CS2100 Computer Organization
AY2023/24 Semester I
Assignment 1

Instructions

1. There are FOUR (4) questions in this assignment, totaling THIRTY-SEVEN (37) marks. Please do all parts of every question. Marks are indicated against each part.

2. An additional 3 marks is awarded for putting in your name, student ID and tutorial group number in your answers and for submitting in PDF format with the correct naming convention (see below). Thus, the total will be FORTY (40) marks. If you fail to do any of these, you will lose the 3 marks.

3. This assignment is due on **Monday, 18 September 2023, 1 pm.** You will be given until 1.15 pm to submit, **after which no submission will be accepted and you will receive ZERO for this assignment, regardless of how hard you've worked on it.**

4. **Plan to submit at least 2 hours early (i.e., by 11 am on 18 September)** in case there are technical issues with submission. If, by 1 pm on 18 September you are unable to submit due to technical reasons, please email your answers to Colin at colintan@nus.edu.sg before 1.15 pm. **No submission over email will be accepted after 1.15 pm.**

5. Complete your answers on the provided **CS2100Assg1AnsBk.docx** file. Save it as AxxxxxxY.pdf before submitting.

6. **Zip your AxxxxxxxY.pdf file together with your parity.c file from Question 1 into a file called AxxxxxxxY.zip, and submit that on Canvas.** You will forfeit the 3 marks if you do not do this, or if you fail to fill in your name, student ID and tutorial group number. **In addition if you do not submit your parity.c file, you will not receive marks for the programming portion of Question 1.**

7. You should do these assignments on your own. Do not discuss the assignment questions with others.

8. Please use the Canvas Discussion Forums for clarifications.

Question 1. (13 MARKS)

In Tutorial 2 Question 5 we looked at calculating the parity for a 32-bit word.

In this question we will write a program that computes parity *across* a block of bytes to create a *parity byte*. To understand what we mean by this, let's consider a block of four bytes: 0x1A, 0x30, 0x4B and 0x1C. We stack them up like this (for neatness the "0b" prefix for binary numbers is omitted):

0x1A: 0001 1010
0x20: 0010 0000
0x4B: 0100 1011
0x1C: 0001 1100

Let's assume that we are using the ODD parity scheme. In the odd parity scheme, if there are even number of 1's in the data bits, the parity bit generated is 1; if there are odd number of 1's in the data bits, the parity bit generated is 0. In other words, the number of 1's in the data bits and parity bit must be odd.

We now compute an *odd parity byte* where each bit at position *k* in the parity byte is a parity bit generated for the corresponding position-*k* bits of each byte in the block. Let's illustrate using the block of 4 bytes above, starting from left (MSB) to right (LSB):

The leftmost column has 0 bits that are '1'. Hence, the parity for that column is '1':

0 0 0 1 1 0 1 0
0 0 1 0 0 0 0 0
0 1 0 0 1 0 1 1
0 0 0 1 1 1 0 0

1

There is 1 bit in the next column that is '1'. Since there is an odd number of '1' bits, the partiy for this column is 0.

0 0 0 1 1 0 1 0
0 0 1 0 0 0 0 0
0 1 0 0 1 0 1 1
0 0 0 1 1 1 0 0
---
1 0

Continuing in this manner, the parity bits for the 8 columns are computed as follows:

0 0 0 1 1 0 1 0
0 0 1 0 0 0 0 0
0 1 0 0 1 0 1 1
0 0 0 1 1 1 0 0
---
1 0 0 1 0 0 1 0

Thus, the parity byte is 0b1001 0010 or 0x92.

a. Calculate the odd parity byte for the block below (you are advised to do this manually so that you can check the output of your program later. Write your answer in hexadecimal. (1 mark)

0x3C 0x4A 0x34 0x98 0x5B 0x4E 0x7F 0x4B

You are given five files:

i. test.c – Test harness for your code.
ii. test.txt – Test file containing strings of bytes and their parities.
iii. q1.c – A program just to ask for a string of bytes and print out the parity.
iv. parity.h – Contains prototypes for the functions in parity.c.
v. parity.c – You need to implement your odd parity byte calculation codes here.

What we want to do is to take hexadecimal values as a string in this form: "3A 2B 10 4A" and compute the odd parity byte for this string of bytes. We will leave this parity as an 8-bit unsigned integer between 0 and 255.

To do this you need to implement the following functions in **parity.c** (do not change any other file!):

Function	**Description**
uint8_t **findParity**(uint8_t *array, uint8_t len)	Calculates the odd parity byte for an array of bytes "array" of length "len".
uint8_t **hex2dec**(char *byte)	Converts the 2-hexadecimal-digit in "byte" to a decimal. E.g. if byte contains "2F", this function should return 47 (2×16+15). Complete this function in a single line to gain credit.
uint8_t **calculateParity**(char *str)	Calculates the odd parity byte for a string of 2-hexadecimal-digit values in "str" and returns it in **decimal**. For example, if str contains "1A 20 4B 1C" your function should return 146. Note: Do not include the 0x in the hex bytes! Also do not omit the leading zero, e.g. "4 A B" should be represented as "04 0A 0B".
void **string2bytes**(char *str, uint8_t *bytes, uint8_t *len)	Converts a string of hexadecimal values in parameter "str" into bytes in the array "bytes". The parameter len returns the number of bytes converted. For example, if str="03 0B 1C", then bytes will contain the values 3, 11 and 28 (hex 1C = 1×16 + 12×1 = 28), and "len" will contain the value 3 since 3 bytes were converted.

Note: The difference between **calculateParity** and **findParity** is that while **findParity** takes in an array of unsigned 8-bit integers, **calculateParity** takes in a string. The two functions are related but not the same.

To play around with your parity code, ensure that **parity.h**, **parity.c** and **q1.c** are in the same directory, then compile and run using:

```
gcc parity.c q1.c -o q1
./q1
```

This program will read in a string of bytes and print out the parity in decimal and hexadecimal.

To test your implementation, ensure that **parity.h**, **parity.c**, **test.c** and **test.txt** are in the same directory, then compile and run using:

```
gcc parity.c test.c -o test
./test
```

This program tests your parity function (part c) for a maximum of 5 marks, and your full implementation that takes a string of bytes and calculates the parity, for another 5 marks.

b. Cut and paste your code for **hex2dec** here. You will not receive any marks if the body of your function consists of more than one C statement. (2 marks)

c. Correctness of code: Test by using **test.c** and **test.txt**. Your graders will use a different set of test data from you, and to get full marks you must pass all of **their** test cases. Each test case is 1 mark. (10 marks)

Question 2. (10 MARKS)

a. Fill in the blanks for each of the questions below (1 mark per part). (3 marks)

i. $152_7 + 341_7 = X_7$. X in base 7 is _____.

ii. $312_5 + Y_5 = 441_5$. Y in base 5 is _____.

iii. $411_Z - 232_Z = 168_Z$. The mystery base Z is _____.

b. We have a 16-bit fixed-point number system in sign-and-magnitude representation. We use 8 bits for the fraction portion and the remaining bits for the integer portion. Write all answers below in decimal. (1 mark per part). (4 marks)

i. What is the smallest positive number that can be represented? Note that 0 is not a positive number.
ii. What is the largest positive number that can be represented?
iii. What is the most negative number that can be represented?
iv. What is the absolute error in representing the number 17.143? Calculate up to a maximum of 8 fraction bits; **do not** calculate to 9 fraction bits to decide rounding.

c. What is 17.143 in 32-bit IEEE-754 format? Write your answer in hexadecimal. Use the version of IEEE-754 as taught in the lectures. You do not need to consider rounding in your answer. (3 marks)

Question 3. (5 MARKS)

For each of the following segments of MIPS code, write down the equivalent C code fragment. You may assume that all variables listed in each question have been properly declared and you do not need to declare any of them. You may also use additional variables without declaring them, and similarly you may not need to map all registers to C variables. For example, registers that are just used to store temporary intermediate results may not need to be mapped to a C variable.

Though some variation from the MIPS code is unavoidable, your C code should follow the MIPS code as closely as possible.

a. Variable to register assignments: $s1 mapped to integer variable `ctr`, $s2 mapped to integer variable `x`. (1 mark)

```
     addi $t0, $zero, 5
     andi $s2, $s2, 0

a:   slt  $t1, $t0, $s2
     bne  $t1, $zero, b
     srl  $s1, $s1, 1
     addi $s2, $s2, 1
     j a
b:   …
```

b. Variable to register assignments: $s1 mapped to integer variable `ctr`, $s2 mapped to integer variable `x`. (1 mark)

```
     addi $t0, $zero, 5
     addi $s2, $t0, 10

a:   srl  $s1, $s1, 1
     addi $s2, $s2, -1
     slt  $t1, $s2, $t0
     beq  $t1, $zero, a
b:   …
```

c. Variable to register assignments: $s1 mapped to the integer variable `ctr` (which contains some previously assigned value that is not necessarily 0), $s2 mapped to the base address of an array *A* of 32-bit integers, $s3 mapped to the base address of an array *B*, also of 32-bit integers. Arrays *A* and *B* are of the same length, given by variable `v` which is mapped to $s4. (3 marks)

```
      sll  $t0, $s4, 2
      add  $s5, $s2, $t0
      sll  $t0, $s1, 2
      add  $t1, $s3, $t0
      add  $t0, $s2, $t0

a:    slt  $t2, $t0, $s5
      beq  $t2, $zero, c
      lw   $t2, 0($t0)
      lw   $t3, 0($t1)
      slt  $t4, $t2, $t3
      bne  $t4, $zero, b
      sw   $t2, 0($t1)
      sw   $t3, 0($t0)
b:    addi $t0, $t0, 4
      addi $t1, $t1, 4
      j a
c: …
```

Question 4. (9 MARKS)

We are given the following code in MIPS assembly. Register $s0 contains the base address for an array *A* of 32-bit integers. The LSB of every byte is 1 and hence every ulw results in loading an odd integer. Answer the questions that follow:

```
      add  $t0, $s0, $zero
      addi $t1, $s0, 16
      addi $s1, $zero, 0
a:    slt  $t2, $t0, $t1
      beq  $t2, $zero, c        # THIS LINE
      ulw  $t2, 0($t0)
      andi $t3, $t2, 1
      beq  $t3, $zero, b
      addi $s1, $s1, 1
b:    addi $t0, $t0, 2
      j a
c:    ...
```

a. How many times is the `beq` at the line marked # THIS LINE executed? (2 marks)

b. How many times is the `beq` in part a actually taken? (2 marks)

c. How many instructions are executed by this code at the point of completion? (2 marks)

d. How many unique bytes are read from array *A* (See slide 39 of Lecture 8 for example of what we mean by "unique bytes read") (3 marks)

=== END OF PAPER ===