CS 451 Project 2 Fall 2011

Assemble a Single-Cycle CPU

Due: Friday, 14 October

For this project, you will use JLS to assemble the Single Cycle CPU described in Chapter 7 of the Harris and Harris textbook (Chapter 5 in the Patterson and Hennessey textbook).

Pre-requisites

You will need to understand

- how to turn assembly language statements into machine language statements (you did this for Lab 4), and
- how the single-cycle CPU shown in Figure 7.14 on page 375 (Figure 5.24 on page 314 in Patterson and Hennessy) works.

Instructions

For this project, you are "assembling" the single-cycle CPU from the components shown in Figure 7.11 on page 375 (Figure 5.24 on page 314 in Patterson and Hennessy): a register file, an ALU, instruction memory, and a starter control unit. The register file, ALU, and instruction memory, are not difficult to implement; however, they will take too long to implement relative to the amount you would learn from the exercise. For the next project, you will implement your own control.

Begin by using JLS to assemble the single-cycle CPU shown in Figure 7.11 on page 375 (Figure 5.24 on page 314 in Patterson and Hennessy). Your CPU should have all the data paths to support the following instructions:

Name	Mnemonic	Format	OpCode (in hex)	Func Code (in hex)
Add	add	R	0	20
Add Immediate	addi	I	8	
And	and	R	0	24
And Immediate	andi	I	c	
Branch if Equal	beq	I	4	
Halt	halt		20	
Jump	j	J	2	
Load Word	lw	I	23	
Load Upper Immediate	lui	I	F	
Nor	nor	R	0	27

Or	or	R	0	25
Or Immediate	ori	I	d	
Set Less Than	slt	R	0	2a
Set Less Than Immediate	slti	I	a	
Store Word	sw	I	2b	
Subtract	sub	R	0	22

Interface

In order for your CPU to work seamlessly with the test suite, it should use the following interface:

- Name the circuit for the entire CPU `PHSingleCycleCPU.jls`.
- When you import the Instruction Memory, name it `InstMemory`.
- When you import the Register File name it `RegisterFile`.
- Main Memory is simply a built-in RAM. Name it `MainMemory`.

Other Instructions

- The provided control unit has two outputs in addition to the control lines: `Error` and `Halt`. The control recognizes a `halt` instruction (op code `0x20`). In response, it places a `1` on the `Halt` output line. The starter control sets the `Error` line to 1 when it receives an input that does not correspond to an instruction. I included the `Error` line to make debugging easier. Your CPU need not utilize it.
- Attach a "stop simulator" widget (the stop sign) to the Control's "Halt" line. This is what causes the simulator to stop at the end of a program.
- Your Main Memory must contain at least `2^20` words.
- Put a comment in `PHSingleCycle.jls` showing the clock cycle time.

Hints

- The built-in Register file simulates *negative*-edge triggered flip-flops. Your registers (e.g., the program counter) should also use negative-edge triggered flip flops. When using negative-edge triggered flip-flops, the clock's falling edge defines the beginning of each CPU cycle.
- Think carefully when configuring your clock. The clock period needs to be long enough for all calculations to complete within one cycle.
- The clock's "one time" and "zero time" need not be equal.
- Be careful when connecting Main Memory. You don't want to set `WE` to 0 until the `data` and `addr` elements are set. (Otherwise, you may end up writing to a random spot in memory.) One trick is to make the clock's "one time" small, and wait to set `WE` until the clock rises from 0 to 1. If the "one time" is small compared to the "zero time", the rising of the clock from 0 to 1 will come at the very end of each cycle.
- Remember, the `WE`, `OE`, and `CS` inputs to RAM and ROM are active low.
- Even though the RAM and ROM control inputs are active low, none of the Control wires are active low. Specifically, the `memWrite` and `memRead` wires are 1 when a memory write or memory read is desired.
- The built-in RAM and ROM circuits are *word* addressable, whereas the memories in the textbook are *byte* addressable. Make sure your circuit converts between them correctly. (The Instruction Memory circuit I provide takes bytes as input, not words.)

- Read carefully about how MIPS implements jump and branch. In particular, notice (1) how MIPS fills bits 28-32 of the jump target and (2) that the branch offset is added to `PC + 4`, not `PC`.
- JLS's un-bundle feature puts the low-order bits at the top. This is different from the diagrams in the book. Read carefully.
- Make use of probes, watches, and displays to help debug your circuit.

Testing

To test your CPU, you have it execute assembly code, then check that the registers and memory contain the expected values. **You** are responsible for writing code to thoroughly test your CPU. I am providing only the demonstration test files below. They are nowhere near a complete test of your CPU:

- `basicOpsDemoTest.a`.
- `basicLoadStoreTest.a`.
- `Multiply.a`.

Running tests automatically

The `JLSCircuitTester` suite contains a program, `jlsCPUTester`, that takes as input (1) your `JLS` circuit and (2) a file containing MIPS assembly code and

1. assembles the file into MIPS machine code,
2. simulates the running of your CPU with the machine code,
3. runs the machine code using MARS (a SPIM-like CPU simulator), and
4. compares the final state of your CPU to that calculated by MARS.

As with previous projects, all the executable code needed for this project is in `/home/kurmasz/public/CS451/bin`. If you add this directory to your path, you will be able to run all test scripts directly from the command line. If you choose not to add this directory to your path, you will have to use the full path name when launching the scripts below. To run the test scripts on your own machine, you must follow the `JLSCircuitTester` installation instructions.

As always, please tell me as soon as possible if you have any problems.

Two Notes:

1. Registers 0, 1, and 25-31 have special purposes in MIPS. Be careful when writing MIPS assembly code using these registers explicitly. Although it won't affect the running of your CPU under JLS, it may affect how MARS simulates your code and, thus, affect whether `jlsCPUTester` detects some bugs.
2. Be aware of pseudo instructions when writing MIPS code. For example, `lw $16, val1` is a pseudo instruction. It won't work unless both `lui` and `lw` work.

Running tests by hand

To debug your CPU, you may want to run test programs by hand. The Instruction Memory I provided is configured to load instructions from a file named `instructions`. You will want to maintain several different test programs. Make sure these programs are not named `instructions`, and simply copy (not move) the program you want to test to `instructions`. Once you have copied the file, JLS will automatically load instruction

memory with the desired program. (The file `instructions` must be in the current working directory. This is the directory from which you launched JLS. Be careful when launching JLS from an icon because it may not be obvious which directory is the current working directory.)

The `JLSCircuitTester` suite contains `marsAssembler`, a program that will take assembly files as input and generate MIPS code formatted for use as an `instructions` file: The file contains two columns. The first is the *word* address of the instruction in hex, and the second is the MIPS binary instruction in hex. Currently `marsAssembler` writes only to the standard output, so you will want use file redirection to save the output to a file.

To examine the results of your program, place a watch on any registers of interest and the RAM unit for Main Memory.

Submission and grading

This project will be worth 100 points, divided as follows:

I-type	15 points	
R-type	15 points	
lw	10 points	
sw	10 points	
branch	10 points	
jump	10 points	
overall design	10 points	
neatness / documentation	5 points	(This means add comments! Please put the clock configuration in one of the comments.)
testing	15 points	

You loose "overall design" points for inefficiencies in your design. For example, you will loose points for setting your clock period ridiculously high instead of calculating the critical path length.

Note: It may not be possible to effectively test some instructions if others are broken. For example, it is not possible to test R-type instructions without first using an I-type instruction to load some registers. The list above reflects my testing order. If it is not possible to easily test category `y` because category`x` is broken, you will receive 0 points for category `y`. Thus, if your I-type instructions are broken, you will likely receive a 0 for the assignment.

Testing

Remember, you are responsible for writing code to thoroughly test your CPU. When you are convinced your CPU works, e-mail me your `PHSingleCycleCPU.jls` file, along with a list of any features you did not implement. I will run my test scripts on your file. If my scripts show no errors, you will receive 15 points for your testing component. If my scripts turn up an error, you will get a chance to fix your code, but will loose 5 points. If your second submission also shows bugs, you will loose 5 more points. Your third submission will simply be graded, regardless of whether there are bugs. If you know a particular instruction doesn't work, I will exclude it from testing. You will loose points only for bugs that you have not identified yourself.

- You are encouraged to consult with other groups when designing your test programs.
- You are also encouraged to test your CPU using programs written by other groups.
- Save your test programs for use in future projects.

Deliverables

After your CPU has passed my tests:

1. Add a text box to your `jls` files that includes
    - your name,
    - the file name,
    - the assignment name, and
    - the assignment due date.
2. Submit all relevant `.jls` files, bug reports, and assembly code using Assignment Manager.
3. Print and submit hard copies copies of all relevant circuit diagrams. (When printing diagrams use the "File -> Print -> Just Visible Window" option. There should be one page per `.jls` file.)