CS 241 – Week 5 Tutorial Solutions

Writing an Assembler, Part 2 & Regular Language Part 1

Spring 2017

1 Assembling instructions automatically

1. `slt $d, $s, $t`.

   Solution: `$d`, `$s`, and `$t` all fit in 32-bit signed integers since they are 5-bit unsigned ints, so we can keep them unchanged. So we can do the following:

   ```
   int s, t, d //assume these are initialized appropriately
   //Begin with the opcode for slt, which is 101010
   int slt = 0x2A //0000 0000 0000 0000 0000 0000 0010 1010
   slt = slt | (s << 21) //0000 00ss sss0 0000 0000 0000 0010 1010
   slt = slt | (t << 16) //0000 00ss ssst tttt 0000 0000 0010 1010
   slt = slt | (d << 11) //0000 00ss ssst tttt dddd d000 0010 1010
   ```

   We could also make sure that register integers only contain 5-bit integers by first taking `s & 0x1F` and similar for `t` and `d`, but this isn't necessary since our scanner already checks this.

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

   Solution: Proceeding similar to the above:

   ```
   int s, t, i //assume these are initialized appropriately
   //Begin with the opcode for beq, which is 000100
   //However, we need to make sure it's on the left 6
   //bits rather than the right 6 bits since beq is immediate-format!
   beq = 0x10000000 //0001 0000 0000 0000 0000 0000 0000 0000
   //Now add s, t in the same way as before
   beq = beq | (s << 21) //0001 00ss sss0 0000 0000 0000 0000 0000
   beq = beq | (t << 16) //0001 00ss ssst tttt 0000 0000 0000 0000
   //Finally, add i. We need to make sure to clear its most significant bits first!
   beq = beq | (i & 0xFFFF) //0001 00ss ssst tttt iiii iiii iiii iiii
   ```

   Note that we need to make sure to zero out the high bits of `i` before adding it to the instruction! In what cases can we get undesirable behaviour if we forget?

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

Solution: have generic functions for any register or immediate-format instruction which simply accept `s`, `t`, `d`, `f` or `s`, `t`, `i`, `o` respectively as arguments.

2 Regular Expression Solutions

Regular languages:

1. $\{0,1\}\{0\}\{0,1\}\{0,1\}\{1\}\{0,1\}^*$
2. $\{0,1\}^*\{1\}\{1\}\{0\}\{1\}\{0\}\{1\}\{0,1\}^*$

Regular expressions:

1. Naive solution: $(xx|xy|yx|yy)$. Better solution: $(x|y)(x|y)$
2. $(G|C|A|T)^*GACAT(G|C|A|T)^*$
3. (a) $(0|1)0(0|1)(0|1)1(0|1)^*$

   (b) $(0|1)^*110101(0|1)^*$

   Doesn't that feel much better?

4. The regular expression for a term is: $(a|b)$

   The regular expression for an operator is: $(+|-|\cdot|/)$

   The structure of an arithmetic expression is term operator term operator term ... operator term. An expression must start with a term: $(a|b)$

   Then we have "operator term" repeated 0 or more times after the initial term:
   $(a|b)((+|-|\cdot|/)(a|b))^*$

5. The first thing we realize is that we must have at least one 1. This will be the basis for our regular expression: 1

   What can come before the first 1? As many even counts of 0's as we like mixed with any number of 2's: $2^*(02^*02^*)^*$. Note that we need the leading 2^* since otherwise we could never have leading 2's.

   What can proceed the first 1? The same as can precede it plus any number of 1's as well: $(1|2)^*(0(1|2)^*0(1|2)^*)^*$

   So we have: $2^*(02^*02^*)^*1(1|2)^*(0(1|2)^*0(1|2)^*)^*$. Does this solve our problem?

   No. Because we could have an odd number of 0's on the left and right of the first 1 and still have an even number of 0's. How do we fix this? By either have 1 or having 1 with a 0 on the left and a zero on the right. Note that we will also have to allow for additional 2's on the left and additional 1's and 2's on the right. The new core becomes $(1|(02^*1(1|2)^*0)$

   Combining this with our original left and right pieces we get:
   $2^*\,(02^*02^*)^*\,\left(1|\,\left(02^*1\,(1|2)^*\,0\right)\right)\,(1|2)^*\,\left(0\,(1|2)^*\,0\,(1|2)^*\right)^*$