Garcia, Nikolic — CS 61C — Quest
Fall 2020 — Machine Structures

Note (August 10, 2022): These are extremely rough drafts of rewritten solutions. They definitely contain errors and unfinished sections, but might have some useful parts for studying.

PRINT your name: _____, _____
(first) (last)

PRINT your student ID: _____

Read the following honor code and sign your name.

I understand that I may not collaborate with anyone else on this exam, or cheat in any way. I am aware of the Berkeley Campus Code of Student Conduct and acknowledge that academic misconduct will be reported to the Center for Student Conduct and may further result in, at minimum, negative points on the exam and a corresponding notch on Nick's Stanley Fubar demolition tool.

SIGN your name: _____

You have ??? minutes. There are 5 questions of varying credit, for a total of 0 points.
For questions with **circular bubbles**, you may select only one choice.

- ○ Unselected option
- ● Only one selected option

For questions with **square checkboxes**, you may select one or more choices.

- ■ You can select
- ■ multiple squares

Anything you write that you ~~cross out~~ will not be graded. Anything you write outside the answer boxes will not be graded.

Question 1 ***Base Conversion*** **(0 points)**

FOR THIS ENTIRE SECTION (Q1 TO Q3), YOU ARE NOT ALLOWED TO USE THE C PROGRAMMING LANGUAGE AS A CALCULATOR (OR A CALCULATOR EITHER, FOR THAT MATTER).

Q1.1 Convert 45_8 to base 3

Solution: 1101_3

$45_8 = 4 \times 8^1 + 5 \times 8^0 = 37$

$37 = 27 + 9 + 1 = 3^3 + 3^2 + 3^0$

$37 = 1101_3$

Question 2 ***Number Representation*** **(0 points)**

How would you encode −18 using 6 bits in the following representations? Write `N/A` if it is impossible. Please write exactly 6 bits – include any leading zeros if needed, and DO NOT include a leading `0b`.

Q2.1 Unsigned

Solution: N/A

Negative numbers cannot be encoded in unsigned representation.

Q2.2 One's Complement

Solution: `101101`

18 in 6-bit unsigned binary is `0b010010`. In one's complement, to negate the number, we flip the bits.

Q2.3 Two's Complement

Solution: `101110`

In two's complement, to negate a number, we flip the bits and add one. The previous subpart has 18 with the bits flipped. We can add one to get −18 in two's complement.

Q2.4 Sign-magnitude

Solution: `110010`

18 in 5-bit binary is `0b10010`.

The sign bit is `0b1` since the number is negative.

Q2.5 Bias (bias value: -31)

Solution: `001101`

First subtract the bias to get $-18 - (-31) = 49$.

13 in unsigned 6-bit binary is `0b001101`.

Question 3 ***Matrix Addressing*** **(0 points)**

Matrix stored as 2D array

A matrix representation stores its values (for this problem, only bytes) such that the offset address of the element in row i and column j can be calculated by concatenating the binary representations of i and j, where row and column indicies start from 0. The size of the bit fields for row and column number should be big enough to accommodate the *highest possible row and column numbers*. **The amount of space allocated for this array will be the maxiumum number of elements that can be represented by this bit field, even if we won't use all of them.**

For example: A 3 (row) x 5 (column) matrix, A, will have a 2 bit row field and a 3 bit column field. The row bits will have values `0b00` through `0b10` (0-2), and the column bits will have values `0b000` through `0b100` (0-4). The element in row 2 and column 3 will have an offset address of `0b10011`. The element in row 1 and column 2 will have an offset address `0b01010`. **Even though this array only stores 15 bytes, space will allocated for 32 bytes due to the size of the 5 bit wide field.**

Q3.1 Matrix M has `maxrows` = 13 rows and `maxcols` = 65 columns. Find the offset address of the element in row `row` = 7 and column `col` = 12. Put your answer in *binary*, without the leading `0b`, and **without any leading zeros**, so if we calculated an offset address (from above) as `0b01010`, we would enter `1010`.

Solution: `1110001100`

To uniquely identify one of 13 rows, we need 4 bits (which can represent $2^4 = 16$ possible values).

To identify row 7, we write 7 in unsigned 4-bit binary: `0b0111`.

To uniquely identify one of 65 columns, we need 7 bits (which can represent $2^7 = 128$ possible values).

To identify row 65, we write 65 in unsigned 7-bit binary: `0b1000001`.

Concatenating these two bitstrings together gives us `0b0111 1000 001` (removing the leading zero for the final answer).

Q3.2 Which of the following C expressions will produce the desired result in Q3.1? (Choose one)

- ○ `(maxrows << ceil(log2(maxcols))) & col`
- ○ `(row << floor(log2(maxcols))) | col`
- ○ `(row << ceil(log2(maxcols))) || col`
- ○ `(maxrows << ceil(log2(maxcols))) && col`
- ○ `(col << floor(log2(maxcols))) && row`
- ○ `(col << ceil(log2(maxcols))) | row`
- ● `(row << ceil(log2(maxcols))) | col`
- ○ `(maxrows << ceil(log2(maxcols))) || col`
- ○ `(col << ceil(log2(maxcols))) || row`
- ○ `(col << ceil(log2(maxrows))) && row`

Solution:

Note that when calculating the number of bits required to identify a column, we rounded up. For example, $\log_2(65)$ falls between $\log_2(64) = 6$ and $\log_2(128) = 7$, but we needed to use 7 bits to uniquely identify one of 65 different columns. Therefore we should use `ceil(log2(maxcols))` instead of `floor(log2(maxcols))`.

We're trying to encode the number of the selected row and column, not the maximum number of rows and columns, so we should use `row` and `col` instead of `maxrows` and `maxcols`.

The upper bits of the matrix should correspond to the row, and the lower bits of the matrix address correspond to the column. Therefore, we should shift the row number (`row`) left to take up the upper bits of the number.

Once we take the row number and left-shift it to the upper bits of the number, we have the row number followed by a sequence of zeros. We need to replace these zeros with the column number. To do this, we can use bitwise OR, because ORing any number with 0 produces the same number. Bitwise ORing the left-shifted row number with the column number will replace the zeros with the column number.

Note that bitwise OR is denoted by `|`. Logical OR is denoted by `||` but is not what we want. If we used logical OR instead, we'd get true (1) or false (0), not the bitstring we're looking for.

Q3.3 What integer would you pass to `malloc` when allocating space for matrix `M`, which only stores a byte per value in the matrix? (Yes, we know there might be a lot of wasted space.)

Solution: 2048

From the first subpart, we found that there are $4 + 7 = 11$ bits in the matrix address. The 11-bit address can address one of $2^{11} = 2048$ different bytes in memory.

Matrix as 1D Array

Another way of storing this array would be to store it in a column-major order one-dimensional array. A column-major order one-dimensional array is an array in which the columns of the given matrix are stacked one after another in the array. For example, given the matrix below:

1	2	3
4	5	6
7	8	9

We would get the array `[1, 4, 7, 2, 5, 8, 3, 6, 9]`.

The amount of space allocated for an array is equivalent to the number of elements in the array. For example, if we were to store a 3 x 5 matrix, we would only need to allocate space for 15 elements. The rows would be numbered 0-2, and the columns would be numbered 0-4.

Q3.4 Recall matrix has `maxrows` = 13 rows and `maxcols` = 65 columns. Find the offset address of the element in row `row` = 7 and column `col` = 12. Just write the integer value, no need to write it in binary.

Solution: 163

Note that in the example, the first column is stored in indices 0-2, the second column is stored at indices 3-5, and the third column is stored at indices 6-8. More generally, in a matrix with `maxrows` rows, the `col`th column is stored starting at index `maxrows` × `col`. In this question, the 12th column is stored starting at index $13 \times 12 = 156$.

Within the 12th column, we want the 7th row, which is at index $156 + 7 = 163$.

Q3.5 Which of the following C expressions will produce the desired result in Q3.4? (Choose one)

- ○ `col + maxrows + row`
- ○ `row + maxcols + col`
- ○ `row + col`
- ● `(col * maxrows) + row`
- ○ `(row * maxcols) + col`
- ○ `row + maxcols`

Solution: See previous solution for how to derive `col * maxrows`. Then we added `row` to find the row within the column.

Q3.6 What integer would you pass to `malloc` when allocating space for matrix `M`, which only stores a byte per value in the matrix? (Yes, we know there might be a lot of wasted space.)

Solution: 845

As the question notes, "the amount of space allocated for an array is equivalent to the number of elements in the array." In this case, to store a 13×65 matrix, we need $13 \times 65 = 845$ bytes.

Question 4 ***Bit Manipulation*** **(0 points)**

Write a function `bit_manip` that takes in an arbitrary 64-bit unsigned integer, and for every `N = 5` set of bits starting from LSB (considering the number zero-extended to a multiple of `N` bits), we rotate them right by `R = 3` bits and then turn on bit 2 and turn off bit 1 (within each group of 5 bits) and then return the final value.

As an example, if our input were a 16-bit number whose bits were `0bABCDEFGHIJKLMPQS` and `N = 5`, we rotate it right by `R = 1` bit, and `ON = 1` and `OFF = 3` then:

- `0bABCDEFGHIJKLMPQS`...being throught of as groups of `N = 5` would become
- `A BCDEF GHIJK LMPQS`...then zero-extended to a multiple of `N = 5` bits would become
- `0000A BCDEF GHIJK LMPQS`...after the rotate right by 1 bit, `R = 1`, would become
- `A0000 FBCDE KGHIJ SLMPQ`...after turning on bit `ON = 1`
- `A0010 FBC1E KGH1J SLM1Q`...after turning off bit `OFF = 3`
- `A0010 F0C1E K0H1J S0M1Q`...and returning the lowest 16 bits means we'd return...
- `0B0F0C1EK0H1JS0M1Q`

Download the included files (shown below)

You may edit `main.c` to test your code. Do not edit the `bit_manip` signature. You will only be submitting `bit_manip.c` and only the code in that file will be graded.

```c
/* bitmanip.c */
#include <inttypes.h>

uint64_t bit_manip(uint64_t num) {
    //YOUR CODE HERE
    return 0;
}
```

```c
/* main.c */
#include <inttypes.h>

extern uint64_t bit_manip(uint64_t);

int main(int argc, char *argv[]) {
    return 0;
}
```

Solution:

Solution Walkthrough: here

```c
/* bitmanip.c */
#include <inttypes.h>

// Replace with values given in question prompt
int GROUP_SIZE = 5;
char *ROT_DIR = "right";
int ROT_AMT = 3;
int ON_BIT = 2;
int OFF_BIT = 1;
```

```
unsigned get_bit(uint64_t x, uint64_t n) {
    // n = 2
    //     ABCDE
    //     ABC   (>> n)
    //       C   (& 1)
    return 1 & (x >> n);
}

void set_bit(uint64_t *x, uint64_t n, uint64_t v) {
    // n = 2
    //     ABCDE
    // AND 11011 // n = 2, (1 << 2) = 0b...00100, ~(1 << 2) = 0b...11011
    //     -----
    //     AB0DE
    //  OR 00v00 // (v << 2) = 0b...00v00
    //     -----
    //     ABvDE
    *x = (*x & ~(1 << n)) | (v << n);
}

// Not the most efficient solution,
// but written for slightly better readability
uint64_t bit_manip(uint64_t num) {
    uint32_t num_groups = 64 / GROUP_SIZE + 1;
    uint64_t mask = 0;
    uint64_t vals[num_groups];

    // Build a mask of GROUP_SIZE 1's
    for (int i = 0; i < GROUP_SIZE; i++) {
        mask = mask | (1 << i);
    }

    // Separate all 64 bits of `num` into groups of GROUP_SIZE bits
    for (int i = num_groups; i > 0; i--) {
        vals[i - 1] = num & mask;
        num = num >> GROUP_SIZE;
    }

    uint64_t result = 0;

    // For each group
    for (int i = 0; i < num_groups; i++) {
        // Rotate group by ROT_AMT in ROT_DIR ("right" in this case)
        for (int j = 0; j < ROT_AMT; j++) {
            // Rotate 1 bit at a time
            // ________
```

(Question 4 continued...)

```
            // /       \
            // |  0110  1 = 01101
            //  \  \\\\
            //   \  \\\\
            //    \  \\\\
            //     1  0110 = 10110
            unsigned bit_0 = get_bit(vals[i], 0);
            vals[i] = vals[i] >> 1;
            set_bit(&(vals[i]), GROUP_SIZE - 1, bit_0);
        }

        // Set the on/off bits
        set_bit(&(vals[i]), ON_BIT, 1);
        set_bit(&(vals[i]), OFF_BIT, 0);

        // Insert group into result
        // result =                            01010
        // result << GROUP_SIZE =              01010 00000
        // (result << GROUP_SIZE) | vals[i] = 01010 00000
        //                                 OR      10110
        //                                    -----------
        //                                    01010 10110
        result = result << GROUP_SIZE;
        result = result | vals[i];
    }
    return result;
}
```

Question 5 ***Split*** **(0 points)**

Write a function `split` that takes a singly-linked list of words (strings of lower cased characters terminated appropriately) and splits it into two new singly-linked lists based on whether the **first** letter of the word is a vowel (a, e, i, o, u) or consonant (everything else). **All of the initial storage (nodes and strings) must be freed; you must copy strings to new locations in memory. Also, the output list should keep the same relative ordering of the input strings.** You must use `CS61C_malloc()` and `CS61C_free()` instead of `malloc()` and `free()`.

As an example, consider the following list (Using Python-style list representation):

`words = ["zebra", "ant", "walrus", "bat", "emu"]`

If you called `split(words, &consonants, &vowels)`, it would set `consonants` to `["ant", "walrus", "bat"]`, and `vowels` to `["zebra", "emu]` (with the strings stored in new locations in memory). Additionally, all of the space (nodes and strings) originally allocated for `words` will be freed.

You may find the following functions useful: `strcpy` and `strlen`:

- `char* strcpy(char* destination, const char* source);`
- `size_t strlen(const char *str);`

Download the included files: (shown below)

You may edit `main.c` to test your code. Your code should be able to run without modifications in `split.h`, or the `split` signature. You will only be submitting `split.c` and only the code in that file will be graded.

```c
/* split.c */
typedef struct node {
    char *data;
    struct node *next;
} Node;

void split(Node *words, Node ** consonants, Node ** vowels);
```

```c
/* split.h */
#include "split.h"
#include <string.h>

void *CS61C_malloc(size_t size);
void CS61C_free(void *ptr);

/*
For reference, this is the Node struct defined in split.h:
typedef struct node {
    char *data;
    struct node *next;
} Node;
*/
void split(Node *words, Node **consonants, Node **vowels) {
        //YOUR CODE HERE
```

```
    return;
}
```

```
/* main.c */
#include "split.h"
#include <stdlib.h>

void *CS61C_malloc(size_t size) {
    return malloc(size);
}

void CS61C_free(void *ptr) {
    free(ptr);
}

int main(int argc, char *argv[]) {
    return 0;
}
```

Solution:

Solution Walkthrough: here

```
/* split.c */
#include "split.h"
#include <string.h>

void *CS61C_malloc(size_t size);
void CS61C_free(void *ptr);

/*
For reference, this is the Node struct defined in split.h:
typedef struct node {
    char *data;
    struct node *next;
} Node;
*/

void split(Node *words, Node **consonants, Node **vowels) {
    Node *consonants_last = NULL;
    Node *vowels_last = NULL;

    // While we haven't reached the list's NULL terminator
    while (words != NULL) {
        // Allocate memory for the node and its data, and copy the data string
        Node *item = (Node *) CS61C_malloc(sizeof(Node));
        item->data =
            (char *) CS61C_malloc(sizeof(char) * (strlen(words->data) + 1));
        item->next = NULL; // Make sure there is no next item (for now)
```

```
        strcpy(item->data, words->data);

        // Check which list the item belongs in
        char first = words->data[0];
        if (first == 'a' || first == 'e' || first == 'i'
              || first == 'o' || first == 'u') {
            // If this is the first item in `vowels`,
            // set it as the head of the `vowels` list
            if (vowels_last == NULL) {
                *vowels = item;
                // Otherwise, append this item to the end of the list
            } else {
                vowels_last->next = item;
            }
            // Save this item so the next item can be appended after it
            vowels_last = item;
        } else {
            // If this is the first item in `consonants`,
            // set it as the head of the `consonants` list
            if (consonants_last == NULL) {
                *consonants = item;
            // Otherwise, append this item to the end of the list
            } else {
                consonants_last->next = item;
            }
            // Save this item so the next item can be appended after it
            consonants_last = item;
        }

        // Free the current node in the words list, and move to the next node
        Node *temp = words->next;
        CS61C_free(words->data);
        CS61C_free(words);
        words = temp;
    }
    return;
}
```