CS/CoE 0447 Computer Organization

Spring 2016

Project 2: JrMIPS Processor
Assigned: March 30, 2016.
Due: April 20, 2016 by 11:59 PM.

Updated project description. The minor change is marked in red in Section 2.2.

1 Introduction

In this project, we'll implement in Logisim a single-cycle processor that resembles MIPS. We'll call the new processor and instruction set, JrMIPS, version 0.9, March 17, 2016. Your processor will be capable of running small programs.

2 JrMIPS Programmer's Reference Manual

JrMIPS is a simplified architecture. The native word size is 16 bits. That is, instructions and data values are 16 bits wide. Two's complement and unsigned data types are allowed for data values. There are eight 16-bit general-purpose registers ($r0 to $r7) and two special-purpose 16-bit registers ($HI and $LO). There are separate instruction and data memories. The instruction memory can hold up to 256 instructions and the data memory can hold up to 256 data values.

2.1 Instructions

JrMIPS has a small number of instructions:

In this table, "X" indicates the *Subop* field (see below) is not applicable and the value doesn't matter. Some opcodes are not listed because these codes are reserved for future instructions.

There are some differences between JrMIPS and the regular MIPS instructions. First, JrMIPS is a "two-operand" instruction set. An instruction has at most two operands, including source and destination operands. In this instruction set style, one of the source operands (registers) is also the destination. For example, consider the `add` instruction:

```
add $r2,$r3
```

This instruction will add the contents of source registers $r2 and $r3 and put the result into destination register $r2. Register $r2 is used both as a source operand and a destination operand.

Most instructions behave like their MIPS counterparts. An important exception involves branches, which use absolute addressing to specify a target address rather than PC-relative addressing. The branches also test the conditions "equal to zero" (branch zero), "not equal to zero" (branch not zero), "less than zero" (branch negative) and "greater than zero" (branch positive).

The `put` instruction causes the contents of $rs to be output to a LED hexadecimal display. This instruction will assist in debugging.

The `halt` instruction causes the processor to stop and a stop LED to turn red.

2.2 Instruction Format

JrMIPS has two instruction formats: R and I. R is used for instructions that have only registers and I is used for instructions with an immediate. The formats are:

Rs is the first source register and *Rt* is the second source register. *Rs* is the destination register.

Imm is an 8-bit immediate. The immediate is signed in `addi` and unsigned in `addui`, `bn`, `bx`, `bp`, `bz`, `jal`, and `j`. In `addi`, the bit *Subop* controls whether the immediate is sign or zero extended.

Opcode	Subop	Format	Instruction	Definition
0010	0	R	`and $rs,$rt`	`$rs` ← `$rs & $rt`
0010	1	R	`nor $rs,$rt`	`$rs` ← ¬(`$rs | $rt`)
0000	0	R	`add $rs,$rt`	`$rs` ← `$rs + $rt`
0000	1	R	`sub $rs,$rt`	`$rs` ← `$rs - $rt`
0001	0	I	`addi $rs,imm`	`$rs` ← `$rs + sxt(imm)`
0001	1	I	`addui $rs,imm`	`$rs` ← `$rs + zxt(imm)`
0011	0	R	`div $rs,$rt`	`$LO` ← `$rs` ÷ `$rt`; `$HI` ← `$rs` mod `$rt`
0011	1	R	`mul $rs,$rt`	`$HI:$LO` ← `$rs` × `$rt`
0100	0	R	`mlo $rs`	`$rs` ← `$LO`
0100	1	R	`mhi $rs`	`$rs` ← `$HI`
1100	0	R	`lw $rs,$rt`	`$rs` ← `MEM[LeastSignificantByte($rt)]`
1100	1	R	`sw $rs,$rt`	`MEM[LeastSignificantByte($rt)]` ← `$rs`
1111	0	I	`li $rs,imm`	`$rs` ← `zxt(imm)`
0101	X	I	`bp $rs,imm`	`PC` ← (`$rs > 0 ? imm : PC + 1`)
0110	X	I	`bn $rs,imm`	`PC` ← (`$rs < 0 ? imm : PC + 1`)
0111	X	I	`bx $rs,imm`	`PC` ← (`$rs` ≠ `0 ? imm : PC + 1`)
1000	X	I	`bz $rs,imm`	`PC` ← (`$rs = 0 ? imm : PC + 1`)
1110	X	R	`jr $rs`	`PC` ← `$rs`
1001	X	I	`jal $rs,imm`	`$rs` ← `PC + 1`; `PC` ← `imm`
1101	X	I	`j imm`	`PC` ← `imm`
1011	X	R	`halt`	`stop fetching and set halt LED to red`
1010	X	R	`put $rs`	`output $rs to Hex LED display`

R Format Instruction					
Bit posn	15-12	11-9	8-6	5-1	0
Field	*Opcode*	*Rs*	*Rt*	*unused*	*Subop*

When *Subop* is 1, then *Imm* is zero extended to implement the `addui` instruction. Otherwise, *Imm* is sign extended to implement `addi`. *Imm* is zero extended for branches and jump (`j`).

In branches, `jal` and `j`, *Imm* specifies the target address. Both branches and jumps use absolute addressing for the target address. So, for example, if a branch is taken and *Imm* is 0x1a, then the target address for the branch is 0x1a.

2.3 Registers

There are eight general-purpose registers, labeled $r0 to $r7. There are two special-purpose registers, labeled $HI and $LO. These registers are used to hold the result from multiplication and division; see the semantics of the `mul` and `div` instructions (see the table in Section 2.1). The instructions `mhi` and `mlo` can be used to copy the contents of these registers into a general-purpose register. The registers are 16 bits wide.

2.4 Instruction Addresses

The instruction memory holds 256 instructions. Each instruction is 16 bits wide. An instruction address references a single instruction as a whole. Thus, an instruction address has 8 bits to specify one of 256 instructions in the memory.

I Format Instruction				
Bit posn	15-12	11-9	8-1	0
Field	*Opcode*	*Rs*	*Imm*	*Subop*

2.5 Data Addresses

The data memory holds 256 16-bit data words. A data address references a single data word as a whole. Thus, a data address has 8 bits to specify one of 256 words.

3 Project Requirements

Your job is to implement this architecture! Your processor will be a single cycle implementation: in one cycle, the processor will fetch an instruction and execute it.

Your implementation will need several components: 1) a program counter and fetch adder; 2) an instruction memory; 3) a register file; 4) an instruction decoder; 5) one or more sign extenders; 6) an arithmetic logic unit; 7) a data memory; 8) an LED hexadecimal display; and, 9) an LED to indicate the processor has halted. You'll also need muxes as appropriate. For the most part, these components are quite similar to what we've talked about in lab and lecture. **You will find it helpful to consult the book and class slides, particularly the diagram of the MIPS processor with control signals and the decoder and data path elements.**

For the project, you may use any component (e.g., an adder) from Logisim's built-in libraries. This makes the project much simpler! All other components must be implemented from scratch. Don't use or look at components that you might find on the Web or *any past CS 0447 project*! If you do look at this past material, this is considered cheating according to the course policy.

The usual policy about outside help applies for this assignment: It is not allowed. You may talk about how to approach the project with others, but you are not allowed to show your design or discuss specific decisions (e.g., control signal settings) with any one else other than the TAs and instructors.

4 Instruction Implementation

It is easiest to do this project with Logisim's subcircuits. For the more complicated components in the data path and control (e.g., ALU), define a subcircuit. A subcircuit is like a function in a programming language. It can be added, or "instantiated", multiple times in a design.

4.1 Clock Methodology

A clock controls the execution of the processor. On each clock cycle (a rising and falling edge), an instruction is fetched from the Instruction Memory. The instruction is input to the Decoder to generate control signal values for the data path. The control signals determine how the instruction is executed. You'll need a single Clock element in your design. This clock should be tied to all state elements (registers and ROM). The state elements in Logisim let you define the "trigger event" when a state element captures its inputs. I recommend that you use the default.

4.2 Program Counter

The program counter is a register that holds an 8-bit instruction address. It specifies the instruction to fetch from the instruction memory. It is updated every clock cycle with PC + 1 or the target address of a taken branch (or jump).

4.3 Instruction Memory

This component is a ROM configured to hold 256 16-bit instructions. You should use the ROM in Logisim's Memory library. In your implementation, the ROM must be visible in the main circuit.

The ROM's contents will hold the instructions for a JrMIPS program. You can set the contents with the Poke tool or load the contents from a file.

4.4 Data Memory

This component is a RAM configured to hold 256 16-bit words. You should use the RAM in Logisim's Memory library. In your implementation, the RAM must be visible in the main circuit. Be sure to read Logisim's documentation carefully for this component! To simplify the implementation, configure the RAM's Data Interface as Separate Load and Store Ports. This configuration is similar to what was described in lecture and the book. Hint: You'll need to set the RAM's *Sel* signal.

4.5 Register File

For the general-purpose registers, JrMIPS has 8 registers. An R-format instruction can read 2 source registers and write 1 destination register. Thus, the register file has 2 read ports and 1 write port. The register file is the same one that you implemented in lab, except it has more registers.

For the special-purpose registers, $HI and $LO, you can implement these as "standalone" registers that are updated by the multipier or divider (simply add a couple registers to your design and connect them to the outputs of the multiplier and divider). You will need to mux the input into the $HI and $LO registers from the multiplier and the divider. To implement the `mhi` and `mlo` instructions, you'll need to allow these registers to be muxed into the data input (i.e., the write port) of the register file.

4.6 Arithmetic Logic Unit (ALU)

The ALU is used to execute the arithmetic instructions: NOR, AND, ADD, and SUB. It may also be used to do branch comparison. Build the ALU as a subcircuit; it is similar to the ALU described in lecture. Be sure to use Logisim's multi-bit Arithmetic library subcircuits; most of what you need is already here! A mux is also needed.

4.7 Multiplier and Divider

Logisim includes a multiplier and divider. You can use these components to implement the `MUL` and `DIV` instructions.

The multiplier has two output pins: result and carry-out. The low portion of the full result from multiplication is Output; the high portion is Carry-Out. The low portion is written to register $LO and the high portion is written to register $HI. Likewise, the divider has two output pins: the Output pin is the quotient (put in $LO) and the Remainder pin is the remainder (put in $HI).

Division-by-zero is undefined. The processor should ignore this error situation. Logisim's Divider treats division-by-zero as division by 1 (divider = 1).

You will also need to introduce the two special-purpose registers ($LO and $HI) and a way to copy these registers to the general-purpose registers (in the register file).

4.8 Sign Extender

Logisim has a built-in sign-extender (signed and zero extension) component, which you can use.

4.9 Decoder

This component takes the instruction opcode as an input and generates control signal values for the data path as outputs. It's easy to make a decoder subcircuit with Logisim's Combinational Analysis tool (Window→Combinational Analysis). This tool will automatically build a subcircuit from a truth table. To make the decoder, list the opcode bits (from the instruction) as table inputs and the control signals as outputs. For each opcode, specify the output values of the control signals. Once you've filled in the table, click Build to create the circuit. To make your main circuit prettier,

the opcode inputs and ALU operation outputs can be combined into multi-bit input and output pins using Splitters. Hint: Logisim has a limit on the number of fields in a truth table. So, you may need two (or more!) decoders.

4.10 LED Hexadecimal Display

JrMIPS has a four digit hexadecimal (16 bit) display. `put` outputs a register value to this display The contents of a `put`'s source register (16-bit value) is output on the display. A value that is "put" must remain until the next `put` is executed.

To implement the LED Hexadecimal Display, you should use Logisim's Hex Digit Display library element (in the Input/Output library). You'll need four Hex Digit Displays, where each one shows a hex digit in the 16-bit number. A way is also needed to make the display stay fixed until the next `put` is executed (don't simply wire the hex digits to the register file!). Hint: Use a separate register. The display should only be updated when the `put` instruction is executed.

4.11 Halting Processor Execution

When `halt` is executed, the processor should stop fetching instructions. The main circuit must have an LED that turns red when the processor is halted. Hint: A simple way to stop the processor is to AND the program counter control with a halt control signal (that also turns on the LED).

4.12 Extracting Instruction Bit Fields

Individual fields in a 16-bit instruction need to be extracted. For example, *Opcode* needs to be extracted for the decoder. Likewise, register numbers and the immediate have to be extracted. This operation can be done with a subcircuit that has splitters connected to appropriate input and output pins. This component will simplify your main circuit drawing.

5 Assembler

I wrote a rudimentary assembler in Perl. You can get the assembler from the project directory in CourseWeb. A separate document describes the assembler.

6 Project Suggestions

Building the JrMIPS processor is like writing a program. Plan your design carefully before trying to implement anything. Once you have a good plan, implement the design in small parts. Test each part independently and thoroughly to make sure it behaves as expected. Once you have the different parts, put them together to implement various classes of instructions. I started with `put`, `li` and `halt` to make testing easy. After these worked, I added the arithmetic instructions. Next, I tackled branches, and then loads and stores. Finally, I implemented jumps.

I strongly suggest that you implement simple test cases as you progress in the project. The test cases should check that each of the instructions works. By starting with `put` and `li`, you will have an easy way to check the remaining instructions. To test instructions, use `li` to set registers to specific values. Next, use the instruction being tested on those values. Finally, use `put` to output the result of the instruction to the Hex display so you can visually check that the instruction under test computed the expected result. Repeat this process for each instruction; be sure to test all cases (e.g., does sign extension in `addi` work with a negative value?). Assuming `li`, `put` and `halt` work, here is an example to test `add`:

```
li $r0,0x1  # source operand 1: value 1
li $r1,0x2  # source operand 2: value 2
add $r0,$r1 # result in $r0 should be 3 (1 + 2)
put $r0     # hex display should show 0x0003
halt        # end program
```

Subcircuits will make the project easier. To define a subcircuit, use the Project→Add Circuit menu option. Next, draw the subcircuit. Finally, add input and output pins to the subcircuit. Be sure to label each pin (i.e., set the pin's Label attribute). Once you're done with the subcircuit, double click the main circuit in the design folder (on the left side of the window). This will switch to the main circuit. Now, the subcircuit will appear in the design folder. It can be instantiated (added) in the main circuit by selecting and placing it in the main circuit. The subcircuit should be wired to the main circuit through its input and output pins. Logisim isn't smart about how it handles changes to instantiated subcircuits. If you make changes to a subcircuit that is already instantiated, I recommend that you delete it from the main circuit first. Then, make your changes and re-instantiate it. See Logisim's subcircuit tutorial.

When you build your main circuit, follow a few conventions to make it easier to understand. First, wire your data path in a way that data signals flow from left to right (west to east). Second, wire control inputs so they run bottom to top (south to north). As a consequence of these two guidelines, put data input pins on the left and data output pins on the right in a subcircuit. Control input pins should be on the bottom of the subcircuit. Third, leave plenty of space between different elements in the design. This will make it easier to add more elements and route wires cleanly. Finally, try to route wires in straight lines as much as possible.

7 Turning in the Project

7.1 What to Submit

You must submit two files:

`jrmips.circ` (the circuit implementation)
`README.txt` (a help file; see next paragraph)

Put your name and e-mail address in both files. For the circuit file, use the text widget to list your name and e-mail address. In the README, put the name and e-mail address at the top of the file. The README file should list what works (i.e., instructions implemented) and any known problems with your design. You should also explain the purpose of each control signal with sufficient detail that the teaching assistant can understand your approaches without further inspecting the circuit.

If you have known problems/issues (bugs!) with your design (e.g., certain instructions don't work, odd behavior, etc.), then you should clearly specify the problems in this file. The explanation and bug list are critical to grading. So, if you're uncertain whether to include something, then err on the side of writing too much and just include it.

Please submit by April 20, 2016 at 11:59 PM. Per the course policy on late work, we will accept late projects through April 23, 2016. We will not accept late projects after this date because we need to complete all project grading during finals week. It is strongly suggested that you submit your file well before the deadline. If you have a problem during submission, we cannot guarantee to respond at the last minute before the deadline.

7.2 Where to Submit

Submit your files to CourseWeb by the deadline.

8 Collaboration

In accordance with the policy on individual work for CS/CoE 0447, this project is strictly an individual effort. Unlike labs, you must not collaborate with a partner. Please see the course web site (syllabus) for more information. Do not view or use past solutions for CS/CoE 0447 in completing this project. It is your responsibility to secure and back up your files. See the course

web site for more details about the course policy on collaboration.

8.1 Grading the Project

We will grade the project with multiple test cases covering all the instructions. To ensure that your processor is working as expected, you will find it very useful to write test cases for each instruction.