CS 241 — Lecture 5

Bartosz Antczak *Instructor: Kevin Lanctot* *January 17, 2017*

5.0.1 Remember

For all of your MIPS assembly instructions, end it with `jr $31` (a.k.a., return).
To clear a register, simply add zero to it: `add $r, $0, $0`.

Recall

The **stack** uses part of the RAM to store temporary values. Since registers are global, any function can access them, so to prevent any function from manipulating any data (also called "context") stored in the registers, we use the stack to temporarily store these values whenever a function is called. When the function returns, we pop these stored values off the stack and place them back in their respective registers.
A **subroutine** is a defined function in MIPS assembly language.

5.1 Saving and Restoring Context on the Stack

5.1.1 Saving Context

When we save an item on the stack, we define it as *pushing onto the stack*. It follows a two step process:

1. Store the respective values from the registers
2. Decrement the stack pointer by $4i$, where i is the number of registers that we're saving data from

5.1.2 Restoring Context

Once a function returns, it restores i values from the stack, this is called *popping off the stack*. It follows two steps (the opposite of saving context):

1. Increment stack pointer by $4i$
2. Load values back into registers

5.2 Working with Subroutines

How do we call and return from a subroutine? We use the `jalr` instruction. Recall that this address jumps to a particular section in memory (i.e., a function call) and then stores the return address in $31 (i.e., return). This approach is beneficial; however, we must ask the question, *what if the function is a recursive function*? This causes the return address to be overwritten.
The **solution** to to save the contents of $31 in the stack! We do this by

1. Pushing $31 (the current return address) onto the stack and updating the stack pointer
2. Jump to the respective subroutine using `jalr` (this now causes $31 to store a new return address, with the previous return address being saved on the stack)

After returning from the respective subroutine, we restore the previous value of $31 using the restoration steps defined previously.

5.2.1 Passing arguments and returning results

In CS 241, we'll pass all parameters on the stack. These procedures follow the same process as defined above.

— We have covered everything needed for assignment 2 —

5.3 Assemblers

5.3.1 Introduction

An **assembler** converts an assembly language program into its corresponding machine code (on assignment 1 for problems 2-6, *we* were the assembler, converting assembly language to binary). For assignments 3 and 4, we'll *build* a small assembler (i.e., a variation of the provided `cs241.binasm` assembler).
An assembler takes in a file containing instructions in assembly language (more formally, *an input file represented in ASCII text that can be edited with a text editor*), and outputs a binary file (which is something that can't be edited with a text editor) which contains encoded MIPS instructions.

5.3.2 Steps in the Process

When we analyse the MIPS instructions, we'll read it twice: first to *analyse*, second to *synthesize*:

Analyse

When we initially read the instructions, we break each line into components, parse components, and check for errors. When we recognize components in the instructions, we place each line into *tokens*. Some examples of tokens include:

- LABEL: any declaration of a label
- ID: an opcode or the use of a label without a colon
- REGISTER: a register
- COMMA: a comma

Note: this step of placing each component of a line into these tokens will be done by a provided function, so we don't need to worry about doing this step on assignments.

5.3.3 Synthesis

After the initial read, now we will actually construct the binary MIPS machine code from the components and direct it to output.

More on this in the next lecture