CS 241 – Week 3 Tutorial

Writing an Assembler

Spring 2018

1 Symbol tables in MIPS

Recall that the *address of a label* is given by 4 times the number of non-blank lines preceding it in the program, where a blank line is a line not containing an instruction or `.word` directive. The *symbol table* for a MIPS assembly program is a list of labels in that program and their associated addresses.

Construct the symbol table for the following MIPS assembly program.

```
begin:
label: beq $0, $0, after
jr $4

after:
sw $31, 16($0)
lis $4
abc0: abc1: .word after

loadStore:
lw $20, 4($0)
sw $20, 28($0)

end:
```

2 Bitwise operations in MIPS

Bitwise operations enable precise manipulation of bit patterns. The bitwise operations that will be useful in this course are the following

- And (`&`), which produces the word where bits are 1 if they were 1 in both inputs:

```
  1011
& 1101
------
  1001
```

- Inclusive Or (`|`), which produces the word where bits are 1 if they were 1 in either input:

```
  1010
| 1001
------
  1011
```

- Left shift (`<<`), which removes the number of bits given on the right hand side from the beginning of the number on the left hand side, and adds that many zeros on the end of that number.

```
   0110
<<    2
-------
   1000
```

- Right shift (`>>`), which works the same way as left shift except that bits are removed from the end and added to the beginning.

```
   0110
>>    2
-------
   0001
```

  Note that the `>>` operator in C may work differently on signed integers, so you should be careful to always use unsigned integers when right shifting.

Note that in Racket these operations are `bitwise-and`, `bitwise-ior`, and `arithmetic-shift` with positive and negative shift values respectively.

1. Assume unsigned 4 bit integers. What should each of the following computations produce?
   (a) `3 & 5`
   (b) `3 | 5`
   (c) `3 << 2`
   (d) `3 >> 2`
   (e) `13 << 2`
   (f) `13 >> 2`
2. Give an example of where each bitwise operation might be useful when writing an assembler.

3 Binary output

1. Write pseudocode for a function called `output_word` that takes a 32-bit integer as input and outputs each of its four bytes to standard output. You can assume a function called `output_byte` is available.

   Assume the `output_byte` function takes an integer as input. If the integer is small enough to fit in a byte it outputs that byte to standard output; otherwise it produces an error. The `output_byte` function corresponds to:

   - `putchar` in C/C++.
   - `write-byte` in Racket.

2. How would you use this above function (in conjunction with the symbol table) to assemble the `.word foo` directive, where foo is a label in an assembly program?

4 Assembling instructions automatically

Recall that instructions follow two basic formats, register and immediate, as detailed in the MIPS reference sheet:

```
Register:  0000 00ss ssst tttt dddd d000 00ff ffff
Immediate: oooo ooss ssst tttt iiii iiii iiii iiii
```

In both cases, `sssss`, `ttttt`, `ddddd` are registers encoded as 5-bit unsigned numbers, `iiii iiii iiii iiii` is a 16-bit two's complement number, and `oooooo` or `ffffff` are 6-bit opcodes (unsigned numbers) specified for each instruction in their row on the sheet.

Using the bitwise operations from last week's tutorial and the ideas from assignment 1, give pseudocode to assemble the following instructions:

1. `slt $d, $s, $t`.
2. `beq $s, $t, $i`, where *i* is an immediate value (INT or HEXINT token).

How could we design our code to maximize code reuse between various instructions?