Lam Ngo

Final Project

Chiron CPU

- Notes: When you use my assembler to run codes for my machine, be sure to read my coding convention section first.

I. Design Choices:

I started this final project with laboratory 4 CPU design as my basis. It was a 4-bit CPU with 14-bit instruction words, and was capable of basic R type instructions, such as add, sub and basic I type instructions, such as addi, lw and sw. That CPU had only 8 registers. However, as I wanted to support function calls, the stack pointer and bgt/blt instructions, I needed more registers than that: 1 register for $sp, 1 register for $a0, 1 register for $a1, 1 register for $v0, 1 register for $v1, 1 register for $at and 1 register for $ra. Thus, I decided to have 16 registers in my CPU. Because of that, I also increased the range of intermediate values that my CPU can handle from [-4,3] to [-8,7].

I chose to design a 16-bit CPU, so I had to build a 16-bit ALU. Compared to the 4-bit ALU I had in lab 4, my ALU now operates on 16-bit inputs, and has more operations: shift left logical, shift right logical and comparing. With this, I was able to design more instructions for my CPU. Later, as an extra credit, I chose to build a carry look ahead adder to use in my ALU instead of the stock adder in Logisim. I also had to remove my substractor (which was provided by logisim), and used my carry look ahead adder to do substraction (due to the increase in speed, though not noticeable at all).

For implementation of a stack, I hard-coded into my assembler three lines of code to initialize the stack pointer at the start of every program. I used addi and 2 sll instructions, to get my stack pointer as far as possible, so users' code cannot touch

it. Though, since the address bit width of my instruction ram is 16-bit, that is unlikely to happen.

To implement jump instructions, I decided to use the 12 -bit address in j instruction words, and concatenate it with the first 4 bit of PC, so I can have 16-bit address to update PC, and also jump relatively to PC. This is similar to what is implemented in MIPS.

Originally, beq could only have offsets ranging from -8 to 7 and can branch to labels. However, I felt that this was not enough, as there are many codes that demand branching to a label far away (such as in Professor John Rieffel's recursion test). Thus, I decided to change my assembler so that whenever the branch offset to a label is larger than 7 or less than -8, it will have two jumps followed close ly to make sure that it can jump to the correct place. For example, from beq $1 $2 base to:

```
beq $1 $2 1
j 2
j base
```

By doing this, branch can branch much further. However, due to lack of time, currently this only works with label, not with intermediate value offset.

Also, since currently there are no way to load an address into a register in my CPU, jr is hard coded to jump to $ra.

II. Machine Code instruction format:

- First 9 bits are the OPCODE.

Intermediate	RegWrite	LW	SW	Branch	Jump	ALU	RegDest	RegSource1	RegSource2 /intermediate
1 bit	1 bit	1 bit	1 bit	1 bit	1 bit	3 bit	4 bit	4 bit	4 bit

- Note:

- RegDest = Destination Register. RegSource = Source Register and RegWrite = Enalbe RegWrite.
- Machine code format for jump instruction is different, refers to the next section for more details.

III. Mapping of assembly code instructions to coressponding machine code:

1. Basic R-type:

- General form:

Intermediate	RegWrite	LW	SW	Branch	Jump	ALU	RegDest	RegSource1	RegSource2
0	1 bit	0	0	0	0	3 bit	4 bit	4 bit	4 bit

- Add: add the contents in the two source registers and save it in the destination register.

  example: add $1 $1 $1

Int	RegW	LW	SW	Branch	Jump	ALU	RegD	RegS1	RegS2
0	1	0	0	0	0	000	4 bit	4 bit	4 bit

- Sub: subtract the contents in the two source registers and save it in the destination register.

  example: sub $1 $1 $1

Int	RegW	LW	SW	Branch	Jump	ALU	RegD	RegS1	RegS2
0	1	0	0	0	0	011	4 bit	4 bit	4 bit

- And: logical and the contents in the two source registers and save it in the destination register.

  example: and $1 $1 $1

Int	RegW	LW	SW	Branch	Jump	ALU	RegD	RegS1	RegS2
0	1	0	0	0	0	001	4 bit	4 bit	4 bit

- OR: logical or the contents of the two source registers and save it in the destination registers.

  example: or $1 $1 $1

Int	RegW	LW	SW	Branch	Jump	ALU	RegD	RegS1	RegS2
0	1	0	0	0	0	010	4 bit	4 bit	4 bit

- SLT: Set the destination register to 1 if the first source register's content is less than the second source register's content and 0 other wise.

  example slt $1 $2 $3

Int	RegW	LW	SW	Branch	Jump	ALU	RegD	RegS1	RegS2
0	1	0	0	0	0	110	4 bit	4 bit	4 bit

2. Basic I-type:
   - LW: load the content in the memory address base + offset into the destination register.

     Example: lw $1 0($1)

Int	RegW	LW	SW	Branch	Jump	ALU	RegD	Base	Offset
1	1	1	0	0	0	000	4 bit	4 bit	4 bit

   - SW: set the content of the memory address base + offset the content of the source register.

     Example: sw $1 0($1)

Int	RegW	LW	SW	Branch	Jump	ALU	RegS	Base	Offset
1	0	0	1	0	0	000	4 bit	4 bit	4 bit

- Addi: add the content of the source register with the immediate value and save it in the destination register.

  Example: addi $1 $1 1

Int	RegW	LW	SW	Branch	Jump	ALU	RegD	RegS	Int Value
1	1	0	0	0	0	000	4 bit	4 bit	4 bit

- Subi: subtract the content of the source register with the immediate value and save it in the destination register.

  Example: Subi $1 $0 0

Int	RegW	LW	SW	Branch	Jump	ALU	RegD	RegS	Int Value
1	1	0	0	0	0	011	4 bit	4 bit	4 bit

- Andi: Logical and the content of the source register with the immediate value and save it in the destination register.

  Example: andi $1 $1 1

Int	RegW	LW	SW	Branch	Jump	ALU	RegD	RegS	Int Value
1	1	0	0	0	0	001	4 bit	4 bit	4 bit

- Ori: Logical Or the content of the source register with the immediate value and save it in the destination register.

  Example: ori $1 $1 1

Int	RegW	LW	SW	Branch	Jump	ALU	RegD	RegS	Int Value
1	1	0	0	0	0	010	4 bit	4 bit	4 bit

- SLL: Shift left logical the content of the source register with the amount of the immediate value and save it in the destination register.

  Example: sll $1 $1 2

Int	RegW	LW	SW	Branch	Jump	ALU	RegD	RegS	Int Value
1	1	0	0	0	0	100	4 bit	4 bit	4 bit

- SRL: Shift right logical the content of the source register with the amount of the immediate value and save it in the destination register.

  Example: srl $1 $1 2

Int	RegW	LW	SW	Branch	Jump	ALU	RegD	RegS	Int Value
1	1	0	0	0	0	101	4 bit	4 bit	4 bit

3. J-type:

- J: jump to an address.

  Example: J 12

Int	RegW	LW	SW	Branch	Jump	ALU	Jump Adr
0	0	0	0	0	1	000	12 bit

- JAL: jump to an address and save PC + 1 into $ra.

  Exam

Int	RegW	LW	SW	Branch	Jump	ALU	Jump Adr
0	1	0	0	0	1	000	12 bit

- JR: Jump to the address contained in a register. Currently Hard coded to jump to $ra.

Example: jr $ra

Int	RegW	LW	SW	Branch	Jump	ALU	RegD	RegS	RegS
1	0	0	0	0	1	000	Xxxx	4 bit	xxxx

4. More instructions:

- BEQ (Added last so it stays here): Branch to PC + 1 + offset if the RegD's content and RegS' content are equal

Int	RegW	LW	SW	Branch	Jump	ALU	RegD	RegS	Offset
0	0	0	0	1	0	000	4 bit	4 bit	4 bit

5. Pseudoinstructions:
    - SGT: derivation from SLT.
    - BGT: Combination of SLT and BEQ.
    - BLT: Combination of SLT and BEQ.

IV. Coding convention:

Since this is my first time making an assembler, there is still a lot of things to be done. Thus, to have a enjoyable and error-free experience with my CPU, this guideline should be followed closely:

- When using labels, be sure to include a *":" after the label* and write your code immediately on the same line. That is, you should not enter a new line after declaring a label.
- Registers 0, register 15 ($sp), register 14 ($ra), register 13 ($at) must not be touched.
- Registers 12 ($a0) and registers 11 ($a1) can be used to pass arguments for a subroutine. Registers 10 ($v0) and 9 ($v1) can be used to return values.
- Registers from 1 to 8 can be used freely by you.
- To use the stack, do addi $sp $sp -1 instead of -4, and push and pop are the same with MIPs assembly.
- Also, you can branch freely to any labels, but if you use offset, please use from -8 to 7.

V. Directions for translating assembly code into hexadecimal machine code:

To do so, users can choose between writing their own assemblers for my CPU, or they can use my assembler.

First, users will need to write Assembly code and save them as .asm files. I recommend using Atom to do this, as it is fast, light and supports color-coded writing.

My assembler requires Python 2.7 or above, which comes with macOS, but Windows users will have to download it from Python website. After doing that, they can choose to move python.exe to my assembler folder, or add python.exe to their PATH environment variable. This can be done easily following instructions on the internet. Using command line is preferable. To translate any assembly files into their corresponding hex files, type this inside the terminal (provided that you have cd inside the assembler folder):

python lamassembler.py inputfile.asm outputfile.hex

If it is translated successfully, this line will appear:

Number of arguments: 3 arguments.

Arguments list: …

If there is any error, it will be displayed in the terminal console. The outputfile can be used to load into Logisim immediately.

VI. More information:

0000	$0
0001	$1
0010	$2
0011	$3
0100	$4
0101	$5
0110	$6
0111	$7
1000	$8
1001	$v1
1010	$v0
1011	$a1
1100	$a0
1101	$at
1110	$ra
1111	$sp

Table 1: Chiron CPU's register list.