CS 241 — Lecture 6

Bartosz Antczak *Instructor: Kevin Lanctot* *January 19, 2017*

Recall

Calling and returning from a subroutine:

```
main:  sw $31, -4($30)     ; push $31 onto the stack and update SP ($30)
       lis $31
       .word 4
       lis $5              ; load address of the subroutine we're calling
       .word func          ; (named func) and jump to it
       jalr $5

       list $31            ; Returning from func
       .word 4
       add $30, $30, $31   ; update SP by adding 4 and
       lw $31, -4($30)     ; pop top of stack and return
       jr $31
```

Recall that we use `jalr` to *call* a function and `jr` to *return* from one.

6.1 What Assemblers Do

Recall that an *assembler* converts an assembly language to machine code (i.e., binary). The assembler reads through the assembly language *twice*, once for *analysis* and the other time for *synthesis*

- **Analysis**: break each line from the assembly language into components and also scan for errors:
  - Breaking each line into components: a component is a particular part of code in the line, such as whitespace or opcode. We assign a token to every component (i.e., label every item in the line into a particular group). A program that assigns tokens to components is provided for us on assignments, called `asm`)
  - Error-checking: we make sure the code is *syntactically* and *semantically* correct. Syntax is the structure of the code (or language), an example of a syntax error in MIPS would be `lw $1` (this opcode takes in two arguments, not one). Semantics is the meaning of the code (or again, language). A prime example of a semantics error in MIPS is defining the same label twice (you would not know to which label to jump to; no meaning!)

  At the end of this step, this process passes an *intermediate representation* (the set of tokens that store all the components of each line) and a *symbol table* (a table that maps labels to addresses) to the synthesis portion of reading through the assembly language

- **Synthesis:** receives the *intermediate representation* and *symbol table* and translates to machine code

6.2 Implementing an Assembler

For our assignments, we'll be using high-level language (either C++, Racket, or Scala) to construct an assembler which runs an analysis and a synthesis read on your MIPS assembly language. As an example of implementing an assembler, we'll walk through how to convert `bne $2, $0, top` in detail:

6.2.1 Converting MIPS to Machine Code Example

For `bne $2, $0, top`:

1. Look up `top` in the symbol table to determine its address
2. As an example, let's assume `top` is in address `0x0C`. To find the number of instructions needed to jump to reach `top`, we calculate $(\texttt{top} - PC)/4 = (\texttt{0x0C} - \texttt{0x20})/4 = -5$ (we divide by 4 because the PC increments by 4)
3. Now, the instruction becomes `bne $2, $0, -5`
4. Referring to our MIPS reference sheet, `bne` in binary is

   `0001 01ss ssst tttt iiii iiii iiii` iiii

   Before we continue on, let's learn how to *build* a binary string using a few methods:

   Bitwise 'And'

   Turning particular bits in a binary string "off". For example, if we want to turn "off" the first three bits in the binary string $a = 01011$, we'd apply a *bitwise and* operation on it using another binary string, particularly $b = 00001$. Applying the operation, we produce another binary string, denoted as $a\&b$:

   $$\begin{aligned} a &= 01011 \\ b &= 00011 \\ a\&b &= 00011 \end{aligned}$$

   This is called *masking off* certain bits in the string. What we're doing here, is lining up every individual bit in a with an individual bit in b and simply applying the *and* operation on those two bits. We then place the new bit in our new string that represents a with some of its bits turned off (in this case, the first three bits).

   Bitwise 'Or'

   It follows the same procedure at the *bitwise and* operation, except when we compare individual bits, we use the OR operator (i.e., $a \vee b$), rather than the AND operator.

Shift Left Operator

This operation, denoted as $a << n$, cuts the first n binary digits in the string off and then replaces them by introducing n 0's on the right hand side. For instance, if $a = 01101001$ then,

$$a << 3 = 01001\underline{000}$$

The underline 0's are the three additional zeros we concatenated to the string (also we removed the first three digits '011')

5. Now, to build the binary string representation of `bne $2, $0, top`!

```
0001 01ss ssst tttt iiii iiii iiii
```

Observe that there are four specific sequences in the binary string representation — the first six digits (000101), followed the s, t, and i sequence. What we'll do is create four 32-bit binary strings that represent each sequence, shift them to their appropriate spot in the string, and then use the *bitwise or* operation on all four of them. In the end, we get the binary string

$$0001\ 0100\ 0100\ 0000\ 1111\ 1111\ 1111\ 1011$$

If we want to output the binary string instruction, we'll need use the C function `putchar`, which takes an `int` as an argument and outputs a converted `char`. We'll output each byte as a char using this function.

You should now have all the information needed to complete assignment 3 and 4, which involves creating most of a small assembler