CS 241 – Week 4 Tutorial Solutions

Writing an Assembler Pt. I

Fall 2018

1 Symbol tables in MIPS

The value of a label is defined to be the number of non-null lines (lines containing an instruction) that precede the label multiplied by 4. To begin, we mark all the non-null lines in the program.

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

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

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

end:
```

Now we make a list of labels in the program.

```
begin
label
after
abc0
abc1
loadStore
end
```

Finally, we count the number of non-null lines before each label to get the values.

```
begin 0
label 0
after 8
abc0 16
abc1 16
loadStore 20
end 28
```

Remember to print it to standard error.

We solved this problem by examining the whole program at once. When writing code that constructs the symbol table, you will have to approach this problem a little differently. The code will probably scan through the program line by line and keep a counter of the number of non-null lines. When the program sees the label, it can compute the label's value using the counter and then add the label to the symbol table.

2 Error-checking MIPS programs

```
1  label: label: .word label
2  .word ; 0
3  .word aaaaa
4  .word 1 2 3
5  .word 2147483648 abcde:
6  .word ,
```

Line 1: Duplicate definition of "label".

Line 2: No operand for the `.word`. A `.word` must have exactly one operand.

Line 3: The label "aaaaa" is used here, but it was never defined.

Line 4: Too many operands for the `.word`.

Line 5: Label definitions must appear *before* instructions on a line.

Line 6: The operand must be an identifier (label name), a decimal integer or a hex integer. A comma is none of those.

Note that there is no "out of range" error on line 5. The number here is outside the range of 32-bit two's complement values, but it is within the unsigned range, and the .word operand can be an unsigned *or* two's complement integer.

3 Bitwise operations in MIPS

1.
```
  3 = 0011
& 5 = 0101
----------
      0001 = 1 (signed and unsigned)
```

2.
```
  3 = 0011
| 5 = 0101
----------
      0111 = 7 (signed and unsigned)
```

3.
```
3 = 0011
    << 2
    ----
    1100 = 12 (unsigned)
         = -4 (signed)
```

4.
```
3 = 0011
    >> 2
    ----
    0000 = 0 (signed and unsigned)
```

5. 13 = 1101
   << 2

   0100 = 4 (signed and unsigned)

6. 13 = 1101
   >> 2

   0011 = 3 (signed and unsigned)

Left shifting allows you to move the pieces of the assembly representation of an instruction to the correct position in the bit pattern that describes the format of the machine code representation of the instruction.

Right shifting allows you to move the bits so that they can be printed by a function which outputs a single byte. Be careful when right shifting in C++; some compilers will use arithmetic shift (preserves sign bit) and others will use logical shift (does not preserve sign bit).

Bitwise Or is the glue that sticks together the various pieces that make up a machine code instruction. Note that when a bit is 'on' in either input, that bit will always be on in the output. Thus if you fix one of the inputs, it is possible to turn on bits in the other.

Bitwise And produces a value in which the only bits that are 'on' are the ones that were on in both inputs. Thus if you fix one of the inputs, it is possible to turn off bits in the other.

4 Binary output

The code will look like this:

```
int output_word(unsigned int word) {
    output_byte((word >> 24) & 0xff);
    output_byte((word >> 16) & 0xff);
    output_byte((word >> 8) & 0xff);
    output_byte(word & 0xff);
    return;
}
```

Explanation: Suppose the input word is `0xabcd1234`. In binary, this is:

```
1010 1011 1100 1101 0001 0010 0011 0100
a    b    c    d    1    2    3    4
```

We want to output all four bytes of this word from left to right: first `0xab`, then `0xcd`, then `0x12`, then `0x34`. But `output_byte` can only output one byte at a time.

We start by figuring out how to output `0xab`. We need to manipulate the bits of "word" to get the following binary value:

```
0000 0000 0000 0000 0000 0000 1010 1011
0    0    0    0    0    0    a    b
```

This value fits into 8 bits, so we can print it with `output_byte`.

To move the upper 8 bits into the right position, we can use an arithmetic right shift by 24 bits, which gives us:

```
1111 1111 1111 1111 1111 1111 1010 1011
f    f    f    f    f    f    a    b
```

Why all the 1s? Since `0xabcd1234` begins with a "1" bit, it is a negative two's complement number. The arithmetic shift will preserve the sign, so the number gets padded with 1s instead of 0s. If we had used an `unsigned int` rather than an `int`, the compiler would likely typically use a logical shift and the number would get padded with 0s.

Now we need to get rid of all the 1s. We can do this by doing a bitwise and with the following number:

```
0000 0000 0000 0000 0000 0000 1111 1111
0    0    0    0    0    0    f    f

    1111 1111 1111 1111 1111 1111 1010 1011
AND 0000 0000 0000 0000 0000 0000 1111 1111
-------------------------------------------
    0000 0000 0000 0000 0000 0000 1010 1011
```

Wherever there is a 0 bit in the bottom number, the result will have a 0. Whenever there is a 1 bit, the result will have a copy of the bit in the top number. So this bitwise and operation will zero out everything but the last 8 bits, which it leaves alone.

After this we can print the byte. So to print the first byte, we do:

```
output_byte((word >> 24) & 0xff);
```

To print the second and third bytes, we basically do the same thing, but we shift by different amounts. To print the last byte, we don't need to shift at all; we just need to zero out everything but the last 8 bits. Note that `output_byte` only considers the last 8 bits so zeroing out the other bits is not needed.

To assemble the `.word foo`, we would look up the label "foo" in the symbol table and pass the integer we get (the value of the label) into `output_word`.

```
output_word(table_lookup("foo"));
```