CS 447

Project 2: μMIPS

Released: 11:59 PM Monday, November 12th, 2018.

Due: 11:59 PM Friday, December 7th, 2018.

Let's make a CPU

Clarification/Errata: The HALT and PUT instructions (very obviously) had their descriptions reversed.

Clarification/Errata: The register file does, in fact, need 2 write ports (but just 1 write destination.)

Introduction

In this project, we'll implement in Logisim a single-cycle processor that resembles MIPS. We'll call the new processor and its implementation μMIPS, version 0.9.2, November 9, 2017. Your processor will be capable of running small programs.

Start early

The deadline is close to the end of the semester. Life happens, sickness happens, so if you start early you can minimize the impact.

Do a little bit every day! 1 hour every day! 30 minutes every day! SOMETHING!

You know you will have questions, and if you decide to ask them in the last week, I may not be able to answer them all!

μMIPS Programmer's Reference Manual

μMIPS is a simplified architecture. The native word size is 16-bits. That is, instructions and data values are 16-bits wide. μMIPS data can be both unsigned and signed using two's complement

values are 16-bits wide. μMIPS data can be both unsigned, and signed using two's complement. There will be 8 registers ($r0-$r7) for general-purpose use. Instruction and data memory will be separate! The instruction memory can hold up to 256 instructions, and the data memory can hold up to 256 data values.

Instructions

μMIPS has a small number of instructions:

Opcode	Subop	Format	Instruction	Definition
0000	0	R	and $rs, $rt	$rs ← $rs & $rt
0000	1	R	nor $rs, $rt	$rs ← !($rs | $rt)
0001	0	R	add $rs, $rt	$rs ← $rs + $rt
0001	1	R	sub $rs, $rt	$rs ← $rs - $rt
1000	0	I	addui $rs, imm	$rs ← $rs + zero_extend(imm)
1000	1	I	addi $rs, imm	$rs ← $rs + sign_extend(imm)
0011	0	R	div $rs, $rt	$rs ← $rs ÷ $rt; $rs+1 ← $rs mod $rt
0011	1	R	mul $rs, $rt	$rs+1:$rs ← $rs × $rt
0010	0	R	sllv $rs,$rt	$rs ← $rs << ($rt & 0x0F)
0010	1	R	slrv $rs,$rt	$rs ← $rs >> ($rt & 0x0F)
0100	0	R	lw $rs, $rt	$rs ← MEM[$rt & 0xFF]
0100	1	R	sw $rs, $rt	MEM[$rt & 0xFF] ← $rs
1001	X	I	li $rs, imm	$rs ← zero_extend(imm)

Opcode	Subop	Format	Instruction	Definition
1010	1	I	bn $rs, imm	PC ← ($rs < 0 ? imm : PC + 1)
1011	0	I	bx $rs, imm	PC ← ($rs ≠ 0 ? imm : PC + 1)
1011	1	I	bz $rs, imm	PC ← ($rs = 0 ? imm : PC + 1)
0101	X	R	jr $rs	PC ← $rs
1100	X	I	jal $rs, imm	$rs ← PC + 1; PC ← imm
1101	X	I	j imm	PC ← imm
0110	X	R	halt	stop fetching and set halt LED to red
0111	X	R	put $rs	output $rs to Hex LED display

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 μMIPS and the regular MIPS instructions. First, μMIPS 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

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.

Instruction Format

μMIPS 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 position	15-12	11-9	8-6	5-1	0
Field	*Opcode*	*Rs*	*Rt*	*unused*	*Subop*

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

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. For the addition instructions with an immediate (i.e., addi and addui), the bit Subop controls whether the immediate is sign or zero extended. 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.

Registers

There are eight general-purpose registers, labelled $r0 to $r7. The registers are 16-bits wide.

Because multiply produces a 32 bit result (from multiplying two 16 bit values), this instruction has two destination registers. These two destination registers are called a "register pair". The multiply instruction has a single register, $rs, specified as the destination in the instruction. The second register is implied. The register pair for the 32-bit result is the register number for $rs and the register number of $rs+1. $rs holds the low 16 bits and $rs+1 holds the high 16 bits of the result from the multiplication. For example, consider this instruction: mul $r2,$r5. When this instruction is executed, the register pair $r2 and $r3 will hold the resulting value of the multiply. $r2 is the low 16 bits of the multiply and $r3 is the high 16 bits. $r2 is specified in the instruction; $r3 is implied from the semantics of the instruction for a register pair.

Divide also has a register pair for the destination. In this case, $rs is the result (quotient) of the divide and $rs+1 is the integer remainder of the division (i.e., the mod).

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.

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.

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 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., a 16-bit 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 instructor, TAs or the CS resource center help desk.

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.

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.

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).

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 μMIPS program. You can set the contents with the Poke tool or load the contents from a file.

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.

Register File

For the general-purpose registers, μMIPS 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 2 write ports (the second write port is used for multiplication and division.) The register file is the same one that you implemented in lab, except it has more registers.

To determine the two registers in a register pair (see instructions mul and div), you will need to add 1 to the $rs register number to get the second register number ($rs+1) of the pair. Only the low 3 bits of the result from this addition are significant (there are only 8 registers).

Arithmetic and Logic Unit (ALU)

The ALU is used to execute the arithmetic instructions. 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.

For the shifts, you might want to have a separate functional unit to do the shift operations, or you can combine the functionality into the ALU.

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 $rs and the high portion is written to register $rs+1. Likewise, the divider has two output pins: the Output pin is the quotient (put in $rs) and the Remainder pin is the remainder (put in $rs+1).

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).

Sign Extender

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

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 Combinatorial Analysis tool (Window→Combinatorial 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.

LED Hexadecimal Display

μMIPS 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.

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).

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.

Assembler

Prof. Bruce Childers wrote a rudimentary assembler in Perl. You can get the assembler from the course website here. A separate document describes the assembler here. You can download two example programs from here, and here. If you want the pre-assembled versions: example_1.m example_2.m.

Project Suggestions

Building the μMIPS 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. E.g. start with put, li and halt to make testing easy. After these work, add the arithmetic instructions. Next, tackle branches, and then loads and stores. Finally, implement 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.

Submission instructions

Submit a single ZIP file with your project named username_proj2.zip (e.g., dww9_proj2.zip). In the zip file, there should be no folder!, just the following files:

Your umips.circ file containing your μMIPS implementation.

- Put your name and username at the top of the file using a label!

A readme.txt file. DO NOT SUBMIT A README.DOCX. DO NOT SUBMIT A README.PDF. SUBMIT A PLAIN TEXT FILE. PLEASE. It should contain:

- Your name
- Your Pitt username
- Your Pitt e-mail address
- Anything that works (i.e., instructions implemented)
- Any known problems with your design
- The purpose of each control signal with sufficient detail that the grader can understand your approaches without further inspecting the circuit.

Note: If you have known problems/issues (bugs!) with your design (e.g., certain instructions don't work, odd behaviour, 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.

Submit into the cs447-submissions Box folder I have shared with you. Your Box folder (you have 50GB!) is accessible through http://my.pitt.edu. When you see your file within the shared folder, you know you have uploaded it successfully. If you would like to resubmit, you can copy the file in again. Let me know immediately if there are any problems submitting your work.

Rubric

Here is the rubric for assignment of points out of a maximum 100 points. *Points relate to both the presence of the necessary components **and** their correctness*.

Processor Component	Maximum Points
Clock, Program counter (must be visible on main circuit)	3 pts

Processor Component	Maximum Points
ROM (instructions), RAM (data) both in main circuit	3 pts
Register File	4 pts
ALU (support for arithmetic and logic operations)	4 pts
Appropriate sign extender(s)	3 pts
Branch and Jump Controls	3 pts
Control/decoder, including choice of control signals	4 pts
Hex Displays and “halt” LED	3 pts
Instruction Set Functionality	
Arithmetic and logical operations (and, nor, add, etc)	18 pts
Memory operations (lw, sw)	10 pts
Branch operations (bp, bn, bx, bz)	10 pts
Multiplications and division operations (div, mul)	10 pts
Jump operations (jr, jal, j)	9 pts
Output and halt instructions	6 pts
Documentation	
README that describes problems and strategy employed.	10 pts

Test Programs

Some of you want some test programs?? Ok. You got it. You probably still want to write small programs to test individual instructions. This is **not** exhaustive of all instructions. Having all of these programs “*work*” **does not** imply your processor is correct.

		Machine	

Program	Assembly	Machine Code	Expected
Program	Assembly	Machine Code	Expected

Iterative Fibonacci	test_itr_fibo.asm	test_itr_fibo.m	Shows “8” as output
Recursive Fibonacci	test_rcr_fibo.asm	test_rcr_fibo.m	Shows “5” as output
Multiply Sequence	test_mul.asm	test_mul.m	Multiplies 1 x 2 x 3 x 4 x 5, showing each step, shows “78” as output
Divide Sequence	test_div.asm	test_div.m	Divides 120 by 5, then that result by 4, by 3, by 2, and then 1, showing each step, shows “1” as output
Bitfields	test_bitfields.asm	test_bitfields.m	Encodes an ‘add $rs, $rt’ instruction. Outputs: “14C0”

Collaboration Policy

You **cannot** work with one other person on your project. According to course policy! Do not view or use past solutions for CS/CoE 0447 in completing this project. See the course web site for more details about the course policy on collaboration.

Backups!!!!!!!!!!!!!!

It is your responsibility to secure and back up your files. Please backup! Accidents happen! BACKUP!!!

From the syllabus:

It is your responsibility to backup your work regularly. Please consider using reliable and multiple ways to protect your files! **I.e.: Please make backups, and backups of your backups, and…** (you get the point!) The University offers space that you can use for backups through Box, if you like. Both Google Drive and Dropbox have free tiers that also work well. No extensions for

if you like. Both Google Drive and Dropbox have free tiers that also work well. No extensions for assignments will be approved due to failed laptops, hard drive crashes, lost USB drives, or other calamities that lead to lost or corrupted data. Per the policy on voluntary late assignments, you may turn in a project up to five days late (with a penalty), which should give sufficient time to recover a lost/corrupted project.

CS 447

CS 447
dwilk@cs.pitt.edu

wilkie

CS 447 (26147); T-Th 9:30AM; 123 Victoria