1 Symbol tables in MIPS

Let’s begin by marking all non-blank 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:

Next, make a list of labels in the program.

    begin
    label
    after
    abc0
    abc1
    loadStore
    end

Finally, we count the number of non-blank lines before each label to get the address associated with each label.

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

Remember for A3P3 that these numbers should be printed to stderr, not stdout.
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 ,

The errors are as follows:

1. Duplicate definition of “label”.
2. No operand for .word: .word expects exactly one operand.
3. The label “aaaaa” is undefined.
4. Too many operands for .word.
5. Label definitions must precede instructions on a line.
6. The operand to .word must be a decimal integer, hexadecimal integer, or identifier.

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

(b) \(3 = 0011\)
   | 5 = 0101
   ---------
   0111 = 7 (signed and unsigned)

(c) \(3 = 0011\)
   << 2
   ----
   1100 = 12 (unsigned)
   = -4 (signed)

(d) \(3 = 0011\)
   >> 2
   ----
   0000 = 0 (signed and unsigned)

(e) \(13 = 1101\)
   << 2
   ----
   0100 = 4 (signed and unsigned)
13 = 1101
   >> 2
   ----
0011 = 3 (signed and unsigned)

2. • Bitwise and can be used to “turn off” parts of a word. For example, in the next question we want to remove all but the last byte of a word, which can be done by computing its bitwise and with 0xff.
• Bitwise or can be used to combine different pieces of an instruction. For example, if we have a partially-assembled instruction but need to insert the right value for $s$, we can do this using bitwise or.
• Left shift can be used to position pieces of information when assembling a word, usually immediately before applying bitwise or.
• Right shifting can be used to position pieces of information when taking apart a word, usually immediately before applying bitwise and.

4 Binary output

1. The code will look like this:

```c
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 a logical right shift by 24 bits, which gives us:

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

Just in case the leading bytes are not 0 (such as if we used an arithmetic right shift instead of a logical right shift) we can remove them with a bitwise and.

The byte is now ready to be printed with output_byte.
For the next byte, we need to shift by 16 to position things correctly:

```
 0000 0000 0000 0000 1010 1011 1100 1101
  0  0  0  0  a  b  c  d
```

Now notice that we still have the unwanted leading ab. We can remove this with a bitwise and.

```
 0000 0000 0000 0000 1010 1011 1100 1101
  0  0  0  0  a  b  c  d
AND 0000 0000 0000 0000 0000 0000 1111 1111
  0  0  0  0  0  0  f  f
-------------------------------------------
  0000 0000 0000 0000 0000 0000 1100 1101
  0  0  0  0  0  0  c  d
```

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 zeroes out all but the last byte, which is left unchanged. The byte is now ready to be printed with `output_byte`.

The remaining two bytes are similar.

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

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