CS 241 — Lecture 4

Bartosz Antczak *Instructor: Kevin Lanctot* *January 12, 2017*

4.0.1 Today's lecture plan

- Labels
- Arrays
- Structure of Assembly files
- Output
- Function calls

4.1 Additional MIPS instructions

4.1.1 Branch Labels in MIPS

Rather than including the immediate variable in a branching instruction, we can use labels as a substitute for calculating the offset (how many instructions we're skipping). For example:
Without label (what we're used to):

0. `mult $2, $3`

4. `mflo $2`

8. `sub $3, $3, $1`

12. `bne $3, $0, -4`

With a label (we called it "loop"):

0. `loop: mult $2, $3`

4. `mflo $2`

8. `sub $3, $3, $1`

12. `bne $3, $0, loop`

Here, loop is the label. It's placed in the first column and it ends with a colon. As shown, it replaces the immediate value that determines how many instructions to skip.

Labels and Scope

Labels can be created manually and automatically. They must also be unique within scope.

Example 4.1.1. *Another example of using a label:*

0. `beq $1, $0, useGST`

4. `add $2, $2, $4`

8. `beq $0, $0, final`

12. `useGST: add $2, $2, $3`

16. `final:` *; continue with program*

4.1.2 What's in an Assembly File?

- Assembly instructions
- Label declarations
- Data definitions (such as `.word`)
- Comments (start with a semi-colon)

A suggested format for an assembly file is to divide your data into three columns: label, instructions, and comments:

Labels:	**Instructions**	**Comments**
`loop:`	`add $2, $2, $4`	*; $2 = 32 in decimal, $4 is zero*

4.1.3 Arrays in MIPS

Recall: to access a particular element in an array, we need to know the element's address in the array. The size of each element in the array is 4 bytes.
The *base address* of the array is the address of the first element (0th index in the array). To access each successive element, add 4 to the current element's address. For instance, if the base address is in register $1, then the i-th element has an address of $\$1 + 4i$.

4.1.4 Output

Input and output from devices are treated just like reading from and writing to memory, which means that we'll use the MIPS instructions `lw` and `sw`; however, to specify that we want to input or output data, we'll load and store data to specific locations in memory.
In CS 241, to output a char to the screen, store the ASCII value of the character to memory address $\text{FFFF000C}_{\text{hex}}$ (note that this address is fixed, but in reality this address is different for various devices). This is also called the *video output*. This method outputs one character at a time.

Example 4.1.2. *Print "C" on the screen*

```
lis $1              ; address of output buffer
.word 0xFFFF000C
lis $2              ; ASCII C
.word 67
sw $2, 0($1)        ; write to screen
```

4.1.5 Writing Functions in MIPS

In MIPS, we don't have functions; instead, we have *subroutines*. We will implement `jr`, `jalr`, labels, and registers to implement functions (i.e., we have to build them).

Subroutines vs. Functions

We define

- Labels as function names
- Arguments and return values as values stored in certain registers or a specific memory location

Since subroutines can access any register they want, there is no local scope.
Some challenges using subroutines:

- The subroutine call is static (since the address of it is always in the same location)
- The return is dynamic, because the location of the function call could happen anywhere (consider a C++ program, you can call a function from anywhere)
- Nested calls, recursion

To navigate through the code, we'll use two instructions:

- `jalr $s`: means *jump and link register*. It behaves just like `jr` (i.e., sets the PC to the address stored in $s) with an additional action: it also copies the address of the next instruction to $31 (this instruction is used to jump into a function)
- `jr $s`: already defined from previous lecture. This instruction is used to exit a function

Where do we store essential data in the registers? Since registers are not local variables, any function can access them. What if we have important data in $2, but a certain function accesses $2 and changes the value. Where do we store this essential data before we call that function? We store it in a *stack*.

The Run-time Stack

We store the stack in a part of memory (i.e., RAM). It follows a last-in, first-out queue. Some things to note about the stack:

- The stack grows downward in memory. This means that in order to access the ith item in the stack, we subtract $4i$ from the base address. For instance, the address of the first item below the base address is (base address-4)

- The address of the bottom of the stack is stored in the *stack pointer* (SP) register. In our MIPS simulator, we use $30 (however typically it's $29)

— At this moment, you can do 7 out of 8 questions on assignment 2 —