CO224 Computer Architecture

Lab 4 - 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 component. Follow the guidelines given to complete. Skeleton codes are 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 (nits7-0)

- 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

The prototypical instructions will be as follows:

`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, you will implement the control logic which will put all the components together to work as a complete processor.

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

Part 1 - ALU

The heart of every computer 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.

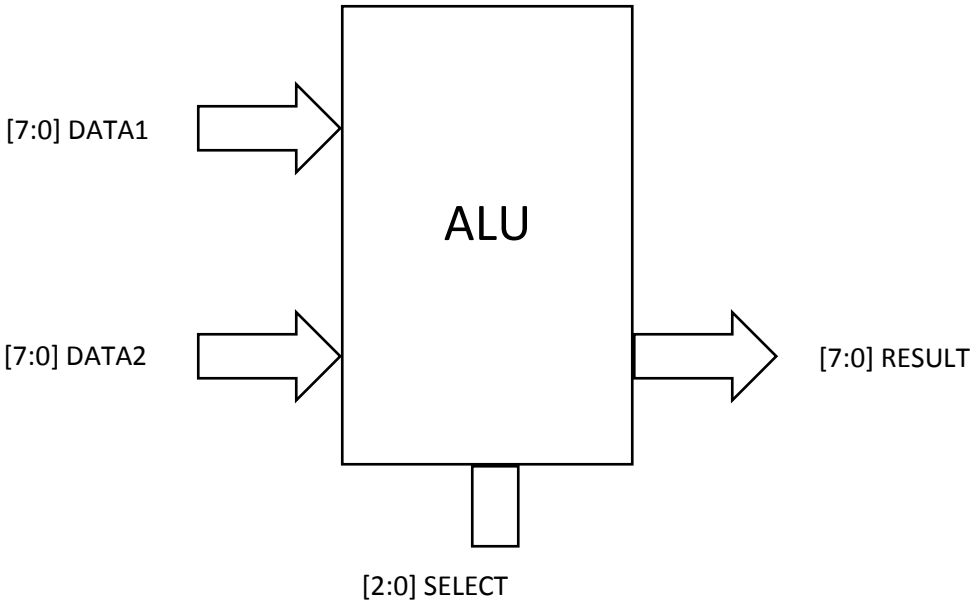


Figure 1: Interfaces of the ALU

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 control input (SELECT) which should be used to pick the required function inside the ALU out of the available five functions, based on the instruction's OP-CODE.

The 3-bit SELECT signal is derived from 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"

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

The Table below shows the XXX functions (operations) that your ALU should be able to perform.

Table 1: ALU Functions

SELECT	Function	Supporting 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*	(subtract DATA2 from DATA1) DATA1 – DATA2 => RESULT
011	OR	*or*	(logical AND on DATA1 with DATA2) DATA1 & DATA2 => RESULT
1XX	reserved	–	Reserved for future instructions

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 data in DATA1 and DATA2 registers and write the output to the RESULT register. (When completing the processor in part 3, you should provide the ALU with two's complement values of SOURCE 1 and SOURCE to correctly to implement both *add* and *sub* instructions using only the ADD functional unit).

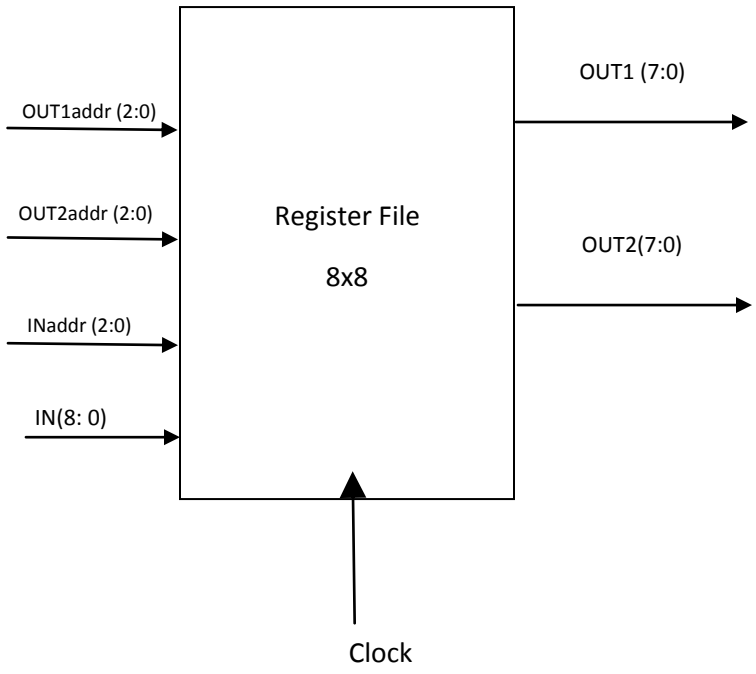
1. Design the ALU using Verilog. Make sure you deal with any unused bit combinations of the Select lines. (Hint: using the *case* structure will make this job easy)
2. Write a test bench and Simulate the ALU and test different combinations of DATA1 and DATA2.

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. 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 below.



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

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

Signals IN, OUT1 and OUT2 represent the single data input and dual data outputs respectively. INaddr , OUT1addr , and OUT2addr are the register numbers for storing input IN data and for retrieving OUT1 and OUT2 data, respectively. A falling edge on clk causes the data input on IN to be written to the input register specified by the INaddr. At the rising edge of the clock, registers identified by OUT1addr and OUT2addr are read from OUT1 and OUT2 respectively.

1. Implement the behavioral model for the register file entity. 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.

Part 3 - Control

Now you should connect the ALU and register file you made. To do this, you will need to implement some control logic.

You need an instruction fetching mechanism with an instruction register and a PC (program counter) register which points to the next instruction. Since we do not have a memory yet, you can hardcode the instructions in your testbench file, and let your processor fetch instructions from there on every falling edge.

You need logic to decode a fetched instruction, extract the OP-CODE, source/destination registers and immediate values. Depending on the OP-CODE, control signals should be generated and sent to the register file and ALU appropriately. For arithmetic instructions (add and sub) you must perform two's complement on the operands before supplying them to the ALU. You will need to use MUXs to achieve the desired control.

An overall block diagram is provided below. If you want an extra challenge, feel free to change anything you feel appropriate there, and add more functionality.