BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
CS F342 COMPUTER ARCHITECTURE
First Semester 2014-2015
Lab Sheet-4

Learning Objectives:

1. To design a Simple MIPS ALU in a step by step manner
2. To design the main Control PLA to generate control signals
3. To design the ALU control Logic

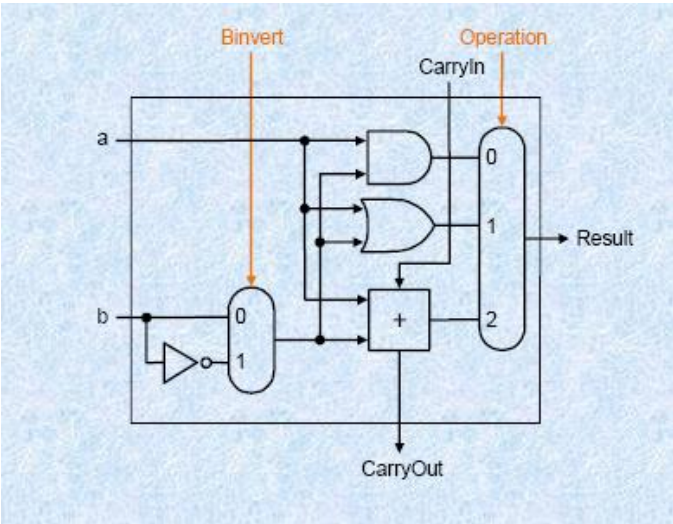
In the last three labs, you have designed various digital blocks of combinational and sequential circuits. This lab onwards, we will start designing MIPS processor step-by-step progressively. Therefore you are advised to carry the Verilog design files that you have completed in a particular lab to next week's lab session. Follow this practice for all the upcoming labs.

Today's lab will be dedicated to the design of the MIPS ALU.

TASK 1: First task is to design the ALU for MIPS ISA, such that it will implement the following instructions.

(i) and $s1, $s2, $s3
(ii) or $s1, $s2, $s3
(iii) add $s1, $s2, $s3
(iv) sub $s1, $s2, $s3

As we know that each of these registers are of 32-bit and thus the ALU must be capable of handling 32-bit values. Our aim in TASK1 is to design the following ALU discussed in the class. It should perform above four operations i.e. AND, OR, ADD and SUB on the operands a and b. So we would divide this task into sub-tasks. We will design various components of the ALU in these sub-tasks and integrate them finally. The ALU primarily comprises of AND gate, OR gate, Full Adder and two multiplexers.

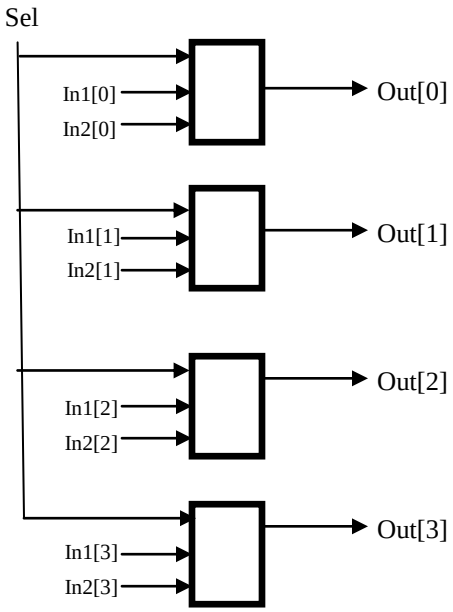


Sub Task 1.1: In this sub-task let us design the Multiplexer. We begin with a 2:1 Multiplexer.

You have already worked on 2:1 mux in Lab-1. The Verilog description is as follows:

```
module mux2to1(out,sel,in1,in2);
  input in1,in2,sel;
  output out;
  wire not_sel,a1,a2;
  not (not_sel,sel);
  and (a1,sel,in2);
  and (a2,not_sel,in1);
  or(out,a1,a2);
endmodule
```

Note that the width of the inputs and output data has been just one bit. So to design a 2:1 Mux for MIPS, the input and output data lines must be 32-bit wide. First consider 2:1 Mux for a 4-bit wide data. How would we design it? The select line will be common for 4 different 2:1 Muxes. So the output Out[i] will be In1[i] or In2[i] depending on the value of Select line 'Sel'. The structure for 4-bit wide Mux is as follows.



Let us design 8-bit wide 2:1 Mux using the above mux2to1 module.

```
module bit8_2to1mux(out,sel,in1,in2);
  input [7:0] in1,in2;
  output [7:0] out;
  input sel;
  mux2to1 m0(out[0],sel,in1[0],in2[0]);
  mux2to1 m1(out[1],sel,in1[1],in2[1]);
  mux2to1 m2(out[2],sel,in1[2],in2[2]);
  mux2to1 m3(out[3],sel,in1[3],in2[3]);
  mux2to1 m4(out[4],sel,in1[4],in2[4]);
  mux2to1 m5(out[5],sel,in1[5],in2[5]);
```

```
  mux2to1 m6(out[6],sel,in1[6],in2[6]);
  mux2to1 m7(out[7],sel,in1[7],in2[7]);
endmodule
```

Test the above design using the following test bench and observe the output.

```
module tb_8bit2to1mux;
  reg [7:0] INP1, INP2;
  reg SEL;
  wire [7:0] out;
  bit8_2to1mux M1(out,SEL,INP1,INP2);
  initial
  begin
    INP1=8'b10101010;
    INP2=8'b01010101;
    SEL=1'b0;
    #100 SEL=1'b1;
    #1000 $finish;
  end
endmodule
```

Note: Instead of writing the statement to instantiate the mux2to1 module 8 times, you can use a generate statement and a for loop. Refer to Section 7.8 of Samir Palnitkar on Generate blocks. The module `bit8_2to1mux` can be written as follows using generate statement.

```
module bit8_2to1mux(out,sel,in1,in2);
  input [7:0] in1,in2;
  output [7:0] out;
  input sel;
  genvar j;
//this is the variable that is be used in the generate
//block
  generate      for   (j=0;   j<8;j=j+1)   begin:   mux_loop
//mux_loop is the name of the loop
  mux2to1 m1(out[j],sel,in1[j],in2[j]);
 //mux2to1 is instantiated every time it is called
end
endgenerate
endmodule
```

Exercises:

1) Design a 32 bit wide 2:1 Mux, using the above 8-bit wide 2:1 Mux module.
2) Using the above idea, design a MUX that has three 32-bit wide inputs.

{*Hint: You have to change the basic block of the above design and reuse the design in above exercise)*.

Sub Task 1.2:

In this task we will design the 32-bit wide AND and OR gates. One of the easiest ways of modelling it will be to do it in data flow modelling. Consider the following design of AND gate and its test bench.

```
module bit32AND (out,in1,in2);
  input [31:0] in1,in2;
  output [31:0] out;
  assign {out}=in1 &in2;
endmodule

module tb32bitand;
  reg [31:0] IN1,IN2;
  wire [31:0] OUT;
  bit32AND a1 (OUT,IN1,IN2);
  initial
  begin
    IN1=32'hA5A5;
    IN2=32'h5A5A;
    #100 IN1=32'h5A5A;
    #400 $finish;
  end
endmodule
```

It is very simple to do it using dataflow modelling.

Exercise:

1) Design the 32-bit OR gate in a similar way.

Subtask 1.3:

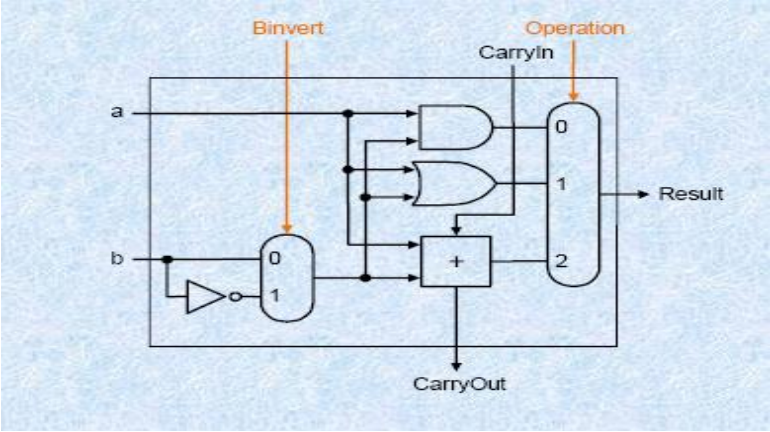
Refer to Lab sheet2 where you have designed a Full Adder using decoders and further you have designed an 8-bit and 32-bit adders. So you may use this 32 bit adder in this above ALU. Otherwise you may design a full adder using the dataflow modeling.

```
module FA_dataflow (Cout, Sum,In1,In2,Cin);
  input In1,In2;
  input Cin;
  output Cout;
  output Sum;
  assign {Cout,Sum}=In1+In2+Cin;
endmodule
```

Now expand it to make a 32-bit adder. One way is to instantiate it 32 times.You may use generate statement for this. Else consider expanding the input and output data ports to 32 bit wide. Will it work?

Final Task:

Now integrate the above modules to make the final ALU. The *Binvert* will be 1 for subtraction operation and 0 for add operation.
The *Carryin* will directly go to the 32 bit Full adder's *Cin*. The operation will be 00 for AND 01 for OR and 10 for ADD/SUB.



Test your design using the following test bench.

```
module tbALU()
reg Binvert, Carryin;
reg [1:0] Operation;
reg [31:0] a,b;
wire [31:0] Result;
wire CarryOut;
ALU (a,b,Binvert,Carryin,Operation,Result,CarryOut);

initial
begin
a=32'ha5a5a5a5;
b=32'h5a5a5a5a;
Operation=2'b00;
Binvert=1'b0;
Carryin=1'b0; //must perform AND resulting in zero
#100 Operation=2'b01; //OR
#100 Operation=2'b10; //ADD
#100 Binvert=1'b1;//SUB
#200 $finish;
end
endmodule
```

TASK 2: Construct Main Control unit which takes as input 6 bits opcode and produces nine different control signals. Also display the name of the control signals produced for specific opcode. Implement a PLA circuit as shown below.

It can be easily implemented using the data-flow modelling.

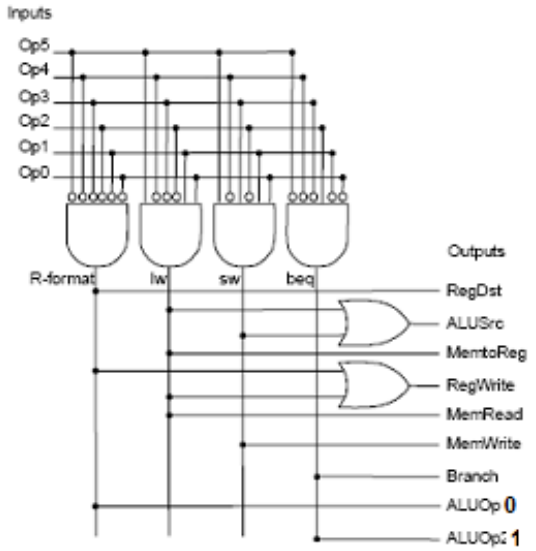
```
module ANDarray (RegDst,ALUSrc, MemtoReg, RegWrite,
MemRead, MemWrite,Branch,ALUOp1,ALUOp2,Op);
input [5:0] Op;
output RegDst,ALUSrc,MemtoReg, RegWrite, MemRead,
MemWrite,Branch,ALUOp1,ALUOp2;
wire Rformat, lw,sw,beq;

assign Rformat= (~Op[0])& (~Op[1])& (~Op[2])& (~Op[3])&
(~Op[4])& (~Op[5]);
// complete rest of the module
assign RegDst=Rformat;
endmodule
```

Complete the above and test it with a suitable test bench. Also Display the control signals produced using $display or $monitor and cross-verify with the truth table.

	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	x	x
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	x
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp0	1	0	0	0
	ALUOp1	0	0	0	1

Truth table for main control signals

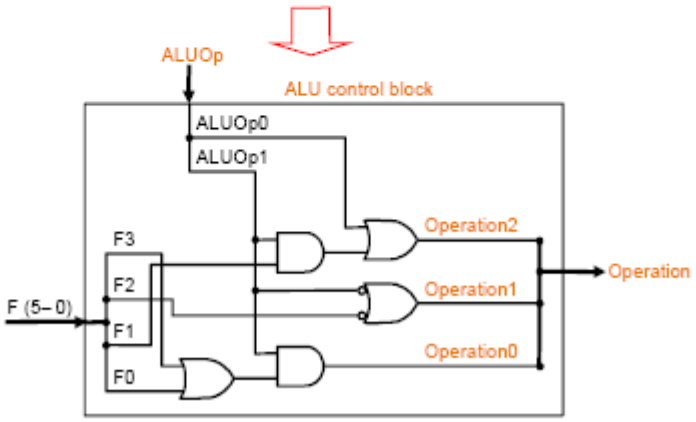


Main control PLA (programmable logic array): principle underlying PLAs is that any logical expression can be written as a sum-of-products

<u>TASK-3</u>: Construct ALU control unit which takes as input 2 bits ALU Ops and 4 Function code field bits and gives the 3 bit output. Also display the desired ALU operation along with output bits. Truth table and circuit diagram is as follows :

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Truth table for ALU control bits



ALU control logic

The ALUOp0 and ALUOp1 bits generated in TASK 2 will be used in this above design. The design is straight forward and can be done using data-flow or gate-level modelling.
