Warm-Up Problem

Questions to ponder:

- Why do we save and restore registers?
- Who saves the registers of whom?

CS 241 Lecture 5

Assembler and Formal Languages
With thanks to Brad Lushman, Troy Vasiga, Kevin Lanctot, and Carmen Bruni

Unresolved Questions

Most of our original questions have been resolved except for one: **How do we pass parameters?**

- Typically, we'll just use registers. If we have too many, we could push parameters to the stack and then pop them from the stack. Documentation is vitally important here!
- If we can do this correctly, then everything, including recursion, should just work properly.

Sum Evens 1 to N Slide 1 of 2

```
; sumEvens1ToN adds all even numbers from 1 to N
; Registers:
; $1 Scratch Register (Should Save!)
; $2 Input Register   (Should Save!)
; $3 Output Register  (Do NOT Save!)
sumEvens1ToN:
    sw $1, -4($30) ; Save $1 and $2
    sw $2, -8($30)
    lis $1
    .word 8
    sub $30, $30, $1 ; Decrement stack pointer
    add $3, $0, $0 ; Don't forget to initialize $3!
    lis $1
    .word 2
    ; ....continued on next slide!
```

Sum Evens 1 to N Slide 2 of 2

```
div $2, $1 ; Is N even?
mfhi $1
sub $2, $2, $1 ; Sub 1 if not
lis $1
.word 2 ; Restore 2
top:
add $3, $3, $2
sub $2, $2, $1
bne $2, $0, top
lis $1
.word 8
add $30, $30, $1
lw $2, -8($30)
lw $1, -4($30) ; Reload $1 and $2
jr $31 ; Back to caller
; End sumEvens1ToN
```

Another outstanding problem: How to we print to the screen or read input?

Input and Output

Another outstanding problem: How to we print to the screen or read input?

We do this one byte at a time!

- Output: Use sw to store words in location `0xffff000c`. Least significant byte will be printed.
- Input: Use lw to load words in location `0xffff0004`. Least significant byte will be the next character from stdin.

Example

Printing CS241 to the screen followed by a newline character:

```
lis $1
.word 0xffff000c
lis $2
.word 67 ; C
sw $2, 0($1)
lis $2
.word 83 ; S
sw $2, 0($1)
lis $2
.word 50 ; 2
sw $2, 0($1)
lis $2
```

```
; Continued from left
.word 52 ; 4
sw $2, 0($1)
lis $2
.word 49 ; 1
sw $2, 0($1)
lis $2
.word 10 ; \n
sw $2, 0($1)
jr $31
```

The Assembler

Recall: part of our long-term goal is to convert assembly code (our MIPS language) into machine code (bits).

- 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 (because that's what it is).
- 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:

- 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 problem is that `myLabel` is used before it's defined: we don't know the address when it's used!
- What is the best fix to this?

Standard Solution:

Perform two passes:

- Pass 1: Group tokens into instructions and record addresses of labels (data structure?).
  - 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 compute the value.

Your Assembler

When writing your assembler, you will do one things:

- Output the machine code coming from the assembled MIPS code to stdout.
- NOT THIS: 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

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

Creating Binary In C++

How do we *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 (figuring out the actual binary), but let's generalize the above and do it completely in C++.

Bit-wise Operations

Our instruction (bne) can be broken down as follows:

------	------	------	----------------
Opcode (6 bits)	**Register s (5 bits)**	**Register t (5 bits)**	**Offset (16 bits)**

In this case,

- bne has opcode `000101 = 5`
- Register s is `00010 = 2`
- Register t is `00000 = 0`
- Offset is `1111111111111101 = -3`

Bit Shifting!

We can use bit shifting to put 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. Why?

Recall in C++, ints are 4 bytes. We only want the last two bytes. First, we need to apply a “mask” to only get the last 16 bits:

```
offset = -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!):

```
    0001 0100 0100 0000 0000 0000 0000 0000
|   1111 1111 1111 1111 1111 1111 1111 1101
-------------------------------------------
    1111 1111 1111 1111 1111 1111 1111 1101
```

With Masking:

```
    0001 0100 0100 0000 0000 0000 0000 0000
|   0000 0000 0000 0000 1111 1111 1111 1101
-------------------------------------------
    0001 0100 0100 0000 1111 1111 1111 1101
```

Are We Done?

Finally, presumably, we just need to

```
cout << instr
```

right?

No! This would output **9 bytes** corresponding to the **ASCII code** for each digit of the instruction as interpreted in decimal! We want to output the four bytes that correspond to this number!

Printing Bytes in C++

Finally, 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 into assembly language.
- Assembly language has a simple and universal way to be translated to machine code for a specific architecture.

End of MIPS

- 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 and translate strings of text.