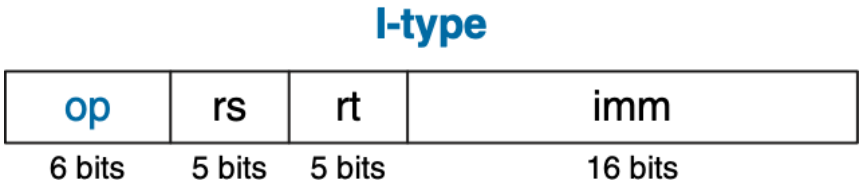
Digital Design and Computer Architecture

Lab ~~7~~ 6: Digital System Design – Single Path RISC Architecture – Part II

The Goal of Lab ~~7~~ 6 is to implement I-type instructions.
We will follow the **single datapath RISC architecture** to achieve 2 types of operations: **LW**, **SW**.

Textbook Reading
Read Chapter 6 on I Type instructions (6.3.2)



and Chapter 7 on Single Cycle Microarchitecture (see the appendix for more information).

To simplify the design, we modify the original RISC instruction **op** field for I-type and R-type instructions. Here, to control the 3 MUXs and ALU (RegDst, ALUSrc, $ALUControl_{2:0}$, MemtoReg), we use **op**. The fields of **op** are shown below.

Examples on how to represent our operations (**LW, SW)** into I-type instructions and how the instructions work in RISC datapath architecture:

Note*: Assume all the data in Register File is 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. **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.

2.1. Lab Tasks

[picking up from lab 5] Based on the "Register File" and "ALU" we implemented in Lab 5, in Lab 6, to achieve I-type instructions, we need to:

1. **Implement the Data Memory**

The module of the RISC data memory should look like this:

```
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
```

Notice for the Data Memory used here, we assume there are totally 256 32-bit registers, which means the width = 32 and the depth = 256.

2. **Initialize the Data memory**

Please initialize the Data Memory similarly to how you initialized the register file for Lab 5, following the example in Appendix 1.3.

3. **Implement MUX**

There are three MUXs here. MUX_MemtoReg, MUX_ALUSrc, MUX_RegDst.

They are very similar. Here is an example of the 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
```

See Figure 7.9 in Appendix 1.0 for more information on these MUXs.

4. **Implement Sign Extend**

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

//you logic

endmodule
```

Sign Extend module is used to extend the sign bit of the 16-bit immediate to make it a 32-bit immediate for the ALU to do operations. See Appendix 1.2. for details.

5. **Implement a top module**

The top module should include: register file, data memory, 3 MUXs, Sign Extend, Display. You can take the top.sv provided in lab 5 as a reference.

Notice that for the Display module, you will only need it for in-lab 6, which means that it is not mandatory for pre-lab 6 implementation/report.

6. **Use ~~Testbench~~ Waveform to verify**

~~Write a testbench for the "top.sv" provided and verify your design in simulation. Follow the "testbench_top.sv" provided in Lab 5 as an example.~~

5

Use waveform file from lab ~~6~~ from Canvas.

Notice: For simulation result, please use the instructions from the above examples, i.e.

Show you correctly load Data_Memory[5] to Register_File[1]. The base address rs = 0, offset imm = 5.

Show you correctly Store Register_File[6] to Data_Memory[2]. The base address rs = 0, offset imm = 2

2.2 What to Turn In?

Prelab – 2 points

- Due Date**:** Check Canvas

Pre-lab report:

You must submit an electronic copy of the following items (in the correct order, and each part clearly labeled) via Canvas. Submit it to the Lab 6 Pre-Lab Assignment. All items should be included in a single file (.pdf).

1. A screenshot of your SystemVerilog Code for Data Memory.
2. A screenshot of your SystemVerilog Code for 3 MUXs.
3. A screenshot of your SystemVerilog Code for Sign Extend.
4. A screenshot of your SystemVerilog Code for Top.
5. Screenshots of your output waveforms using Waveform file ~~in Modelsim~~
   (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)

In-Lab – 8 points

- Due Date: Check Canvas

In-lab report:

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

1. To the pdf document from your prelab assignment, just add a section for the in-lab assignment.
2. Take 2 pictures showing your results of LW, SW on the 7-segment display separately.
   The LW will be loading data_memory[i] to register_file[1], where i is the first (MSB) digit of your board number.
   The SW will be storing register_file[i] to data_memory[2], where i is the last (LSB) digit of your board number.

   E.g. Your board number is 503, for LW, you will load data_memory[5] to register_file[1].
   Make sure you record the board number and explain which registers you use in the in-lab report.

Appendix:

1.0. Introduction - General

For the following labs, we are going to design a simple digital system which is a demo of a modern computer with RISC architecture. Lab 7 is part 2 of the final lab, which is to implement a single datapath RISC architecture (shown below as Figure 7.9) using SystemVerilog.

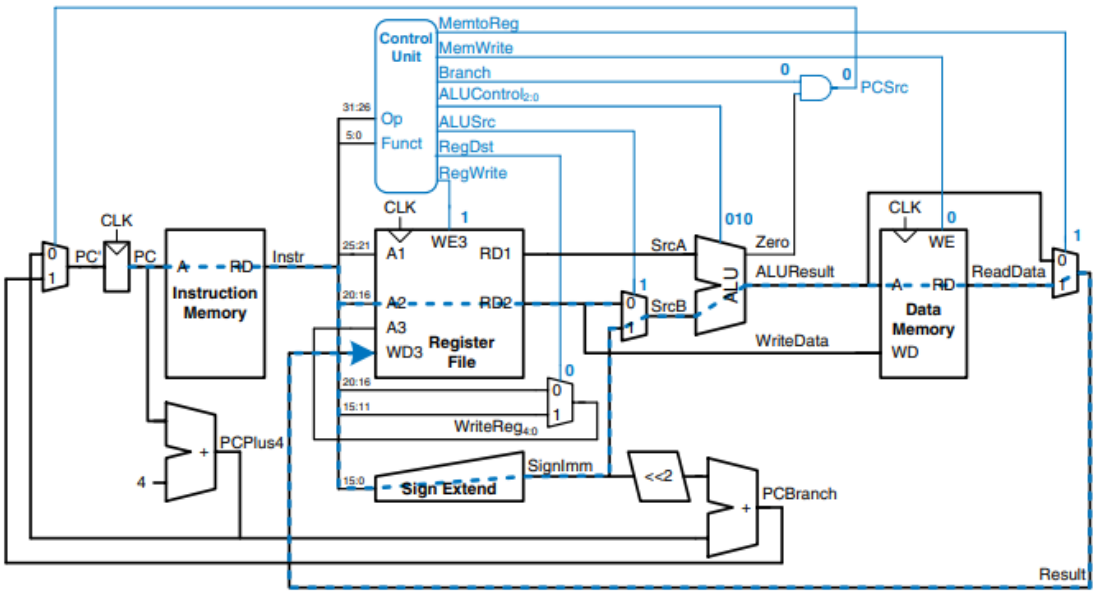


Figure 7.15 Critical path for `lw` instruction

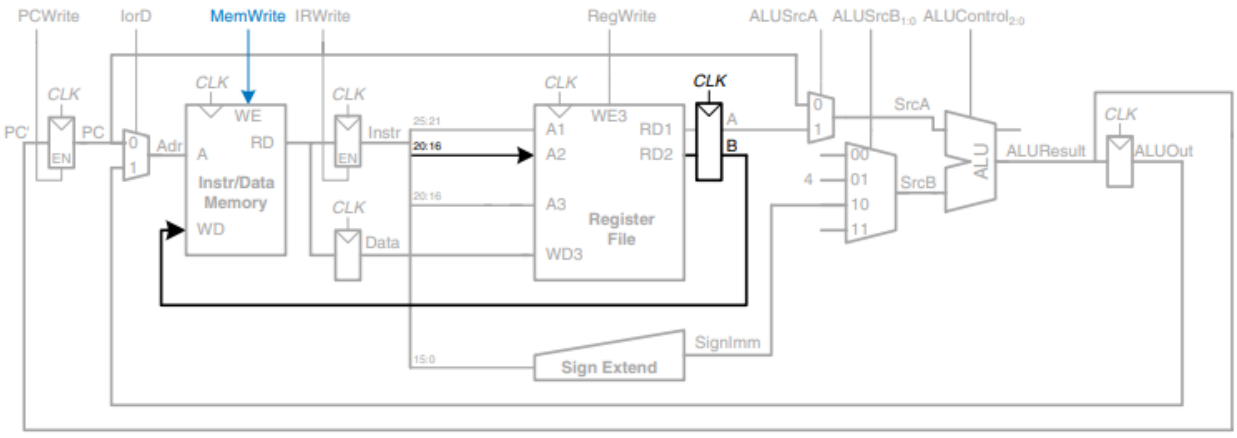


Figure 7.24 Enhanced datapath for `sw` instruction

We use the **single datapath RISC architecture** to achieve 2 types of operations: **LW**, **SW**.
These 2 operations are represented into 32-bit I-type machine code in modern RISC computer architecture.

I-type Instructions

I-types instructions can be used to achieve **LW** and **SW** operations.

Figure 6.8 shows the format of an I-type instruction.

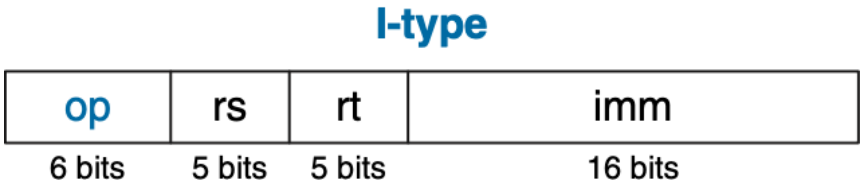


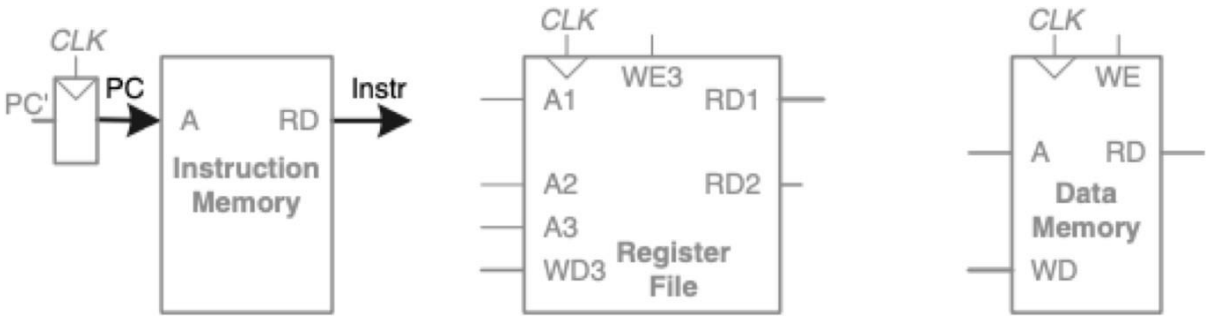
Figure 6.8 I-type instruction format

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**. The example is given in the beginning.

1.1. Introduction - Single Datapath RISC Architecture

The main components in RISC architecture are shown 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** reads the instruction data. As shown here, **PC** connects 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 to 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** above.

1.2. Introduction - How LW, SW works in RISC Architecture

Please refer to Chapter 7, Single-cycle Datapath (7.3.1) in textbook to see how **LW**, **SW** 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 the textbook stated in the beginning).**

***Note**: To clarify, 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], and so on.*

LW and **SW** are I-type instructions and mainly provide data communications between the 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 the 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. The ALU result, in this case, is inputted to port A (see **Figure 7.6 in the 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 the 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 review section 7.3.1 of the textbook.