Computer Science 230
Computer Architecture and Assembly Language
Fall 2021

Assignment 1

Due: Sunday October 16, 11:55 pm by Brightspace submission
(Late submissions **not** accepted)

Programming environment

For this assignment you must ensure your work executes correctly on the MIPS Assembler and Runtime Simulator (MARS) as was installed during Assignment #0. Assignment submissions prepared with the use of other MIPS assemblers or simulators will not be accepted. ***Solutions which prompt the user for input will not be accepted.***

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. **However, sharing of code fragments is strictly forbidden without the express written permission of the course instructor.** If you are still unsure regarding what is permitted or have other questions about what constitutes appropriate collaboration, please contact me as soon as possible. (Code-similarity analysis tools will be used to examine submitted work.) **The URLs of significant code fragments you have found and used in your solution must be cited in comments just before where such code has been used.**

Objectives of this assignment

- Understand short problem descriptions for which an assembly-language solution is required.
- Use MIPS32 assembly language to write solutions to three such small problems.
- Use MARS to implement, simulate and test your solution.
- Hand draw a flowchart corresponding to your solution to the first problem.

A few requirements for all of the code in Assignment #1

As these may be your first assembly-language programs, I want to focus on only those aspects of the MIPS architecture that you need to start. Therefore your submissions must obey the following:

- **Please keep your code within the area indicated for student work in the provided files.** Code outside of this area (these areas) will be modified during evaluation of student work (i.e., for different test cases).
- **You must refer to registers by number, not by name**. Register names will make more sense once we start to use procedures. For example, in your solutions you can use register $6, but you must not refer to this register as $a2.
- **You must not use `$0`, `$1`, `$2`, and `$31`**. You may otherwise use any other general-purpose register (i.e., registers $3 to $30 are available for your use). You are *not*, of course, required to use all of them!
- **The only branching or branching-support instructions** you may use are b, beq, bne, bgtz, blez, bgez, bltz, and slt. You are *not*, of course, required to use all of them!
- **You must not use functions/procedures for this assignment.**

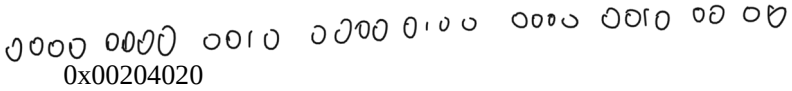
Part (a): Computing *odd parity*

Digital-communication hardware and software often include circuitry and code for detecting errors in data transmission. The possible kinds of errors are large in number, and their detection and correction is sometimes the topic of upper-level Computer Science courses. In a nutshell, however, the concern is that bits sent by a data transmitter are sometimes not the same bits that arrive at the data receiver.

That is, if the data transmitter sends this word:

0x00200020

but this is received instead:



0x00204020

then the data receiver has the wrong data, and would need some way to determine that this was the case. In response to the error, the same receiver might discard the word, or ask the sender to retransmit it, or take some other combination of steps.

A technique for detecting one-bit errors is to associate with each word a *parity bit*. If an *odd parity bit* is transmitted along with the word, then *the number of all set bits in the word* plus *the value of the parity bit* must sum to *an odd number*. In our example above involving the data transmitter, the chosen parity bit would be 1 (i.e., two set bits in 0x00200020, plus one set parity bit, equals three set bits). The data receiver has the corrupted byte as shown (0x00204020) plus the parity bit value, and determines that the number of set bits is four(i.e., three bits in the word, plus the set parity bit,

resulting in four set bits, which is an even number). Given that four is not an odd number, the receiver can conclude there was an error in transmission. Note that the receiver can only detect an error with the parity bit; more is needed to correct the error. (Also notice that if two bits were in error, then our scheme would not detect this event.)

Your main task for part (a) is to complete the code in the file named parity.asm that has been provided for you. You must also draw by hand the flowchart of your solution and submit a scan or smartphone photo of the diagram as part of your assignment submission; please ensure this is a JPEG or PDF named "parity_flowchart.jpg" or "parity_flowchart.pdf" depending on the file format you have chosen.

Some test cases are provided to you.

Part (b): Reversing the order of bits in a word

Recall that in our course we define a word to be a 32-bit sequence (i.e., four consecutive bytes). For some algorithms it is useful to have a reversed version of that 32-bit sequence. (The deeply curious can read a brief description about such use in Fast Fourier Transform algorithm implementations by visiting Wikipedia at this link: http://bit.ly/2rnvwz6).

Your task for part (b) is to complete the code in reverse.asm that has been provided for you. Please read this file for more detail on what is required.

Some test cases are provided to you.

Part (c): An implementation of *division*

You are familiar with integer division operations. In some processors there is no division instruction. One way to implement division is via repeated subtractions.

NB: This is integer division, NOT floating point division.

For example, consider the expression M / N where M = 370 and N = 120. If we repeatedly subtract until we have a value less than N, then the number of times we do the subtraction is the result. Ie:

- 370 – 120 => 250 (subtraction #1)
- 250 – 120 => 130 (subtraction #2)
- 130 – 120 => 10 (subtraction #3)

When we have the result 10, we have performed 3 subtractions. Therefore, 370 / 120 = 3

Your task for part (a) is to complete the code in division.asm that has been provided for you. In that file there are some comments giving several simplifying assumptions.

For example, the value M will always be a positive two's-complement number; the value N will always be a two's complement positive number less than 128. For greater certainty, N will never be 0. Your code will not be tested with inputs that do not meet these requirements.

Some test cases are provided to you.

What you must submit

- Your completed work in the three source code files (parity.asm, reverse.asm, division.asm). Do not change the names of these files!
- **Your work must use the provided skeleton files**. Any other kinds of solutions will not be accepted.
- The scan / smartphone photo of your hand-drawn flowchart with the name of "parity_flowchart.jpg" or "parity_flowchart.pdf" depending on the format you have chosen.

Evaluation

- 2 marks: Parity solution is correct for different word values (part a).
- 1 mark: Hand-drawn flowchart for parity solution is correctly prepared (part a).
- 3 marks: Reverse solution is correct for different word values (part b).
- 3 marks: Division solution is correct for different values of M and N (part c).
- 1 mark: All submitted code properly formatted (i.e., indenting and comments are correctly used), and files are correctly named. *The meaning and purpose of registers in your code is clearly indicated (one comment per register).*

Total marks: 10