# CS 241 – Week 4 Tutorial

Writing an Assembler Pt. II

Spring 2015

## Summary

- Outputting bytes
- Outputting instructions
- MERL
- Relocating

## Problems

1. The code will look like this:

```
int output_word(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 a right bitwise 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 bitwise shift will preserve the sign, so the number gets padded with 1s instead of 0s. If we had used an `unsined int` rather than an `int` to store the number being shifted the number would always 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 after`, we would look up the label "after" in the symbol table and pass the integer we get (which is 8) into `output_word`.

```
output_word(table_lookup("after"));
// outputs 0x00000008, value of the label "after"
```

2. ```
   int assemble_add(int s, int t, int d) {
       return (s << 21) | (t << 16) | (d << 11) | 32;
   }
   ```

   Explanation: If we look at the MIPS reference sheet, add is encoded as follows:

   ```
   000000 sssss ttttt ddddd 00000 100000
   ```

   In binary, our parameters look like:

   ```
   000000 00000 00000 00000 00000 0sssss
   000000 00000 00000 00000 00000 0ttttt
   000000 00000 00000 00000 00000 0ddddd
   ```

   Notice that to get the bits of s in the right position, we have to shift it to the left by 21 bits. Similarly, we have to shift t by 16 bits. Finally, we have to shift d by 11 bits.

   ```
   000000 sssss 00000 00000 00000 000000 s << 21
   000000 00000 ttttt 00000 00000 000000 t << 16
   000000 00000 00000 ddddd 00000 000000 d << 11
   ```

   Now we combine them with bitwise or. When we do a bitwise or, any time there is a 1 in a column, the result will contain a 1 in that column. This combines the three words into one.

   ```
   000000 sssss ttttt ddddd 00000 000000
   ```
   ```

Now to complete the assembly of the instruction, we just need to add in the function bits, which tell the MIPS machine that this is an add. We can do this by bitwise-or-ing what we have with 32, the integer that corresponds to the function bits for add.

```
    000000 sssss ttttt ddddd 00000 000000 (s<<21)|(t<<16)|(d<<11)
OR  000000 00000 00000 00000 00000 100000 32
----------------------------------------
    000000 sssss ttttt ddddd 00000 100000
```

Then we simply return the assembled integer.

Yes, we can easily generalize the `assemble_add` function to all R-type instructions. We can do this by adding a fourth parameter represnting the f-bits of the R-type instructions and passing the correct value for each R-type instruction in the parameter. This fourth parameter will take the place of the 32 in `assemble_add`. However, we need to esnure that this number does not exceed the value 32. e.g.

```
assemble_rtype(3,4,5,32); // Generating the binary for an add
```

3. First we convert the assembly file into MERL file (shown on the left in assembly language format for readability). Note that only a `.word` instruction whose operand is a label must be relocated.

```
; MERL header
  beq $0, $0, 2              0x10000002
  .word endFile              0x0000006c
  .word endCode              0x00000064

  lis $6                     0x00003014
  .word 0x18                 0x00000018

  sw $31, -4($30)            0xafdffffc
  lis $31                    0x0000f814
  .word 4                    0x00000004
  sub $30, $30, $31          0x03dff022
  lis $3                     0x00001814
rel:
  .word proc                 0x0000005c
  lis $11                    0x00005814
  .word 1                    0x00000001
loop:
  beq $2, $0, end            0x10400003
  jalr $3                    0x00600009
  sub $2, $2, $11            0x004b1022
  beq $0, $0, loop           0x1000fffc

end:
  add $3, $1, $0             0x00201820
  lis $31                    0x0000f814
  .word 4                    0x00000004
  add $30, $30, $31          0x03dff020
  lw $31, -4($30)            0x8fdffffc
  jr $31                     0x03e00008

proc:
  add $1, $1, $6             0x00260820
```

```
  jr $31                        0x03e00008

endCode:
  .word 0x01                    0x00000001
  .word rel                     0x00000028

endFile:
```

Note that this program contains only one relocation entry. We follow the location and relocate the word pointed to by the entry by adding 0x8c to get 0xe8. Be careful when adding numbers in hex. We can add numbers in hex in the same way that we would add in any other base:

```
    1
  0x5c
+ 0x8c
------
  0xe8
```

Alternatively, you can convert to decimal, add, then convert back, but this process is time consuming. In the event that such a question appears on an exam, you would be much better off if you could add directly in hex.