CO224 Computer Architecture - 2020

Department of Computer Engineering

Lab 5 - Building a Simple Processor

In this lab you will be designing a simple 8-bit single-cycle processor which includes an ALU, a register file and control logic, using Verilog HDL. Follow the guidelines given here to build your processor.

The microarchitecture of a processor is designed based on an Instruction Set. Your processor should implement the instructions `add, sub, and, or, mov, loadi, j` and `beq`. All instructions are of 32-bit fixed length, and should be encoded in the format shown below.

OP-CODE (bits 31-24)	DESTINATION (bits 23-16)	SOURCE 1 (bits 15-8)	SOURCE 2 (bits7-0)

- `OP-CODE` field identifies the instruction's operation. This should be used by the control logic to interpret the remaining fields and derive the control signals.
- `DESTINATION` field specifies either the register to be written to in the register file, or an immediate value (jump or branch target offset).
- `SOURCE 1` field specifies the 1st operand to be read from the register file.
- `SOURCE 2` is the 2nd operand from the register file, or an immediate value (`loadi`).

Here are some examples about the usage and descriptions of these instructions:

`add 4 1 2` (add value in register 2 to value in register 1, and place the result in register 4)
`sub 4 1 2` (subtract value in register 2 from the value in register 1, and place the result in register 4)
`and 4 1 2` (perform bit-wise AND on values in registers 1 and 2, and place the result in register 4)
`or 4 1 2` (perform bit-wise OR on values in registers 1 and 2, and place the result in register 4)
`j 0x02` (jump 2 instructions forward from the next instruction to be executed, by manipulating the Program Counter. Ignore `SOURCE 1` and `SOURCE 2`)
`beq 0xFE 1 2` (if values in registers 1 and 2 are equal, branch 2 instructions backward by manipulating the Program Counter)
`mov 4 1` (copy the value in register 1 to register 4. Ignore `SOURCE 1`)
`loadi 4 0xFF` (load the immediate value `0xFF` to register 4. Ignore `SOURCE 1`)

You will be building your processor in four steps:

- In part 1, you will build an 8-bit ALU which implements all the functional units required to support the instructions `add, sub, and, or, mov,` and `loadi`.
- In part 2, you will implement a simple 8×8 register file.
- In part 3, you will implement the control logic and integrate all the components from parts 1 and 2 together to work as a complete processor.
- In part 4, you will upgrade your processor to support `j` and `beq` instructions.

Part 1 – ALU [25 marks]

At the heart of every computer processor is an Arithmetic Logic Unit (ALU). This is the part of the computer which performs arithmetic and logic operations on numbers, e.g. addition, subtraction, etc. Use Verilog language to implement an 8-bit ALU which can perform **four** different functions to support the instructions `add`, `sub`, `and`, `or`, `mov`, and `loadi` (note: we will <u>not</u> support `j` and `beq` at this stage). Figure 1, below, shows the interfaces of the ALU you will be implementing.

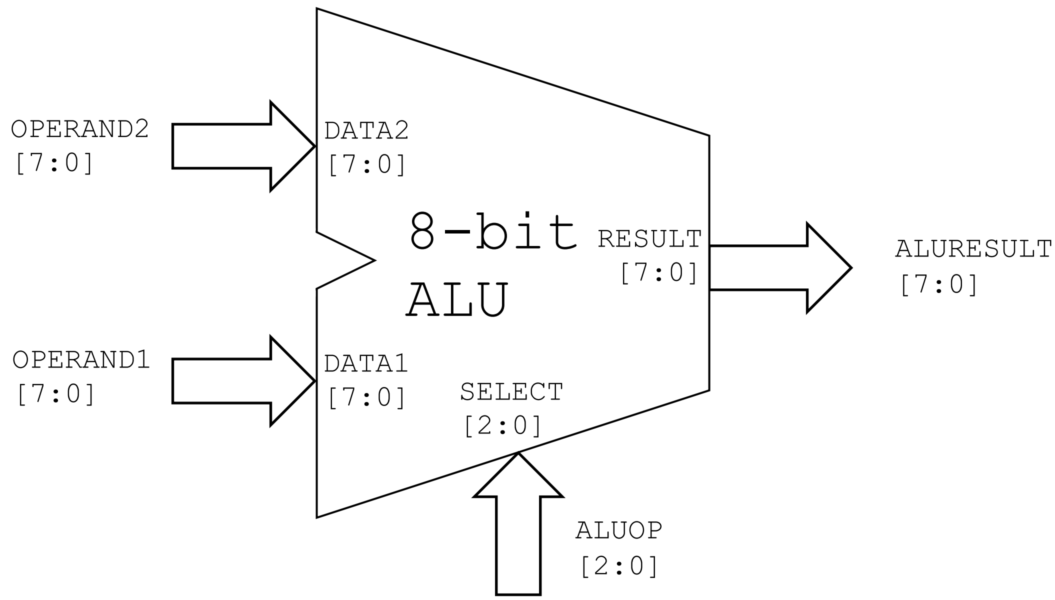

Figure 1: Interfaces of the ALU

The ALU that you are building should work with 8-bit operands. There should be two 8-bit input ports for operands (`DATA1` and `DATA2`), one 8-bit output port (`RESULT`) and one 3-bit control input port (`SELECT`) which should be used to pick the required function inside the ALU out of the available four functions, based on the instruction's `OP-CODE`. (You may notice that two bits are enough for the `SELECT` interface as there are only four function choices. We're reserving the 3rd bit for future use.)

The 3-bit `ALUOP` control signal supplied to the `SELECT` port should be derived from `OP-CODE` using combinational logic, by the control unit. You may define suitable `OP-CODE` values for the given instruction set, and implement an appropriate mapping from `OP-CODE` to the `ALUOP` signal when you design the control logic.

The following module definition gives a template interface for your alu:

```
module alu(DATA1, DATA2, RESULT, SELECT)
```

Make sure you use the same signal and register names as the ones used in this sheet.

The Table below shows the four functions (operations) that your 8-bit ALU should be able to perform.

Table 1: ALU Functions

SELECT	Function	Supported Instructions	Description
000	FORWARD	loadi, mov	(forward DATA2 into RESULT) DATA2 → RESULT
001	ADD	add, sub	(add DATA1 and DATA2) DATA1 + DATA2 → RESULT
010	AND	and	(bitwise AND on DATA1 with DATA2) DATA1 & DATA2 → RESULT
011	OR	Or	(bitwise OR on DATA1 with DATA2) DATA1 | DATA2 → RESULT
1XX	Reserved	-	Reserved for future functional units

FORWARD functional unit should simply copy an operand value from `DATA2` to `RESULT`. This unit will be used by the `loadi` and `mov` instructions to place the respective source operand in the specified destination register.

ADD, AND and OR functional units will use the values in `DATA1` and `DATA2`, perform the corresponding operation, and write the output to `RESULT`. (in part 3 when you implement the control logic, you should include a two's complement unit in the datapath to correctly implement both `add` and `sub` instructions using the same ADD functional unit).

1. Design and implement the *alu* module using Verilog. Include **a lot of comments**. Make sure you properly deal with any unused bit combinations of the `SELECT` port. (Hint: using a *case* structure will make this job easy)
2. Write a testbench and simulate your *alu* module. Test with different combinations of `OPERAND1`, `OPERAND2` and `ALUOP` signal values.
3. Submit a compressed file ***eXXYYY_lab5_part1.zip*** containing your Verilog file with the *alu* module (and any other Verilog files with any sub modules of your design).

Note that any form of plagiarism will result in zero marks for the entire lab.

Part 2 - Register File [25 marks]

Next you should implement a simple 8×8 register file. The purpose of the register file is to store `ALURESULT` values generated by the ALU, and to supply the ALU with operands.

Your register file should be able to store **eight** 8-bit values (`register0 - register7`). It should contain one 8-bit data input port (`IN`) and two 8-bit data output ports (`OUT1` and `OUT2`). To specify which register you are reading or writing with a given port, you must include three address ports (`INADDRESS`, `OUT1ADDRESS`, `OUT2ADDRESS`).

You must also include a control input port `WRITE` to accommodate the `WRITEENABLE` control signal. Since the register file is a sequential unit, you will need `CLOCK` and `RESET` signals for synchronization.

A block diagram of the register file is shown in Figure 2 below.

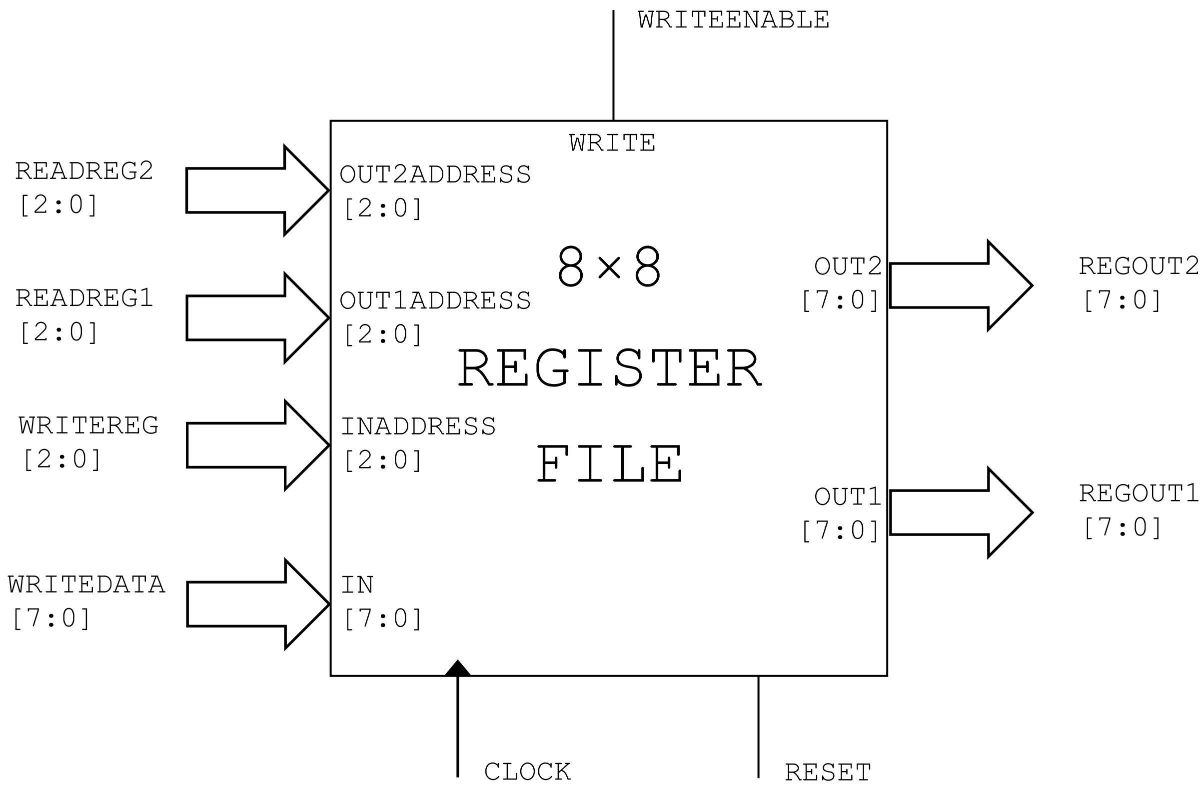

Figure 2: Interfaces of the Register File

The following module definition gives a template interface for your register file:

module reg_file(IN, OUT1, OUT2, INADDRESS, OUT1ADDRESS, OUT2ADDRESS, WRITE, CLK, RESET)

The port `IN` represents the data input, with `INADDRESS` providing the register number to store that data in. The ports `OUT1` and `OUT2` are the dual data outputs, where `OUT1ADDRESS` and `OUT2ADDRESS` respectively provide the register numbers where data should be retrieved from.

Rising edge of `CLOCK` should make the data present on the `IN` port (from the instruction that just completed) to be written to the input register specified by the `INADDRESS`. Just after the same rising edge of `CLOCK`, registers identified by `OUT1ADDRESS` and `OUT2ADDRESS` are read and the values are loaded onto `OUT1` and `OUT2` respectively (for the new instruction fetched at that rising edge).

You need to **pay careful attention to the timings** so that any result sent to the register file by an instruction is available to be used by the next instruction. Test your design thoroughly using several inputs until you make sure the desired behaviors is achieved. Use GTKWave (or any other similar tool that you prefer) to help you visualize the timings.

1. Implement the behavioral model for the Register File. Represent your registers as an array of words and use a structured procedure to update register contents and register file outputs. include **a lot of comments**.
2. Implement a testbench for your Register File, and thoroughly test your design.
3. Submit a compressed file ***eXXYYY_lab5_part2.zip*** containing your Verilog file with the *reg_file* module (and any other Verilog files with any sub modules of your design), and a screenshot of a timing diagram clearly showing the synchronized reading and writing of registers.

Part 3 – Control Unit [25 marks]

Now you should compose a working CPU using your ALU and Register File, supporting the instructions `add`, `sub`, `and`, `or`, `mov`, and `loadi`. To do this, you will need to implement the control logic in a top-level module (you may call this module as *cpu*). Your CPU needs an instruction fetching mechanism and a **Program Counter** (PC) register which points to the next instruction. You may choose to have a *control_unit* module and instantiate it within your top-level *cpu* module, or you may choose to implement all control logic in your *cpu* top-level module itself.

Since we do not have an instruction memory module yet to hold instructions, you can keep the instructions (as an array of hardcoded instruction words) in the testbench file which you use to test your *cpu*. Your instruction fetching mechanism should read the hardcoded instructions from the testbench.

You need combinational control logic to **decode** a fetched instruction, extract the `OP-CODE`, source/destination registers and immediate values. Based on the `OP-CODE`, control signals should be generated and sent to the Register File, ALU and other components appropriately.

For arithmetic instructions (`add` and `sub`), assume that the operands are **signed integers** with negative values presented in Two's Complement format. You will need to perform the Two's Complement operation on the second operand before supplying it to the ALU, in order to support both `add` and `sub` instructions using the same adder functional unit. You will need to use MUXs to achieve the desired control.

Pay careful attention to how you coordinate the timings of instruction fetching, execution and Register File reading/writing. Use GTKWave (or any other similar tool that you prefer) to help you visualize the timings.

You must thoroughly test your *cpu* using several different software programs (instruction sequences). For this, you need to prepare programs as machine code and hardcode them inside your testbench file, one program at a time. Since it is easier to write textual assembly programs (rather than writing machine code), you may use the provided *CO224Assembler* tool to convert textual assembly into machine code. Note that you must add your `OP-CODE` value definitions to the *CO224Assembler.c* file before using it to generate the assembler tool.

An overall block diagram for this simple CPU is provided in Figure 3 (next page). All components except the instruction memory should be housed within your top-level *cpu* module.

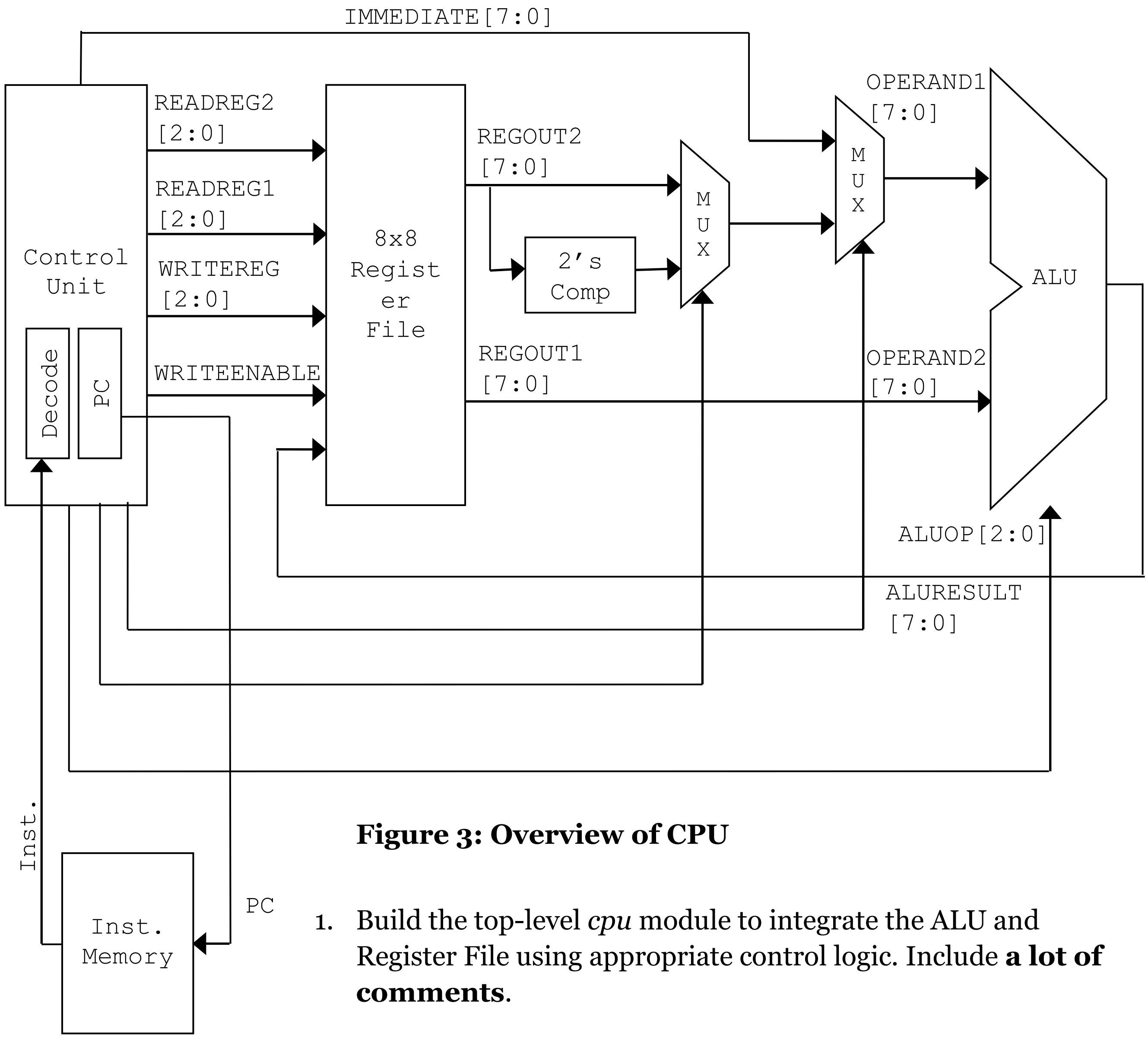

Figure 3: Overview of CPU

1. Build the top-level *cpu* module to integrate the ALU and Register File using appropriate control logic. Include **a lot of comments**.

2. Write a testbench and thoroughly test your completed design. Hardcode your software program (instruction sequence) within the testbench file.

3. Submit a compressed file ***eXXYYY_lab5_part3.zip*** containing your Verilog files with the top-level *cpu module, alu* and *reg_file* modules (and any other Verilog files with any sub modules of your design), testbench, and screenshots of timing diagrams clearly showing the synchronized operation of the datapath and control signals.

Part 4 – Flow Control Instructions [25 marks]

Now that you have a working CPU which supports `add`, `sub`, `and`, `or`, `mov`, and `loadi` instructions, it's time to add microarchitectural support for the flow control instructions `j` and `beq`. For this, you will need to modify your top-level *cpu* and *alu* modules. The *alu* needs an additional output port (`ZERO`) to indicate whether the result of a subtract operation is zero or not, in order to implement the `beq` instruction. The control functionality within the top-level *cpu* module needs to be modified to: manipulate the Program Counter using the immediate jump/branch target offsets provided in `j` and `beq` instructions (you will need to use adders and multiplexers); and generate the additional control signals required.

Before you go and modify your code, it is highly recommended to **draw a complete block diagram** of the datapath and control by extending Figure 3 with the added components. And make sure to keep copies of your original files before modifying.

1. Modify the top-level *cpu* module and *alu* module to support the `j` and `beq` instructions. Include **a lot of comments**.
2. Write a testbench and thoroughly test your upgraded design. Hardcode your software program (instruction sequence) within the testbench file.
3. Submit a compressed file ***eXXYYY_lab5_part4.zip*** containing your Verilog files with the upgraded top-level *cpu module, alu* and *reg_file* modules (and any other Verilog files with any sub modules of your design), testbench, and screenshots of timing diagrams clearly showing the synchronized operation of the datapath and control signals.

Make sure you **add a lot of comments** when coding.

Note that any form of plagiarism will result in zero marks for the entire lab.

Part 5 – Extended ISA [20 marks]

There's a part 5?!? Well, only if you are up for the challenge, extend your processor to support two or more of the following instructions for maximum of **bonus 20 marks**:

`mult 4 1 2` (multiply value in register 1 by value in register 2, and place the result in register 4)
`sll 4 1 0x02` (apply logical shift left 2 times on value in register 1, and place the result in register 4)
`srl 4 1 0x02` (apply logical shift right 2 times on value in register 1, and place the result in register 4)
`sra 4 1 0x02` (apply arithmetic shift right 2 times on value in register 1, and place the result in register 4)
`ror 4 1 0x02` (apply rotate right 2 times on value in register 1, and place the result in register 4)
`bne 0x02 1 2` (if values in registers 1 and 2 are not equal, branch 2 instructions forward)

You must provide a clear description of the instruction encodings, assigned opcodes, and changes made to the datapath+control as a separate report. Submit a compressed file ***eXXYYY_lab5_part5.zip*** containing all files of your design along with a testbench and your description report.

Have fun coding. May the force be with you!