ASSIGNMENT 1
INTRODUCTION TO VERILOG

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

Hardware Description Languages

A hardware description language (HDL) is a convenient, device and technology independent way of representing digital logic. HDLs are helpful describing, simulating and verifying digital circuits.

Why not use C/C++, Java...?

C/C++, Java etc. are programming languages which are very good at what they were designed for, that is programming. However, describing digital circuits in a programming language is difficult and often confusing as it needs more specifications on 'How To' alongside 'What To'.

For example, let's say you want to add 4 1-bit numbers. Now, if you were programming in C/C++, you would have simply written RESULT=a+b+c+d and left the rest to your compiler. But when implementing this logic on hardware, there are more than one ways to do it. You can either cascade three adders or make an adder-tree using three adders. Now both of these implementations have different delays and thus you must have the freedom to decide which one to use. No doubt you could achieve whichever implementation you want by writing some extra lines of C++ to specify exactly what you want, but, as the problem complexity grows, this extra effort to exactly specify what we want grows drastically. This led to development of HDLs where WYWIWYG(What You Write Is What You Get). NOTE: Although this may not be the case if you are lazy and leave ambiguity in your HDL code leaving it to the synthesizing software to decide upon.

Which HDL to use?

Many hardware description languages are available today. Each of them is different from the other in terms of functionality, semantics and grammar. In this course however, we will stick to Verilog 2001.

Using Verilog 2001

As a beginner, we are going to describe and simulate simple combinational circuits with Verilog.

Design Flow

Verilog Syntax

Modules

Module, the basic building block in Verilog, helps in organizing and structuring designs in logical and human readable form. A module can be considered as analogous to functions in programming languages(NOT EXACTLY though. Functions, by definition, act in BATCH MODE i.e, you give them inputs and get output after sometime. Batch mode here means before getting the output once, you can't send in new inputs. But in HDL, you may write PIPELINED modules. Pipelining can be understood by the working of a car assembly line. The job is divided into smaller sections. Now while a car is being painted, some other cars may be in the welding and testing sections. This naturally leads to faster production of cars if the company has a large order.) Basic structure of a Verilog module is given in Listing 1

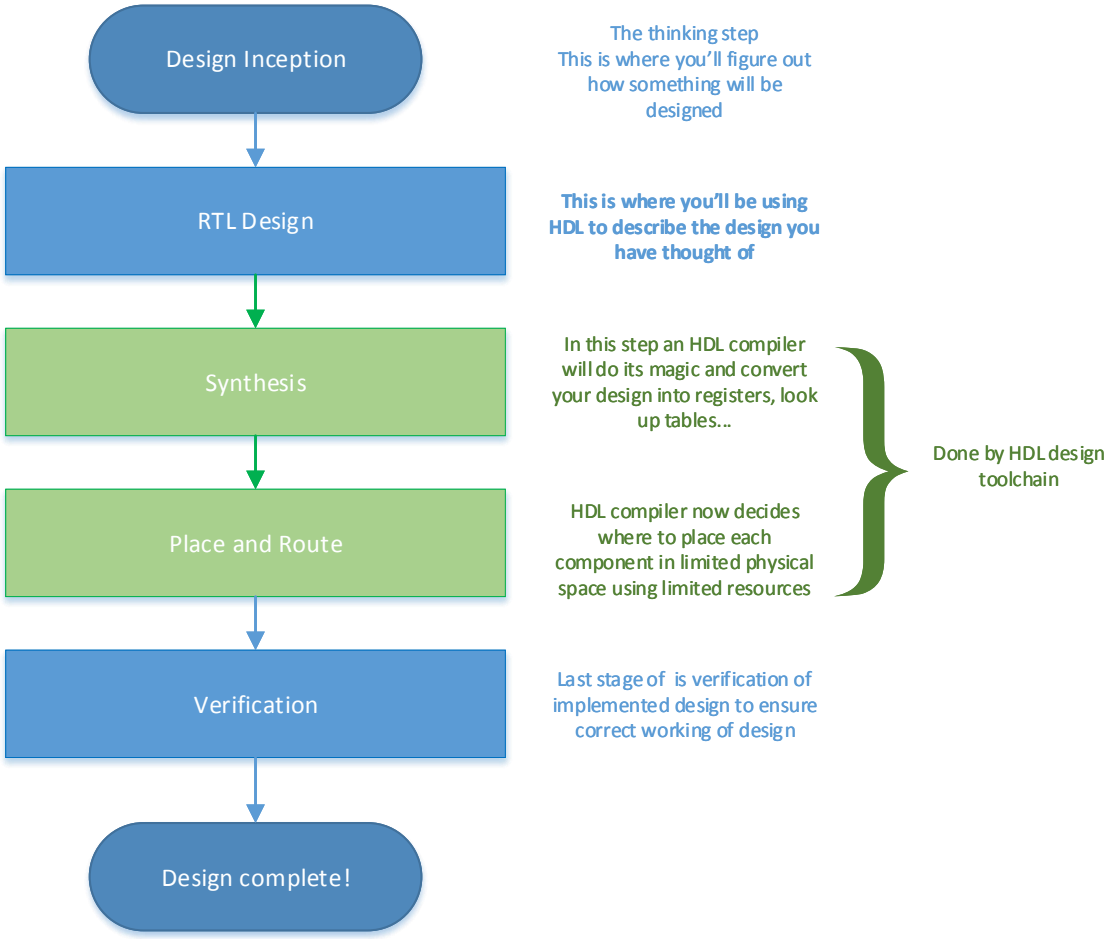

Figure 1: Basic design flow of a design from its inception to implementation.

Listing 1: Sample module indicating its structure

```
1 module module_name( ...module_input_and_outputs... );
2   ...
3   // Module functionality
4   ...
5 endmodule;
```

An example of AND gate is given in Listing 2

Listing 2: Illustrative AND gate module

```
1 module andGate (inputA, inputB, result);
2         ...
3         // AND gate functionality
4         ...
5 endmodule
```

Instantiating modules

The process of creating objects of modules is called instantiation in Verilog. Modules are like blueprints and instantiating them means constructing hardware using the blueprints.

Listing 3: Illustrative AND gate module

```
// Define the top-level module called AndGate3 which is a three
// input AND gate. It instantiates 2 2-input AND gates.
// Interconnections will be explained later
module AndGate3(input1, input2, input3, result);
   input input1, input2, input3; // Declare variables as input
   output result; // declare result as an output

   wire   result1; // To be explained later

    // Instatiation of 2-input AND gates
   AndGate2(input1, input2, result1);
   AndGate2(result1, input3, result);
endmodule

// Define a module AndGate2 which is a 2-input AND gate.
module AndGate2(input1, input2, result);
   input input1, input2;
   output result;

   // Assignment would be explained later
   assign result = input1 & input2;
endmodule
```

Comments

Comments in Verilog are of two kinds:

Single Line Comment

Two forward slashes represents beginning of a single line comment in Verilog, anything in a line after those two characters will ignored by the compiler

Listing 4: Single line comment

```
...
// This is a single line comment
...
```

Block Comment

Block comments in verilog are used to comment a block of code, they start with '/*' and ends with '*/'. Anything between these two character sequences will be ignored by the compiler.

Listing 5: Block comment

```
...
/*
 This is a block comment
 */
...
```

Numerical Literals

Sized Numbers

To represent digital circuits accurately Verilog allows defining numbers of fixed size. These numbers have the following format:

<size>'<character for base><number>

For example:

Listing 6: Example of sized numbers

```
6'b010010    // (18) 32-bit number in binary
24'hc0ffee   // (12648430) 24-bit number
16'd255      // (255) 16-bit decimal number
```

Unisized Numbers

Verilog also includes support for unsized numbers. These numbers are assumed to be of a particular size depending on the compiler/machine.

Constants

Global constants can be declared in Verilog. When Verilog code is processed all these constants will be replaced by their respective values. NOTE: Verilog constants always starts with backtick ‘ ` ’.

Listing 7: Declaration and use of constants

```
`define A 2'200 // NOTE: Constant declarations do
`define B 2'b01 // not end with semicolon!
`define C 2'210
`define D 2'b11

// Using constants
wire [A:0] wire_a = `A; // Notice the backtick
wire [B:0] wire_b = `B;
wire [C:0] wire_c = `C;
wire [D:0] wire_d = `D;
```

Wires

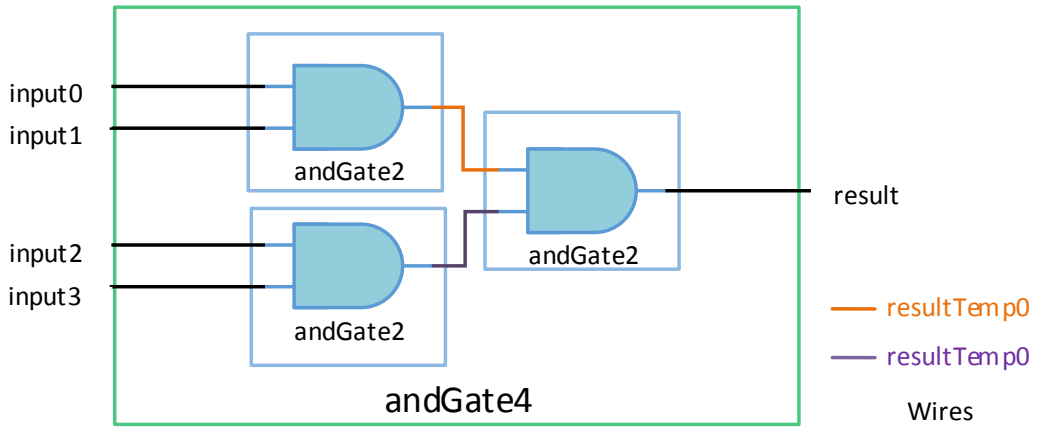

Figure 2: Gate level representation of code in Listing 8.

Wires are something very important in verilog. Use of wires in Verilog is for what wires are used in real life ... to connect two things electrically! Wires will be used extensively in future assignments to connect two modules, registers and even wires together. This concept is explained using Listing 7 where wires are used to connect the output of two 2-input AND gates to inputs of single 2-input AND gates. Gate level diagram of which is shown in Fig. 2.

Listing 8: Use of wires to connect output of one module to input of another

```
1  // andGate4 is a four input AND gate that uses 3 2-input AND
2  // gates to produce its result. resultTemp0 and resultTemp1 wires
3  // are used to connect output of two andGate2 to a another andGate2
4  // This module uses andGate2 module from previous listing.
5  module andGate4(input0, input1, input2, input3, result);
6     // Declare input and output
7     input input0, input1, input2, input3;
8     output result;
9
10    // Declare wire for connecting gates
11    wire resultTemp0;
12    wire resultTemp1;
13
14    // Declare instances of 2-input AND gate
15    andGate2(input0, input1, resultTemp0);
16    andGate2(input2, input3, resultTemp1);
17
18    andGate2(resultTemp0, resultTemp1, result);
19 endmodule
```

Vectors

Verilog supports declaration of vectors to represent a group of wires (or registers which will be explained later). Vectors are declared by specifying their type, range and then their name:

<type> <range> <name>;

Example declaration of 6 bit wide vector of type wire:

wire [5:0] new_wire; // Both the range specifier digits are inclusive

Here new_wire[5] is MSB while new_wire[0] is LSB.

Part Select

What if you wanted to access 2^{nd}, 3^{rd} and 4^{th} elements of a vector? This is when part select of verilog will help. Part select in Verilog allows extraction of a smaller vector from an existing vector.

Example:

Listing 9: Using bit extract to extract lower, higher and middle 16 bits from a 32 bit input

```
1  module bitExtract(input1, bitsHigh, bitsLow, bitsMid);
2     input [31:0] input1;          // Declare 32 bit input
3     // Declare 3 output having 16 bit width
4     output [15:0] bitsHigh;       // Higher 16 bits from input1
5     output [15:0] bitsLow;        // Lower 16 bits from input1
6     output [15:0] bitsMid;        // Mid 16 bits from input1
7
8     // Assign values to output
9     assign bitsHigh = input1[31:16]; // <-- Note that bit select is inclusive
10    assign bitsLow  = input1[15:00];
11    assign bitsMid  = input1[21:06];
12
13 endmodule // bitExtract
```

Problem 1

Shift operation is one of the important operation in a digital system. Various different designs exist for performing shifts of arbitrary amount within a range(Search the internet for Barrel Shifters for one implementation). But, a fixed amount shifter (e.g. a shift by 5 element) is extremely easy to implement.

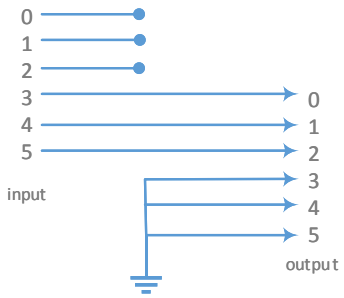

Figure 3: A shift right by 3 element design for bus width of 6.

This is done by moving connections from input to output along the desired direction by the desired amount. An example of which is given in Fig. 3

Write a Verilog module named MODULE_1_1 which uses part select, takes a 32bit vector as input named INPUTSIGNAL and outputs a 32bit vector RESULT which is simply inputSignal shifted by 5 towards left. Use testbench_1_1.v to verify the output. (*Hint: Similar to part select, Verilog supports part assign which allows assigning values to a part of a vector*)

Operators

Verilog supports a wide range of operators to represent mathematical operations. These are high level representation of what later would be converted to a gate level implementation by your HDL compiler.

Airthmetic Operators

Veriog allows use of various airthmetic operators to perform calculations on vectors (wires & reg). Following example shows its usage:

Listing 10: Functioning of airthmetic operator

```
1  module airthmeticOperators
2    (
3     output o1, o2, o3, o4, o5, o6, o7, o8
4     );
5
6     assign o1 =  1 +  2; // = 3
7     assign o2 =  1 -  2; // = -1
8     assign o3 = 10 *  5; // = 50
9     assign o4 = 10 /  5; // = 2
10
11    assign o5 =  5 / 10; // = 0 <--- NOTE
12
13    assign o6 = 10 %  3; // = 1
14    assign o7 = +7;      // = 7
15    assign o8 = -7;      // = -7
16 endmodule // airthmeticOperators
```

Problem 2

Hierarchical design is often helpful in verifying large project easily by verifying each individual module separately. Make a module for 2 input NAND gate named NANDGATE. Now, make a 2 input AND gate named ANDGATE using multiple of these NAND gates. Use testbench_1_2_1.v for verifying the output of NAND gate and testbench_1_2_2.v for verifying the output of the AND gate.

Table 1: List of common operators used in Verilog

Operator	Description	Functional Group
[]	bit select or part select	
()	parenthesis	
! ~ & | ~& ~| ^ ~^ or ^~	negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	logical bitwise reduction reduction reduction reduction reduction reduction
+ −	Unary Plus (plus sign) Unary minus (minus sign)	airthmetic
* / %	multiply divide modulus	airthmetic
+ −	Binary Plus Binary minus	airthmetic
<< >>	shift left shift right	shift
> >= < <=	greater than greater than or equal to less than less than or equal to	relational
== !=	case equality case inequality	equality
& | ^	bitwise AND bitwise OR bitwise XOR	bitwise

Reduction Operators

Reduction operators in Verilog reduces a vector into a single bit by repeatedly performing a specified operation. Following example explains its usage:

Listing 11: Functioning of reduction operator

```
1  module reductionOperators
2    (
3     input [15:0] i,
4     output       r1, r2, r3, r4, r5, r6
5     );
6
7     // 16 bits of i are ANDed together and set as the value
8     // of r1
9     assign r1 = &i; // r1 = i[0] & i[1] & i[2] ... & i[15]
10    // Other reduction operator follows same logic
11    assign r2 = ~&i; // r2 = i[0] ~& i[1] ~& i[2] ... ~& i[15]
12    assign r3 =  |i; // r3 = i[0]  | i[1]  | i[2] ...  | i[15]
13    assign r4 = ~|i; // r4 = i[0] ~| i[1] ~| i[2] ... ~| i[15]
14    assign r5 =  ^i; // r5 = i[0]  ^ i[1]  ^ i[2] ...  ^ i[15]
15    assign r6 = ~^i; // r6 = i[0] ~^ i[1] ~^ i[2] ... ~^ i[15]
16 endmodule // reductionOperators
```

Problem 3

Reduction operators are often used when some computation has to be performed on complete signal, particular example of which is calculating XOR of a signal to get its parity bit. Even parity bit of a signal is 1 if signal has odd number of 1 and 0 if number of 1s is even, similarly odd parity bit of number is 1 if signal has even number of 1 and 0 if number of 1s is odd.

Write a Verilog module named PARITYBITS which uses reduction operator, takes a 32bit vector as input named SIGNAL, two single bit output named PARITYEVEN and PARITYODD. Use testbench_1_3.v to verify output. (*Hint: Calculating XOR of all bits of a signal gives even parity of the signal.*)

Relational Operators

Relational operators in verilog are used to compare two values, usage of all four type of relational operators supported by Verilog are given in Listing 12.

Listing 12: Functioning of relational operator

```
module relationalOperators
  (
   output o1, o2, o3, o4
   );

   assign o1 = 1<2;              // = 1'b1
   assign o2 = 1>2;              // = 1'b0
   assign o2 = 1>=1;             // = 1'b1
endmodule // relationalOperators
```

Logical Operators

Logical operators are used in conjunction with relational and equality to perform multiple comparisons within a single expression. Example usage of logical operators are provided in 13.

Listing 13: Functioning of logical operator

```
module logicalOperators
  (
   output o1, o2
   );

   assign o2 = (2'b10 & 1'b01) && (2'b00 & 1'b11);               // = 1'b0
   assign o1 = ((1'b1 == 1'b0) || ((2'b11 && 2'b01) == 2'b01)); // = 1'b1
endmodule // logicalOperators
```

Bitwise and Shift Operators

Verilog Bitwise and Shift operators works in the same way as they work in Java, except that in Verilog only left shift (<<) and right shift (>>) are supported.(There is an arithmetic right shift operator too which can be searched on the internet for details!)

Problem 4

Multiplying using digital logic is a challenging task, one very intuitive approach is to use 'shift-and-add approach'; in this method, two binary numbers are multiplied by shifting one and adding to other. This is repeated till the desired multiplication is obtained. This approach is described here:

https://en.wikipedia.org/wiki/Binary_multiplier#Multiplication_basics

Write a module named MULTIPILER for multiplying a 32-bit binary number by 10(4'b1010 in binary) without using the multiply operator. Assume the input is such that output doesn't overflow. Use testbench_1_4.v to verify output. RTL schematic of your module should resemble Fig. 4. The input to the module should be named INPUTSIGNAL and the output should be named RESULT to allow the testcases to run.

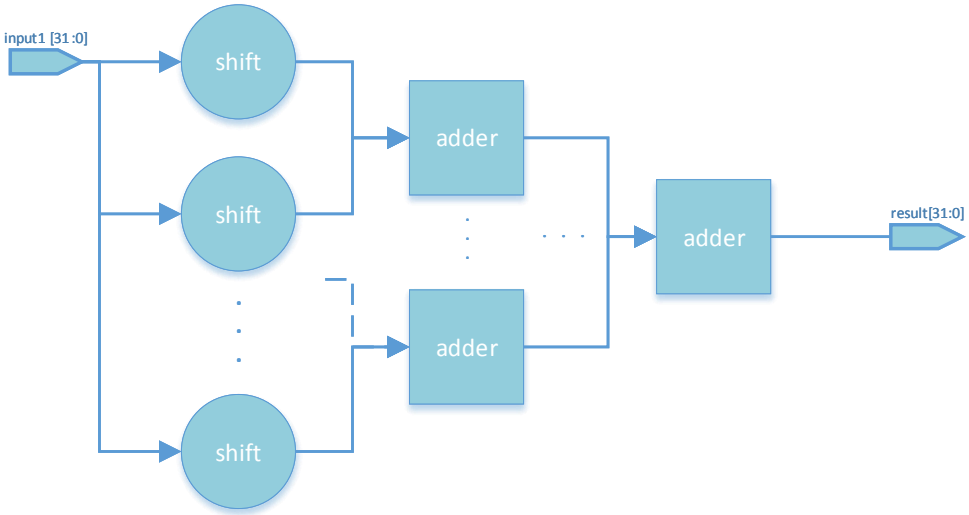

Figure 4: RTL schematic for Problem 4. **Note:** This is representative RTL schematic, solution to the problem will have fixed number of shift and add blocks.

References

- http://cva.stanford.edu/people/davidbbs/classes/ee108a/winter0607%20labs/ee108a_nham_intro_to_verilog.pdf
- Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition. By Samir Palnitkar. Publisher: Prentice Hall PTR. Pub Date: February 21, 2003. ISBN: 0-13-044911-3
- https://www.utdallas.edu/~akshay.sridharan/index_files/Page5212.htm

Submission Instructions

The submissions of solutions to the assignment 1 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