CoE 0147 Computer Organization and Assembly Language

1 Introduction

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

2 CoEMIPS Programmer's Reference Manual

CoEMIPS is a greatly simplified architecture. It has 8 registers, instructions are 16 bits, the native word size is 16 bits, 2's complement and unsigned numbers are used, the instruction memory can hold 256 instructions and the data memory is 256 data words.

2.1 Instructions

CoEMIPS has a small number of instructions:

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

In this table, "X" indicates the *Subop* field (see below) is not applicable for I-format instructions. Opcode 0010 is not listed because it is reserved for a future instruction.

There are some differences between CoEMIPS and the regular MIPS instructions. First, CoEMIPS 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).

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

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

2.2 Instruction Format

CoEMIPS 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:

R Format Instruction						
Bit posn	15-12	11-9	8-6	5	4-1	0
Field	*Opcode*	*Rs*	*Rt*	*unused*	*Shamt*	*Subop*

I Format Instruction				
Bit posn	15-12	11-9	8	7-0
Field	*Opcode*	*Rs*	*Unsigned*	*Imm*

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 *Unsigned* controls whether the immediate is sign or zero extended. When *Unsigned* 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 16d, then the target address for the branch is 16d.

Shamt is used by shift instructions. There are two varieties of shifts. One version encodes the shift amount as a 4-bit unsigned immediate in *Shamt*. The other version provides the shift amount as a source register. Notice that the left and right shifts of each variety have the same opcode. The shift direction is encoded in *Subop*. When *Subop*=0, the direction is left and when *Subop*=1, the direction is right.

2.3 Registers

There are 8 registers, labeled $r0 to $r7. In CoEMIPS, $r0 is not 0. It's a general register and can be used like any other 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.

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) a sign extender; 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 CoE 0147 or CS 447 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 CoEMIPS 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

CoEMIPS has 8 registers. A 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.

4.6 Arithmetic Logic Unit (ALU)

The ALU is used to execute the arithmetic instructions: AND, NOR, 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 Shifter

Logisim includes a shifter. You can use this component to implement left and right shift operations.

4.8 Sign Extender

Logisim has a built-in sign-extender 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

CoEMIPS 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 instruction register write control with a halt control signal (that also turns on the LED). An alternative way to halt is to set the program counter's enable signal to 0 when a halt instruction is executed.

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 borrowed a rudimentary assembler jrmipsasm.pl from the CS 447 folks and it also part of this email. The documentation for the assembler is at the end of this PDF (pages 7–11).

6 Project Suggestions

Building the CoEMIPS 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` 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 shifts and jumps. You can find numerous test case programs on the project web site. I recommend that you write and try additional test cases as well.

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 w[illegible]ight lines as much as possible.

7 Turning in the Projec

7.1 Files to Submit

You must submit a compress[illegible]ile (`.zip`) containing:

- `coemips.circ` (the [illegible]cuit implementation)
- `README.txt` (a help [illegible]; see next paragraph)

Put your name and e-mail address in both files. For the circuit file, write your name and e-mail address with the text widget. 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.

The filename of your submission (the compressed file) should have the format

`firstname.lastname-p2.zip`

Files submitted after April 17, 2015 at 11:59 PM will not be graded. 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

Courseweb

7.3 Grading the Project

We will grade the project with multiple test cases, some of which will come from the tests that were distributed as part of this email.

CoE 0147 Spring 2013

Assembler Reference Manual
Borrowed with gratitude from the CS 447 folks, i.e.,
Prof. Childers and Co.

This document describes the rudimentary assembler for CoEMIPS.

The assembler uses Perl. You'll need access to Perl to use the assembler. You can get Perl from: http://www.perl.com/download.csp

1) Assembly Language Syntax

The CoEMIPS assembler supports the instructions and assembly language syntax:

```
and   $rs,$rt
nor   $rs,$rt
add   $rs,$rt
addi  $rs,imm
addi  $rs,dlabel (this is equivalent to la below)
sub   $rs,$rt
sll   $rs,shamt
srl   $rs,shamt
sllv  $rs,$rt
srlv  $rs,$rt
lw    $rs,$r1
sw    $rs,$r1
bp    $rs,label
bn    $rs,label
bz    $rs,label
bx    $rs,label
jal   $rs,label
jr    $rs
j     label
not   $rd,$rs (pseudo instruction)
or    $rd,$rs,$rt (pseudo instruction)
clr   $rs (pseudo instruction: set $rs to 0)
li    $rs,imm (load immediate into $rs)
la    $rs,dlabel (load address of data labl dlabel into $rs)
put   $rs
halt
```

where, `$rs` and `$rt` are one of names `$r0,$r1,$r2,$r3,$r4,$r5,$r6,$r7`;
`shamt` is an unsigned value;
`label` is a symbolic label name declared in the text segment;
`dlabel` is a symbolic label name declared in the data segment; and,
`imm` is a signed constant decimal or hexadecimal value.

The assembler supports hexadecimal notation for values (immediates). To indicate hexadecimal, use "0x" before the value (e.g., 0xF00F is valid). A hex value may only be used in a location where an immediate value is expected, including instructions with an immediate (the last operand) or labels in the data segment.

Labels are allowed, but they must start with a letter and end with a colon. Labels are case insensitive. For example, the label `LABEL0` is the same as `Label0`.

The JrMIPS assembler supports a text and data segment. For the data segment, use the directive ".data" by itself on a line to begin the data segment. In this segment, you can declare initialized 16-bit word values. Each value may have a label. The syntax is:

label: value

where *label* is a symbolic name and *value* is the value to associate with the label. Unlike MIPS, you do not declare a type. All items in the data segment are 16-bit words. Every value must be declared on a line by itself with its own label. Hence, you cannot have a sequence of values separated by commas.

The data in the data segment must be processed and loaded separately from the text segment. The assembler option "-d" will output the data values in a form that can be loaded into a Logisim RAM component.

The text segment is the default. If you use a data segment, be sure to switch to the text segment with the ".text" directive.

Comments and blank lines are allowed. A comment is given with a semicolon (";").

There is almost no error checking by the assembler, so tread carefully.

2) Assembling a Program

To run the assembler, use the command:

```
perl jrmipsasm.pl progname.asm
```

This will cause `progname.asm` to be assembled. The assembler will output hex encoding for the instructions to the display. The output is in a format that can be loaded directly from within Logisim into a ROM component.

To save the output from the assembler to a file, use the command:

```
perl jrmipsasm.pl progname.asm > progname.dat
```

Then, load `progname.dat` into ROM from Logisim. To load a program into ROM, open your processor design in Logisim, poke the ROM, and click on the ROM's Load contents attribute. Select the filename with the encoded program.

To output the data segment, use:

```
perl jrmipsasm.pl -d progname.asm > progdata.dat
```

Then, load `progdata.dat` into RAM. Click on the RAM and press Control (control-click). This will bring up a menu option "Load Image", which will set the RAM to the contents in the file. You need to reload the RAM every time you reset the simulation.

The assembler supports some command line options.

An option ("-p") disables all pseudo-instructions.

A second option ("-v") causes the assembler to print verbose information about the assembled program. This option is useful to verify that the assembler generated the correct output.

A third option ("-l") will cause the assembler to dump the addresses it used for the labels in the input assembly language program.

A final option ("-d") causes the assembler to output data segment information.

Appendix: Example Programs

Here is an example program:

```
           clr   $r0
           # increment from 0 to 15
           # lowest hex digit will cycle from '0' to 'F'
loop0:     put   $r0,0      # output current value
           addi  $r0,1      # increment value by 1
           mov   $r1,$r0    # check for loop end
           addi  $r1,-16    # end of loop?
           bn    $r1,loop0
           halt
```

The output from the assembler (without any command line options) is:

```
v2.0 raw
# JrMIPS ISA version v3 11/16/11
# to load this file into Logisim:
# 1) save the output from the assembler to a file
# 2) use the poke tool in Logisim and control-click ROM/RAM
# 3) select Load Image menu option
# 4) load the saved file
0001
F000
1001
0241
0200
12F0
B201
7000
```

This is a second program with a data section:

```
      .data
a:    10
b:    0
      .text
      la $r0,a
      lw $r1,$r0
loop: add $r2,$r1
      addi $r1,-1
      bp $r1,loop
      la $r0,b
      sw $r2,$r0
      lw $r3,$r0
      put $r3,0  # answer should be 37h
      halt
```

The assembled instruction file is:

```
v2.0 raw
# JrMIPS ISA version v3 11/16/11
# to load this file into Logisim:
# 1) save the output from the assembler to a file
# 2) use the poke tool in Logisim and control-click ROM/RAM
# 3) select Load Image menu option
# 4) load the saved file
0001
1100
6200
0440
12FF
A203
0001
1101
6401
6600
F600
7000
```

The data section file (loaded into the RAM) is:

```
v2.0 raw
# to load this file into Logisim:
# 1) save the output from the assembler to a file
# 2) use the poke tool and control-click the ROM/RAM component
# 3) select Load Image menu option
# 4) load the saved file
000A
0000
```