Compter Architecture
CECS 440

LAB 4

CECS Department
California State University, Long Beach

Spring 2020

Due on Sunday 03/22/2020 11:00 pm

In Part 1 and 2 we are going to review some aspect of Verilog language. In part 3 we will talk about the ALU which we ask you to design.

1 Arithmetic Operators

Arithmetic operators are used to create arithmetic functions. Verilog provides the following arithmetic operators:

Operator	Description
a + b	a plus b
a - b	a minus b
a * b	a multiplied by b
a / b	a divided by b
a % b	a modulo b
a ** b	a to the power of b

The arithmetic operators in Verilog are defined for numeric types (integer and real).

Code 1: 32-bit Full Adder

```
1  `timescale 1ns / 1ps
2  // Module definition
3  module FA32(Sum, Cout, A, B, Cin);
4  // Define I/O signals
5    output reg [31:0] Sum;
```

```
6     output reg Cout;
7     input [31:0] A;
8     input [31:0] B;
9     input Cin;
10  // Describe FA behaviour
11    always @(A, B, Cin)
12      {Cout, Sum} = A + B + Cin;
13  endmodule  // 32-bit FA
```

2 Concurrent Vs Sequential

Contrary to regular computer programs which are sequential, Verilog statements are inherently concurrent (unless they are inside a sequential block as discussed later). Concurrent means that the operations described in each line take place in parallel. The commonly used concurrent constructs are gate instantiation and the continuous assignment statement. Concurrent statements are evaluated independently of the order in which they appear. Concurrent statements include the followings:

- Wire assignments
- Component instantiations
- Generate statements
- Always statements
- Procedure and function calls

Sequential statements differ from concurrent statements in that they are executed in the order they are written. Sequential statements always appear within a `always@` block statement or within a function or procedure. The term Sequential in Verilog refers to the fact that the statements execute in order, rather than to the type of logic generated.

- All sequential Verilog statements must be inside a procedural block.
- Sequential statements are placed inside a begin/end block and executed in sequential order within the block. However, the blocks themselves are executed concurrently.
- Verilog statements outside any `always@` block are interpreted as concurrent statements and different procedural blocks execute concurrently.

The commonly used sequential constructs are:

- Initial and always procedural blocks: `initial` blocks start execution at time zero and execute only once. `always@` blocks loop to execute over and over again, and as the name suggests, they execute always.
- if, case structures, for and while loops: they always appear inside an `always@` block. Note that multiple assignment of values is allowed inside a procedural block.

2.1 `always@` Blocks

A `always@` block is a sequential section of a Verilog code which is followed by a set of paranthesis, a `begin`, some code, and an `end`. Its syntax is shown below.

```
always @( ... sensitivity list ... ) begin
    ... elements ...
end
```

The `sensitivity` list specifies which signals should trigger the elements inside the `always@` block to be updated. The `elements` describe elements that should be set when the sensitivity list is satisfied. When the sensitivity list is satisfied, the elements inside the `always@` block are set/updated. They are not otherwise. In the multiplexer example, the `always@` block will run whenever there is a change in `select or d`.

Following is an example of a 4 to 1 Mux code which includes sequential statements (case) in an `always@` block statement:

Code 2: 4-to-1 Multiplexer with case statement

```
 `timescale 1ns / 1ps
// Module definition
module mux41( S, D, Y );
//Define I/O signals
 input[1:0] S;
 input[3:0] D;
 output     Y;

 reg        Y;
 wire[1:0] S;
 wire[3:0] D;
 // Describe 4to1 Mux behaviour
 always @( S or D )
  begin
     case( S )
      0 : Y = D[0];
      1 : Y = D[1];
      2 : Y = D[2];
      3 : Y = D[3];
     endcase
  end
endmodule   // 4to1 MUx
```

2.2 Wires and Regs

`wire` elements are simple wires (or busses of arbitrary width) in Verilog designs. The following are syntax rules when using wires: `Wire` elements:

- Are used to connect input and output ports of a module instantiation together with some other element in your design.
- Are used as inputs and outputs within an actual module declaration.
- Must be driven by something, and cannot store a value without being driven.
- Cannot be used as the left-hand side of an = or <= sign in an `always@` block.
- Are the only legal type on the left-hand side of an `assign` statement.
- Are a stateless way of connecting two pieces in a Verilog-based design.
- Can only be used to model combinational logic.

`reg` elements are similar to wires, but can be used to store information ('state') like registers. The following are syntax rules when using reg elements. `reg` elements:

- Can be connected to the input port of a module instantiation.
- Cannot be connected to the output port of a module instantiation.

- Can be used as outputs within an actual module declaration.
- Cannot be used as inputs within an actual module declaration.
- Are the only legal type on the left-hand side of an `always@` block = or <= sign.
- Are the only legal type on the left-hand side of an `initial` block = sign (used in Test Benches).
- Cannot be used on the left-hand side of an `assign` statement.
- Can be used to create registers when used in conjunction with `always@(posedge Clock)` blocks.
- Can, therefore, be used to create both combinational and sequential logic.

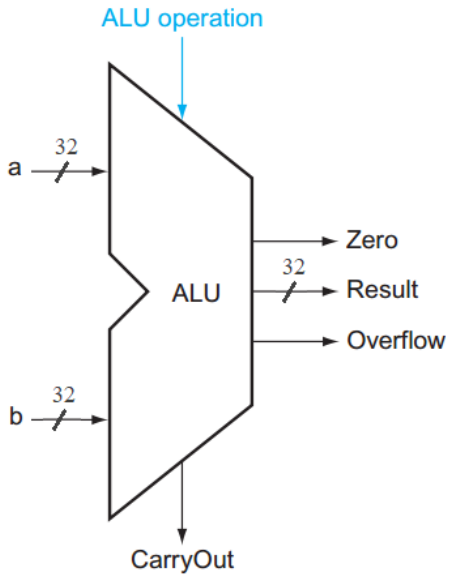
2.3 When `wire` and `reg` Elements are Interchangable

wire and reg elements can be used interchangably in certain situations:

- Both can appear on the right-hand side of `assign` statements and `always@` block = or <= signs.
- Both can be connected to the input ports of module instantiations.

3 Design a 32-bit ALU

The arithmetic logic unit (ALU) is the brain of the computer, the device that performs the arithmetic operations like addition and subtraction or bitwise operations like AND and OR. In this section, we want to design a 32-bit ALU. Below you see the block diagram.



The ALU receives two 32-bits inputs (`a` and `b`) and produces one 32-bits output (Result). The ALU based on the 4-bits Control Code decides which operation should be executed. Below you can see the value of the ALU control lines and the corresponding ALU operations.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	Set less than
1100	NOR
1111	Equal comparison

The **Set Less Than** operation tests the two inputs (`a` and `b`). If `a` is less than `b` then the Result would be 1, otherwise 0.
The **Equal Comparison** operation tests the two inputs (`a` and `b`). If `a` is equal to `b` then the Result would be 1, otherwise 0.
The **Zero** output is 1 when all of the 32 bits of the results are 0.
The **Overflow** output is 1 when there is an overflow in Add or Subtract operations (for other operations overflow is always 0).
The **Carry_Out** output is set (Only in ADD operation) when you have carry out on Left-most bit (MSB).
SLT, ADD, and SUB operations need to be implemented as signed operations.

Use this sample code for your Module (Entity) definition.

Code 3: ALU module definition

```
1  `timescale 1ns / 1ps
2  // Module definition
3  module alu_32(A_in, B_in, ALU_Sel, ALU_Out, Carry_Out, Zero, Overflow);
4  //Define I/O ports
5
6
7  // Describe ALU behaviour
8
9
10 endmodule   // 32-bit ALU
```

Write a testbench for your design and run the tests below.

test1 (Run for 20ns):
tb_din_a= 32'h086a0c31 , tb_din_b= 32'hd785f148, tb_alu_sel = "4'b0000"

test2 (Run for 20ns):
tb_din_a= 32'h086a0c31 , tb_din_b= 32'h10073fd4, tb_alu_sel = "4'b0001"

test3 (Run for 20ns):
tb_din_a= 32'h086a0c31 , tb_din_b= 32'h90073fd4, tb_alu_sel = "4'b0010"

test4 (Run for 20ns):
tb_din_a= 32'h086a0c31 , tb_din_b= 32'h90073fd4, tb_alu_sel = "4'b0110"

test5 (Run for 20ns):

tb_din_a= 32'h086a0c31 , tb_din_b= 32'h90073fd4, tb_alu_sel = "4'b0111"

test6 (Run for 20ns):
tb_din_a= 32'h086a0c31 , tb_din_b= 32'h90073fd4, tb_alu_sel = "4'b1100"

test7 (Run for 20ns):
tb_din_a= 32'h086a0c31 , tb_din_b= 32'h086a0c31, tb_alu_sel = "4'b1111"

test8 (Run for 20ns):
tb_din_a= 32'h086a0c31 , tb_din_b= 32'h10073fd4, tb_alu_sel = "4'b1111"

Check the outputs (Result, Carry_Out, Zero, Overflow) to see if they are correct. Put a screenshot of the wave in your report.

4 Assignment Deliverables

Your submission should include the following:

- Block designs and testbenches. (ALU.v, tb_ALU.v)
- A report in pdf format. Your report should have all details for your design + screenshot of the wave. Each report should includes group members.

Note1: Your work will be evaluated based on your report and correct functionality of your code. Allocate your time and your effort for both.

Note2: Compress all files (3 files : 2 .v files + report) into zip and upload to the *Beachborad* before deadline. One submission per group.

Note3: Use the code samples that are given in the lab description. The Entity (module definition) part of your code should exactly look like the code sample otherwise you lose points.