Computer Architecture
CECS 440

LAB 7

CECS Department
California State University, Long Beach

Spring 2020

Due on Friday 05/08/2020 by 11:00 pm

In this lab, we want to complete the design of our single cycle processor. As you see in Figure 1 there are three sub-modules in this processor.

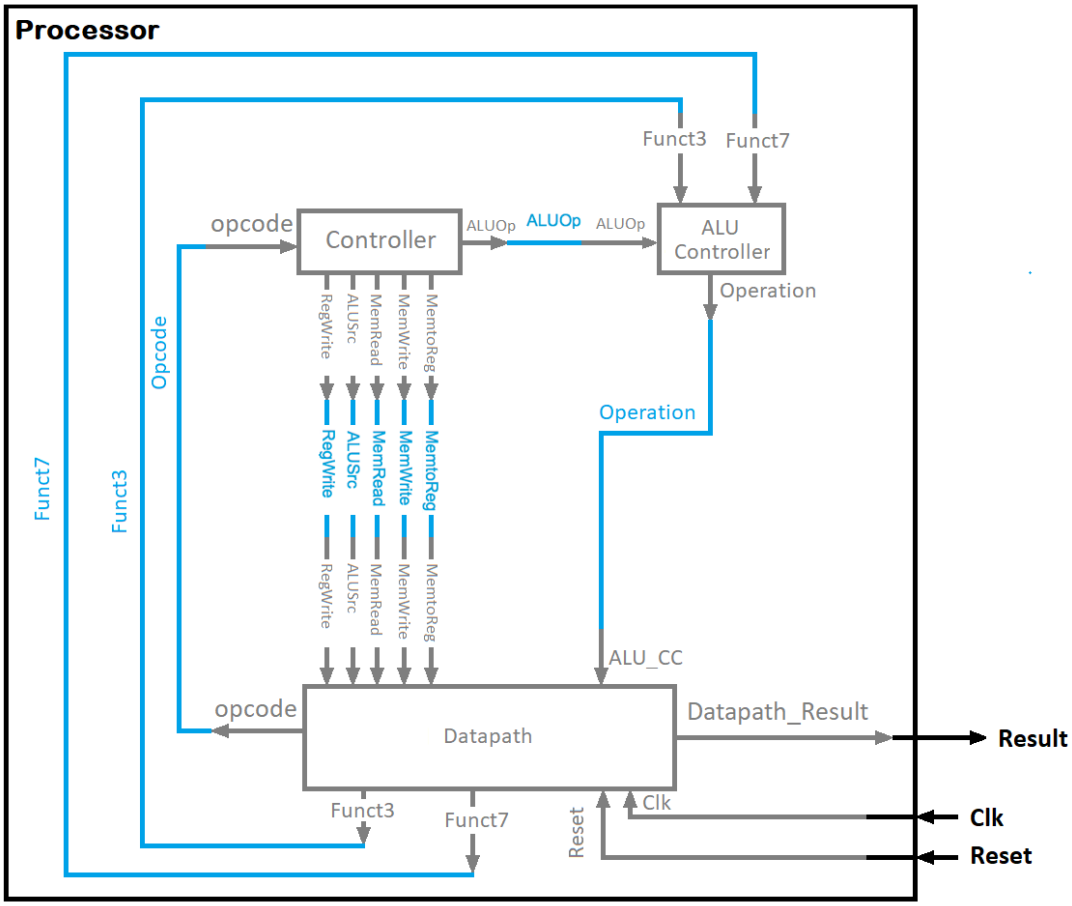


Figure 1 : Processor.

We have designed the Datapath in Lab6. The Controller takes Opcode from the Datapath and pro-

duce five control signals (MemtoReg, MemWrite, MemRead, ALUSrc, RegWrite) and a 2-bits ALUOp. The ALUController takes Funct3 and Funct7 from the Datapath and ALUOp from the Controller and produces a 4-bits operation input (ALUCC) for the Datapath.
In the first part of this lab, we will design two remained sub-modules (Controller and ALU Controller). Then in the second part, we will use these submodules and the datapath to complete the processor.

Before starting the design process, please review the instruction set we have already implemented:

Table 1 : Instruction Set.

Funct7			Funct3		Opcode		
Imm[11:5]	Rs2	Rs1	010	Imm[4:0]	0100011	SW	S-type
Imm[11:0]		Rs1	010	rd	0000011	LW	I-type
Imm[11:0]		Rs1	000	rd	0010011	ADDI	
Imm[11:0]		Rs1	010	rd	0010011	SLTI	
Imm[11:0]		Rs1	100	rd	0010011	NORI	
Imm[11:0]		Rs1	110	rd	0010011	ORI	
Imm[11:0]		Rs1	111	rd	0010011	ANDI	
0000000	Rs2	Rs1	000	rd	0110011	ADD	R-type
0100000	Rs2	Rs1	000	rd	0110011	SUB	
0000000	Rs2	Rs1	010	rd	0110011	SLT	
0000000	Rs2	Rs1	100	rd	0110011	NOR	
0000000	Rs2	Rs1	110	rd	0110011	OR	
0000000	Rs2	Rs1	111	rd	0110011	AND	

1 Lower Level Modules

1.1 Controller

From figure 1 you see the control signals which we used in the datapath design are driven by a module named "Controller". The input to this module is a 7-bits Opcode field of the instruction (comes from the Datapath). The outputs of the control unit are:

- Two signals that are used to control multiplexers in the Datapath (ALUSrc and MemtoReg)
- Three signals for controlling reads and writes in the register file and data memory in the Datapath (RegWrite, MemRead, and MemWrite)
- A 2-bits control signal for the ALUController unit (ALUOp)

Here is the block diagram of the Controller.

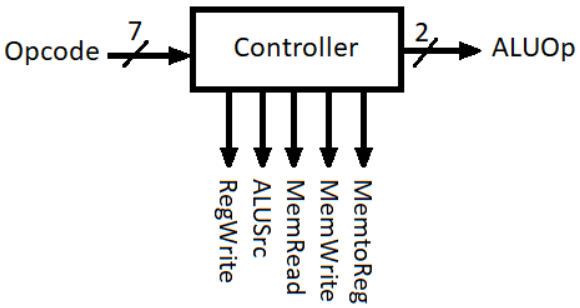


Figure 2 : Controller.

With the information in Table 1, we can design the Controller.

- The MemtoReg signal is '0' for all instructions except for the "Load" instruction
- The MemWrite signal is '0' for all instructions except for the "Store" instruction
- The MemRead is also '0' for all instructions except for the "Load" instruction
- The ALUSrc is '1' when the opcode is "0010011" or "0000011" or "0100011". For these instructions, source B operand comes from the imm_Gen. For other instructions with opcode "0110011", ALUSrc is '0' Because both of the Source operands of the ALU come from the register file.
- RegWrite is '1' except for the "Store" instructions. This is because for all of these instructions we want to write back the result to the register file.
- The ALUOp(1 downto 0) is "10" when the opcode is "0110011" and "00" when the opcode is "0010011". It is also 01 for the "Load" and "Store" instructions with opcodes "0000011" and "0100011" respectively.

Table 1 : Control Signals.

		Opcode			
		0110011	0010011	0000011	0100011
		AND, OR, ADD, SUB, SLT, NOR	ANDI, ORI, ADDI, SLTI, NORI	LW	SW
Control Signals	MemtoReg	0	0	1	0
	MemWrite	0	0	0	1
	MemRead	0	0	1	0
	ALUSrc	0	1	1	1
	Regwrite	1	1	1	0
	ALUOp	10	00	01	01

Use this code for the Module part of the Controller.

Code 1: Controller.

```
`timescale 1ns / 1ps

// Module definition
module Controller (
    Opcode,
    ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite,
    ALUOp
    );

   // Define the input and output signals


  // Define the Controller modules behavior


endmodule //Controller
```

1.2 ALUController

The inputs to the ALUController are the ALUOp, funct3, and funct7. ALUOp comes from the Controller and funct3 and funct7 come from the Datapath. The output of the ALUController is the 4-bits operation code which goes to the ALU_CC input of the datapath. You see the block diagram of the ALUController in figure 3.

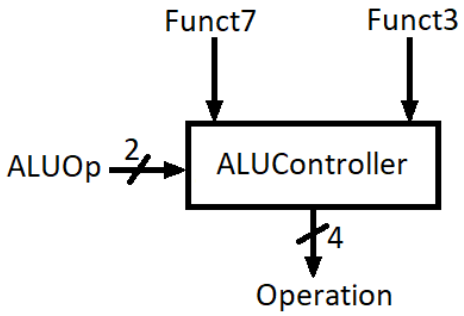


Figure 3 : ALUController.

Table 2 shows the list of the instructions and their operation code.

Table 2 : Instruction and the operation code.

operation	Operation code
AND, ANDI	0000
OR, ORI	0001
ADD, ADDI, SW, LW	0010
SUB	0110
SLT, SLTI	0111
NOR, NORI	1100

We need to find a relation between the inputs and the output of the ALUController. Table 3 shows this relation.

Table 3 : The truth table for the 4 operation code.

	Funct7	Funct3	ALUop	Operation			
AND	0000000	111	10	0	0	0	0
OR	0000000	110	10	0	0	0	1
NOR	0000000	100	10	1	1	0	0
SLT	0000000	010	10	0	1	1	1
ADD	0000000	000	10	0	0	1	0
SUB	0100000	000	10	0	1	1	0
ANDI	-	111	00	0	0	0	0
ORI	-	110	00	0	0	0	1
NORI	-	100	00	1	1	0	0
SLTI	-	010	00	0	1	1	1
ADDI	-	000	00	0	0	1	0
LW	-	010	01	0	0	1	0
SW	-	010	01	0	0	1	0

"-" in Table 3 means there could be any values there.

Here you see the equation for the first bit of the operation as an example:

```
assign Operation[0] = 1'b1 ? ((Funct3 == 3'b110) | ((Funct3 == 3'b010) & (ALUOp[0] == 1'b0))) : 1'b0;
```

Here is the sample code for the ALU Controller.

Code 2: ALUController.

```
`timescale 1ns / 1ps

// Module definition
module ALUController (
    ALUOp, Funct7, Funct3, Operation
    );

   // Define the input and output signals


  // Define the ALUController modules behavior


endmodule //ALUController
```

1.3 Datapath

This is the Datapath module we have already designed in the Lab6.

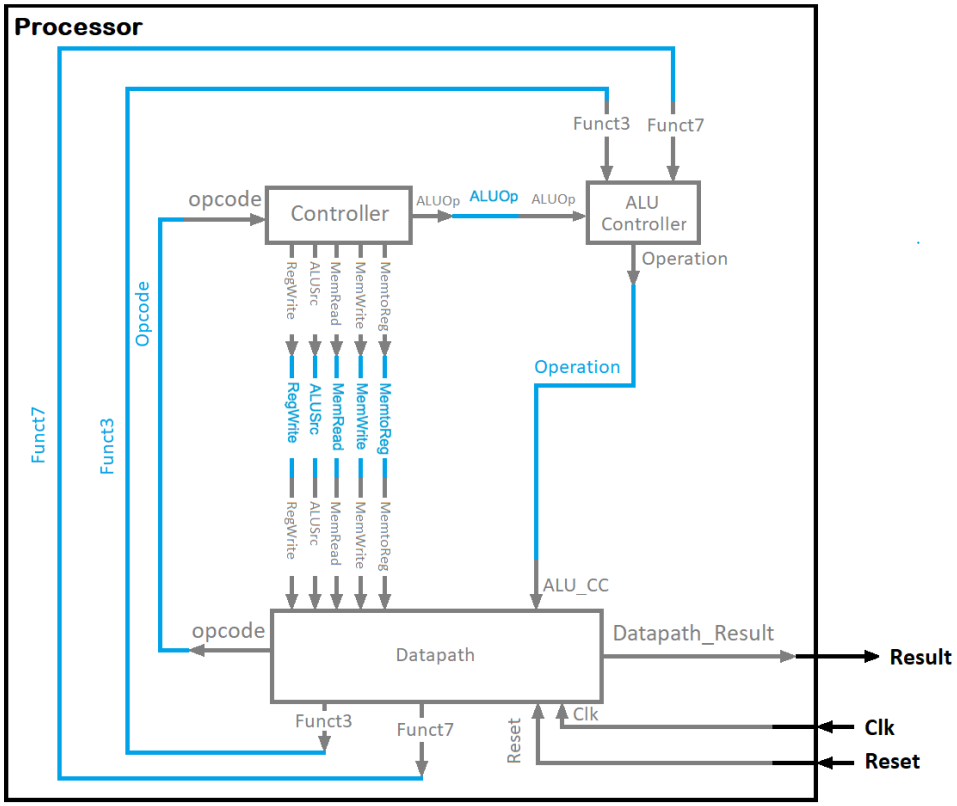
Code 3: Datapath.

```
`timescale 1ns / 1ps

module data_path #(
    parameter PC_W = 8,       // Program Counter
    parameter INS_W = 32,     // Instruction Width
    parameter RF_ADDRESS = 5, // Register File Address
    parameter DATA_W = 32,    // Data WriteData
    parameter DM_ADDRESS = 9, // Data Memory Address
    parameter ALU_CC_W = 4    // ALU Control Code Width
 )(
    input                   clk ,
    input                   reset,
    input                   reg_write,
    input                   mem2reg,
    input                   alu_src,
    input                   mem_write,
    input                   mem_read,
    input  [ALU_CC_W-1:0]   alu_cc,
    output          [6:0]   opcode,
    output          [6:0]   funct7,
    output          [2:0]   funct3,
    output   [DATA_W-1:2]   alu_result);

wire [PC_W-1:0]   pc, pc_next;
wire [INS_W-1:0]  instruction;
wire [DATA_W-1:0] l_alu_result;
wire [DATA_W-1:0] reg1, reg2;
wire [DATA_W-1:0] l_read_data;
wire [DATA_W-1:0] l_reg_wr_data;
wire [DATA_W-1:0] srcb ;
wire [DATA_W-1:0] extimm;

//  next pc
    assign pc_next = pc + 4;
    FlipFlop  pcreg(clk, reset, pc_next, pc);

//  instruction memory
    InstMem instruction_mem (pc, instruction);

    assign opcode = instruction[6:0];
    assign funct7 = instruction[31:25];
    assign funct3 = instruction[14:12];

//  register file
      RegFile rf(
        .clk          ( clk                ),
        .reset        ( reset              ),
        .rg_wrt_en    ( reg_write          ),
        .rg_wrt_addr  ( instruction[11:7]  ),
        .rg_rd_addr1  ( instruction[19:15] ),
        .rg_rd_addr2  ( instruction[24:20] ),
        .rg_wrt_data  ( l_reg_wr_data      ),
        .rg_rd_data1  ( reg1               ),
        .rg_rd_data2  ( reg2               ));

    assign l_reg_wr_data = mem2reg ? l_read_data : l_alu_result;

//  sign extend
    ImmGen ext_imm (instruction, extimm);

//  alu
    assign srcb = alu_src ? extimm : reg2;
    alu_32 alu_module( .A_in(reg1), .B_in(srcb), .ALU_Sel(alu_cc),
    .ALU_Out(l_alu_result), .Carry_Out(), .Zero(), .Overflow());

    assign alu_result = l_alu_result;

//  data memory
    DataMem data_mem(.addr(l_alu_result[DM_ADDRESS-1:0]), .read_data(l_read_data),
    .write_data(reg2), .MemWrite(mem_write), .MemRead(mem_read) );
endmodule
```

2 Higher Level module

2.1 Processor

We have designed the three sub-modules of the processor (Datapath, Controller, ALUController) and now we want to use them to design a single cycle processor. Create a new source and define these three components and connect them to make the processor.
Here again, you see the Processor. Blue lines are some wires which we used to connect the submodules. Define these blue lines as signals and connect the components to complete the Processor.



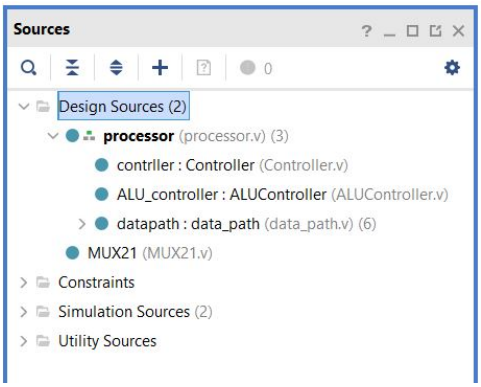
Use this code for the Module part of the Processor.

Code 4: processor.

```
‘timescale 1ns / 1ps

module processor
(
   input clk, reset,

   output [31:0] Result
);


// Define the input and output signals

// Define the processor modules behavior


endmodule //processor
```

After finishing Processor design the sources window in the Vivado should look like this.

Important: we want you to have separate source files for each of the submodules.



2.2 Test the Processor

To test our code we only need to provide 2 inputs (clk and reset) and then read the outputs to see if they are as expected or not. In every rising edge of the clock, the processor will read a new instruction from the instruction memory and send the result to the output. Use the code below as test bench.

Code 5: tb_processor.

```
`timescale 1ns / 1ps

module tb_processor();

/** Clock & reset **/
reg clk , rst ;
wire [31:0] tb_Result;

processor processor_inst(
    .clk        (clk),
    .reset      (rst),
    .Result  (tb_Result)
);

always begin
#10;
clk = ~clk;
end

initial begin
clk = 0;
@( posedge clk );
    rst = 1;
@( posedge clk );
    rst = 0;
end

integer point =0;

always @(*)
begin

    #20;
    if (tb_Result == 32'h00000000) // and
    begin
        point = point + 1;
    end

    #20;
    if (tb_Result == 32'h00000001) // addi
    begin
```

```
    point = point + 1;
end

#20;
if (tb_Result == 32'h00000002) // addi
begin
    point = point + 1;
end

#20;
if (tb_Result == 32'h00000004) // addi
begin
    point = point + 1;
end;

#20;
if (tb_Result == 32'h00000005) // addi
begin
    point = point + 1;
end;

#20;
if (tb_Result == 32'h00000007) // addi
begin
    point = point + 1;
end

#20;
if (tb_Result == 32'h00000008) //addi
begin
    point = point + 1;
end

# 20;
if (tb_Result == 32'h0000000b)// addi
begin
    point = point + 1;
end

# 20;
if (tb_Result == 32'h00000003) // add
begin
   point = point + 1;
end

# 20;
if (tb_Result == 32'hfffffffe) // sub
begin
   point = point + 1;
end;

# 20;
if (tb_Result == 32'h00000000) // and
begin
    point = point + 1;
end;

# 20;
if (tb_Result == 32'h00000005) // or
begin
    point = point + 1;
end;

# 20;
if (tb_Result == 32'h00000001)// SLT
begin
    point = point + 1;
end;

#20;
if (tb_Result == 32'hfffffff4) // NOR
begin
    point = point + 1;
end

#20;
if (tb_Result == 32'h000004D2) // andi
begin
   point = point + 1;
end
```

```
    #20;
    if (tb_Result == 32'hfffff8d7) // ori
    begin
       point = point + 1;
    end

    #20;
    if (tb_Result == 32'h00000001) // SLT
    begin
       point = point + 1;
    end;


    # 20;
    if (tb_Result == 32'hfffffb2c) // nori
    begin
        point = point + 1;
    end;

    # 20;
    if (tb_Result == 32'h00000030) // sw
    begin
        point = point + 1;
    end;

    # 20;
    if (tb_Result == 32'h00000030) // lw
    begin
        point = point + 1;
    end;



    $display("%s%d","The number of correct test cases is:" , point);


end

initial begin
#430;
$finish ;

end

endmodule
```

Check the output (Result) to see if they are correct. Put a screenshot of the wave in your report.

3 Assignment Deliverables

Your submission should include the following:

- Block designs and testbenche:
  (tb_processor.v, processor.v, Controller.v, ALUController.v, Datapath.v, FlipFlop.v, HA.v (if you designed this module), Instr_mem.v, RegFile.v, Imm_Gen.v, Mux.v, ALU.v, Data_mem.v)
- A report in **pdf** format.

Note1: Compress all files (.v files + report) into zip and upload to the **BB** before deadline.
Note2: Use the code samples that are given in the lab description. **The Module part of your code should exactly look like the code sample otherwise your code will not be graded**.