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

1.1 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 implementation.

1.1.1 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
```

1.1.2 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 time unit while 1ps is the time resolution. Which will be explained in later 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.

```verilog
/* Compiler directive to specify time unit, which will be
 * used for assigning propogation delays of logic gates.
 */
`timescale 1ns/1ps

module delayExample(input1, input2, result);
   input input1, input2;
   output result;

   // Specify a NAND gate having propogation delay of 2ns
   nand #(2) nd1(result, input1, input2);
endmodule // delayExample
```

Problem 1

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 is 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 (NOT gate), you have to use verilog primitive not for the implementation, use the testbench testbench2.1.1.v for testing your design.

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. This buffer seems to be of no use since it cannot do any processing. This leads us to its one of the common use case, using buffer to level translation. Level translation is used whenever two parts of a circuit are powered with different Vdd. Now that you know what a buffer is, write a Verilog module to implement a buffer 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 . 1 be t_d, intuitively one can say that the delay between change of values at 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.

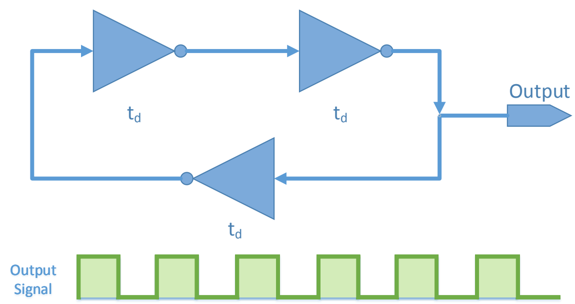

Figure 1: Simulation result for Listing 1, output changes 15ns after the change in input.

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 should be 30ns, assume that each of the NOT gate has a propagation delay of 5ns. Use testbench2.1.4.v to test your design.

5. 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 should be 30ns, assume that each of the NOT gate has a propagation delay of 5ns. Use testbench2.1.4.v to test your design.

6. Will a single not gate ring oscillator work? Why?

1.2 Data-flow Modelling

Data flow modelling is a higher level of abstraction. 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 used data flow modelling.

1.2.1 Continuous Assignment

Continuous assignment in Verilog are 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:

Note: 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.

```
/*
 * Following module uses continuous assignement to model
 * an AND gate
 */
module continuousAssignment(input1, input2, result);
   input input1, input2;
   output result;

   /* Use continuous assignment to set result
    * This is equalent to:
    *     and (result, input1, input2);
    */
   assign result = input1 & input2;
endmodule // continuousAssignment
```

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

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

1.2.2 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.

```
`timescale 1ns/1ps

module assignmentDelay(input1, input2, result);
   input input1, input2;
   output result;

   // Defines a circuit which will have output value
   // input1 | input2 15ns after changing input
   assign #15 result = input1 | input2;
endmodule // assignmentDelay
```

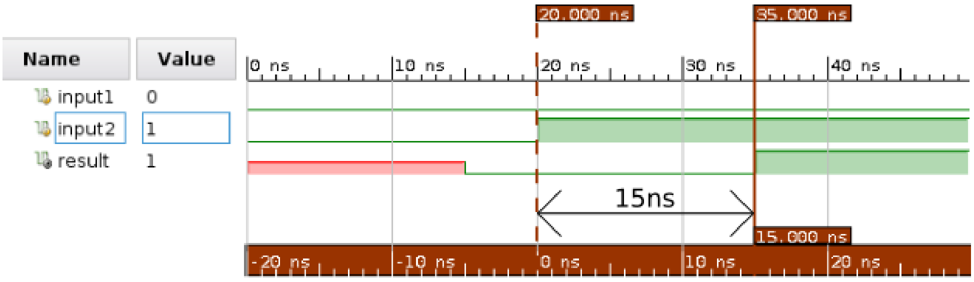

Figure 2: Simulation result for Listing 5, output changes 15ns after the change in input.

Problem 2

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 $= \text{in}_1$ else output $= \text{in}_2$.

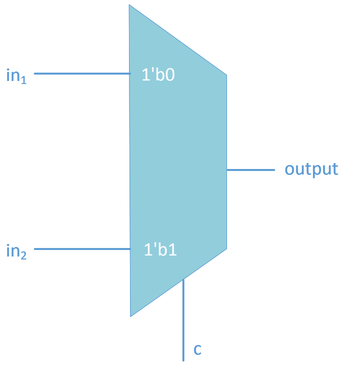

Figure 3: Simulation result for Listing 5, output changes 15ns after the change in input.

1. Write the hardware description of a 2-to-1 multiplexer using Verilog, you have to implement this using only behaviourial modelling. (*Hint: Verilog supports the Ternary operator, which has same behaviour as in programming languages.*)
2. To verify the behaviour of multiplexer you just described, write a testbench to test its behaviour. The testbench should generate all different permutations of the inputs. To keep things simple, you can for now verify the behaviour using waveform viewer.
3. Now suppose you want to model a multiplexer you just purchased from the market, which has a prpogation delay of 2ns. Modify the module from part 1 such that the behaviour of your description matches the one you bought.

---

2 Behavioral Modelling

Behavioral modelling is a 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 circuit description.

2.1 `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 vector). We'll get back to usage of `reg` and will differentiate it with `wire` after studying procedural blocks.

2.2 Procedural Blocks

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

2.3 `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 `end`. Initial blocks gets executed only once when the simulation is started.

Note: 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
...
```

2.4 `always@` block

`always@` block in Verilog are used for describing a 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 one 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
```

2.5 Blocking and Non-Blocking Assignment

2.5.1 Blocking Assignments

Blocking assingments are assignments which block the simulation while their value is being calculated. This means that theexecution 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.

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

2.5.2 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!

```
// Following example is for swapping bytes
// using non-blocking assigment, this module
// unlike last example uses non-blocking assignment
// this allows swapping bytes correctly.

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

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

```

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

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

3.1 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. However they cannot store it, you'll have to drive them with values (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.
- `wire` can be used to assign a value to a `reg` or a `wire`.

3.2 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 `wire` can only be assigned a value in a procedural block.
- `wire` can be used to assign a value to a `reg` or a `wire`.

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

Problem 3

Turing machines are mathematical abstraction and model of computational machines, Turing machine was proposed by Alan Turing in 1973. The basis of invention of turing machine was Alan's interest in the philosophy of computation, any algorithm which couldn't be implemented on a Turing machine is never been implemented using some other programming language or computational device. 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.

Turing machine consist of a infinitely long tape which has certain shapes 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.

Turing machines can be used to implement Fine state machines, to use a Turing machine as a finite state machine we need following assumptions:

(a) We have a finite length tape (since it would be impossible to write description of an infinite tape).

(b) Each character on the tape can only be 'A', '+' or '_' where 'A' and '+' are used for decisions while '_' represents a halting condition.

(c) 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.

(d) At every clock cycle the Turing machine can execute only one instruction.

(e) Each instruction execution changes decides the next position of the head and can write to the tape where the head is.

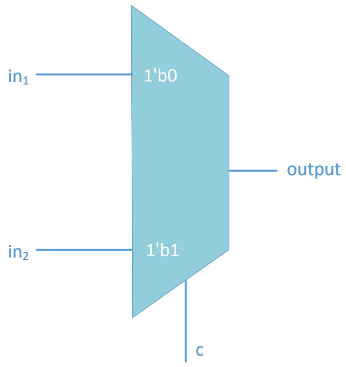

Figure 4: Example of a Turing machine.

Value under head	Direction of movement	Write value
A	Right	-
+	Right	A

Now consider the following turing machine rules, for a machines which adds up two numbers. These numbers are represented in base 1 $(1111_1$ to represent $4_{10})$:

Using this rule set addition of two numbers can be performed, 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:

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**)

Questions

1. Briefly describe the algorithm for the addition of which the trace is give above.
2. Write Verilog description for the Turing machine described above, use 2 bit registers to represent single location in the tape. The tape of the machine will be finite in length and should be enough to store the above example.
3. Simulate the addition of 2 and 4 in the above turing machine.

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

References

- http://inst.eecs.berkeley.edu/~cs150/fa08/Documents/Always.pdf
- Aenean in sem ac leo mollis blandit.

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

These resources are great for expanding your understanding, but...