## Part 3 –Integration & Control [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.

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

```
module cpu(PC, INSTRUCTION, CLK, RESET)
```

Since we do not have an instruction memory module yet to hold instructions, you should keep the instructions as an array of hardcoded instruction words (array size = 1024 bytes, i.e. 256 instructions) in the testbench file which you use to test your *cpu*. Your *cpu*'s instruction fetching mechanism should read the hardcoded instructions asynchronously from the testbench, based on the address provided by PC.

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` (bits 31-26), all control signals should be generated and sent to the Register File, ALU and other components appropriately. Bits 25-0 need to be directly sent to appropriate places where they may be used, without needing to wait.

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, decoding, register reading, execution and register writing. To realistically simulate the latencies of instruction fetching, decoding and execution, you should include the following artificial timing delays in your CPU:

- PC Update (write to PC register) = One time unit (#1)
- Instruction Memory Read = Two time units (#2)
- Instruction Decode (generating control signals) = One time unit (#1)
- Two's complement operation = One time unit (#1)

In order to automatically increment the PC value by 4, you will need to include a dedicated adder. We will assume that this adder has a latency of one time unit (#1), which will work in parallel to instruction memory reading.

Use GTKWave (or any other similar tool that you prefer) to help you visualize the timings.

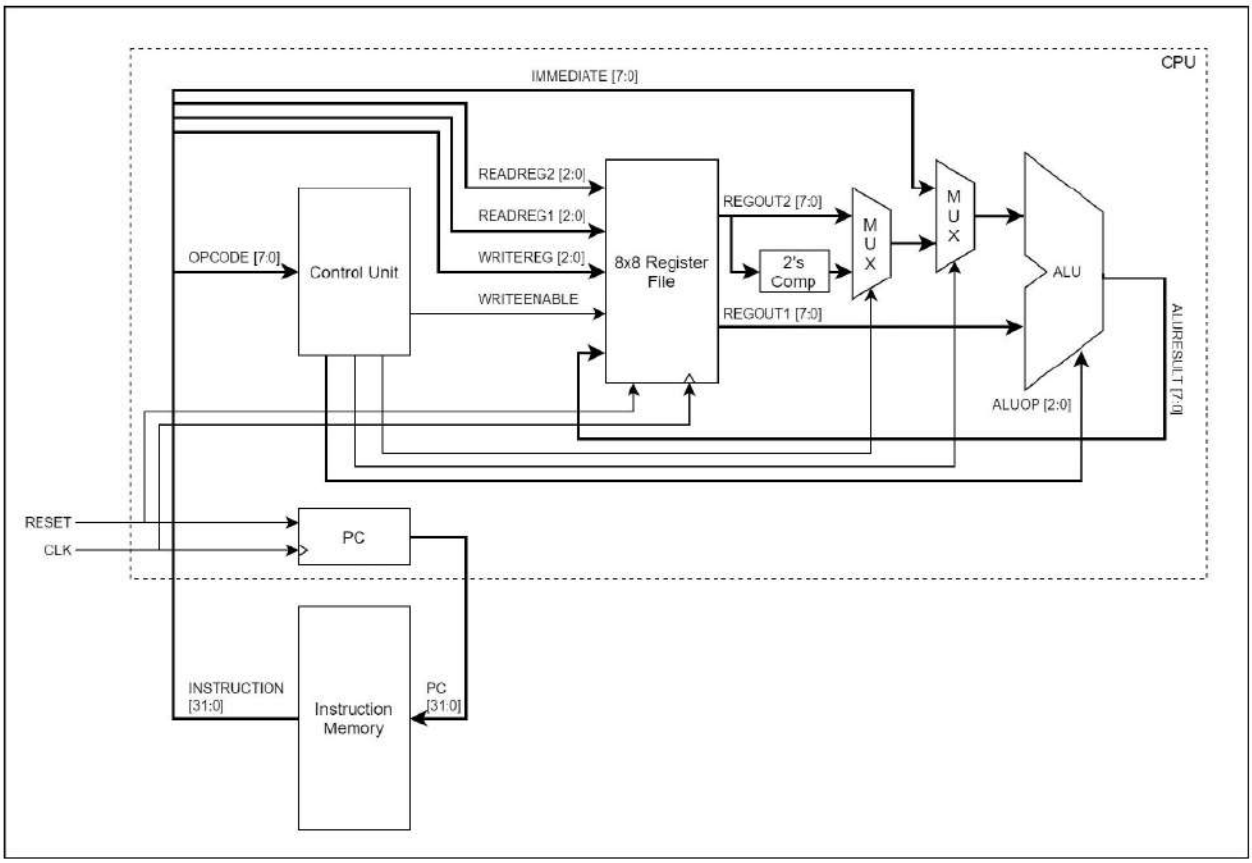


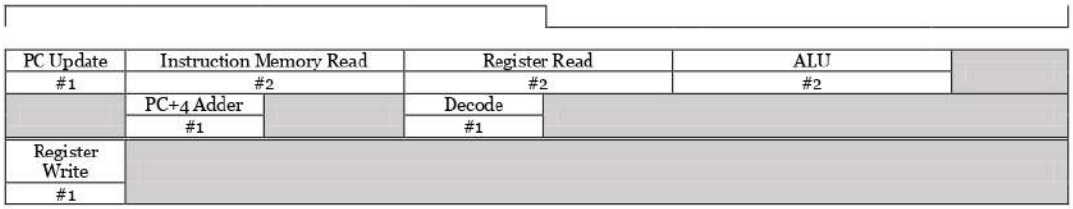

**Figure 3: Overview of CPU**

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

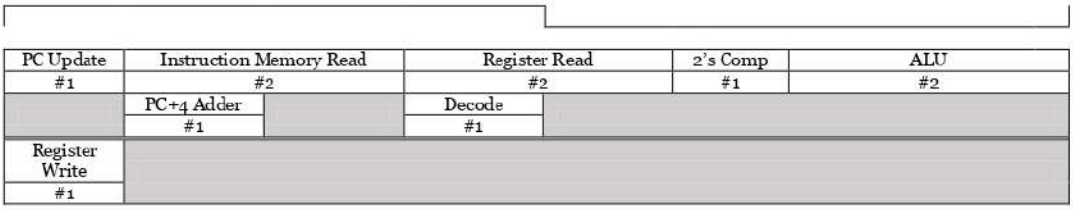
You must thoroughly test your *cpu* using several different software programs (instruction sequences). For this, you need to prepare programs as machine code and hardcode them inside your testbench file, one program at a time. Since it is easier to write textual assembly programs (rather than writing 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. A shell script is provided to you, which can convert assembled machine code into a memory image which you can easily copy onto your testbench.

The diagram given in Figure 4 shows the worst-case timing of your single-cycle CPU for the `add`, `sub`, `and`, `or`, `mov`, and `loadi` instructions. One clock cycle spans for eight (8) time units duration, rising edge to rising edge. Every instruction should complete within one clock cycle, and any data to be written to the register file should be ready by the rising edge at the end of the clock cycle. Writing to registers and PC should be synchronized to rising edges of the clock, while reading of registers should happen asynchronously. The given timing delays should be artificially added to the corresponding operations in order to realistically simulate the latencies observable in a synthesized datapath. For the sake of simplicity, we will be assuming that our multiplexers and wires have negligible delays.

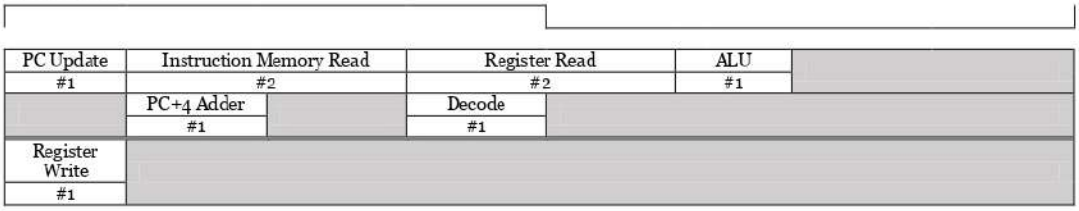
`add:`




`sub:`




`and/or/mov:`




`loadi:`

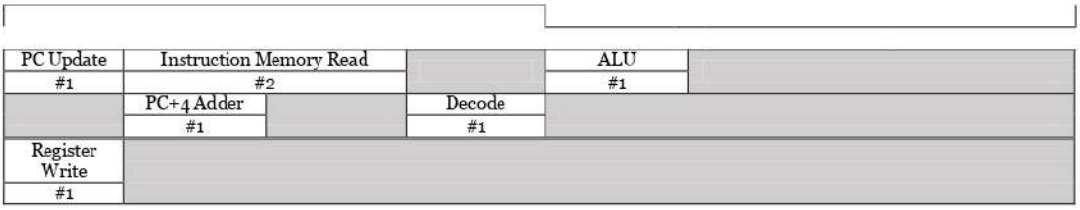



**Figure 4: Timing Details for the Datapath**

You must also implement the reset functionality of the CPU, so that the program can be restarted by setting the `RESET` signal high for a short period of time. If the `RESET` signal is high when writing to PC at a positive clock edge, write zero to PC instead of the next PC value in order to restart the program. So the CPU will read the instruction memory at address zero. In addition to this, your register file content should also be reset at the same time.

**Note that data memory reading and writing is not implemented in Lab 5.** You will be adding those functionalities in Lab 6.

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 ***groupXX_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 for the given six instructions.

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