Homework #2 – Assembly Programming

Due date: see course website

Directions:

- For short-answer questions, submit your answers in PDF format as a file called <NetID>-hw2.pdf. ***Word documents will not be accepted.***
- For programming questions, submit your source file using the filename specified in the question.
- **You must do all work individually, and you must submit your work electronically via Sakai.**
    - All submitted code will be tested for suspicious similarities to other code, and the test will uncover cheating, even if it is "hidden" (by reordering code, by renaming variables, etc.).

Q1. MIPS Instruction Set

(a) [5 points] What MIPS instruction is this? 0000 0010 0010 0010 0100 1000 0010 0110

(b) [5] What is the binary representation of this instruction? `lw $t3, 8($s1)`

Q2. C compilation and MIPS assembly language

In Homework 1 Q3 ("Compiling and Testing C Code"), you observed how changing the level of compiler optimization altered execution time. In this question, we will examine how that works.

The computer used in Homework 1 was a Duke Linux system, which is a PC based on the Intel x86 64-bit architecture, so the compiler generated instructions for that CPU. In this question, we want to examine the resulting assembly language code, but we aren't learning Intel x86 assembly language, so we'll need a compiler that produces MIPS code instead. **Such a tool has been developed for you here**. This web-based tool acts a front end to a `g++` compiler which has been set to produce MIPS code. Further, it's been set up to show us the assembly language code (.s file) rather than build an executable binary.

A small piece of the program from Homework 1, **prog_part.c**, is linked on the course website. Compile this code to assembly language with optimization disabled (-O0) and set to maximum (-O3). Locate the get_random function (labeled "_Z10get_randomv" or similar) in the two versions of the code to answer the following questions:

(a) [1] How many total instructions (not dot-prefixed directives like .frame) are in each of the two versions?

(b) [1] How many memory accesses (loads and stores) were in each of the two versions?

(c) [1] Which version of the code uses more registers?

(d) [7] Based on the above, what are some general strategies you suspect the optimizer takes to improve performance?

Note: Do not try to use the web-based MIPS C compiler to "automate" the programming questions below. In addition to being academically dishonest, it also won't work very well for the technical reasons listed on the web-based tool itself.

Q3. Assembly Language Programming in MIPS

For the MIPS programming questions, use the QtSpim simulator that you used in Recitation #3.

These programming questions are almost the same as those from Homework 1 with the following key differences:

- In HW1, input came from command line arguments. In this assignment, input is typed into the console.
- Because the program is now prompting for input interactively, your program will output prompts before reading values. We intend to use an automated tool to assist with grading, so **please end all your user prompts in a colon**.
- Like HW1, an **automated self-test tool** is provided. This tool relies on a command-line version of QtSpim (simply called 'spim') which is pre-installed on Duke Linux machines. Therefore, you'll need to get your work over to your Duke home directory for automated testing. The automated tester is not meant to *diagnose* issues with your code; for that, you'll need to get hands-on with QtSpim to trace the root cause, manually typing inputs and observing program flow and results. Also, the automated tests provided are not meant to be exhaustive, and additional tests will be used by the instructors for grading. See the Homework 1 write-up for details on the tester; as this one works the same way.
- Some test cases from HW1 no longer apply, and have been eliminated.

The test files are in `homework2-kit.tgz`, linked on the course site.

Each of your programs should prompt for input and display output via the **QtSpim Console window**. To execute an instructor-provided test manually, type in the **Input** listed in the tables associated with each program when prompted. After you have run your program, your program's output should match the **expected output** from the file indicated.

For problem (c), the input comes from the file listed in the **Input file** column. Each line in the file should be typed in individually per prompt as instructed in problem (c).

Q3a: byseven.s

[20] Write a MIPS program called byseven.s that prints out the first N positive integers that are divisible by 7, where N is an integer that is input to the program. Your program should prompt the user for the value of N via the console and receive input from the user via the console using syscalls.

You will upload byseven.s into Sakai (via Assignments). The following tests cases are provided:

Test #	**Input**	**Expected output file**	**What is tested**
0	1	byseven_expected_0.txt	input of 1
1	2	byseven_expected_1.txt	input of 2
2	3	byseven_expected_2.txt	input of 3
3	4	byseven_expected_3.txt	input of 4

Q3b: recurse.s

[40] Write a MIPS program called recurse.s that computes f(N), where N is an integer greater than zero that is input to the program. f(N) = 3*(N-1)+f(N-1)+1. The base case is f(0)=2. Your code must be recursive, and it must follow proper MIPS calling conventions. **The key aspect of this program is to teach you how to obey calling conventions; code that is not recursive or that does not follow MIPS calling conventions will be severely penalized (-75% penalty)!** Your program should prompt the user for the value of N via the console and receive input from the user via the console using syscalls.

Note: the calling conventions must be followed in every function, *including* ***`main`****!* Your main function should return (`jr $ra`) when finished rather than using the exit syscall. Caller- and callee-saved registers should be saved as needed at the appropriate times. There should be no data sharing via registers between functions other than $a for arguments and $v for return values.

You will upload recurse.s into Sakai (via Assignments). The following tests cases are provided:

Test #	**Input**	**Expected output file**	**What is tested**
0	1	recurse_expected_0.txt	input of 1
1	2	recurse_expected_1.txt	recursion of 2
2	3	recurse_expected_2.txt	deeper recursion
3	4	recurse_expected_3.txt	even more recursion

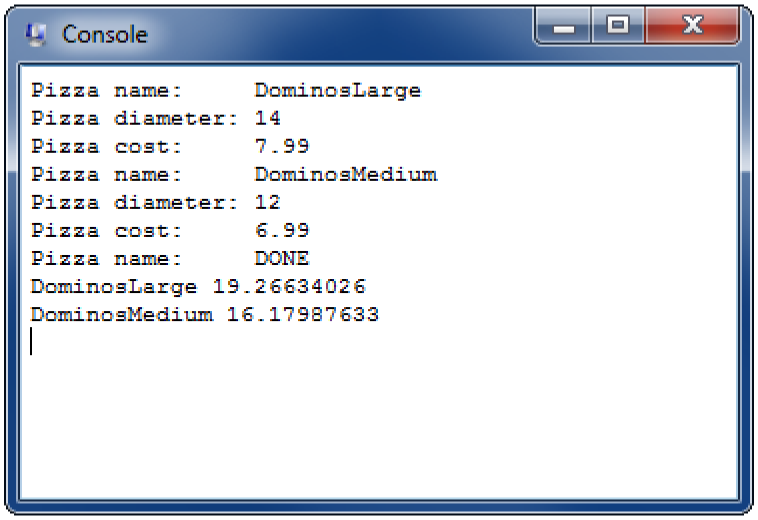
Q3c: PizzaCalc.s

[50] Write a MIPS program called PizzaCalc.s that is similar to the C program you wrote in Homework #1. However, instead of reading in a file, your assembly program will read in lines of input from the console. Each line will be read in as its own input (using spim's syscall support for reading in inputs of different formats). The input is a series of pizza stats, where each pizza entry is 3 input lines long. The first line is a name to identify the pizza (a string with no spaces), the second line is the diameter of the pizza in inches (a float), and the third line is the cost of the pizza in dollars (another float). After the last pizza in the list, the last line of the file is the string "DONE". For example:

```
DominosLarge
14
7.99
DominosMedium
12
6.99
DONE
```

Your program should prompt the user each expected input. For example, if you're expecting the user to input a pizza name, print to console something like "Pizza name: ".

Your program should output a number of lines equal to the number of pizzas, and each line is the pizza name and pizza-per-dollar (in^2/USD) , which should be set to zero if either diameter or price are zero). **The lines should be sorted in *descending* order of pizza-per-dollar, and in the case of a tie, *ascending* order by pizza name.** An example execution of the above input is shown below:



```
Pizza name:     DominosLarge
Pizza diameter: 14
Pizza cost:     7.99
Pizza name:     DominosMedium
Pizza diameter: 12
Pizza cost:     6.99
Pizza name:     DONE
DominosLarge 19.26634026
DominosMedium 16.17987633
```

IMPORTANT: There is no constraint on the number of pizzas, so you may not just allocate space for, say, 10 pizza records; you must accommodate an arbitrary number of pizzas. You must allocate space on the heap for this data. **Code that does not accommodate an arbitrary number of pizzas will be penalized (-75% penalty)!** Furthermore, you may NOT first input all names and data into the program to first find out how many pizzas there are and *then* do a single dynamic allocation of heap space. Similarly, you may not ask the user at the start how many player records will be typed. Instead, you must dynamically allocate memory on-the-fly as you receive names. To perform dynamic allocation in MIPS assembly, I recommend looking here.

Note: You must follow calling conventions in this program. This includes every function, *including* ***`main`***! Your main function should return (`jr $ra`) when finished rather than using the exit syscall. Caller- and callee-saved registers should be saved as needed at the appropriate times. There should be no data sharing via registers between functions other than $a for arguments and $v for return values.

You'll also be using floating-point MIPS instructions, which are different from the integer ones we learned. Like many processors, MIPS considers floating point separate from the "real" CPU; it's instead "co-processor 1". You can find floating-point-specific operations are here, but if you want to get data in/out of the floating point unit or to take branching decisions based on floating point comparisons, you'll need instructions like mtc1 (move to coprocessor 1), bc1t (branch if coprocessor 1 comparison is true), etc., which are described here. Also, the different floating point registers have different roles similar to the integer registers (caller-saved, callee-saved, etc.); these are distinguished on the second page of this document. Lastly, as there is no load-immediate instruction for floats, the easiest way to specify a floating point constant such as π is to put it in memory as shown below and load it with `l.s`:

```
    PI: .float 3.14159265358979323846
```

You will upload PizzaCalc.s into Sakai (via Assignments). The following tests cases are provided:

Test#	Parameter Passed	Expected output file	What is Tested
0	tests/PizzaCalc_input_0.txt	tests/PizzaCalc_expected_0.txt	One pizza
1	tests/PizzaCalc_input_1.txt	tests/PizzaCalc_expected_1.txt	Two pizzas, in order
2	tests/PizzaCalc_input_2.txt	tests/PizzaCalc_expected_2.txt	Two pizzas, out of order
3	tests/PizzaCalc_input_3.txt	tests/PizzaCalc_expected_3.txt	Six pizzas
4	tests/PizzaCalc_input_4.txt	tests/PizzaCalc_expected_4.txt	Ensure we stop reading at "DONE"
5	tests/PizzaCalc_input_5.txt	tests/PizzaCalc_expected_5.txt	Correct output with diameter of zero
6	tests/PizzaCalc_input_6.txt	tests/PizzaCalc_expected_6.txt	Correct output with cost of zero
7	tests/PizzaCalc_input_7.txt	tests/PizzaCalc_expected_7.txt	100 pizzas, some stats are zero

BONUS: EFFICIENCY COMPETITION

In many real-world scenarios, it is essential to optimize assembly language code in some way. Developers may write small pieces of speed-critical code in assembly language for performance reasons, and developers of software for embedded systems often need to fit their code into very small amounts of storage or memory (such as the ATtiny4 microcontroller, which has only 512 bytes of storage and 32 bytes of RAM). Despite this, they must still produce readable, maintainable code.

In this spirit, we're going to have a competition to see which student can complete the PizzaCalc program in the fewest instructions possible (i.e., the shortest program).

The student who fulfills the requirements in the fewest instructions in the course will be recognized in class and win a copy of Code Complete, by Steve McConnell, a fantastic reference in modern software engineering. (No additional points will be awarded.)

Rules:

- Your solution must pass all student and instructor test cases.
- Your solution must have sufficient documentation and readability that we can follow its logic.
- Your solution must still follow all calling conventions (e.g., saving \$s/\$t as needed, no register data sharing between functions other than via \$a/\$v, etc.).
- Your solution to this contest (like all your assembly language programs) must ***NOT*** be generated in whole or in part by a compiler.
- Our definition of "instruction" is instructions written in source code. This means that "pseudo-instructions" that expand to multiple hardware instructions (e.g., `li` → `lui`+`ori`) will be counted as one.
- On the command line, the following perl command can count instructions in your program[1]:
  ```
  perl -ne 's/^\s*[\$\w]+://; s/^\s*\..*//; /^\s*\w/ and $n++; END{print "$n\t$ARGV\n"}' <FILENAME>
  ```
- In the event of a tie, all winning students will be recognized, but the instructor reserves the right to select a subset of winners to receive the prize based on ingenuity and/or readability of their submissions.

Note: This competition originated with Dana Lasher's Computer Architecture course (CSC236) at NC State University, where it has been running for decades.

[1] Perl is an incredibly ugly language, but it can do fancy things quick if you come to understand it.