Total points assigned to all lab 6 assignments- 10 points

Lab 6(Including Prelab and Inlab) is all online sections! You will need to do it at home

0. Introduction

For the following labs, we are going to design a simple digital system which is a demo of modern computer with RISC architecture. Lab 6 is the #2 checkpoint towards final project
The Goal of Lab 6 is to understand the Single Cycle Datapath RISC Architecture and I-types and R-types instructions which we will be implementing in Final Project.

Before you read
ALL the content in Introduction are based on Textbook Chapter 6, R Types(6.3.1) and I types(6.3.2) Instructions, and Chapter 7, Single Cycle Datapath(7.3.1), the purpose of this instruction is to help you understand the content in textbook. So please read the above three sections(6.3.1, 6.3.2, 7.3.1) in textbook! Note that this material should not be fundamentally new to you as it is a material that is typically covered in the Computer Architecture class which is a prerequisite for DSD. Notice also that below we modify the instruction formatting a little bit to make it easier for you to understand and implement.

We will need to use **single datapath RISC architecture** to achieve 4 types of operations, **ADD**, **SUB**, **LW**(Load) and **SW**(Store).
These 4 operations are represented into 2 types of machine instructions, R- and I-type, in modern RISC computer architecture.

A. R-type Instructions

ADD and **SUB** are R-type Instructions. Figure 6.5 (below) is the format of R-type instruction. All instructions have a total of 32 bits.

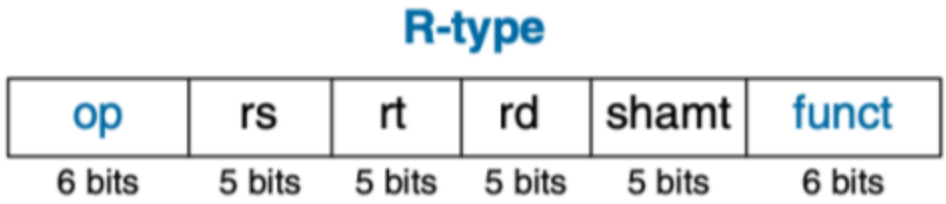


Figure 6.5 R-type machine instruction format

The fields **rs** (instr[25:21]), **rt**(instr[20:16]), **rd**(instr[15:11]) are pointers to operands. The field **op** (instr[31:26]) is used for operations. In this Lab, we will not use the **shamp** and **funct** fields so we simply set their bits as logic 0(instr[5:0] = 6'b000000, instr[10:6] = 5'b00000).
[Important]. The values stored in **rs**, **rt**, and **rd** are **address are 5 bit fields that address one of 32 registers in the register file (R)**.
op in R-type instructions tells us to **ADD** or **SUB**. An example will be given after we explain the **single datapath RISC architecture**.

B. I-type Instruction

I-types instructions can be used to achieve **LW** and **SW** operations.
The Figure 6.8 is the format of I-type instruction.

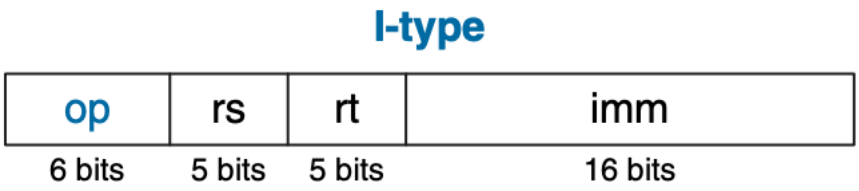


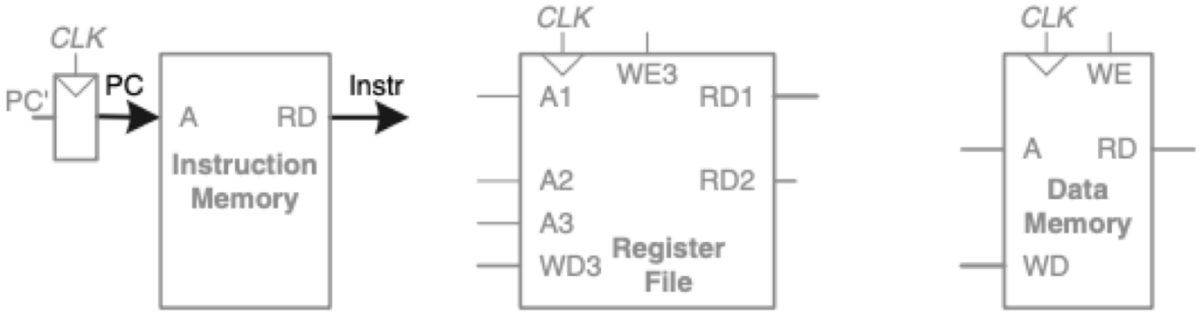
Figure 6.8 I-type instruction format

Within these **32** bits, the field **rs** (instr[25:21]), **rt**(instr[20:16]), **imm**(instr[15:0]) are used to point to operands. The field **op** (instr[31:26]) is used for operations.

[Important]The values stored in **rs**, **rt** also point to registers **in the RF**.
Op in I-type instructions tells us to **LW** or **SW**. An example will be given after we explain the **single datapath RISC architecture**.

1. Introduction of Single Datapath RISC Architecture

The general components in RISC architecture are shown in below:



PC(Program Counter) contains the address of the instruction to execute.
Instruction Memory stores all the instructions. Port **A** requires the address of instructions. Port **RD** reades the instruction data. As shown here, **PC** connect to **A** and the **Instruction Memory** reads out the instruction from **RD**, labelled **Instr.**

Register File can read 2 registers and write 1 register simultaneously. For **Register File**, Port **A1, A2, A3** are the input ports for address. Port **RD1, RD2** are the output ports for read registers pointed by A1 and A2, respectively. **WD3** is the input port for writing data into a RF at the rising edge of the clock. **WE3** is the write enable signal port.
Data Memory stores the data. Port **A** is the address input port. Port **RD** is the data output port. Port **WD** is the data input port. Port **WE** is the write enable port.

[Important] The above figure only shows the main component of a RISC architecture. A complete single datapath RISC architecture is given in **Figure 7.9**. below.

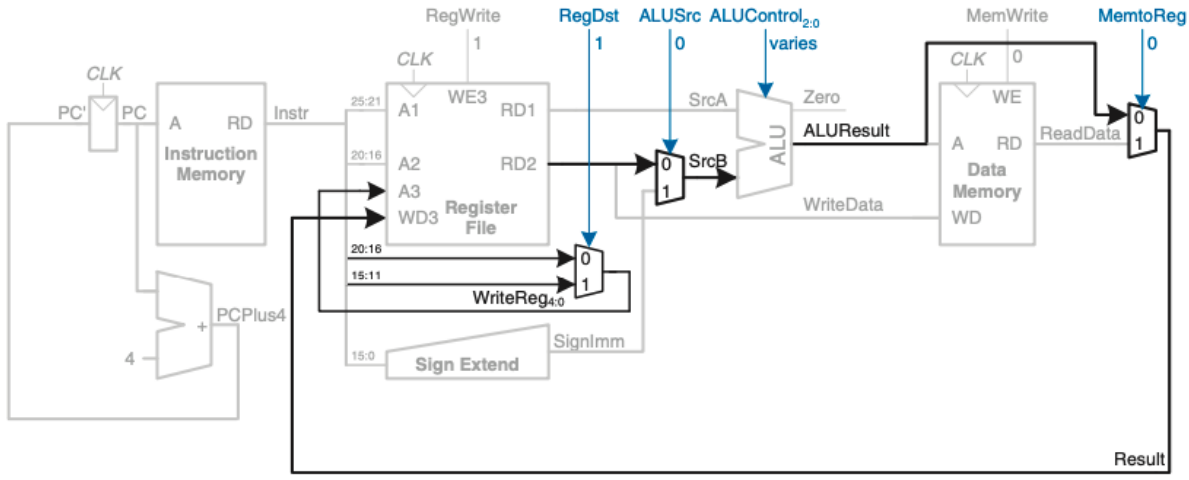


Figure 7.9 Datapath enhancements for R-type instruction

3. How LW, SW, ADD, SUB works in RISC Architecture

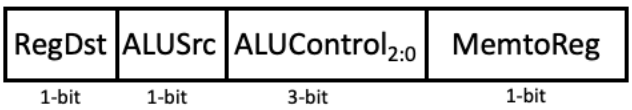
Please refer to Chapter 7, Single-cycle Datapath(7.3.1) in textbook to see how **LW**, **SW**, **ADD** and **SUB** work in the single path RISC architecture. Here we give you a brief description on how those operations work in the RISC architecture(**For details, you still need to go over the chapters/sections in textbook stated in the beginning)**

***Note**: To better describe, data in memory is represented as Memory[address]. E.g. the 1st data in Register File is Register_File[0], the second data in Data memory is Data_Memory[1].*

LW and **SW** achieved by I-type instruction, are mainly data communications between Register File and Data Memory. For **LW, rs** (instr[25:21]) are inputted in port A1 of Register File and **Register_File[rs]**(32-bit) is read out from port RD1, at the same time, **imm**(instr[15:0]) is extended through Sign Extender and **SignImm**(32-bit) is obtained. **Register_File[rs]** and **SignImm** are added together through ALU to get the memory address for data memory. So you can see the ALU result, in this case, is inputted to port A(see **Figure 7.6 in Textbook**) and the Data Memory reads out **Data_Memory[Register_File[rs] + SignImm]** from port RD of data memory and inputted in port WD3 of register file. Meanwhile **rt**(instr[20:16]) is inputted in port A3 as the destination address for register file. So **Data_Memory[Register_File[rs] + SignImm]** is loaded from Data Memory to **Register_File[rt]**. For **SW**, it is very similar, please go through the section 7.3.1.

ADD and **SUB** achieved by R-type instruction, are mainly operated for the registers in Register File. For **ADD**, **rs** (instr[25:21]) is inputted in port A1 of Register File, **rt**(instr[20:16]) is inputted in port A2 of Register File. So **Register_File[rs]** and **Register_File[rt]** are read out from port RD1 and RD2 respectively to ALU to finish the addition. The ALUResult will be inputted back to Register File(Here MemtoReg = 0) from port WD3. At the same time, **rd**(instr[15:11]) is inputted in port A3 of Register File. So **Register_File[rs]+Register_File[rt]** is stored in **Register_File[rd].** For **SUB**, it is very similar, please go through the section 7.3.1.

The only thing we modify for our Lab, which is a little bit different from the textbook, is the **op** field for I-types and R-types instruction. Here to control the 3 MUXs and ALU(RegDst, ALUSrc, $ALUControl_{2:0}$, MemtoReg), we use **op**. The fields of **op** is shown below.



Now we give some examples for you how to represent our operations(**LW, SW, ADD, SUB)** into I-type and R-type instruction and how the instruction work in RISC datapth architecture.

***Note**: Assume all the data in Register File are initialized as 0.*

1. **LW**(Load) in I-type instruction

Load Data_Memory[5] to Register_File[1]. The base address **rs** = 0, offset **imm** = 5. So Address of Data_Memory = Register_File[rs] + imm = 5

The address of Register File **rt** = 1

In this case, RegDst = 0, ALUSrc = 1, ALUControl[2:0] = 010(ADD), MemtoReg = 1, so:
op = 6'b010101
rs = 5'd0 = 5'b00000
rt = 5'd1 = 5'b00001
imm = 16'd5 = 16'b0000_0000_0000_0101
So the I-type instruction should be: 32'b010101_00000_00001_0000_0000_0000_0101.

2. **ADD** in R-type instruction

ADD Register_File[1] and Register_File[2], and store the result to Register_File[6]. The first Address **rs** = 1, The second address **rt** = 2, the destination address **rd** = 6;
In this case, RegDst = 1, ALUSrc = 0, ALUControl[2:0] = 010, MemtoReg = 0, so:
op = 6'b100100
rs = 5'b00001
rt = 5'b00010
rd = 5'b00110
rest = 11'b00000_000000(**shamp** and **funct** are set as logic 0)
So the R-type instruction should be: 32'b100100_00001_00010_00110_00000_000000.

3. SW(Store) in I-type instruction

Store Register_File[6] to Data_Memory[2]. The base address **rs** = 0, offset **imm** = 2. So Address of Data_Memory = Register_File[rs] + imm = 2.
The address of Register file **rt** = 6

In this case, RegDst = 0, ALUSrc = 1, ALUControl[2:0] = 010, MemtoReg = 0, so:
op = 6'b010100
rs = 5'b00000
rt = 5'b00110
imm = 16'b0000_0000_0000_.0010
So the I-type instruction should be: 32'b010100_00000_00110_0000_0000_0000_0010.

4. SUB in R-type instruction

For SUB operation, it is similar to ADD in R-type instruction. Notice that ALUControl for SUB should be 110 for ALU to do substraction.

Assignment in Inlab 6

In Prelab6 assignment, we translate LW, SW, ADD, SUB operations into I-types and R-types instruction. In Inlab6 we are going implement 2 key component of single path RISC architecture, **Register File** and **ALU** by writing the System Verilog code for 2 modules, **Register File** and **ALU.** And you need to write testbench to verify your Register File and ALU.

1. Register File

The module of register file should look like this,

```
module register_file(input logic clk, rst,
input logic[4:0] A1, A2, A3, //A1,A2,A3 are the address input
input logic[31:0] WD3, //data from data memory
input logic WE3, //WE3 = 1, write register file
output logic[31:0] RD1, //output port one for register file, lw operation
output logic[31:0] RD2 //output port two for register file, add or sub operation
);

//.......

endmodule
```

Notice for the Register File used here, there are totally 32 32-bit registers, which means the width(number of bits for each register) and depth(number of registers) are both 32.

You should initialize the Register File first. There are a lot of ways to initialize a register file. Here we provide the simplest way to initialize it, which is hardcode it. Below is an example of how to initialize the register file:

```
//suppose a register file with width = 8, depth = 3
reg[7:0] registers[2:0];

always@(posedge clk or negedge rst)
begin
   if(!rst) begin
      for(int i = 0; i < 3; i++) begin
         registers[i] <= i;
      end
   end
   else begin
   //write other logics here

   end
end
```

Please follow the example and initialize the Register File as Register_File[i] = i. (It means for the 1st register, the value it stores is 1, for the 3rd register, the value it stores is 3, etc...)

2. ALU

The module of ALU should look like this:

```
module ALU(
input logic[31:0] SrcA,
input logic[31:0] SrcB,
input logic[2:0] ALUControl,
output logic[31:0] ALUResult
);

//.....

endmodule
```

Notice that you only need to operate ADD and SUB for ALU.

ALUControl = 010, do ADD, ALUControl = 110, do SUB, (see Table 5.1 in Textbook)

3. Testbench

Notice that there are 2 testbenches you need to write, one for Register File, one for ALU.

a. Testbench for Register File

In testbench for Register File, you should be able to show that(There is not specific input values for A1, A2, A3, WD3, but you need to explain in the report what are these values in your testbench)

While Reading(WE3 = 0)

RD1 = Register_File[A1], RD2 = Register_File[A2],

While Writing(WE3 = 1)

Register_File[A3] = WD3.

Put your waveforms result in your report, also explain during reading, what values are read out from Register File, during writing, what values is written into the Register File

b. Testbench for ALU

In testbench for ALU, there is no specific input values and output values required. But you should be able to perform one ADD and one SUB (Also there is not specific input values for SrcA and SrcB, but you need to explain in the report what are these values in your testbench, besides make sure the SrcA and SrcB is not too large to cause overflow).

Put your waveforms result in your report, also explain it(What is your input values, what is the result)

What to turn in

In the pdf, you will need to include the following sections(In order);

1. **Attach your Prelab report as 1st section**
   Attach your Waveform Results for Register File and ALU and explain it(What are your input values and what are your output values)
   Input values: A1, A2, A3, WE3, WD3 for Register File, SrcA , SrcB, ALUControl for ALU
   Output values: RD1, RD2, Register_File[A3] for Register File, ALUResult for ALU
2. **System Verilog code for your 2 testbenches.sv, one for Register File, one for ALU**

Notice you don't need to submit other files for your codes. Your code needs to be attached as sections in your InLab report.

Appendix

How to check if the writing is successful in Register File, or to say, how to check Register_File[A3]? -> Try to add an output in your module, connect this output to Register_File[A3]:

```
module register_file(
  input logic clk, rst,
  input logic[4:0] A1, A2, A3,
  input logic[31:0] WD3,
  input logic WE3,
  output logic[31:0] RD1,
  output logic[31:0] RD2,
  output logic[31:0] probe //this probe signal is used to check Register_File[A3]
);

  //here you define your register file.....
  ..........

  //you can assign probe to any register in register file to see the value of the register!
  assign probe = Register_File[A3];

  ....
```

1. Design, and Simulation

[Picking up from Lab 6]

In Lab 6, we have designed "**Register File**" and "**ALU**", Now we need to implement the rest components in **Figure 7.9** to create a complete single datapath RISC architecture.

1. Data memory

```
module data_memory(
input logic clk,rst,
input logic[31:0] A, //address
input logic[31:0] WD, //input data
input logic WE, //enable input
output logic[31:0] RD,
output logic[31:0] prode //to check the data in data memory
); //output data

//your design...

endmodule
```

Note 1: For the size of the data memory, please set it the same size of Register File: which means there are totally 32 32-bit registers, or to say, the width and depth are both 32.

Note 2: Also initialize your data memory as **data_memory[i] = i**.(In terms of How to initialize it, it is the same as initialization of register file as shown below). Also again, initialize your Register File as **Register_File[i] = i**(Same as Lab 6)

```
//suppose a register file with width = 8, depth = 3
reg[7:0] registers[2:0];

always@(posedge clk or negedge rst)
begin
   if(!rst) begin
      for(int i = 0; i < 3; i++) begin
         registers[i] <= i;
      end
   end
   else begin
   //write other logics here

   end
end
```

Note 3: For **prode** output usage, Please check the appendix in Lab 6.

Note 4: Please also add **prode** signal for **Register File** module so we can check the register value in **Register File** too!

2. MUX

There are three MUXs here. MUX_MemtoReg, MUX_ALUSrc, MUX_RegDst

They are very similar. Here is an example of module definition of MUX_MemtoReg:

```
module MUX_MemtoReg(
input logic MemtoReg,
input logic[31:0] ALUResult,
input logic[31:0] RD, //from data memory
output logic[31:0] MemtoReg_out
);

//your logic here

endmodule
```

3. Sign Extend

```
module sign_extend(
input logic[15:0] Imm,
output logic[31:0] SignImm
);

//you logic

endmodule
```

4. Instruction memory

```
module instruction_memory(
input logic clk, rst,
input logic[2:0] A, //address
output logic[31:0] RD //output data, instruction
);
reg[31:0] ins_regs[4:0];
always@(posedge clk or negedge rst)
begin
    if(!rst) begin
    //I-type instruction = op[5:0] + rs[4:0] + rt[4:0] + imm[15:0]
    //R-type instruction = op[5:0] + rs[4:0] + rt[4:0] + rd[4:0] + rest[10:0]
        //set Inst[0] as 0, so when A = 0, the instruction does nothing
        ins_regs[0] <= 0;
        //1st Inst: load data_regs[5] to rf_regs[1], rs = 2, rt = 1, imm = 3 (result: rf_regs[1] = 5)
        ins_regs[1] <= ...
        //2nd Inst: store rf_reg[1] to data_regs[9],rs = 8, rt = 1, imm = 1 (result: data_regs[9] = 5)
        ins_regs[2] <= ...
        //3rd Inst: ADD rf_regs[3] and rf_regs[4] to rf_regs[1] (result: rf_regs[1] = 3 + 4 = 7)
        ins_regs[3] <= ...
        //4th Inst: SUB rf_reg[10] - rf_regs[8] = to rf_reg[1] (result: rf_regs[1] = 2)
        ins_regs[4] <= ...
    end
    else begin
        //you logic
    end
end
endmodule
```

Here we give you almost the whole part of instruction memory. But you still need to fill up the 4 32-bit instruction codes as it is told in notation, also you need to fill up the logic left blank after “else”.

5. PC

PC is the top module of your design,

```
module PC(
input logic clk, rst,
input logic[2:0] instruction_A, //address for instruction memory
input logic RegWrite, MemWrite, //write enable signals for register_file and data_memory
output logic[31:0] prode_register_file,
output logic[31:0] prode_data_memory
);

//your design here(instantiate each component and connect them)

endmodule
```

6. Simulation

Make PC as your top-level entity. Write the testbench for it and verify the result through Modelsim waveforms. For the waveforms, you need to show the 4 results(preferred in decimal raidx) of the 4 instructions in instruction memory.

2. What to Turn In

2.1. For Lab 7 Inlab Report

- **Due Date:** Lab 7 in-lab report is due in the week of November 15.

- **Points assigned to inlab Assignment:** 10 points for Lab 7 come from inlab assignment report submission

- **Penalty for late submission:** You have one extra week to submit the in-lab report late. But this will attract a penalty of 4 points. So, if you miss your deadline in the week of November 15, you can still submit it a week later, in the week of November 22 but with a penalty.

- **Content of the in-lab report:**

You must submit an electronic copy of the following items (in the correct order, and each part clearly labeled) via Sakai. Submit it to the Lab 7 In-Lab Assignment. All items should all be included in a single file (.pdf).

1. A Screenshot of System Verilog Code for Instruction Memory.
2. A Screenshot of System Verilog Code for Register File.
3. A Screenshot of System Verilog Code for ALU.
4. A Screenshot of System Verilog Code for the 3 MUXs.
5. A Screenshot of System Verilog Code for Sign Extend
6. A Screenshot of System Verilog Code for data memory.
7. A Screenshot of System Verilog Code for Testbench(for PC)
8. A Screenshot of System Verilog Code for PC;
9. Screenshots of your output waveforms in Modelsim, there should be 4 results corresponding to the 4 instructions in instruction memory.
   (Please Set your output result radix as decimal to look clearer. It's fine if you cannot fit in all your results in one screenshot, you can separate them into several screenshots but make sure you provide all the correct results shown in notation)