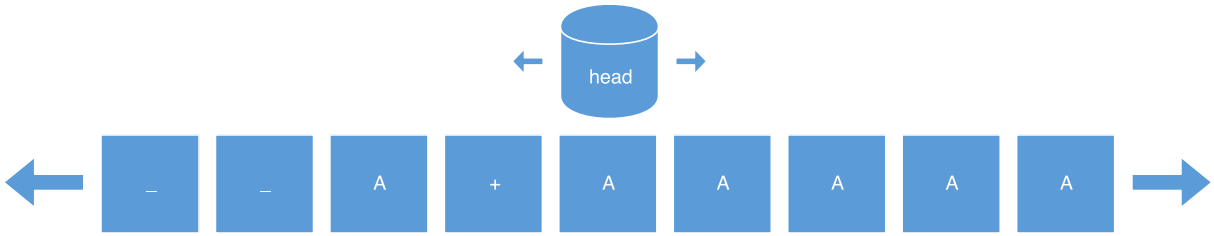

Figure 4: Example of a Turing machine.

Turing machines can be used to implement Finite state machines. To use a Turing machine as a finite state machine we need following assumptions:

1. The tape is finite in length (since it would be impossible to write description of an infinite tape).
2. Each character on the tape can only be 'A', '+' or '_' where 'A' and '+' are used for decisions while '_' represents an unused tape slot.
3. The input to the machine will be the initial configuration of the tape while the output of the machine will be the state of the tape when the machine halts.
4. At every clock cycle the Turing machine can execute only one instruction.
5. Here each possible arrangement of characters on the tape represents a single state in the FSM.

Now consider the above mentioned Turing machine rules for a machine which adds up two numbers. These numbers are represented in base 1 (e.g. 1111_1 to represent 4_{10}):

Using this rule set addition of two numbers can be performed on a Turing machine, say for example you want to add 2_{10} (AA_1) and 4_{10} ($AAAA_1$) the trace of the Turing machine to perform such addition would be (Assuming the halting condition to be a state where head returns to the position one left to the first number):

Value under head	Direction of movement	Write value
A + _	Right Right Left	A

1. __[A]A+AAAA___
2. __A[A]+AAAA___
3. __AA[+]AAAA___
4. __AAA[A]AAA___
5. __AAAA[A]AA___
6. __AAAAA[A]A___
7. __AAAAAA[A]___
8. __AAAAA[A]____
9. __AAAA[A]A____
10. __AAA[A]AA____
11. __AA[A]AAA____
12. __A[A]AAAA____
13. __[A]AAAAA____
14. _[_]AAAAAA____ (**HALT**)

Here [.] represents the head of the machine.

Questions

1. Briefly describe the algorithm which would give the above trace (you can use psuedo code for your description of the algorithm).
2. Write Verilog description for the Turing machine described above, use 2 bit registers to represent a single location on the tape. The tape of the machine will be finite in length and should be enough to store the above example. Your machine should be able to calculate this sum and return the result as 32 bit vector. When this result is generated the machine should also set the 'done' output high indicating that the computation is completed. Please use names listed in Table 2. To get started you can use Hint 3 and code provided in Listing 11.
3. Simulate the addition following two numbers using the turing machine you designed:
    1. Your enrollment number's numerically largest digit.
    2. Your enrollment number's numerically largest digit + your enrollment number's least significant digit.

    For an example:

    If your enrollment number is 16116069, your **Number 1** would be the 9 since the largest digit would be 9 and **Number 2** would be 18 since $9 + 9 = 18$.

    NOTE: If you fail to follow this convention your question will not evaluated.

If you want to read more interesting stuff on Turing Machines, you can look up for 'The halting problem' on the internet.

Hint - 3

1. You can use a 2 bit wide vector to store the characters.
2. Sample code to get you started with the problem is provided in Listing 11
3. The sample code also includes a few lines of code to automatically print your tape's states on every negative edge of the clock. To enable this you'll have to name your tape 'tape', head's current position 'current_pos'. This is already done for you in the sample code.
4. Please note that you should use the states already defined in Listing 11 to allow the already included helper code to run.
5. To calculate the 'result' you can count how many 'A's the machine's traverses when it moves back from left to right beforing halting.

Table 2: Module attributes for problem 3

Atrribute	Name	Size (in bits)
Module Name	turing	_
Clock Input	clk	1
Is Halted?	done	1
Final count	result	32

Listing 11: Code to get you started with problem 4.

```
1  `timescale 1ns/1ps
2
3  /* Sample code for turing module, **DO NOT** change the module name or
4  /* the I/O declaration.*/
5  module turing(clk, result, done);
6     input wire clk;
7     output reg [32-1:0] result;
8     output reg          done;
9
10    /* Define few constants to represent the states of each tape
11    /* location.*/
12
13     `define SYMB_A      2'b00
14     `define SYMB_ADD    2'b01
15     `define SYMB_BLANK  2'b10
16
17    /*******************************************************/
18    /* All changes to this code should be below this line */
19    /*******************************************************/
20
21    /* Declare a tape and other required variables/nets here, please
22    /* note that since we have declared the symbols to be 2 bit wide,
23    /* your tape should also have 2bit wide locations. To assign a
24    /* symbol to any location use the following syntax:
25    /*
26    /*     your_tape_name[your_counter] = `SYMB_A
27    /*
28    /* To assign the location at your_counter the symbol 'A'*/
29
30    initial begin
31       /* Initialize the tape and other global variables */
32    end
33
34    always @(posedge clk) begin
```

```
35        /* Describe your turing machines here */
36    end
37
38 endmodule // turing
```

Problem 4

[Weightage: 35%]

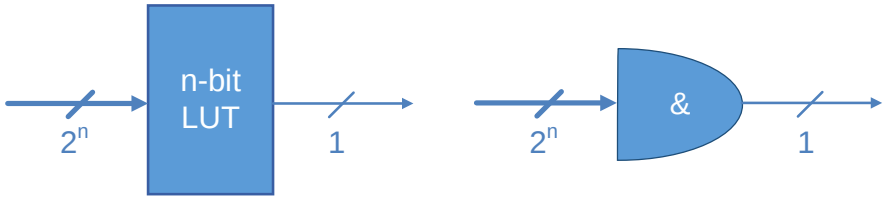

Figure 5: Example of an n-bit LUT which represents an equivalent 2^n input AND gate.

A lookup table or LUT in short is a digital structure which resembles a RAM, it takes an address of the location where to lookup an input and produces an output stored at that location. An LUT can be used to mimic any boolean function, an n-bit LUT table is one which can implement any n-bit boolean function. For example, a 2 bit LUT will have 4 inputs and a single bit output. Visual representation of an n-bit LUT is shown in Figure 5. Using this knowledge answer the following questions:

1. Design and test a 4 input AND gate using a 2 bit LUT. You should use names from Table 3. Take hint from the AND gate's truth table.
2. Use the LUT that you designed in Problem 4.1 to implement and test a 4 input AND gate. You should use names from Table 4.
3. Design appropriate modules and implement and test the following logic equation:

$$\text{result} = [(a \wedge b \wedge c \wedge d) \vee (c \oplus a)] \odot (d) \tag{1}$$

   For symbol reference, visit: https://en.wikipedia.org/wiki/Logic_gate#Symbols . You should use names from Table 5.

Table 3: Module attributes for problem 4.1

Atrribute	Name	Size (in bits)
Module Name	and2_lut	_
Address input	address	2
Lookup Result	result	1
Testbench Name	and2_lut_tb	1

Table 4: Module attributes for problem 4.2

Atrribute	Name	Size (in bits)
Module Name	and4_lut	_
Address input	address	4
Lookup Result	result	1
Testbench Name	and4_lut_tb	1

Table 5: Module attributes for problem 4.3

Atrribute	Name	Size (in bits)
Module Name	equation_lut	_
Input a	a	1
Input b	b	1
Input c	c	1
Input d	d	1
Result	result	1
Testbench Name	equation_lut_tb	_

Submission Instructions

The submissions of solutions to the assignment 2 should strictly follow the following guidelines. Please read them carefully to allow proper evaluation of the assignment.

- **Full screen - Screen shots** of Elaborated design, Source code and simulation window. Please note that only full screen screenshots will be considered for the evaluation. You are required to provide these screen shots in a PDF file with file's name being your enrollment number. `.doc`, `.docx`, `.odt` or any other document format will not be accepted as a substitute for the PDF file.
- **Source code of each problem**: You are also required to submit the source code file for each of the problem, if your solution to a single problem has more than one source code files, then submit each one of them.
- **Single zip file submission**: Please note that the submission should have only a single zip file. This zip file would contain all the other files that you want to submit.

  `.rar`, `7z`, `tar.gz` or any other archiving format will not be accepted and can lead to disqualification of your submission.

For example: If your enrollment number is 1234 and you have created two files for solving Problem 1, while rest of the problems have single file each:

- `module_1_1.v` - Which contains your main module, namely `module_1_1`.
- `andGate.v` - Which contains description of module andGate and is used by module `module_1_1.v`
- `module_1_2.v`
- `module_1_3.v`
- `module_1_4.v`

Then your submission should be a **zip** having the following structure:

1. `1234.zip`
   (a) `1234.pdf` - containing all the screen shots.
   (b) `module_1_1.v`
   (c) `andGate.v`
   (d) `module_1_2.v`
   (e) `module_1_3.v`
   (f) `module_1_4.v`

Please note that only a single zip file will be accepted for the submission. Using any other file format will lead to a penalty. All your files, source codes and the pdf should be within the zip file for the submission to be considered complete.

Document History

1. **Sat Feb 9 14:25:45 IST 2019**:
   (a) Add expected module names to each problem
   (b) Fix typos
2. **Mon Nov 29 15:57:13 IST 2017**: First Edition

Additional resources

- Stack Exchange: How are Verilog always statements implemented in hardware?
  ▷ http://bit.ly/verilogAlwaysStatement
- Stack Overflow: Difference between behavioral and dataflow in verilog
  ▷ http://bit.ly/modellingDiff
- Stack Overflow: How can I know if my code is synthesizable? [Verilog]
  ▷ http://bit.ly/synthCode

References

- Understanding Verilog Blocking and Non-blocking Assignments
  ▷ http://www.sutherland-hdl.com/papers/1996-CUG-presentation_nonblocking_assigns.pdf
- Verilog: always @ Blocks
  ▷ http://inst.eecs.berkeley.edu/~cs150/fa08/Documents/Always.pdf