CS 241, Lecture 5 - Assembler and Formal Languages

1 The Assembler

- We wish to input assembly code and output machine code.
- This translation process involves two phases: Analysis (understand what is meant by the input source) and Synthesis (output the equivalent target code in the new format).
- An assembly file is just a fuckton of characters.
- So in order to parse it correctly, we need to write a tokenizer that understands our tokens.
- Examples could be .word, MIPS instructions, labels, numbers, etc.
- This is already done for us - we don't have to write a tokenizer (for now at least, thank god)
- Thus, all we have to really do is group tokens into instructions in our analysis, then in synthesis, output code.
- Assembler problem: Assume we have the following code:

```
1 beq $0, $1, myLabel
2 myLabel:
3     add $1, $1, $1
```

- The issue is that when we see myLabel for the first time, we don't know what the correct address is.
- What we'll do is that we'll go through the code twice - once to store all addresses of labelled instructions, second time when a label is referred to, we'll then look up the associated address on our symbol table/map and actually translate instructions to machine code.
- For us, we'll output our symbol table to stderr.
- This also means if we have a branch instruction we have to manually calculate the hard value of lines we jump by on our second iteration if it had a label as its third parameter.

2 Binary in C++

- Recall we have three types of instruction formats - R, I, J.
- R-format follows the following format:
  - opcode - 6 bits

  - rs - 5 bits
  - rt - 5 bits
  - rd - 5 bits
  - shift(shamt) - 5 bits
  - funct - 6 bits
- I-format follows the following format:
  - opcode - 6 bits
  - rs - 5 bits
  - rt - 5 bits
  - immediate - 16 bits, in twos complement usually
- J-format follows the following format:
  - opcode - 6 bits
  - psuedo-address - 26 bits
- bne and beq are in I-format (has an immediate value), for example. Let's use this for the following example.
- We can store an instruction as a single 32-bit integer, where we bitshift it the correct number of bits left, and bitwise or to join all the required values.
- For example, bne $2, $0, -3, would correspond to (5 << 26) | (2 << 21) | (0 << 16) | offset
- An issue with the offset for an I-format is that the offset could be a 32-bit integer, but our offset is limited to a 16-bit integer.
- This could cause a lot of problems with negative numbers, as we aren't shifting it or anything - just using a bitwise or will cause the offset to overwrite.
- In order to solve this we'll use a mask to only get the last 16 bits. We can do this by using a bitwise and with 0x0000ffff to forcefully zero out 4 of the 8 bytes.
- Another problem to overcome is that C++ will see our final result as just this big 9-byte integer. This is NOT what we want - we want the binary!
- We want to do what happens with MIPS - when we sw to print, we print the LEAST SIGNIFICANT BYTE.
- Thus! We will have to do MORE bit shifting, where we shift the bytes we want to print by the correct number of spots, in order to make them the LSBytes, then print them as an unsigned char.

3 Formal Languages

- An **alphabet** is a non-empty finite set of symbols often denoted by $\sum$.
- A **string/word** w is a finite sequence of symbols denoted from $\sum$. The set of all strings over an alphabet $\sum$ is denoted by $\sum *$.
- A **language** is a set of strings.
- The **length of a string** w is denoted by $|w|$.
- For example, we can define $\sum = \{a, b, c, ..., z\}$ to be our **normal English alphabet**.
- We could define ϵ to be the **empty string**, it is in $\sum *$ for any $\sum$ and $|\epsilon| = 0$.
- For $\sum = \{0, 1\}$, **strings** include $w = 011101$, or $x = 1111$. $|w| = 6$ and $|x| = 4$.
- A **language** could be $L = \{\}$ (the empty language), $L = \{ab^n a : n \in \mathbb{N}\}$, which would be the set of strings over the alphabet $\sum = \{a, b\}$ consisting of an a followed by 0 or more b characters followed by an a.
- Another language example could be that for alphabet {., -}, L = {words in Morse Code}.
- Our objective is that given a language, we want to determine if a string belongs to it (membership).
- The difficulty depends on the complexity of the language, from very easy to impossible. In order of relative difficulty:
  - finite
  - regular
  - context-free
  - context-sensitive
  - recursive
  - impossible languages
- For a finite language, membership is very easy as we just need to check for equality with every single word in the language. How you do this search would affect efficiency but it is definitely doable.