CO224 Computer Architecture - 2018

Department of Computer Engineering

Lab 5 - Building a Simple Processor

In this lab you will be designing a simple 8-bit single-cycle processor using Verilog HDL, which includes an ALU, a register file and other control logic. Follow the guidelines given to complete. Skeleton codes will be provided for your convenience.

Every processor is designed based on an Instruction Set. Your processor should implement the instructions `add`, `sub`, `and`, `or`, `mov` and `loadi`. All instructions are of 32-bit length, and should be coded in the format shown below.

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

- `OP-CODE` identifies the instruction's operation. This should be used by the ALU to perform the corresponding function.
- `DESTINATION` field specifies the destination register number.
- `SOURCE 2` is the second source register number
- `SOURCE 1` is either the first source register number, or an immediate value

Your instructions should be as shown in following examples:

`add 4 2 1` (add value in register 1 to value in register 2, and place the result in register 4)
`sub 4 2 1` (subtract value in register 1 from the value in register 2, and place the result in register 4)
`and 4 2 1` (perform bit-wise AND on values in registers 1 and 2, and place the result in register 4)
`or 4 2 1` (perform bit-wise OR on values in registers 1 and 2, and place the result in register 4)
`mov 4 X 1` (copy the value in register 1 to register 4. Ignore `SOURCE 2`)
`loadi 4 X 0xFF` (load the immediate value 0xFF to register 4. Ignore `SOURCE 2`)

You will be building your processor in three steps:

- In part 1, you will build and test the ALU which implements all the functional units required to support the given instruction set.
- In part 2, you will implement a simple register file.
- Finally in part 3, you will implement the control logic which will put all the components together to work as a complete processor.

You may implement your Verilog modules as **behavioral** models.

Part 1 - ALU

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. In this part you will use the Verilog language to implement an ALU which can perform **four** different functions to support the six instructions specified above. Figure 1 below shows the interfaces of the ALU you will be implementing.

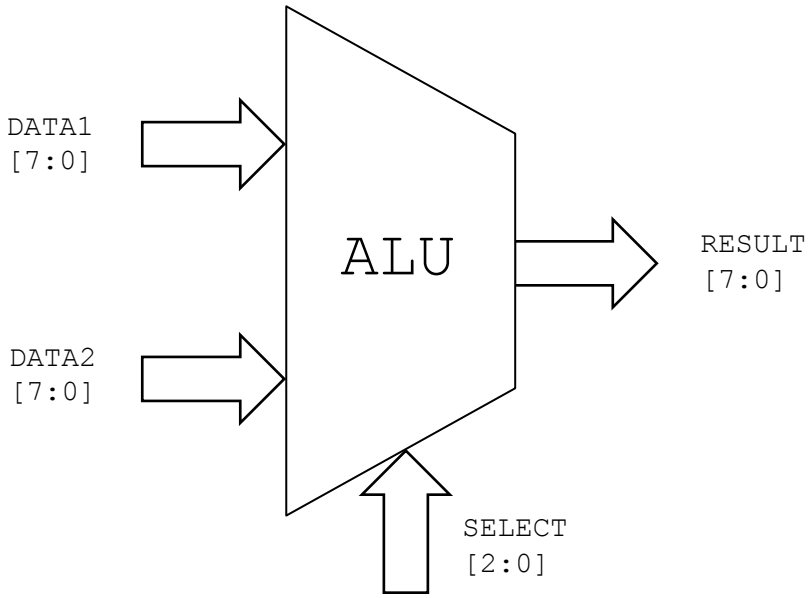

Figure 1: Interfaces of the ALU

The ALU that you are building should work with 8-bit operands. There should be two 8-bit input operands (`DATA1` and `DATA2`), one 8-bit output (`RESULT`) and one 3-bit control input (`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`.

The 3-bit `SELECT` signal should be derived from `OP-CODE`. One possible way is to use the three least significant bits of the `OP-CODE`.

eg:- When `OP-CODE` is "`00000000`", `SELECT` should be "`000`"
When `OP-CODE` is "`00000011`", `SELECT` should be "`011`"

You may implement any appropriate mapping from `OP-CODE` to the `SELECT` signal.

Please 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 ALU should be able to perform.

Table 1: ALU Functions

SELECT	Function	Supported Instructions	Description
000	FORWARD	loadi, mov	(forward DATA1 into RESULT) DATA1 → 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 DATA1 to RESULT. This unit will be used by the *loadi* and *mov* instructions to place the required 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. (When completing the processor in part 3, you should provide the ALU with two's complement values of SOURCE 1 and SOURCE 2 to correctly to implement both *add* and *sub* instructions using only the ADD functional unit).

1. Design the ALU using Verilog. Make sure you deal with any unused bit combinations of the SELECT lines. (Hint: using a *case* structure will make this job easy)
2. Write a test bench and simulate your ALU and test with different combinations of DATA1 and DATA2 values.

Part 2 - Register File

Next you should implement a simple register file. The purpose of the register file is to store `RESULT` values coming from the ALU, and to supply the ALU with operands (for `DATA1` and `DATA2` inputs) for register-type instructions (in our case, `add`, `sub`, `and` and `or`).

Your register file should be able to store **eight** 8-bit values (`register0 - register7`). It should contain one 8-bit input port (`IN`) and two 8-bit output ports (`OUT1` and `OUT2`). To specify which register you are reading/writing with a given port, you must include address signals (`INaddr`, `OUT1addr`, `OUT2addr`).

You may include separate control input signal(s) to **change the read/write mode** of the register file. Feel free to use additional control signals as appropriate.

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

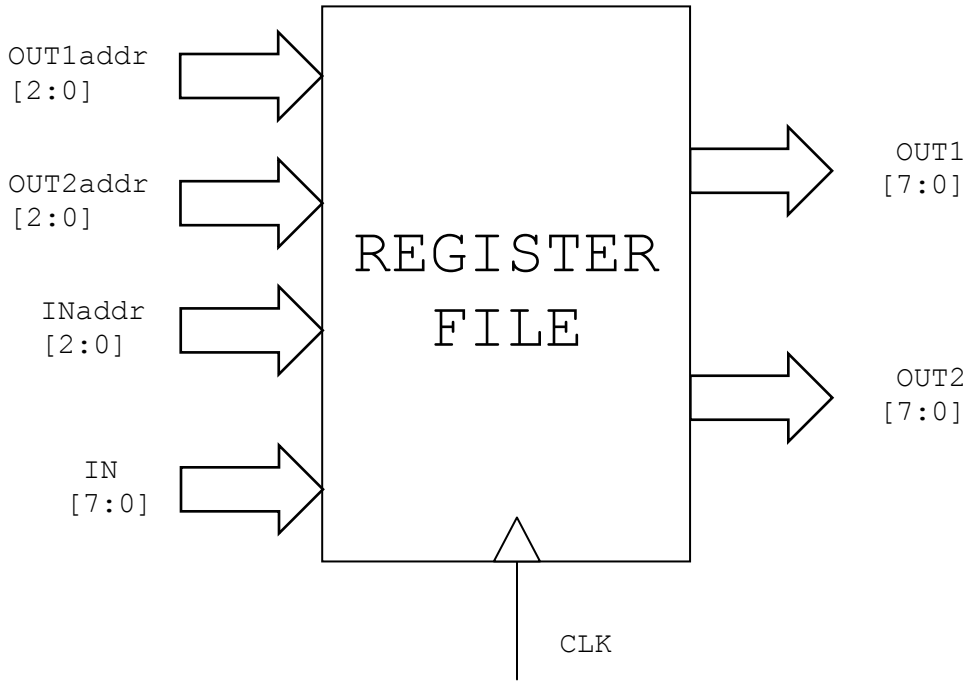

Figure 2: Interfaces of the Register File

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

```
module reg_file(IN, OUT1, OUT2, INaddr, OUT1addr, OUT2addr, CLK, RESET)
```

The signal `IN` represents the single data input, with `INaddr` providing the register number to store the data in. The signals `OUT1` and `OUT2` are the dual data outputs, where `OUT1addr` and `OUT2addr` respectively provide the register numbers where data should be retrieved from. A falling edge of `CLK` causes the data present on `IN` to be written to the input register specified by the `INaddr`. At the rising edge of `CLK`, registers identified by `OUT1addr` and `OUT2addr` are read and the values are loaded onto `OUT1` and `OUT2` respectively.

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.
2. Implement a test bench for your Register File, and test your code.

Part 3 - Control

Now you should connect your ALU and Register File together. To do this, you will need to implement some additional control logic in a top-level module (you may call this module as CPU). Your CPU needs an instruction fetching mechanism with an Instruction Register (IR) which holds the current instruction, and a Program Counter (PC) register which points to the next instruction.

Since we do not have a memory module yet, you can hardcode the instructions in your test bench file. Your instruction fetching mechanism can read instructions from the test bench, every clock cycle.

You need 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 and ALU appropriately.

For arithmetic instructions (`add` and `sub`), assume that the operands are signed integers with negative values presented in Two's Complement format. Therefore, you will need to perform the Two's Complement operation on some operands before supplying them to the ALU. You will need to use MUXs to achieve the desired control.

Pay careful attention to how you coordinate the timings of instruction fetching and Register File reading/writing.

An overall block diagram for this simple CPU is provided below in Figure 3.

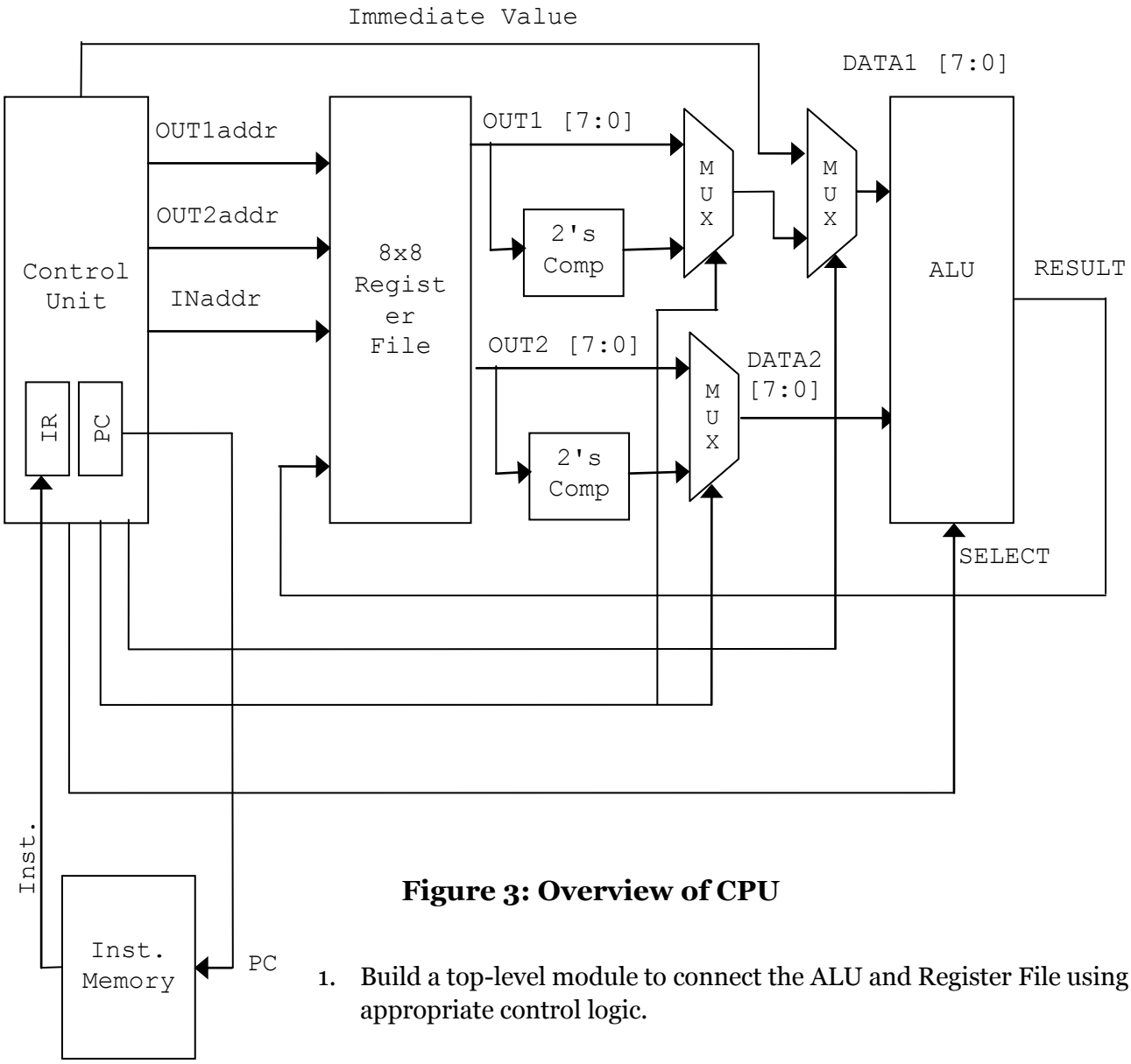

Figure 3: Overview of CPU

1. Build a top-level module to connect the ALU and Register File using appropriate control logic.

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

3. [Bonus exercise] Implement branching instructions such as `beq` (branch if the result of previous operation is zero) and `bne` (branch if the result of previous operation is non-zero). For this you will need to implement a status register as well.

If you still want an extra challenge, change things in this layout as you feel appropriate, and add more functionality.

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