Design of Decode, Control FSM, Program Counter and Associated Datapath Units

ECE/CS 3710 - Computer Design Lab
Deadlines are as follows:
Integrate Memory, Program Counter, and FSM for register-to-register instructions: Tuesday Oct 13
Then, augment the FSM to accommodate Loads and Stores: Tuesday Oct 20
Further augment the FSM to account for Jumps and Branches: Thursday Oct 29
Lab 4 report due on Canvas: Friday, October 30.

I. Objective

In the previous labs, we have designed the ALU, Regfiles and integrated the datapath with memory. In this lab, our objective is to integrate the baseline CPU control logic, which is a finite state machine (FSM), with this ALU datapath. In the coming two weeks, we will discuss the overall architecture of the CPU executions, and various options for organizing the design and control. Your job is to make the design work, not just in simulation, but also on the FPGA board.

For this lab, at least all the baseline CPU operations need to be implemented. Some of you will have to augment the baseline with some extra instructions to access IO: (S)NES controllers, Audio, accelerometers, ultrasound reflectometers, etc. You do not necessarily have to implement those right away for this assignment, but it would help to think ahead: does it affect the PC (and associated control) significantly? Does it change your datapath significantly? For this lab, our machine should be able to run without external IO. So you have to implement:

- All register-to-register (and also immediate) operations;
- Load and Store Operations;
- Conditional and unconditional branches (and if you also want to implement jumps as different from branches);

This way, a programmed control execution is completed, and we can execute most non-IO-reliant instructions (algorithms!) on our computer.

To test our machine, demo a small program execution: initialize the memory with a sequence of baseline instructions, and after reset, fetch instructions from memory, doing loads, stores, reg-to-reg instructions, and finally using branch control, in RTL and on the FPGA board. The final result can be displayed on the 7-segment display. Actually, for your own benefit, you should be testing many different small programs to feel confident about your machine.

Deadlines are given above, and I suggest you *design the CPU control incrementally*, and hit the milestones (deadlines) step-by-step. We should try to have the final FPGA demo completed by Oct 29, and have the report due on Canvas by Oct 30.

II. Guidelines and Approach to Design

Before you begin designing the components for this lab, my suggestions are the following:

- Draw the block-diagram of the ALU-Regfile-Memory-PC, as I showed in class.
- Take a look at the state machine design in Hennessey and Patterson (ECE/CS 3810 textbook) Chapter 5 and see how the state machine FSM for the control is designed. Keep in mind, that the FSM in the textbook may

have one or 2 extra states than yours, due to the fact that they use a separate instruction register (memory data register) outside of memory.

- For each class of instructions (Reg-Reg, Load-Store, Branch/Jump), confirm if the data-flow between memory, ALU, regfile and PC can be accomplished.
- Make a list of all the control signals and their associations (PC-related control, ALU-related control, etc.).
- Design the Verilog code modularly. Instantiate each of the blocks structurally, ensuring that the Verilog design modularly maps/resembles your block-diagram. This will make debugging easier.
- Approach the FSM/decoder design step-by-step.
- *Remember: In each and every state of the FSM, every output/control signal (mux select, reg eneables, etc.) needs to be explicitly assigned.*
  - Consult the 3810 computer architecture book, or any other computer architecture book that you may have used. Ch 5 in the 3810 book describes the control FSM and the datapath.
  - I used a similar approach to describe the FSM in class.
  - Using such a block diagram, and the "state-transition-graph" (STG) of the FSM, first attempt to complete the reg-to-reg instructions as they are easier than the others. Then try loads and stores and branches.
- Keeping the FSM simple is a good design strategy. For this, a detailed and modular datapath design is important.
- FSM can be both a Mealy or Moore type, and the designs **may** differ in one clock-cycle for instruction completion. If you are analyzing both Mealy and Moore type controls, do not get surprised if this happens.
- Pay attention to the PC and related logic. Incrementally design the PC-related hardware, in a step-by-step fashion [PC increments for straight-line code; then extend the design for loads/stores, and then for branches]. It will help a lot if you draw a separate, and detailed, block-diagram for the PC and associated hardware.

III. Design issues for you to consider for implementation

Memory interface: Are you using memory's own internal registers as instruction and data registers? Or are you implementing a separate memory data register (MAR) and instruction register (IR)? If so, how does your FSM control data transfer between memory $\iff$ MAR/IR.

Instruction Decoder: In class, the design I showed had included the decode logic within the FSM. That is not necessary. You may design a combinational logic block that decodes the instruction, and then the FSM just controls the opcode and reg/mux-enables to connect the decoder-output to the rest of the datapath. [I kept it within the FSM because our machine really has 3-types of instructions, so the FSM isn't that complicated.]

Program Counter Design: This is the most "thinking part" of the architecture. You need to take care that the PC is organised in a way that it can do all of the following: i) PC = PC + 1 [standard execution]; ii) PC = PC +/- k for branch targets; iii) and PC updates for loads and stores.

Think about the following: For loads and stores, if the address resides in the regfile, then do you need to load the PC from the regfile? What about Jumps?

Signed or 2's Complement PC: Addresses are unsigned integers, so you need to consider what it means to have an unsigned PC that is operated by a 2's Complement ALU; especially in the case of signed offsets that might require subtraction. The problem can get more tricky with Verilog, particularly when unsigned and signed numbers are added. Refer to our 3700 textbook, sections 5.5.7 and 6.6 to review bit-vector and arithmetic representation.

PC updates: For each (set of) instructions, when does PC get updated? Which state of the FSM takes care of it?

Processor Status Registers/Flags: Does your FSM need access to/from PSR/Flags?

Sign Extension for immediates: How are these being handled? Various instructions in our machine make use of sign-extended immediates. Recall from the instruction set handout that immediates in arithmetic operations are sign-extended from the 8-bits. Logical immediate operations are zero-extended instead of sign-extended. Is this done by the CPU? Or within the ALU already?

IV. Demo and Submission

Once the CPU + datapath (ALU + Regfile) + PC + Memory design is completely simulated and synthesized, demo the correct operation of the machine to the TA by Thursday Oct 29. Your **FINAL test program** should reside in memory, and use the following operations: (i) Load data from memory into the regfile; (ii) Perform a series of arithmetic and logical operations (set flags); (iii) Store the result in memory; (iv) reload the result from memory into the regfile; (v) perform arithmetic to set flags; (vi) use the flags to do branches (BEQ, BLT, etc.); and (vii) write the result into memory and display at the output.

For your own testing, you should be trying different programs of similar type to validate your machine.

For the report, submit a completed Lab 4 (CPU) report. The report should depict the entire datapath, and have CPU interface cleanly described. If you prefer, you could describe multiple block-diagrams to specify the details and a somewhat abstract high-level diagram of the overall integration. Feel free to re-use parts of block diagrams from prior reports.

In the document, make sure that all the issues are properly described — size of the PC, unsigned or 2'C representation and how it is handled, show the STG of the FSM, how many clock cycles does each (type of) instruction takes, when does write-back occur, when does PC get updated, etc. — all the issues that are needed to understand your architecture. Please try to be complete, yet concise!

Report the area/delay synthesis results and FPGA-resource utilization. This should give us an idea of how large or small our baseline RISC CPU really is. Finally, write 1 to 2 paragraphs about how your CPU needs to be augmented (if at all) to accommodate your team's IO/application requirement. Submit one report per team.

In the final conclusion section of your report, please include the contributions of each team member! It is important that every team member contributes to the design and testing.

Have fun.