CS/ECE 3710 — Computer Architecture Lab

Instruction Decoding and Control FSM

Due Monday October 28th

Overview

After completing your ALU, register file, datapath, and memory interface, the only major remaining design tasks for your basic processor are the instruction decoding logic and the control finite state machine. There are still a few logistical tasks (place and routing the FPGA, writing test programs, downloading and uploading memory from the PC, dealing with I/O), but with the addition of some decode logic and a control state machine, your processor should be able to process! This checkpoint is basically to finish off the design of the whole basic machine and have it running code for the check off. With any luck, the code that it is running might even be generated from your own assembler.

For this checkpoint at least all the baseline CPU operations need to be implemented. Some of you will want 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 checkpoint, our machine should be able to run without external IO. So you have to implement:

- All register-to-register (and their immediate versions) operations;
- Load and Store Operations;
- Conditional and unconditional branches and jumps;
- Jump and link

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 or on the LEDs. Actually, for your own benefit, you should be testing many different small programs to feel confident about your machine. It's easier to debug small test programs that use a few instructions than one large test program that uses every single instruction.

Guidelines and Considerations

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

- Draw a block-diagram of the ALU-Regfile-Memory-PC that you've designed so far. Make sure to identify all the control points into that block diagram.
- 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 may use a different memory structure, or decoding structure, than you will use.
- For each class of instructions (Reg-Reg, Load-Store, Branch/Jump), confirm if the data-flow between memory, ALU, RF, and PC can be accomplished. Remember to make sure to check the "tricky ones" like JAL too.

- Make a list of all the control signals and their associations (PC-related control, ALU-related control, etc.).
- Design the Verilog code modularly. You may want to instantiate each of the blocks structurally, ensuring that the Verilog design modularly maps/resembles your block-diagram. This can make debugging easier.
- Approach the FSM/decoder design step-by-step, and structure your Verilog code so it's clear which parts of the code will become sequential circuits and which parts will be combinational.
- *Remember: In each and every state of the FSM, every output/control signal (mux select, reg eneables, etc.) needs to be explicitly assigned.* See the example in the Verilog slides of using default assignments in combinational `always` blocks that can be overridden in later parts of that block as one way to make sure of this.
- 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. For example, in a Moore-style machine, the outputs are asserted once the machine is in state "Foo", which means they "take effect" when the machine leaves that state, not when it enters that state. On the other hand, in a Mealy machine the outputs may update before you enter a state and "take effect" as you enter a state. It can be confusing, especially if you mix FSM types in your design.
- Pay attention to the PC and related logic - the PC changes in multiple different ways as your FSM executes. Straight-line code has the PC increment by one word on each instruction, but then there are offsets to be added (for branches) register contents to be loaded into the PC (for jumps), and PC contents to be loaded back to the RF (for JAL), etc. It may help if you draw a separate, and detailed, block-diagram for the PC and associated hardware.
- It's time to really think about the use of signed offsets for branches using two's complement signed notation acting upon an unsigned 16-bit PC. The problem can get more tricky with Verilog, particularly when unsigned and signed numbers are added. Refer to your 3700 textbook, sections 5.5.7 and 6.6 to review bit-vector and arithmetic representation.
- 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?
- Processor Status Registers/Flags: Does your FSM need access to/from PSR/Flags?

Even More Details...

Control Flow

Since you are (most likely) not building a pipelined machine, the execution model is pretty simple. Basically your control needs to sequence some actions in the machine that will result in the instruction being executed. Some details to think about:

Instruction Fetch: Before you can execute an instruction, you need to fetch it from memory. Think about what this involves using the clocked Block RAM on the FPGA using the M10k blocks. The

returned data (the instruction) might then be latched into an instruction register. The result of this sequence of actions is that you fetch the next instruction into the instruction register so you now know which instruction you are about to execute.

Instruction Decode: In this phase of execution you use the information in the instruction register to set up all the state in the control path that you need to execute the instruction. This may or may not involve a separate clock phase depending on how your datapath is arranged. Things that get decoded include: mux settings, register file addressing, immediate fields (including sign extension or zero extension), and register enables.

Instruction Execution: Now that everything is set up by the instruction decode logic, you can execute the instruction. In this non-pipelined case this simply means allowing the correct data to go through the datapath and compute a result. Make sure you understand each and every instruction. Note that loads and stores may require some extra work here because you need to communicate to the clocked Block RAM to execute the load or store. Also note that a PC operation must be performed somewhere. If the instruction is not a branch or jump, the PC needs to be incremented by one 16-bit word. If it is a taken branch, then the PC must be added to a signed offset from the instruction (the offset is a word-offset from the current PC), and if it's a jump, the PC must be loaded from a register. If it's a jump-and-link, then the incremented PC must be stored in a register. Make sure to get the details of JAL right!

Writeback to the Register File: Once the answer is computed for that instruction, you need, in most cases, to write back the answer to the register file. Presumably you have already set up all the relevant information in the datapath (like destination addresses and mux settings), so this is probably nothing more than enabling the register file to do a write on the next cycle.

This cycle then repeats itself for each new instruction.

Instruction Decoding Logic

The main things that the instruction decode logic does are to pick out the fields of the instruction word that are important, and to use combinational logic to generate datapath control signals from those fields. Some are easy. For example, the src and dst register addresses should always be in the same place in the instruction word. These fields of the instruction register may be able to be wired directly to the address inputs of your register file.

Some are a little trickier. There are at least four different immediate formats: sign-extended immediates, zero-extended immediates, shift amounts (assumed to be 1 or -1 in the baseline), and branch offsets. Each of these need to be extracted from the instruction word and routed to the correct input of the ALU.

Another piece of the instruction decoder looks at the opcode information and uses that to decide how to set the other control bits of the data path. For example, on a MOV instruction, the datapath muxes should be set to allow the src register to be written back without modification to the destination register. This might involve mux settings, it might also involve function code bits to set the ALU function to pass a value through unmodified. The form of this logic is opcodes as input (from the instruction register), and datapath control bits as outputs (mux selects, function codes, etc). Almost certainly this will be combinational logic, so make sure that your Verilog code is structured accordingly.

Finite State Machine Controller

The finite state machine simply sequences actions that are already set up by the instruction decode logic. For example, you start by doing an instruction fetch. This means that you have to select the PC to the Block RAM address (this is independent of the old instruction already in the instruction register so this is a bit of logic that depends both on the decode and on the state machine outputs), do a read cycle (remember that Block RAM reads are synchronous), and transfer the data to the instruction register

(enabling the instruction register to latch on the next clock edge). Note that most of these actions involve enabling certain things to happen. These enable signals are outputs from your state machine.

The instruction fetch part of your state machine is the same for any instruction. Once you fetch the instruction, you may have to have the state machine do something different depending on which instruction is being executed. Most instructions will be pretty similar but see the 3810 example to see that there are subtle differences based on which type of instruction is being executed. On a load, for example, you will have to find the address to load from, send that address to the Block RAM, do a read cycle to memory , and transfer that data into the register file (enable the RF to write on the next clock). A store is also a strange one. You need to find the address and send it to the Block RAM, and also compute the data to store and send that to the Block RAM. Then you do a write cycle to the Block RAM. Make sure you understand the needs of each and every instruction before you design the state machine.

When you write Verilog code for the FSM, I encourage you to be explicit about what parts of the code are sequential and what parts are combinational.

Assembler

At some point you'll need to be able to generate code to run on your machine, and put the generate machine code into a text file that can be used to initialize the Block RAM on the FPGA. You probably want to do this using an assembler rather than write machine code directly. At its simplest, an assembler is just a translator from symbolic assembly code into machine code. So, if you had an instruction like ADD R1 R2 the assembler would convert this into 0000000101010010 as described in the instruction set document (the CS/ECE 3710 ISA document). Of course, you can make an assembler as complex as you like by introducing features like symbolic labels which get bound to actual addresses as the code is assembled, etc.

There is an example assembler from last year on the Canvas page written in Python. I haven't played around with it myself, but it may be a good starting point for your project. You will likely have to add some things to it, especially if you've added instructions beyond the baseline. You may also want to add macros that are shorthand for assembly sequences.

What to do

Once the CPU + FSM + datapath (ALU + Regfile) + PC + Memory design is completely simulated and synthesized, demo the correct operation of the machine to the TA or the insatructor. Your FINAL test program should reside in memory, and use the following operations:

1. Load data from memory into the regfile;
2. Perform a series of arithmetic and logical operations (set flags);
3. Store the result in memory;
4. reload the result from memory into the regfile;
5. perform arithmetic to set flags;
6. use the flags to do branches (BEQ, BLT, etc.); and
7. 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 this checkpoint, because it's basically a working CPU, you should write a short report as a group. 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.

In the document, make sure that all the issues are properly described size of the PC, unsigned or 2C representation and how it is handled, show the state diagram of the FSM, how many clock cycles does

each (type of) instruction take, 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 teams 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.

Also please submit all the Verilog code you wrote for the FSM and other parts of the processor that had to be changed to make things work.

Above all - **have fun!** This is a big checkpoint, but it should be very gratifying to see your CPU actually running code!