CS 61C Summer 2023

Charles, Jero, Rosalie Midterm

Solutions last updated: Saturday, March 2, 2024

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

PRINT your student ID: _____

You have 110 minutes. There are 7 questions of varying credit (100 points total).

Question:	1	2	3	4	5	6	7	Total
Points:	15	24	15	25	12	8	1	100

For questions with **circular bubbles**, you may select only one choice.

- ○ Unselected option (completely unfilled)
- ● Only one selected option (completely filled)

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

- ☐ You can select
- ■ multiple squares
- ■ (completely filled)

Anything you write that you ~~cross out~~ will not be graded. Anything you write outside the answer boxes will not be graded. If you write multiple answers or your answer is ambiguous, we will grade the worst interpretation. For coding questions, you may write at most one statement and you may not use more blanks than provided.

If an answer requires hex input, make sure you only use capitalized letters! For example, `0xDEADBEEF` instead of `0xdeadbeef`. Please include hex (`0x`) or binary (`0b`) prefixes in your answers unless otherwise specified. For all other bases, do not add any prefixes or suffixes.

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.

SIGN your name: _____

Q1 *Potpourri* **(15 points)**

Q1.1 (1 point) An n-bit number in bias notation and an n-bit two's complement number always represent the same range of numbers.

○ True ● False

Solution: Suppose the bias is 0, then the range of bias notation is the same as the range of unsigned.

Q1.2 (1 point) An n-bit two's complement number can represent exactly twice the number of unique values as an n-bit unsigned number.

○ True ● False

Solution: Two's complement represents the same number of unique values as unsigned.

Q1.3 (1 point) For any number representation system, you must have at least $2n + 1$ bits to represent the value 2^{2n}.

○ True ● False

Solution: Using bias notation, you can represent this number with 1 bit and a bias of 2^{2n}.

Q1.4 (1.5 points) Convert `0xA7` to decimal, assuming the data was stored as a two's complement one-byte integer.

Solution: -89
Grading: 0.5 points for the correct sign, 1 point for the correct magnitude (final answer was either 89 or 167). -167 was not awarded partial credit since it is the result of two errors.

Q1.5 (1 point) The output of the compiler never contains pseudoinstructions.

○ True ● False

Q1.6 (1 point) The loader produces an executable.

○ True ● False

For Q1.7 and Q1.8, consider the following code snippet and assume that `int`s are 4 bytes.

```
int x = 257;
int y = strlen((char *) &x);
```

Q1.7 (1.5 points) In a little-endian system, what will `y` contain?

Solution: 2

`x` contains the four bytes `0x00 00 01 01`. Thus on a little-endian machine, the bytes will be stored with the nonzero bytes at lower addresses. When `strlen` interprets this as a string, it will count length until the first null byte - in this case, it will count both `0x01` bytes and report a length of 2.

Grading: All-or-nothing, except partial credit was given for interpreting it as a big-endian system.

Q1.8 (1.5 points) In a big-endian system, what will `y` contain?

Solution: 0

See Q1.7 for the representation of `x`. On a big-endian machine, the bytes are stored with the *zero* bytes at lower addresses. Thus when `strlen` is called on `x`, it will immediately see a null byte and report a length of 0.

Grading: All-or-nothing, except partial credit was given for interpreting it as a little-endian system.

Q1.9 (2 points) Write a Boolean expression that simplifies the Boolean expression below. You may use at most 3 Boolean operations. $\sim$ (NOT), `|` (OR), `&` (AND) each count as one operation. We will assume standard C operator precedence, so use parentheses when uncertain.

`~(A|~C)|((~A&B)|B)`

Solution: `(~A&C)|B`

$$
\begin{aligned}
\texttt{~(A|~C)|((~A\&B)|B)} &= \texttt{(~A\&~~C)|((~A \& B)|B)} && \text{De Morgan's Law (OR)} \\
&= \texttt{(~A\&C)|((~A \& B)|B)} && \text{Idempotence (NOT)} \\
&= \texttt{(~A\&C)|B} && \text{Absorption (AND)}
\end{aligned}
$$

Grading: No partial credit was awarded for an expression that is not equivalent to the original expression. If the answer used more than 3 operators, the score was calculated as $\frac{4}{\texttt{num\_ops}+1} \cdot 2$.

Q1.10 (2 points) Translate the following RISC-V instruction to its hexadecimal counterpart.

```
slli a0 t2 4
```

Solution: `0x00439513`

Grading: Partial credit was awarded for having the correct funct7/immediate, rs1, funct3, rd, and opcode. funct7 and immediate are combined because it may be possible to have the correct funct7 or immediate without demonstrating understanding of what this question was designed to test.

Q1.11 (1.5 points) Represent 1.5×2^{-511} in hex using a binary floating point representation, which follows IEEE-754 standard conventions, but has 10 exponent bits (and a standard bias of -511) and 21 mantissa bits.

Solution: `0x00180000`

Looking at the number, it is equal to $1.1_2 \times 2^{-511}$. Since we can only represent exponents from -510 to 511 with a normal floating point number, this means our number must be represented as a denormalized number, with a fixed exponent of 2^{-510}. Rewriting our number to use this new exponent gives $0.11_2 \times 2^{-510}$. Thus the floating point representation is:

```
  sign  exponent   mantissa
  0     0000000000 11000000000000000000000
  0000 0000 0001 1000 0000 0000 0000 0000
0x0    0    1    8    0    0    0    0
```

Grading: Partial credit was awarded for having the correct sign bit, having the correct exponent bits, and having the correct mantissa.

Q2 *#include "library.c"* **(24 points)**

```
 1 #define MAX_BORROWS 25
 2
 3 typedef struct {
 4   char* book_name;
 5   bool borrowed;
 6 } Book;
 7
 8 typedef struct {
 9   char* user_id;
10   Book* borrowed_books[MAX_BORROWS];
11 } User;
12
13 typedef struct {
14   User* users;
15   int users_len;
16   Book* books;
17   int books_len;
18 } Library;
```

The city of Eddy B.C wants to build a new library! The `init_users` function receives the following input:

- `Library* lib`: A pointer to an uninitialized `Library` struct. You may assume that memory has already been correctly allocated on the heap for the `Library` struct.
- `char** user_ids`: An array of well-formatted strings of nonzero length except the last element. The last element is `NULL`. You may assume that all strings are allocated on the stack.

The function should make sure the following properties are held:

- `users_len` should be set to the number of strings in `user_ids`.
- Each `User` in `users` should be initialized as follows:
  - The `user_id` of the `i`th `User` in `users` should be set to the `i`th string in `user_ids`.
  - `borrowed_books` should be an array of `NULL`s to indicate that no `Book` has been borrowed.
- Every `User` and its contents must persist through function calls.

Useful C function prototypes:

```
void* malloc(size_t size);
void free(void *ptr);
void* calloc(size_t num_elements, size_t size);
void* realloc(void *ptr, size_t size);

size_t strlen(char* s);
char* strcpy(char* dest, char* src);
```

```
// memset sets the first num bytes of the block of memory pointed to by ptr
// to the specified value (interpreted as an unsigned char).
void* memset(void* ptr, int value, size_t num);
```

(15 points) Fill in `init_users` so that it matches the described behavior. Assume that all necessary C libraries are included.

```
1 void init_users(Library* lib, char** user_ids) {
2   int i = 0;

3   while (_____Q2.1_____) {

4     lib_____Q2.2_____ = _____Q2.3_____;

5     User* cur_user = _____Q2.4_____;

6     cur_user_____Q2.5_____ = _____Q2.6_____;

7     strcpy(cur_user_____Q2.7_____, _____Q2.8_____);

8     memset(cur_user_____Q2.9_____, _____Q2.10_____,
                    MAX_BORROWS * _____Q2.11_____);
9     i++;
10  }

11  lib_____Q2.12_____ = i - 1;
12 }
```

Solution:

```
1 void init_users(Library* lib, char** user_ids) {
2   int i = 0;
3   while (user_ids[i] != NULL) {
4     lib->users = realloc(lib->users, sizeof(User) * (i + 1));
5     User* cur_user = &lib->users[i];
6     cur_user->user_id = malloc((strlen(user_ids[i]) + 1) * sizeof(char));
7     strcpy(cur_user->user_id, user_ids[i]);
8     memset(cur_user->borrowed_books, 0,
                    MAX_BORROWS * sizeof(Book *));
```

```
 9     i++;
10   }
11   lib->users_len = i - 1;
12 }
```

Line 11 should have been `i`, not `i - 1`. This was given as a clarification during the exam, and no grading adjustment has been made.

It was ambiguous whether or not that memory was allocated for the `Library` struct pointer's members, such as `users`. Our solution relies on `users` being a pointer returned by `malloc`. As a result, we've awarded full credit for Q2.2, Q2.3, and Q2.4 to everyone.

The `remove_users` function takes in one argument:

- `Library* lib`: A pointer to a `Library` struct where the contents have been allocated on the heap.

The `remove_users` function should free every `User` and any additional memory that might have been allocated for the `User`.

```
1 void remove_users(Library* lib) {
2   for (int i = 0; i < lib->users_len; i++) {
3     free(lib->users[i]);
4   }
5 }
```

Q2.13 (3 points) Is the `remove_users` function implemented correctly? If "Yes", no justification is required. If "No", please explain in two sentences or fewer.

○ Yes ● No

Solution: There are multiple solutions that were accepted. One of the possible answers is pointing out that `lib->users` is a `User *`, and `lib->users[i]` cannot be free'd since it was not returned by `malloc`. The correct way of freeing this memory would be `free(lib->users)`.

For each of the following symbols, choose which section of memory it would live in.

Q2.14 (1 point) `MAX_BORROWS`

○ Stack ○ Heap ○ Data/Static ● Code

Solution: `#define` is a preprocessor directive, therefore it is replaced at compile time and part of the code itself.

Q2.15 (1 point) `init_users`

○ Stack ○ Heap ○ Data/Static ● Code

Solution: `init_users` is a function, which is part of the code segment.

Q2.16 (1 point) `lib`, the parameter for `init_users`

● Stack ○ Heap ○ Data/Static ○ Code

Solution: C parameters are stored on the stack.

Q3 *0x5f3759df* (15 points)

Thanton remembered from class that right-shifting a binary number is the same as dividing it by two (rounding down). They did get a little too eager, though, and have tried applying the concept to arbitrary numbers including floating point numbers! They wrote a function, `mystery`, to logically right shift the binary representation of the input value by 1 and return the result as a floating point value.

For this question, assume that we're working with standard IEEE-754 single-precision floating point numbers.

For Q3.1 to Q3.4, what would each of the following function calls return?

Q3.1 (2 points) `mystery(0)`

Solution: +0

The binary representation of 0 is `0b00000000000000000000000000000000`. Right-shifting this doesn't change it, so `mystery` will return (positive) zero as well.

Grading: All-or-nothing. Due to the ambiguity of our clarification, we gave almost full credit for "(+/-) 0", though it is technically incorrect since there are two 0's in IEEE-754.

Q3.2 (2 points) `mystery(-0)`

Solution: 2

Negative zero is just like zero, but has its sign bit set:

`0b1 00000000 00000000000000000000000`

After shifting (logical shift means no sign-extension!), this gives:

`0b0 10000000 00000000000000000000000`

Splitting this up, this gives us a positive normal number with an exponent of 128 pre-bias (1 post-bias) and 0 mantissa.

$1.0_2 \times 2^1 = 2$

Grading: All-or-nothing.

Q3.3 (2 points) `mystery(`∞`)`

Solution: 1.5

Positive infinity has representation

`0b0 11111111 00000000000000000000000`

and thus is

`0b0 01111111 10000000000000000000000`

after the shift. This is a positive number with an exponent of 127 pre-bias (0 post-bias).

$1.1_2 \times 2^0 = 1.5$

Grading: All-or-nothing.

Q3.4 (2 points) `mystery(`$-\infty$`)`

Solution: NaN
Negative infinity has representation like positive infinity, but negative:
`0b1 11111111 00000000000000000000000`
After the shift, this is
`0b0 11111111 10000000000000000000000`
With a maximal exponent and a nonzero mantissa, this value is a NaN.
Grading: All-or-nothing.

For Q3.5 and Q3.6, "normal numbers" are numbers that are not denormalized, zero, NaN, or infinity.

Q3.5 (3.5 points) Suppose we run `mystery` on every possible NaN value. What are all of the possible return values? Select all that apply.

- ☐ Denormalized numbers
- ■ NaN
- ■ Normal numbers
- ☐ Zero
- ☐ Infinity
- ☐ None of the above

Solution:
Any NaN value has all of the exponent bits set, and the sign bit may or may not be set. If the sign bit is set, then after the shift we will obtain `0b0 11111111 1xxx..` where the `xxx...` represents any arbitrary bits. This is a NaN value. If the sign bit is not set, we get `0b0 01111111 1xxx...` which is normal.
Grading: Each checkbox was graded as it's own true/false question, and selecting "None of the above" was treated as not selecting any of the other choices.

Q3.6 (3.5 points) Suppose we run `mystery` on every possible denormalized value. What are all of the possible return values? Select all that apply.

- ■ Denormalized numbers
- ☐ NaN
- ■ Normal numbers
- ■ Zero
- ☐ Infinity
- ☐ None of the above

Solution:
A denormalized value has all of its exponent bits not set (0), and again the sign bit may or may not be set. If the sign bit is set, we will get `0b0 10000000 0xxx...` where the `xxx...` represents any arbitrary bits. This is a normal number. If the sign bit is not set, we get `0b0 00000000 0xxx...`. If the arbitrary bits are nonzero, this is another denormalized number; if they are zero, then every bit is zero and we have zero.
Grading: Each checkbox was graded as it's own true/false question, and selecting "None of the above" was treated as not selecting any of the other choices.

Q4 *RISC-V, RISC-XVI, RISC-VIII* **(25 points)**

A *wondrous* sequence of positive integers is defined as follows: If n is even, then the next number is $\frac{n}{2}$. Otherwise, the next number is $3n + 1$.

Q4.1 (2 points) We want to create a pseudoinstruction to check whether a number is odd or not. This instruction, written `is_odd rd rs1`, will put the value `1` in `rd` if the value in `rs1` is odd, and the value `0` otherwise. What is the RISC-V instruction that `is_odd rd rs1` would translate to? You may only use one instruction, and you may not use any pseudoinstructions.

Note: Your solution may include `rd` and `rs1`.

> **Solution:** `andi rd rs1 1`, or equivalent. The key idea is that the last bit of a number is sufficient to tell whether or not it is odd — an odd number in binary ends in 1, whereas an even one will end in 0. Therefore, this mask, which zeroes out all but the LSB, will perform our desired operation.

Using `is_odd`, write a function (that follows calling convention) to compute the next wondrous number and return it in `a0`, given the current wondrous number as input in `a0`. Your code may not use `mul` or any `t` registers.

```
1  next_number:
2      _____
           Q4.2
3      _____
           Q4.3
4      is_odd _____
                Q4.4
5      beq s0 x0 else
6      _____
           Q4.5
7      _____
           Q4.6
8      _____
           Q4.7
9      j exit
10 else:
11     _____
           Q4.8
12 exit:
13     _____
           Q4.9
14     _____
           Q4.10
```

```
15      jr ra
```

Solution:

There are many possible solutions, one such solution is included below:

```
 1 next_number:
 2     addi sp sp -4
 3     sw s0 0(sp)
 4     is_odd s0 a0
 5     beq s0 x0 else
 6     slli s0 a0 1
 7     add so s0 a0
 8     addi a0 s0 1
 9     j exit
10 else:
11     srai a0 a0 1
12 exit:
13     lw s0 0(sp)
14     addi sp sp 4
15     jr ra
```

Grading: Credit was given for all equivalent answers, with points deducted for using `t` registers, `mul`, or breaking calling convention.

Using `next_number`, write a function (that follows calling convention) to count the number of steps until a wondrous sequence reaches the number 1 and return the number of steps in `a0`, given a starting number as input in `a0`.

You may assume that `a0` will be a positive integer. You may not use any additional `s` registers beyond those provided in the skeleton code.

```
1 num_steps:
      # Prologue
      # Omitted

2     addi s0 x0 _____________________________
                          Q4.11

3 loop_start:

4     addi t0 x0 _____________________________
                          Q4.12

5     _______________________________________
                        Q4.13
```

```
 6     _____________________________________________
                          Q4.14

 7     _____________________________________________
                          Q4.15
 8     j loop_start

 9 loop_end:

10     _____________________________________________
                          Q4.16
       # Epilogue
       # Omitted
11     jr ra
```

Solution:

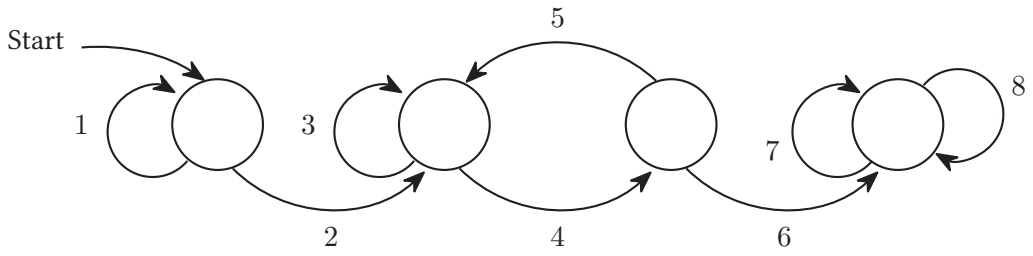
```
 1 num_steps:
       # Prologue
       # Omitted
 2     addi s0 x0 0
 3 loop_start:
 4     addi t0 x0 1
 5     beq a0 t0 loop_end
 6     jal ra next_number
 7     addi s0 s0 1
 8     j loop_start
 9 loop_end:
10     add a0 s0 x0
       # Epilogue
       # Omitted
11     jr ra
```

Grading: Credit was given for all equivalent answers, with points deducted for using `s` registers or breaking calling convention.

Q5 *Fully Secure Machine* **(12 points)**

A garage door opens if it ever sees the password 011 in a transmission. More formally, this FSM takes a bitstring consisting of 0's and 1's as its input, and continually outputs 0's until it sees the substring 011, after which it outputs 1's continuously. Example executions of this FSM are below:

Input:	010101001100001010101	Input:	00000000001000000010
Output:	00000000111111111111	Output:	00000000000000000000



For each of the numbered arrows, mark the correct FSM state transition.

Q5.1 (1.5 points) Arrow 1

○ 0/0 ○ 0/1 ● 1/0 ○ 1/1

Q5.2 (1.5 points) Arrow 2

● 0/0 ○ 0/1 ○ 1/0 ○ 1/1

Q5.3 (1.5 points) Arrow 3

● 0/0 ○ 0/1 ○ 1/0 ○ 1/1

Q5.4 (1.5 points) Arrow 4

○ 0/0 ○ 0/1 ● 1/0 ○ 1/1

Q5.5 (1.5 points) Arrow 5

● 0/0 ○ 0/1 ○ 1/0 ○ 1/1

Q5.6 (1.5 points) Arrow 6

○ 0/0 ○ 0/1 ○ 1/0 ● 1/1

Q5.7 (1.5 points) Arrow 7

○ 0/0 ○ 0/1 ○ 1/0 ● 1/1

Q5.8 (1.5 points) Arrow 8

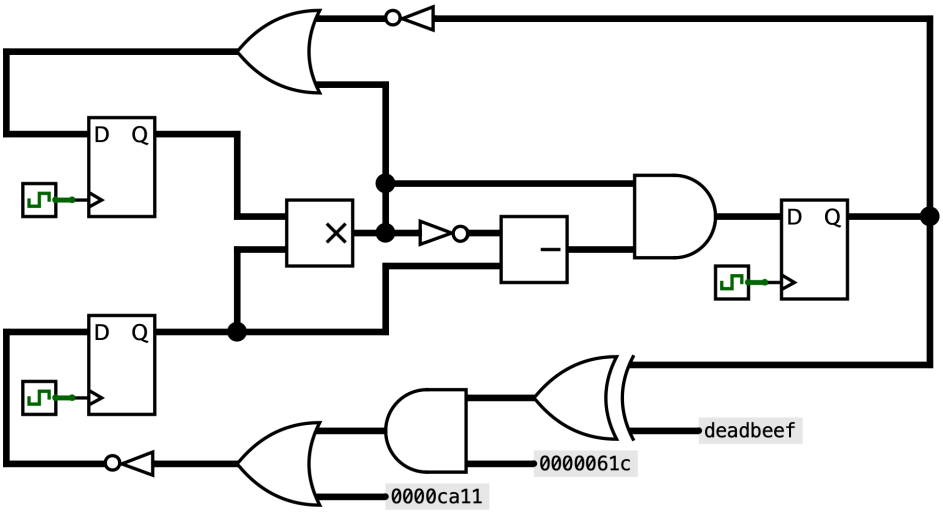
○ 0/0 ● 0/1 ○ 1/0 ○ 1/1

Solution: Note. For the last two questions, their answers can be swapped. Credit is given for both arrangements.

Grading: All-or-nothing, except for the last two questions, where partial was awarded for having one correct answer between the two.

Q6 *SDS* **(8 points)**

Consider the following circuit diagram and component delays:



$t_{\text{AND}} = 20\text{ps}$
$t_{\text{NOT}} = 5\text{ps}$
$t_{\text{OR}} = 20\text{ps}$
$t_{\text{XOR}} = 25\text{ps}$
$t_{\text{multiplier}} = 50\text{ps}$
$t_{\text{subtract}} = 35\text{ps}$
$t_{\text{clk-q}} = 5\text{ps}$
$t_{\text{setup}} = 2\text{ps}$

Q6.1 (2 points) What is the smallest combinational delay of all paths in this circuit, in picoseconds?

Solution: 25ps
The shortest CL path is between the right register and the top left register, consisting of a NOT gate and an OR gate, for a total delay of 25ps.
Grading: All-or-nothing.

Q6.2 (2 points) What is the minimum allowable clock period for this circuit to function properly, in picoseconds?

Solution: 117ps
The longest path between any two registers is between the top left register and the register, consisting of a multiplier, a NOT gate, a subtractor, and an AND gate, for a total of 110ps. Additionally, we need to account for clk-to-q and setup, which gives us 117ps.
Grading: All-or-nothing.

Q6.3 (2 points) What is the maximum hold time the registers can have so that there are no hold time violations in the circuit above?

Solution: 30ps
The shortest CL path is 25ps (see Q6.1), and the maximum hold time is the shortest CL path + clk-to-q, which gives us 30ps.
Grading: All-or-nothing, except full credit was given for Q6.1 + 5 to avoid double jeopardy

Q6.4 (2 points) Suppose this circuit only deals with two's complement integers. Currently, the subtractor component has a delay of 35ps. What is the maximum delay an adder component can have such that we could replace the subtractor component with adders, NOT gates, and constants to achieve the same delay as the subtractor while maintaining the same behavior? You may assume that constants have no delay.

As a reminder, the subtract component does the following operation:
output = top input - bottom input

Solution: 35ps

Since we're dealing with two's complement numbers, subtracting by x is equivalent to adding $\sim x + 1$, where $\sim x$ flips all of the bits of x. As a result, we can chain together a NOT gate, an adder with a constant 1, and another adder (to add the output of the previous adder and the top input of the subtractor) to achieve the same behavior.

The intent of the question is for students to realize that the NOT gate and the first adder does not actually add any additional delay, since the multiplier/NOT gate combo of the top input of the subtractor takes more time than the NOT gate/adder combo for the bottom input. Therefore, the adder can have a delay of 35ps (the same as the existing subtractor) for the circuit to maintain the same timing behavior.

Grading: All-or-nothing, except 15ps was also awarded full credit due to ambiguity raised within this question (assuming that the subtractor should be treated as a black box, and replaced with a black box consisting of two adders, a NOT gate, and a constant).

Q7 *The Finish Line* **(1 points)**

Everyone will receive credit for this question, even if you leave it blank.

Q7.1 (1 point) Where are the RISC and IEEE-754 plaques in Soda Hall?

- ○ 1st Floor
- ○ 2nd Floor
- ● 3rd Floor
- ○ 4th Floor
- ○ 5th Floor
- ○ 6th Floor
- ○ 7th Floor

Q7.2 (0 points) Is there anything you want us to know?