Programming Assignment 4-0: Assembler

CS141 Spring 2019

Due March 29, 2019 at 5pm

Introduction

Useful Reading: Harris & Harris Chapter 6.1-6.5, Appendix B.

As a CPU is running, it must fetch instructions from RAM to determine what computations to perform. In the MIPS instruction set architecture (ISA) instructions are stored as 32-bit values. The representation of MIPS instructions as these 32-bit values is called machine code, and the symbolic representation of the instructions is called assembly code.

An assembler is a program which translates assembly code into its equivalent machine code. In almost all cases this translation is a one-to-one mapping, and your job in this assignment is to implement a MIPS assembler. More information about assemblers can be found in Harris & Harris Chapter 6.3.

Specification

The input to your assembler will be a .asm (assembly) file. Each line is formatted as:

```
[label:] insn arg0 [arg1] [arg2]  [# comment]
```

Where fields in brackets are optional – not every line has a label or comment, and not every instruction has three arguments. Your assembler should be able to figure out which arguments are provided for any instruction. Your assembler must support register arguments either named following MIPS conventions (e.g. `$s0`, `$t1`, `$sp`) or directly named by number (`$0`-`$31`). See page 294 of the book for more details about the MIPS register set.

The instructions you must support are:

- `add`
- `addi`
- `sub`
- `and`
- `andi`
- `or`
- `ori`
- `xor`
- `xori`
- `nor`
- `sll`
- `sra`
- `srl`
- `slt`
- `slti`

- `beq`
- `bne`
- `j`
- `jal`
- `jr`
- `lw`
- `sw`
- `nop`

Each instruction is worth 2 points, for a total of 46 points.

Opcodes for each instruction are provided in Appendix B of your textbook. The exception is the `nop` (no operation) instruction, which should be encoded as 32 zeros.

We have also uploaded a specification document for the entire MIPS ISA (of course you only have to support a small subset of the full ISA). See there or the book for details on every instruction you should support.

The output of your assembler should be a `.machine` file, which contains machine code in hexadecimal, one instruction per line. A hex conversion chart can be found on page 12 of H&H. Do not print `0x` at the start of each line.

Be careful when you are assembling branch and jump instructions, particularly when they contain labels. For simplicity, assume that a label will never exist on its own line. It only refers to the instruction on the same line. We assume that the first instruction of the program is located at address `0x00400000` (this is useful for determining jump target addresses).

We have provided a small test assembly program below and the correct machine code that you can use to verify your assembler. We will use different assembly files to test your program. Your assembler must be written in Python 2, and a template of the assembler (in Python) has been provided on the course website.

For an example of what we are expecting from your assembler: if the input code is:

```
loop: add  $s0,   $s1,    $s2        # $s0 = $s1 + $s2
      lw   $t4,   0($s0)
      lw   $t3,   0($s1)
      bne  $t3,   $t4,    loop
      addi $s0,   $s0,    1          # $0++
```

The output machine code should be:

```
02328020
8e0c0000
8e2b0000
156cfffc
22100001
```

You may want to consider writing your own additional test programs to verify that your assembler works properly.

Deliverable

You should submit on Canvas a Python file containing the code for your assembler.

Please let us know if you have any questions regarding the assignment or Python installation.