Assignment 2: Design of Multi-Cycle and Pipelined Processor

Georgios Lamprinakis
AM: 2018030017

April 18, 2021

Purpose of the Assignment

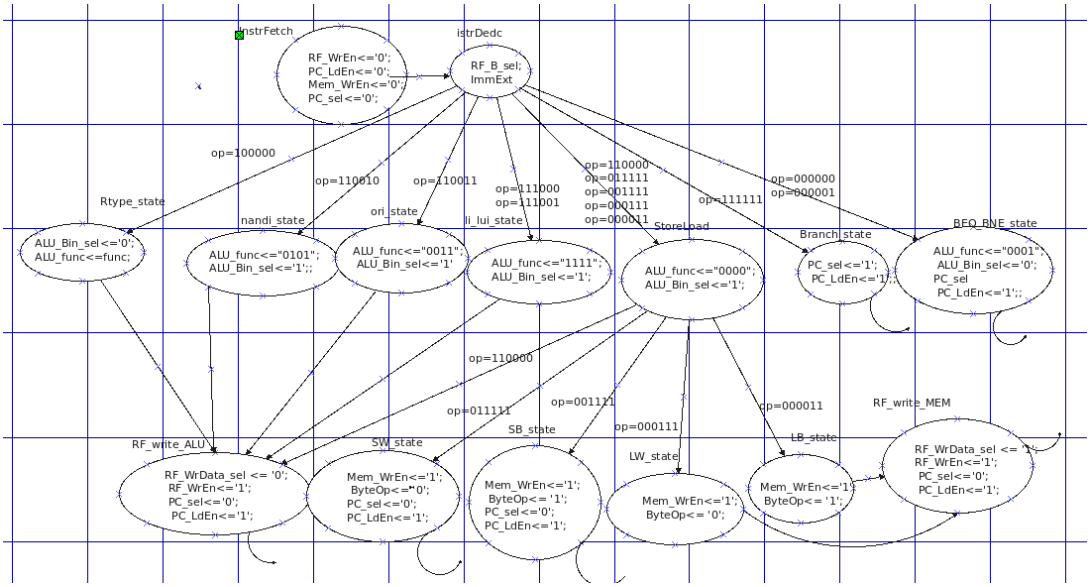
The purpose of this assignment is to understand the functionality of the multi-cycle processor and the pipelined processor, and how they can be constructed by making appropriate modifications to the single-cycle processor.

Multi-Cycle Processor

For the multi-cycle processor, the execution of an instruction is divided into 5 stages, each requiring its own clock cycle (Instruction Fetch, Decode, Execution, Memory Access, Write Back). The same structural components of the single-cycle processor corresponding to each stage (ifstage, decstage, exstage, memstage) were used, and registers were placed between them. These registers store the output of each stage at the end of a clock cycle and provide it as input to the next stage.

- Between `ifstage` and `decstage`, a register is placed to hold the instruction.
- Between `decstage` and `exstage`, three registers are used — two for the contents of the two read registers from the RF, and one for the immediate value.
- Between `exstage` and `memstage`, two registers hold the ALU result and the data to be written to memory.
- After `memstage`, one register holds the data read from memory.

The control was implemented as a finite state machine (FSM), as shown below:



The values `immExt` and `RF_B_sel` of the `instrDec` state, and `PC_sel` of the `BEQ_BNE_state`, depend on the inputs, as shown in the code snippets:

```
if opcode = "100000" then
    RF_B_sel<='0';
    ImmExt<="01";
elsif opcode="110010" or opcode="110011" then
    ImmExt<="00";
    RF_B_sel<='1';
elsif opcode="111001" then
    ImmExt<="11";
    RF_B_sel<='1';
elsif opcode="000000" or opcode="111111"or opcode="000001" then
    ImmExt<="10";
    RF_B_sel<='1';
else
    ImmExt<="01";
    RF_B_sel<='1';
end if;
```

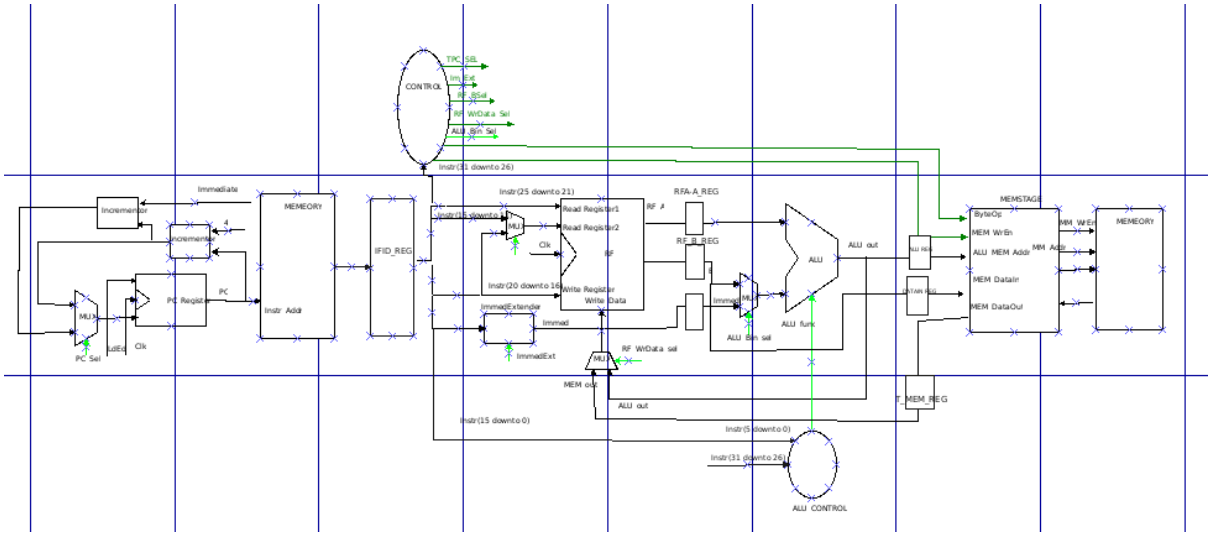
```
if (opcode ="000000" and zero='0') or (opcode ="000001" and zero='1') then
   PC_sel<='0';
elsif (opcode ="000000" and zero='1') or (opcode ="000001" and zero='0') then
     PC_sel<='1';
 end if;
```

Due to the two states mentioned, the FSM is a Mealy machine.

The signal `pc_ld` remains '0' throughout an instruction's duration and becomes '1' only at the end of it, i.e., in one of the states: `branch_state`, `BNE_BEQ_state`, `RF_write_ALU`, `SW_state`, `SB_state`, `RF_write_MEM`. It is then reset to '0' in the `instFetch` state.

It should also be clarified that there is no separate `addi_state`, since it is combined with the `StoreLoad` state (due to identical inputs).

A simulation follows for Reference Program 1:



Pipelined Processor

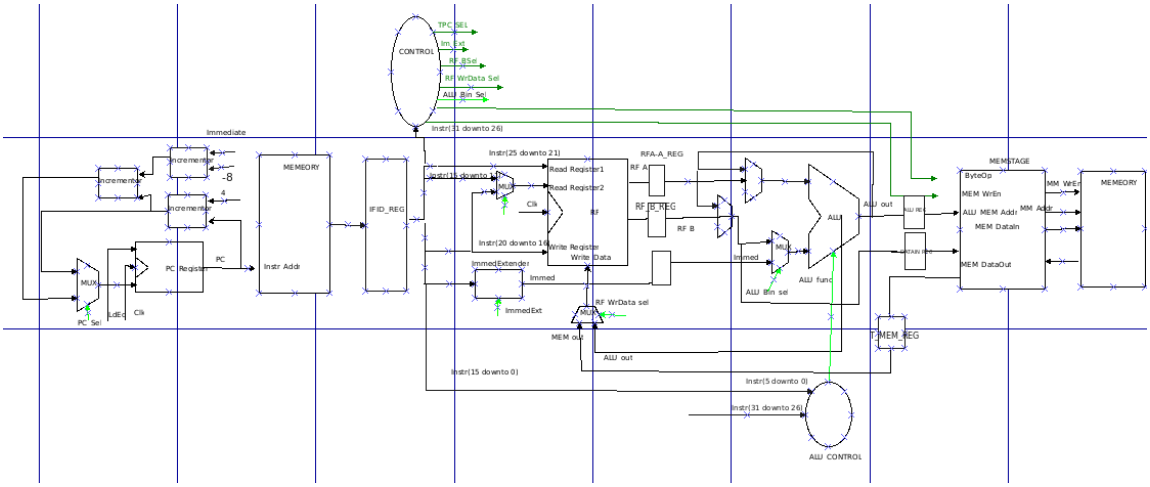
For the pipelined processor's datapath, we retain the above multi-cycle model with modifications to the `decstage` and `ifstage`, and we add additional registers between stages.

- Between `decstage` and `exstage`, three 5-bit registers are added to store register addresses Rs, Rt, and Rd, and one 15-bit register to store all control signals.
- Between `exstage` and `memstage`, a register for Rd and one for the control signals are added.
- Equivalent registers are also added after the `memstage`.
- Before the `exstage`, two 3-to-1 multiplexers (32-bit) controlled by `forward_sel_A` and `forward_sel_B` are added to handle data hazards (explained further in the control section).

In addition, from the `decstage` module, the multiplexers for selecting the second read register address and write data are removed and placed elsewhere in the datapath. This is because the write register address and the write data for the RF will be determined in later stages of the instruction, and the previous implementation was incompatible.

The changes to the `ifstage` relate to handling branch hazards and are explained in the control section.

Note: The datapath diagram is incomplete due to space constraints.



Pipeline Control

The pipeline control is a combinational circuit consisting of the submodules: `MAIN_CONTROL_UNIT`, `FORWARD_UNIT`, and `HAZARDS_DETECTION_UNIT`.

Main Control Unit

This unit is similar to the single-cycle control, but now the main control and ALU control are merged. Additionally, a new signal `branch` is introduced, which informs subsequent stages whether the current instruction is a branch and what type (BNE, BEQ, b).

Forward Unit

This module is responsible for forwarding where necessary, by altering the values of `forward_sel_A` and `forward_sel_B`, thus handling data hazards.

Hazards Detection Unit

This module performs necessary stalls for load instructions and manages branch instructions.

- If a `load` instruction writes to a register that needs to be read by the immediately following instruction, the latter must be delayed by one cycle to allow forwarding. This is done by disabling writing to the `pc_register` and `IFID_register`, and zeroing all control signals via a multiplexer to prevent unintended writes to memory or RF.
- Detection occurs when the `load` is in `exstage` and its following instruction is in `decstage`, where it remains for two cycles.
- For `bne` or `beq` instructions, since we cannot know in advance if the branch will be taken, we assume it will not. The final decision is made in `exstage` based on the `zero` signal. In the meantime, two new instructions are fetched from memory, which must be ignored:
  - The instruction in `decstage` has its control signals zeroed.

– The instruction in `ifstage`, already written to `IFID_register`, is prevented from executing by overwriting it with a new instruction whose opcode is unused.

This is achieved by using a 2-to-1 multiplexer at the input of `IFID_register` with inputs: `instruction` and `x"40000000"`, controlled by `PC_sel`. If `PC_sel` is '1' (i.e., a branch is taken), the "non-existent" instruction is inserted. Such opcodes ensure `MEM_we` and `RF_WrEn` are '0', preventing unintended writes.

- Since the PC will have incremented by 8 (i.e., 2 instructions) by the time we decide whether to branch, we must subtract 8 from the immediate to branch to the correct address. Thus, we add another incrementor in the `ifstage` to handle this.
- Even though for the `b` instruction we know in advance whether to branch, we handle it the same way as `beq` and `bne` for simplicity in implementation.

A simulation follows for Reference Program 1: