Warm Up Problem

Write an assembly language MIPS program that takes a positive two's compliment integer in register $1 and stores the sum of the digits in register $2.

Other questions to ponder:

- What were some of the concerns with creating procedures from last time?
- How did we handle returning from a procedure?
- What do we have to do to make your code above a procedure?

CS 241 Lecture 5

Assembler and Formal Languages

With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot

The Assembler

Recall part of our longterm goal is to convert assembly code (our MIPS language) into machine code (zeroes and ones).

- Input: Assembly code
- Output: Machine code

Any such translation process involves two phases: **Analysis** and **Synthesis**.

- Analysis: Understand what is meant by the input source
- Synthesis: Output the equivalent target code in the new format

Assembly File

- Think of it as a string of characters.
- We want to first break it down into meaningful **tokens** such as labels, numbers, `.word`, MIPS instructions and so on.
- This is done for you in `asm.rkt` and `asm.cc`.

Your job (should you choose to accept it):

- Analysis: Group tokens into instructions if possible
- Synthesis: Output equivalent machine code.

If the tokens are not valid instructions, output `ERROR` to `stderr`.

Assignment Advice

- There are many more incorrect tokens than correct ones.
- Focus on finding correct ones! (More on this in upcoming weeks)
- Later we will discuss parsing, a formal way of grouping tokens.

The Biggest Assembler Problem

How do we assemble this code:

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

The Biggest Assembler Problem

How do we assemble this code:

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

The problem is when we see `myLabel` for the first time, we don't know what the correct corresponding address is!

What is the best fix to this?

Standard Solution:

Make two passes:

- Pass 1: Group tokens into instructions and record address of all labelled instructions. This can be done in a symbol table, that is, a list of label address pairs.
- **Note:** multiple labels are possible for the same line! For example, `f: g: add $1, $1, $1`.
- Pass 2: translate each instructions into machine code. If it refers to a label, look up the associated address in the aforementioned symbol table and perform the computation to compute the correct corresponding needed value.

Your Assembler

When writing your assembler, you will do two things:

- Output the machine code coming from the assembled MIPS code to `stdout`.
- Output the symbol table to `stderr`.

Symbol Table Example

Note: A label at the end of code is allowed (it would be the address of the first line after your program).

```
0x00        main:  lis $2
0x04               .word beyond
0x08               lis $1
0x0c               .word 2
                   ;Ignore
0x10               add $3, $0, $0
0x14        top:
                   add $3, $3, $2
0x18               sub $2, $2, $1
0x1c               bne $2, $0, top
0x20               jr $31
0x24        beyond:
```

label	addr
main	0x00
top	0x14
beyond	0x24

Summary of Passes for Previous Code:

Pass 1:

- Group tokens into instructions
- Build Symbol Table one label at a time
- At the end of code, table is complete.

Pass 2:

- Translate each instructions to machine code
- For each label in an instruction, look up in symbol table and process accordingly (See below)

Creating Binary In C++

How do we then write the binary output:

0001 0100 0100 0000 1111 1111 1111 1101

for `bne $2, $0, -3`?

Creating Binary In C++

How do we then write the binary output:

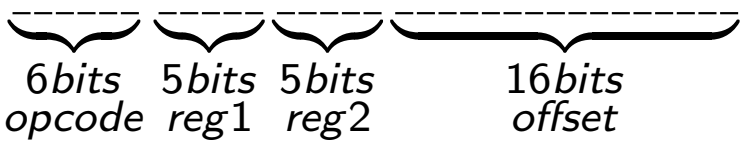
0001 0100 0100 0000 1111 1111 1111 1101

for `bne $2, $0, -3`?

We've done a lot of the heavy lifting for this problem but let's generalize the above and do it completely in C++.

Bit-wise Operations

Our instruction `bne` can be broken down as follows:



For us,

- `bne` has op code `000101 = 5`
- Register 1 is `00010 = 2`
- Register 2 is `00000 = 0`
- Offset is `1111111111111101 = -3`

Bit Shifting!

In this way, we can continue to shift the other information into the correct position and use a bitwise or to join them:

```
int instr = (5 << 26) | (2 << 21) | (0 << 16) | offset
```

We need to be careful with the offset. What can go wrong with the offset?

Bit Shifting!

In this way, we can continue to shift the other information into the correct position and use a bitwise or to join them:

```
int instr = (5 << 26) | (2 << 21) | (0 << 16) | offset
```

We need to be careful with the offset. What can go wrong with the offset?

Recall in C++, integers are 4 bytes - we only want the last two bytes. First we need to apply a "mask" to only get the last 16 bits:

$$\textit{offset} = \texttt{-3 \& 0xffff}$$

and then use this in the formula above. Thus `instr` is 339804157.

Explanation of Offset Masking

Without Masking (Notice the leading ones ruin our work!):

$$\begin{array}{rl} & 0001\,0100\,0100\,0000\,0000\,0000\,0000\,0000 \\ | & 1111\,1111\,1111\,1111\,1111\,1111\,1111\,1101 \\ \hline & 1111\,1111\,1111\,1111\,1111\,1111\,1111\,1101 \end{array}$$

With Masking:

$$\begin{array}{rl} & 0001\,0100\,0100\,0000\,0000\,0000\,0000\,0000 \\ | & 0000\,0000\,0000\,0000\,1111\,1111\,1111\,1101 \\ \hline & 0001\,0100\,0100\,0000\,1111\,1111\,1111\,1101 \end{array}$$

Are We Done?

Having all this all we need to do is:

```
cout << instr
```

... right?

Are We Done?

Having all this all we need to do is:

```
cout << instr
```

... right? right...

Are We Done?

Having all this all we need to do is:

```
cout << instr
```

... right? right...

No!!! This would output 9 bytes corresponding to the ASCII code for each digit! We want to output the four bytes that correspond to this number!

Printing Bytes in C++

So let's print out the bytes of the instruction

```
int instr = (5 << 26) | (2 << 21)
        | (0 << 16) | (-3 & 0xffff);
unsigned char c = instr >> 24;
cout << c;
c = instr >> 16;
cout << c;
c = instr >> 8;
cout << c;
c = instr;
cout << c;
```

Note: You can also mask here to get the 'last byte' by doing `& 0xff` if you're worried about which byte will get copied over.

End of MIPS

- We will now transition to something that looks completely different and is much more theoretical.
- Our goal remember is to translate our high-level language (Watered down C/C++) into assembly language.
- Assembly language has a simple and universal way to be translated to machine code for a specific architecture
- Higher level languages however could have multiple different translations to machine language. These usually have a more complex structure than assembly language code.
- What we need is to formalize our notion of a language and then figure out from this formalized notion how we are to parse strings of text.

Formal Languages

We begin with a few definitions

Definition

An **alphabet** is a non-empty finite set of symbols often denoted by Σ.

Definition

An **string** (or **word**) w is a finite sequence of symbols chosen from Σ. The set of all strings over an alphabet Σ is denoted by Σ^*.

Definition

A **language** is a set of strings.

Definition

The **length of a string** w is denoted by $|w|$.

Examples

Alphabets:

- $\Sigma = \{a, b, c, ..., z\}$ our alphabet.
- $\Sigma = \{0, 1\}$ alphabet of binary digits.
- $\Sigma = \{0, 1, 2, ..., 9\}$ alphabet of base 10 digits.

Examples

Alphabets:

- $\Sigma = \{a, b, c, ..., z\}$ our alphabet.
- $\Sigma = \{0, 1\}$ alphabet of binary digits.
- $\Sigma = \{0, 1, 2, ..., 9\}$ alphabet of base 10 digits.

Strings:

- ϵ is the empty string; it is in Σ^* for any Σ and $|\epsilon| = 0$.
- For $\Sigma = \{0, 1\}$, strings include $w = 011101$ or $x = 1111$. Note $|w| = 6$ and $|x| = 4$.

Languages:

- $L = \emptyset$ or $\{\}$, the empty language
- $L = \{\epsilon\}$ the language consisting of the empty string.
- $L = \{ab^n a : n \in \mathbb{N}\}$ the set of strings over the alphabet $\Sigma = \{a, b\}$ consisting of an a followed by 0 or more b characters followed by an a.
- $\{., -\}$, $L = \{$ words in Morse Code$\}$ (Thanks Brad)

Objective

Given a language, determine if a string belongs to it.

How hard is this question?

Objective

Given a language, determine if a string belongs to it.

How hard is this question? This depends on the language!

- $L = \{ab^n a : n \in \mathbb{N}\}$ Very Easy
- $L = \{$ valid MIPS assembly programs $\}$ Easy
- $L = \{$ valid Java/C/C++ Programs $\}$ Harder
- $L = \{$ set of programs that return "Yes" as a decision problem$\}$ Impossible!

Which languages *a priori* are easier than others?

Membership in Languages

In order of relative difficulty:

- finite
- regular
- context-free
- context-sensitive
- recursive
- impossible languages

Finite Languages

Why are these easy to determine membership?

Finite Languages

Why are these easy to determine membership?

- To determine membership in a language, just check for equality with all words in the language!
- Even if the language is of size $10^{10^{10}}$ this is still theoretically possible.
- However, is there a more efficient way?

A Leading Example:

Suppose we have the language

$$L = \{\text{bat}, \text{bag}, \text{bit}\}$$

Write a program that determines whether or not $w \in L$ given that each character of w is scanned exactly once without the ability to store previously seen characters.