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)	RD / IMM (bits 23-16)	RT (bits 15-8)	RS / IMM (bits7-0)

- `Bits(31-24)`: `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.
- `Bits(23-16)`: A register (`RD`) to be written to in the register file, or an immediate value (jump or branch target offset).
- `Bits(15-8)` : A register (`RT`) to be read from in the register file.
- `Bits(7-0)` : A register (`RS`) to be read from in the register file, or an immediate value.

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 `bits 15-0`)
`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 `bits 15-8`)
`loadi 4 0xFF` (load the immediate value `0xFF` to register 4. Ignore `bits 15-8`)

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 not support `j` and `beq` at this stage). Figure 1, below, shows the interfaces of the ALU you will be implementing.

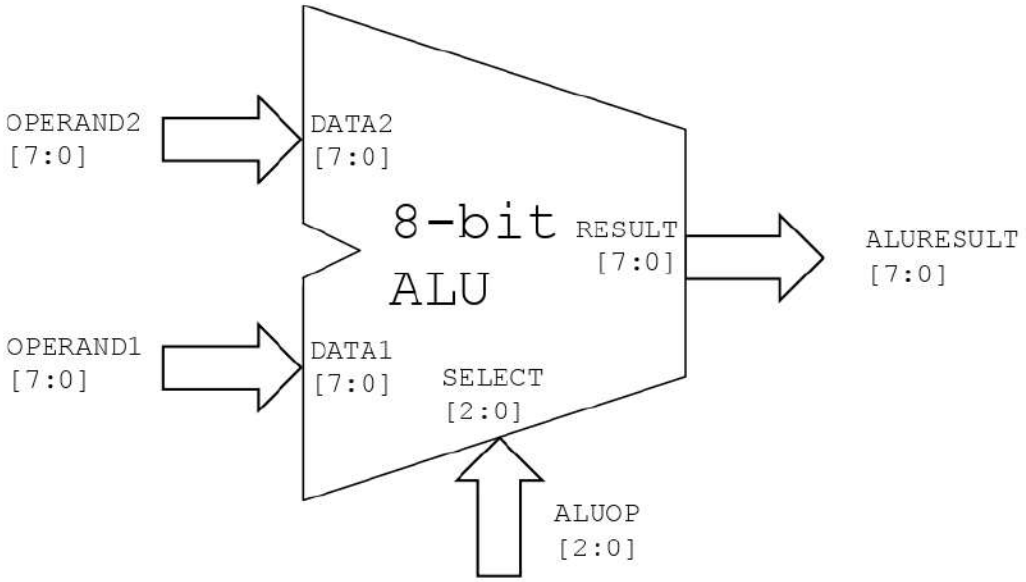

Figure 1: Interfaces of the ALU

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	Description	Supported Instructions	Unit's Delay
000	FORWARD	(forward DATA2 into RESULT) DATA2→ RESULT	loadi, mov	#1
001	ADD	(add DATA1 and DATA2) DATA1 + DATA2 → RESULT	add, sub	#2
010	AND	(bitwise AND on DATA1 with DATA2) DATA1 &DATA2 → RESULT	and	#1
011	OR	(bitwise OR on DATA1 with DATA2) DATA1 | DATA2 → RESULT	or	#1
1XX	Reserved	Reserved for future functional units	-	-

These functional units must be implemented **as separate modules**, and instantiated inside the *alu* module. FORWARD unit should simply send an operand value from DATA2 to its output. 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 send the result to its output. The *alu* module should use a MUX to pick one of the functional units' outputs and send it to RESULT based on the SELECT value. To simulate the ALU latencies realistically, include the given artificial delays in each functional unit (note that the delays are for individual functional units, not the MUX that will choose the RESULT).

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 ***groupXX_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 output values generated by the ALU, and to supply the ALU's inputs 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.

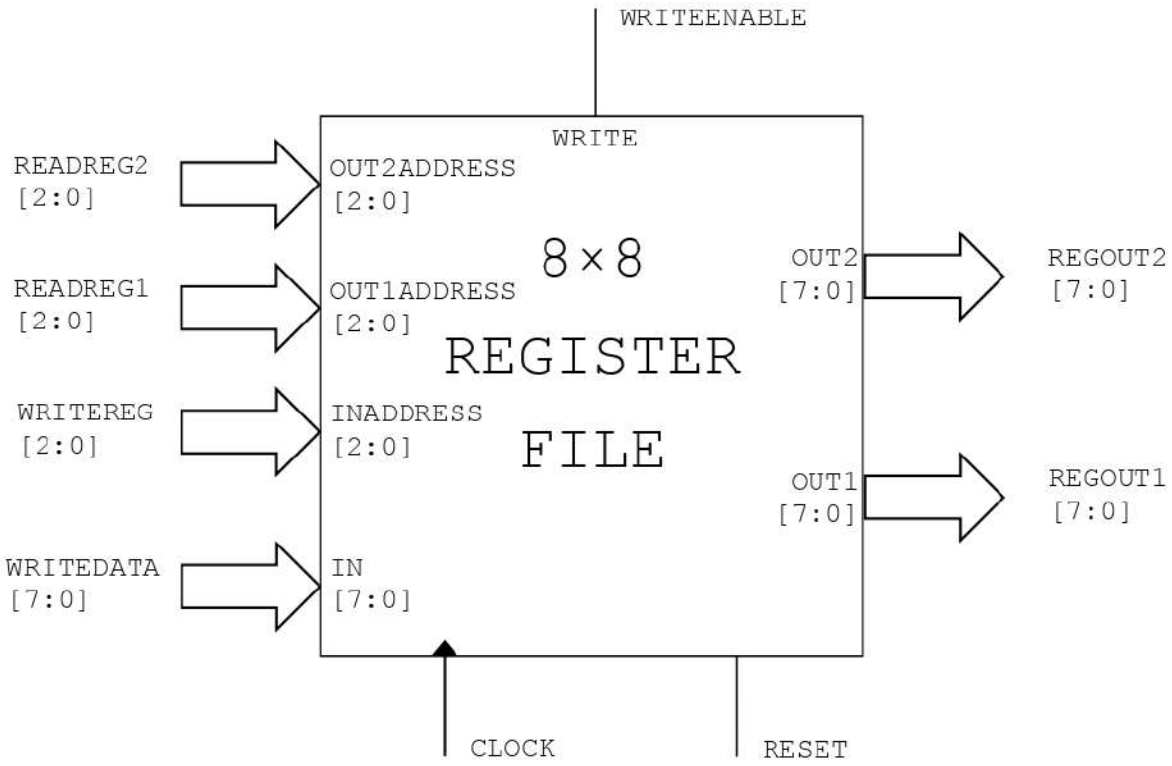

Figure 2: Interfaces of the Register File

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 parallel data outputs, where `OUT1ADDRESS` and `OUT2ADDRESS` respectively provide the register numbers where data should be retrieved from.

Registers identified by `OUT1ADDRESS` and `OUT2ADDRESS` should be read asynchronously and the values should be loaded onto `OUT1` and `OUT2` respectively.

Writing to the register file must be done synchronously. When `WRITEENABLE` signal at the `WRITE` port is set high, rising edge of `CLOCK` should make the data present on the `IN` port to be written to the input register specified by the `INADDRESS`.

Register file reset should happen synchronously at the positive edge of the clock if the `RESET` signal is high. All registers should be cleared (written zero) in a reset event.

To simulate the register file read and write latencies realistically, include artificial delays of two time units (#2) for register reading and one time unit (#1) for writing operations including reset.

You need to **pay careful attention to the timings**. 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 ***groupXX_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 reading and writing of registers.

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