CS 61C Spring 2023

Garcia, Yokota Midterm

Solutions last updated: Saturday, Mar 11, 2023

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

PRINT your student ID: _____

You have 110 minutes. There are 6 questions of varying credit (100 points total).

Question:	1	2	3	4	5	6	Total
Points:	24	24	14	16	10	12	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 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* **(24 points)**

Q1.1 (2 points) For some $n > 0$, n-bit sign-magnitude and n-bit two's complement can represent the same number of values.

○ (A) True ● (B) False

Solution: False. Sign-magnitude has two representations of 0, so it's able to represent one fewer value than two's complement.

Q1.2 (2 points) Offsets for jump instructions will always be resolved in the linker step.

○ (A) True ● (B) False

Solution: False. These are resolved in the assembler and linker.

Q1.3 (2 points) `gcc` executes all four steps of CALL.

○ (A) True ● (B) False

Solution: False. `gcc` is the C compiler.

Q1.4 (2 points) Using dynamic linking may result in a smaller executable size compared to static linking.

● (A) True ○ (B) False

Solution: True. Dynamic linking means you may not need to include the entire library in your executable.

Q1.5 (2 points) Calling convention dictates that you must save `ra` onto the stack in the prologue.

○ (A) True ● (B) False

Solution: False. You just need to make sure `ra` is unchanged by the end of the function.
One common way to do this is saving the value of `ra` on the stack in the prologue, and restoring the value on the stack back into `ra` in the epilogue.
However, other methods of preserving `ra` exist. For example, you could save `ra` on the stack before every instruction that changes the value of `ra` (e.g. function calls). Then you can restore the value on the stack back into `ra` after the instruction.

Q1.6 (3 points) Translate the following RISC-V instruction to its hexadecimal counterpart.

```
jal s3 588
```

Hint: 588 = 512 + 64 + 8 + 4

Solution: `0x24C009EF`

Q1.7 (3 points) Write a Boolean expression that evaluates to the truth table below. You may use at most 2 Boolean operations. $\sim$ (NOT), | (OR), & (AND) each count as one operation. We will assume standard C operator precedence, so use parentheses when uncertain.

W	Y	Z	Out
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Solution: `~Y | Z`
Other solutions may exist.

What is the output of this program on a **32-bit, little endian** system?
Reminder: the reference sheet has a list of common C format specifiers.

```
#include <stdint.h>
#include <stdio.h>

int main() {
    int8_t i = 0xA7;

    // %hhd is like %d except it interprets i as an 8-bit integer
    printf("Q1.8: %hhd\n", i);

    // %hhu is like %u except it interprets i as an 8-bit integer
    printf("Q1.9: %hhu\n", i);

    char* str = "hello!!";

    printf("Q1.10: %x\n", ((int8_t *) str)[1]);
    printf("Q1.11: %x\n", ((int32_t *) str)[1]);
    return 0;
}
```

Q1.8 (2 points)

Solution: -89
`0xA7` interpreted as a signed, two's complement, 8-bit integer is -89.

Q1.9 (2 points)

Solution: 167
`0xA7` interpreted as an unsigned 8-bit integer is 167.

Q1.10 (2 points)

Solution: `'e'` = `0x65`
If we treat `str` as an array of `int8_t`s, then each element is 1 byte. The first element (zero-indexing) is the character `'e'`. Using our ASCII chart, we can convert this to hexadecimal and get `0x65`.

Q1.11 (2 points)

Solution: `0x00212165` or `0x212165`

If we treat `str` as an array of `int32_t`s, then each element is 4 bytes. The first element (zero-indexing) consists of the four bytes `'o'`, `'!'`, `'!'`, and `0x00` (the null terminator at the end of the string). In ASCII, these are the bytes `0x65 0x21 0x21 0x00`, from lowest memory address to highest memory address.

Finally we have to combine these 4 bytes into one 4-byte `int32_t` value. Since the system is little-endian, `0x00` at the highest memory address is the most significant byte, and `0x65` at the lowest memory address is the least significant byte.

Q2 *I Can't C My Cheatsheet;* **(24 points)**

```
1  #define NUM_PAGES 8
2
3  typedef struct Page {
4      int num;
5      char* data;
6  } Page;
7
8  typedef struct Cheatsheet {
9      int student_id;
10     int total_length;
11     Page pages[NUM_PAGES];
12 } Cheatsheet;
```

The function `cheatsheet_init` receives as input the following:

- `Cheatsheet** ch`: The location in memory to store the pointer to the created `Cheatsheet`.
- `int student_id`: The student ID of the owner of the `Cheatsheet`.
- `char** contents`: An array of well-formatted strings of nonzero length. You may assume that this array contains exactly `NUM_PAGES` strings, and that all strings are allocated on the **stack**.

It should create a well-formed `Cheatsheet` with the following properties, and save a pointer to that `Cheatsheet` at the address `ch` points to:

- `student_id` should be set to the `student_id` argument.
- `total_length` should be set to the sum of the `strlen` of all strings in `contents`.
- Each `Page` in `pages` should be set as follows:
  - The `num` of `pages[i]` is set to `i`.
  - The `data` of `pages[i]` is set to the `i`th string of `contents`.
- The resulting `cheatsheet` (and all data in its pages) must persist through function calls, even if the strings in `contents` are deallocated.

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);
```

(15 points) Fill in `cheatsheet_init` so that it matches the described behavior.

```
1  void cheatsheet_init(Cheatsheet** ch, int student_id, char** contents) {

2      Cheatsheet* sheet = ____________________________________;
                                                        Q2.1

3      sheet_____________ = student_id;
              Q2.2
4      for (int i = 0; i < NUM_PAGES; i++) {

5          Page* page = ____________________________________;
                                                    Q2.3

6          page_____________ = i;
                  Q2.4

7          page_____________ = ____________________________;
                  Q2.5                          Q2.6

8          strcpy(page_____________________, contents[i]);
                                  Q2.7

9          sheet_____________________ += strlen(contents[i]);
                          Q2.8
10     }

11     ____________________________________________________;
                                  Q2.9
12 }
```

Solution:

Q2.1: `calloc(1, sizeof(Cheatsheet)`

Note that we need to `calloc` in this case in order to set `total_length` equal to 0.

Q2.2: `->student_id`

Q2.3: `&sheet->pages[i]`

When we allocate memory on the heap for a `Cheatsheet`, we allocate memory for a `Page` array of size `NUM_PAGES`. Therefore, we already allocated memory for each `Page`. In order to get the correct `Page`, we need to index into the correct `Page` in our `Cheatsheet` (`sheet->pages[i]`). To get the pointer to this `Page`, we will use the `&` to get a pointer to this `Page` (`&sheet->pages[i]`)

Q2.4: `->num`

Q2.5: `->data`

Q2.6: `malloc(sizeof(char) * (strlen(contents[i]) + 1))`

Note that we allocated memory for a `char` pointer but we now need to actually allocate memory for the string itself. Also, `strlen` doesn't consider the null-terminator, so we need to add 1.

Q2.7: `->data`
Q2.8: `->total_length`
Q2.9: `*ch = sheet`

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

Q2.10 (2 points) `NUM_PAGES`

○ (A) Stack ○ (B) Heap ● (C) Code ○ (D) Data/Static

Q2.11 (2 points) `sheet`

● (A) Stack ○ (B) Heap ○ (C) Code ○ (D) Data/Static

Q2.12 (2 points) `*sheet`

○ (A) Stack ● (B) Heap ○ (C) Code ○ (D) Data/Static

The `update_contents` function should update the contents of a `Page` if `new_data` has a length that is less than or equal to `MAX_STR_LEN`. `update_contents` does not have to update the `total_length` of a `Cheatsheet`.

Q2.13 (3 points) Is the following implementation of `update_data` correct (follows the described behavior) and memory efficient?

```
1  int MAX_STR_LEN = 100;
2
3  // You may assume that new_data is stored on the heap
4  // and page is well-formed
5  void update_data(Page* page, char* new_data) {
6      if (strlen(new_data) > MAX_STR_LEN) {
7          return;
8      }
9      page->data = new_data;
10 }
```

○ (A) Yes ● (B) No

If you selected "No", provide a **brief** explanation. If you selected "Yes", leave this box blank. We will only grade the first 15 words of your answer.

Solution: Memory leak. We need to `free` the old `data` before reassigning `page->data` to `new_data`.

Note: There were a couple of alternative answers

1. Mentioning that line 6 is incorrect because `strlen` doesn't include the null terminator. There was ambinguity in what "length" meant (Does length include the null terminator or not?)

2. (Out of scope) Mentioning how comparing `size_t` to an `int` could lead to a security vulnerability where the comparison on line 6 would fail, specifically if how `strlen(new_data)` returned a number where, if represented in binary, had a leading 1. This solution is out of scope for this class, and was only deemed correct if the student's answer got every aspect of it correct.

Q3 *A Bit of Floating Point* (14 points)

For this question, assume that we are working with a binary floating point representation, which follows IEEE-754 standard conventions, but has 5 exponent bits (and a standard bias of -15) and 10 mantissa bits.

Q3.1 (2 points) What is the value of the floating point number `0x7F00`?

Solution: NaN

`0b0111 1111 0000 0000`

The exponent bits are all 1s, and the mantissa bits are not all zeroes. This represents a NaN.

Q3.2 (2 points) Consider the floating point number -2.125. What is the largest (closest to $+\infty$) possible value we can represent by modifying a single bit of the floating point representation of this number? Write your answer as a decimal number (e.g. 10.5).

Solution: 2.125

To get the closest to $+\infty$, we can flip the sign bit. Changing any mantissa or exponent bit leaves the number negative.

Q3.3 (5 points) Consider the floating point number 7.625. What is the largest (closest to $+\infty$) possible value we can represent by modifying a single bit of the floating point representation of this number? Write the binary representation of each component of your answer.

Solution: Sign bit: `0b`0

Exponent bits: `0b`11001

Mantissa bits: `0b`1110100000

$7.625 = 61/8 = 61 \times 2^{-3} = \texttt{0b}111101 \times 2^{-3} = (\texttt{0b}1.11101 \times 2^5) \times 2^{-3} = \texttt{0b}1.11101 \times 2^2$

Sign bit: 0 (positive). We know flipping the sign bit will just make the number negative, which isn't helpful.

Mantissa bits: `0b11101 00000`. To increase the number, the most-significant bit we could flip is the 0 to a 1, which would produce $\texttt{0b}1.11111 \times 2^2$. The difference between this number and the original number is $\texttt{0b}0.00010 \times 2^2 = \texttt{0b}0.01 = 1/4$. Flipping any of the less-significant 0s would increase the number by even less.

Exponent bits: $2 - (-15) = 17$, which in unsigned 5-bit binary is `0b`10001. We can increase the number by flipping the most-significant 0 to a 1, which would produce `0b`11001.

The overall solution is to leave the sign and mantissa bits unchanged, and flipping the most-significant zero bit in the exponent.

Q3.4 (5 points) How many non-zero numbers `x` are there in this floating point system where `x` and `2x` differ by exactly 1 bit?

Write your answer as a sum or difference of unique powers of 2 (e.g. $2^3 - 2^2 + 2^1$).

Solution: $2^{15} - 2^{12}$

Note that to double a floating-point number, we have to increase the exponent by 1.

When we increase the exponent by 1, what could happen? If the least-significant bit of the exponent (as represented in bias notation) is 0, then the 0 gets flipped to a 1. For example, `0b11010 + 0b1 = 0b11011`.

If the least-significant bit of the exponent is 1, then the 1 flips to a 0, a 1 carries over into the next place, and other bits must change. For example, `0b11011 + 0b1 = 0b11100`, which changed 3 bits.

In summary, we need to figure out how many floating-point numbers have a least-significant exponent bit of 0. This is half of the floating-point numbers (if you just wrote out all the bit representations, half of them would have a 0 in the exponent LSB). There are 2^{16} floating-point numbers, and 2^{15} of them have a 0 in the exponent LSB.

The last thing we need to do is remove the infinities, NaNs, and denorms, because adding 1 to the exponent does not double these numbers. (In the case of denorms, changing the exponent also introduces the implicit 1, which changes the number in other ways than just a simple doubling.)

Denorms: Exponent is all 0s. The 11 sign/mantissa bits could be anything, so there are 2^{11} denorms we have to remove from our final total.

Infinities and NaNs: Exponent is all 1s. Just like the denorms, there are 2^{11} more values we have to remove.

In total, we throw out $2 \times 2^{11} = 2^{12}$ values from our original total of 2^{15}.

The original idea for this question came from an ex-TA who went on to teach other classes, so you can't blame anyone on the current staff for it. It's a tricky question!

Q4 *The R in RISC-V* **(16 points)**

For each of the following instructions, provide a sequence of RISC-V instructions that computes the equivalent to the given instruction.

- You may **not** use the instruction itself.
- You may **not** use any pseudoinstructions.
- You may **not** have unaligned memory accesses.
- You may assume that `rs1` or `rs2` are not the same register as `rd`.
- You may use **at most** the number of lines we provide.
- You may **only** have one instruction per line.

Each subpart is independent from one another. We've provided `sub` as an example.

`sub rd rs1 rs2`

```
sub_alternative:
    xori rd rs2 -1
    addi rd rd  1
    add rd rs1 rd
```

Q4.1 (5 points) `lbu rd imm(rs1)`

You can use `rs1`, `rd`, and `imm` in your answer. You may not modify any registers other than `rd`.

```
lbu_alternative:

    _____________________________________________

    _____________________________________________
```

Solution:

```
lb rd imm(rs1)
andi rd rd 0x000000FF
```

Recall that `lbu` reads 8 bits (1 byte) from memory, writes them to the lower 8 bits of `rd`, and doesn't sign-extend (leaves the upper 24 bits of `rd` as all zeros).

The closest instruction to `lbu` is `lb`. It fetches the same 8 bits from memory and writes them to the same lower 8 bits of `rd`, except it sign-extends the upper 24 bits of `rd`.

Therefore, our solution is to use `lb` to get the lower 8 bits of the register correctly filled with the value from memory, and then use bitwise AND to zero out the upper 24 bits.

Q4.2 (5 points) `bne rs1 rs2 label`

You can use `rs1`, `rs2`, and `label` in your answer. You may not modify any registers.

```
bne_alternative:

    _____________________________________________________

    _____________________________________________________
continue: # This is a label, but you do not have to use it.
```

Solution:

Solution 1: Use the `continue` label.

```
beq rs1 rs2 continue
jal x0 label
```

One particularly close instruction to `bne` (branches not equal) is `beq` (branch equal). `beq` takes the branch every time `bne` doesn't, and vice-versa.

We use `beq` to force the program to not jump (i.e. go to the `continue` label, the next instruction after the branch) if the register values are equal. Then, if the register values are not equal, we don't go to `continue`, and we instead move to Line 2. At this point, we do want to go to the label so we jump to `label`.

Note: `j label` is a pseudoinstruction, so it has to be expanded out. Partial credit was given if your answer was entirely correct but forgot to expand out the pseudoinstruction.

Solution 2: Use `blt` twice.

```
blt rs1 rs2 label
blt rs2 rs1 label
```

If `rs1` and `rs2` are not equal, one must be greater than the other. We check both cases and jump to `label` if one is greater than the other.

Q4.3 (6 points) `auipc rd imm`

You can use `rd` and `imm` in your answer. You may only modify `rd` and `t0`.

Note: the value of PC used for `auipc` computation should be the PC of the final instruction in your answer. For example, if your answer has four lines, you should add `imm` to the PC of the fourth line.

```
auipc_alternative:

    _____________________________________________________
temp_label: # This is a label, but you do not have to use it.

    _____________________________________________________

    _____________________________________________________

    _____________________________________________________
```

Solution:

```
jal rd temp_label
addi rd rd 8
lui t0 imm
add rd rd t0
```

This one is tricky. The first thing to remember is what `auipc` does. First, it takes a 20-bit `imm`, and creates an immediate with these 20 bits as the upper 20 bits and 0s as the lower 12 bits. Then, it adds this new immediate with the current PC.

First, we have to get the value of the current PC. Looking through the reference card, the only instructions that put the PC in a register are the jump instructions. Here, we use `jal` to get the address of the `jal` instruction, plus 4 (i.e. the address of Line 2 here) into register `rd`.

However, the question says that we should be adding to the PC of the final instruction in our answer. Since our answer uses all 4 lines, and we got the address of Line 2, we need to add 8 bytes = 2 instructions to the PC we got, to find the PC of the final instruction.

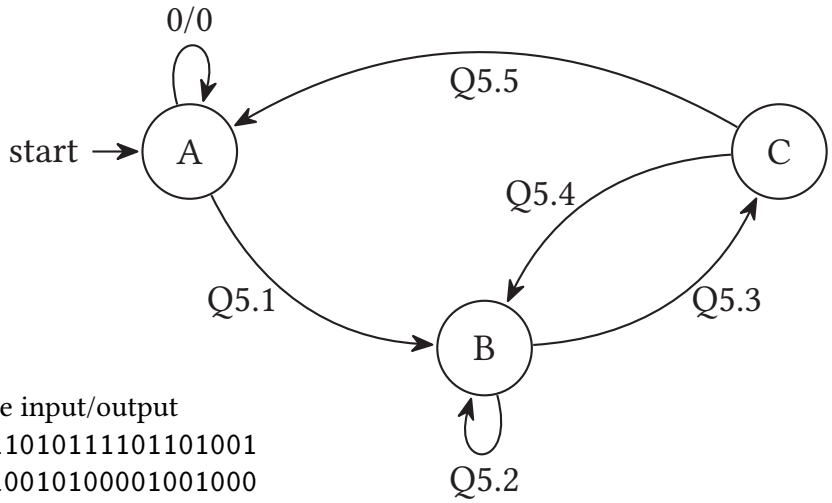
Now, we can take `imm` and build the 32-bit value we'll be adding to PC. Note that `imm` is 20 bits, so an `addi` instruction (with only a 12-bit immediate) cannot handle this number. We have to use `lui` to put these 20 bits into the top 20 bits of a register. We choose to use `t0` because that's the only other register we can modify, and `rd` is already holding our current PC. (If we `lui`'d into `rd`, we'd mess up the PC we found.)

Finally, we use `add` to add the PC and the immediate.

Other answers are possible here, e.g. putting the PC in `t0` and the immediate in `rd` before adding.

Q5 *FSM* **(10 points)**

The following finite state machine (FSM) should output 1 if the last 3 input bits are 101. Otherwise, it should output 0.



Example input/output

Input: 1011010111101101001

Output: 0010010100001001000

For each of the transitions above, fill in the appropriate input and output values such that the FSM behaves as described.

Q5.1 (2 points)

Solution: `1 / 0`

Q5.2 (2 points)

Solution: `1 / 0`

Q5.3 (2 points)

Solution: `0 / 0`

Q5.4 (2 points)

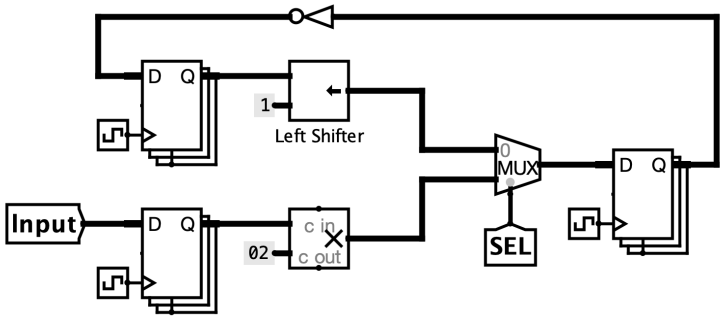
Solution: `1 / 1`

Q5.5 (2 points)

Solution: `0 / 0`

Q6 *SDS* **(12 points)**

Consider the following circuit diagram. `SEL` is a single bit control signal that updates instantaneously at the rising edge of every clock cycle and remains stable during any given clock cycle. You may assume that `Input` will not cause any hold time violations.



$$t_{\text{NOT}} = 8 \text{ ps}$$
$$t_{\text{mux}} = 25 \text{ ps}$$
$$t_{\text{multiplier}} = 1000 \text{ ps}$$
$$t_{\text{shifter}} = 2 \text{ ps}$$
$$t_{\text{clk-q}} = 30 \text{ ps}$$
$$t_{\text{setup}} = 20 \text{ ps}$$

The left shifter combinational logic block shifts the top input by the number of bits indicated by the bottom input. The shifter in the diagram shifts the output of the connected register left by 1 bit.

Q6.1 (3 points) What is the minimum clock period for the circuit above such that it will always result in well-defined behavior?

Solution: 1075 ps

The longest path goes through the multiplier, by far the slowest block in the circuit.

From the rising edge of the clock, we have to wait 30 ps for the signal to show up at the register output. Then, we have to wait 1000 ps for the signal to move through the multiplier, and another 25 ps for the signal to move through the mux. Finally, we have to reach the rightmost register 20 ps early (before the next rising edge) to account for the setup time.

In total, this is $30 + 1000 + 25 + 20 = 1075$ ps.

Q6.2 (3 points) What is the maximum hold time the registers can have so that there are no hold time violations in the circuit above? Reminder: you may assume that `Input` will not cause any hold time violations.

Solution: 25 ps

The shortest path between any two timed elements is actually the path from the SEL signal, which changes instantly at the rising edge of the clock, to the right register. This path has only delay 25 ps from the mux.

If you didn't see this path, the next-shortest path starts from the rightmost register and goes around, through the NOT gate, to the top-left register. This path has a delay of 30 ps (clk-to-q from the rightmost register) and 8 ps (from the NOT gate), for a total of 38 ps. Partial credit was given for this answer.

Eric wants to make one change to increase the circuit's frequency the most without changing the behavior of the circuit. He cannot change the delays of any component.

Q6.3 (3 points) **Briefly** describe a change Eric should make. We will only grade the first 15 words of your answer.

Solution: Replace multiplier with shifter. (Switch constant from 2 to 1.)

The slowest block by far here is the multiplier, so we should do something about that. In particular, we don't need this general-purpose multiplier that can multiply any two inputs together, if all we're ever doing is multiplying numbers by 2.

There are a few simpler ways to multiply a number by 2. You could add the number with itself, using an adder (but an adder isn't provided here). You could also left-shift the number by 1, using the shifter (which is provided, and is very fast).

Q6.4 (3 points) What is the new minimum clock period after implementing the change in Q6.3?

Solution: 77 ps

With the multiplier replaced by a shifter, the new longest path starts from either left-side register, goes through the shifter and the mux, and ends at the rightmost register.

From the rising edge, we wait 30 ps for the output to appear out of the left registers (clk-to-q). Then we wait another 2 ps for the signal to travel through the shifter, and another 25 ps for the multiplexer. Finally, we have to get to the rightmost register 20 ps before the next rising edge to account for setup time. In total, this is $30 + 2 + 25 + 20 = 77$ ps.