CSC258H Week 10 Lab: Control Structures

1 Introduction

When programming in a high-level language like Python, Java, or even C, we use three main control-flow constructs: branches, loops, and function calls. Last week, we learned how to create programs in assembly for the MIPS architecture, but these programs did not include any "control". This week, we will write programs with branches and loops to see how these constructs are implemented in assembly. When we finish this lab, we hope you will have a better appreciation for the structures a high level language provides!

Like last week, take your time on this lab and ask your neighbors or the TAs questions whenever you see something you don't understand. We will be practicing writing functions next week, so it's important that you be comfortable with the syntax for system calls and branch and memory instructions. If you need a reference, please check the extra resources posted on the course webpage. I've found Larus's guide particularly useful for a high level overview.

This week's lab assumes that you completed all of the material from last week. If you did not get the program in Section 4 working last week, make sure to complete it before lab time this week.

2 If-Else

An If-Else statement (or *branch* or *conditional statement*) is a control structure that creates conditionally executed code. The structure relies on a *predicate* – an expression that evaluates to a Boolean value – `True` or `False`. The *if* is always followed by a block that is executed when the predicate is `True`. This is the *Then* block. The Then block is optionally followed by an *Else* block that is executed if the predicate is `False`.

In assembly, If-Else is implemented using `labels` and `branch` operations. If necessary, the predicate is simplified using normal arithmetic operators. Then, it is evaluated using a `branch`. The `branch` uses a label to specify what the next instruction to execute should be if the predicate evaluates to `True`. If it evaluates to `False`, then the next instruction is executed (as normal). Here is an assembly implementation of a branch:

```
# if x < 5 {
#     y = 1
# }
# else {
#     y = 2
# }

IF:                     # This label isn't required but is added for clarity.
    addi $t1, $t0, -4   # Prepare to evaluate x - 4 <= 0.
    bgtz $t1, ELSE      # Branch to the label ELSE if the predicate is False.

THEN:                   # This label isn't required but is added for clarity.
    li $t2, 1
    b DONE

ELSE:
    li $t2, 2

DONE:                   # This label marks the end of the If-Else.
```

Note that the predicate is evaluated in an odd way. Branches in MIPS can only compare if two values are equal (beq) or not equal (bne), if one value is $<= 0$ (blez), or if one value is > 0 (bgtz). Some arithmetic has to be done to make most comparisons into comparisons to zero. Furthermore, the branch usually checks

whether the predicate is `False` and then branches to the `Else` block. (Note: In class, we used a different implementation of "if" that used a branch that is actually a pseudoinstruction. Feel free to use those. We're using the native instructions in this example to show you what the assembler must generate.)

Make a copy of your program from Section 4 of last week's lab. This program prompted the user for a maximum string length, allocated space for that string, and then prompted the user for a string. Modify that program to check whether the length the user provides is greater than 0. If the length is less than or equal to zero, modify the program to print an error message and then terminate.

3 Loops

Loops are very similar to branches. If we start with a branch with a Then block and no Else block, then the difference is that we name the Then block the loop's *body* and may execute it multiple times. This means that the bottom of the Then block is an unconditional branch back to the top of the loop. For example:

```
# x = 0
# while x < 5 {
#     x = x + 1
# }

LOOPINIT:               # Many loops have an initialization section.
    li $t0, 0
WHILE:                  # The loop checks the condition, then evaluates the body.
    addi $t1, $t0, -4
    bgtz $t1, DONE

    addi $t0, $t0, 1
    b WHILE
DONE:                   # This label marks the end of the loop.
```

This code breaks the loop into three parts. First, the initialization block sets up loop variables. Second, the loop's predicate is evaluated, and if the predicate is false, control jumps to the code after the loop. Third, the loop body is evaluated, and an unconditional branch is made to the top of the loop.

Start with your code from the previous section. Instead of exiting immediately if the user inputs an invalid length, have your program print the error message and then try again. If the user has entered an invalid length N times, then exit. Make N a parameter in the `.data` section with value 3. (*Hint:* Use the `.word` keyword to create space for an integer. Use `lw` to load the word into a register.)

4 Program: Average

Write a program that asks the user for an integer N. N represents the number of integers to be averaged. After getting a legal value N, the program asks for an integer N times and then prints their average. (You may, for simplicity, assume that the user will not provide inputs that will cause overflow.)

5 Summary of TODOs

Below is a short summary of the steps to be completed for this lab:

1. Modify your program from last week to check the input to diagnose buffer overruns.
2. Further extend your program to ask the user for input again if they provide an invalid input.

3. Write a new program that computes the average of the numbers provided by the user.

Evaluation (3 marks in total): 1 mark for error checking string input, 1 mark for creating the final version of the program from last week (with a loop), and 1 point for demonstrating that your averaging program works. Be prepared to answer questions about your code should the TA ask.

In next week's lab, we'll be writing recursive code that relies on implementing a function call convention.