Programming Assignment 4-1: CPU R-Types

CS141 Spring 2019

Due April 5, 2019 at 5pm

Introduction

We'll be implementing the following subset of the MIPS ISA in the next few programming assignments.

- R-Types
  - `and`
  - `or`
  - `xor`
  - `nor`
  - `sll`
  - `srl`
  - `sra`
  - `slt`
  - `add`
  - `sub`
- I-types
  - `andi`
  - `ori`
  - `xori`
  - `slti`
  - `addi`
  - `beq`
  - `bne`
  - `lw`
  - `sw`
- J-types
  - `j`
  - `jal`
  - `jr` (technically an R-type, but we'll include it here because it's a jump)
- nop

We will be starting with R-types, then moving on to I-types and J-types. Getting R-types to work requires you to write most of the datapath, so make sure to start early. Chapter 7 of your textbook is an excellent resource for writing the CPU.

MIPS CPU Core Components

The start project includes an ALU, register file, and instruction/data memory (sources are in the `remote_sources` directory, if you're not in the Xilinx IDE), along with some useful definitions. Other

than that, you will be writing most of the datapath on your own.

The heart of the MIPS CPU consists of the following core components:

ALU

You could use your ALU from Lab 2 if you didn't have any bugs, but we highly recommend that you instead use the working behavioral ALU in the Xilinx Project.

Register File

We suggest that you use the register file included in the project.

Memory

The memory is your CPU's only real input and output. We will simulate our CPU by preloading machine code into the memory, running the CPU for enough time, and then writing out the memory's final contents to a file. By comparing the simulated final memory state with what the final memory state should be, we can verify that the CPU is doing the right thing.

Creating memories that work properly in both simulation and synthesis in Verilog is a bit tricky (check out the Xilinx XST guide if you are curious about how to do this). Making a testbench that can deal with input and output memory files is also non-trivial in Verilog - a working memory has been provided for you as well as a testbench that you will need to finish for testing your datapath.

The provided memory has 1024 32-bit rows for instructions and 1024 32-bit rows for data. Recall that in your assembler, you assumed that instructions begin from address 0x00400000. Be sure that this is where you start your PC!

The memory is von Neumann (we can read and write to both the instruction and data address spaces) compared to a Harvard architecture (where the instruction address space is read only). The two address spaces are separated to allow for easy intialization in synthesis. It's also a common trick for advanced computer architecture techniques (having separate address spaces for instruction and data allows for separate instruction and data caches, for instance).

> The memory provided changes its outputs on the next positive edge of the clock, while your textbook assumes an ideal memory that can instantly output any row. This slower memory is crucial to how the synthesis rig works, so you have to modify the book's datapath to account for this delay. Although we won't actual demonstrate the CPU on the FPGA, we want to make sure we write synthesizable Verilog.

By default, the memory will read in the `tests/current_test.memh` file (use your assembler to build this from human readable assembly). You will need to modify either the memory file or `tests/current_test.memh` to change what you are simulating! See the `test_mips_multicycle.v` file for more details. You can download a sample memory file which contains just one instruction. Create a tests directory in your project, place the memory file there, and add it to your project.

Datapath

The datapath is the finite state machine that ties the rest of the components together. Just be aware that while your textbook does describe most of the multicycle datapath, it does not cover all of the instructions that you are responsible for implementing, and it assumes a perfectly fast memory (which you do not have!). If you blindly follow the diagrams in the book your CPU will not work — you have to think!

We will be requiring you to sketch out your state machine for the multicycle datapath as you go. This makes debugging and reasoning about your CPU far easier.

Part 1: Just the R-types

We need to incrementally test our CPU to make sure that the pieces work as we implement them. To do this, we need to write assembly level tests, run them, and see how the register file or memory is altered. To do this, first write some assembly code that tests a limited set of instructions. Then use your assembler (or the solution assembler) to generate a file that has one 32-bit hexadecimal instruction per line. At the start of the provided testbench the in.machine file will be loaded into memory - feel free to change the in.machine file name in the code to the name of your assembled test. Once the simulation is done it will output the the status of the memory into the file "out.machine". This won't have any useful information in it until we have our SW command working, so you will need to modify the `test_mips_multicycle.v` testbench to dump the final state of the register file onto the console.

The goal of this part of the lab is to implement the following R-type instructions:

- `and`
- `or`
- `xor`
- `nor`
- `sll`
- `srl`
- `sra`
- `slt`
- `add`
- `sub`

and to write the necessary assembly test files that show that all of these operations work (in simulation). Some tips/suggestions:

- A simple test can make use of the NOR instruction to get `0xFFFFFFFF` (`-1` in decimal) into any register, i.e. `NOR $1, $0, $0` will put the value -1 into register 1
- Before you can do anything in the CPU, you need to be able to fetch instructions! Prioritize implementing your fetch unit and make sure that IR gets loaded with the right instruction every fetch state. Also make sure that your PC increments by 4 by the end of every instruction!
- Implement a decoder module that converts the 32-bits of the current instruction (IR) into meaningful sub-fields (op code, funct, rs, rt, rd, immediate, etc.).
- Implement another decoder module to determine what the ALU should do for the different possible funct codes in the instructions.
- We strongly recommend making liberal use of header files for defining various constants (i.e. op codes, FSM state names, etc.). This will make your code much more readable and will make debugging much easier.

- When debugging, make sure that you can look at how your state machine reg, the PC reg, and the IR regs are easy to see. Reading the isim documentation will provide many useful techniques for analyzing waveforms.

Deliverable

You should submit the Verilog code for your CPU working with the R-type instructions specified above. You will also need to show the assembly files you wrote to test your partial datapath.

Note: You have until April 5 to complete this lab - it is difficult and time consuming, so please get started early! In order to easily create machine code from your assembly test files, you will need your assembler from Lab 4-0.

Rubric (100 points)

Functionality/feature	Points
Schematic of MIPS Datapath/FSM	10
Fetch	20
Decode	20
Execute	20
Writeback	20
Testing methodology	10