CS 61C Summer 2023

Charles, Jero, Rosalie Final

Solutions last updated: Saturday, March 2, 2024

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

PRINT your student ID: ____

You have 170 minutes. There are 9 questions of varying credit (100 points total).

Question:	1	2	3	4	5	6	7	8	9	Total
Points:	9	20	13	11	12	17	7	10	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: ____

This page intentionally left (mostly) blank.

The exam continues on the next page.

Q1 *Potpourri* (9 points)

Suppose we have the following hit times and hit rates for a system:

	Hit Time	Hit Rate
L1 Cache	10ns	40%
L2 Cache	100ns	60%
DRAM	550ns	100%

Q1.1 (2 points) On average, what would our memory access time be for this system?

Solution: $10 + 0.6 \cdot (100 + 0.4 \cdot 550) = 202\text{ns}$

Grading: This question was graded as all-or-nothing.

Q1.2 (2 points) Convert the following RISC-V instruction to hexadecimal:

`lhu t1 32(sp)`

Solution: `0x02015303`

Q1.3 (2 points) Suppose you can speed up 25% of your program by a factor of 5. What fraction of the unoptimized runtime will the optimized program take to run? Express your answer as a simplified fraction.

Solution: $\frac{1}{(1-0.25)+\frac{0.25}{5}} = \frac{1}{0.8}$x speedup, which means that the optimized runtime is $\frac{0.8}{1} = \frac{4}{5}$ of the unoptimized runtime.

Grading: This question was graded as all-or-nothing.

Q1.4 (1 point) True or False: The linker initializes the stack with program arguments.

○ True ● False

Q1.5 (1 point) True or False: During a thread context switch, the entries of the TLB get invalidated.

● True ○ False

Q1.6 (1 point) True or False: Creating more threads is guaranteed to increase the speed of your code.

○ True ● False

Q2 *It's a Jerover!* (20 points)

NASA is planning to launch a new rover to Mars to continue scientific research. They've decided to put a custom RISC-V processor in the rover (lovingly named Jerover), and this means you're in charge of writing its code! However, the thin atmosphere of Mars means that extra protection is needed for the data in memory. They've decided to use a Hamming code with even parity to protect your data. For this question, use the parity table shown below.

Output Bit	6	5	4	3	2	1	0
Stored Bit	d_3	d_2	d_1	p_2	d_0	p_1	p_0
p_0	✓		✓		✓		✓
p_1	✓	✓			✓	✓	
p_2	✓	✓	✓	✓			

Assume that bit 0 is the least significant bit.

Part 1: Implement `calculate_parity`, a RISC-V function that takes in a 4-byte input in `a0` and calculates the parity of its bits, returning in `a0` either 1 for odd parity or 0 for even parity. For example:

Input: `0b 0000 0000 0000 0000 0000 0010 1101 0111` Output: 1

There are seven 1 bits (an odd number), so this number has odd parity, so the return value is 1.

Input: `0b 0000 0000 0000 0000 0000 0010 1010 1101` Output: 0

There are six 1 bits (an even number), so this number has even parity, so the return value is 0.

`calculate_parity` must follow calling convention.

```
calculate_parity:
    li t1 0            # t1 holds current parity value
                       # in least significant bit
loop:
    xor t1 t1 a0       # add also acceptable
                       # adds the current least significant bit
                       # in a0 to the parity value in t1

    srli a0 a0 1       # shift to the next bit in a0
                       # it's important here that this is
                       # a logical shift, since using a
                       # arithmetic shift would possibly
                       # put us in an infinite loop
                       # if bit 31 is set

    bne a0 x0 loop     # loop if there are more than 1 bit left in a0
                       # 0 bits will never affect parity

    andi a0 t1 1       # we only want the one parity bit
                       # in bit 0 of t1
jr ra
```

Part 2: Implement `store_nibble_protected`, a RISC-V function that accepts four bits of data in `a0` and writes the seven encoded bits (as a byte, with 0 in the most significant bit) to the memory address in `a1` (and doesn't return anything). You may assume that `calculate_parity` has been implemented correctly and adheres to calling convention, but you may not assume any specific implementation of `calculate_parity`. You may also assume that the most significant 28 bits of the argument in `a0` are set to 0.

You do not have to follow the recommendations made in the comments.

```
store_nibble_protected:
    # prologue omitted
    mv s0 a0
    mv s1 a1

    srli s7 s0 1                # srai also acceptable
    slli s7 s7 1                # make space for next bit

    andi a0 s0 0b1110           # parity of d1, d2, and d3
    jal ra calculate_parity     # compute p2
    add s7 s7 a0                # xor and or also acceptable
    slli s7 s7 1                # make space for next bit

    andi t0 s0 1                # extract d0
    add s7 s7 t0                # xor and or also acceptable
    slli s7 s7 1                # make space for next bit

    andi a0 s0 0b1101           # parity of d0, d2, and d3
    jal ra calculate_parity     # compute p1
    add s7 s7 a0                # xor and or also acceptable
    slli s7 s7 1                # make space for next bit

    andi a0 s0 0b1011           # parity of d0, d1, and d3
    jal ra calculate_parity     # compute p0
    add s7 s7 a0                # xor and or also acceptable

    sb s7 0(s1)                 # s1 contains the memory address
                                # passed in as a1
    # epilogue omitted
    jr ra
```

Solution: There is an alternate solution where instead of masking off the bits from the saved argument in `s0`, the data bits are extracted from the working copy `s7`. This was awarded equal credit on the exam.

Q2.15 (3 points) List all registers that need to be saved in the prologue and restored in the epilogue in order for `store_nibble_protected` to follow calling convention. If there are none, write "None".

Solution: The required registers are `s0`, `s1`, `s7`, and `ra`. All of these registers are used in the question template and therefore must be saved and restored to follow calling convention. (While `ra` is not strictly a saved register, `store_nibble_protected` is calling functions and therefore needs to save `ra`.) Any additional saved registers used also need to be saved and restored.

Grading: $+0.75$ points was awarded for each of the four required registers. Each additional saved register that was used in the program but not included here or register that was not required to be saved incurred a -0.75 point penalty.

`sp` is a special case, as even though it is callee-saved, it is not usually saved and restored from the stack. Thus it was ignored for grading (no penalty for inclusion or exclusion).

Q3 *Long Jump* (13 points)

AJ is working on implementing jumps and branches into his RISC-V processor.

What values should the control logic output for `jal`?

Q3.1 (0.5 point) PCSel

○ PC+4 ● ALUOut ○ Doesn't matter

Q3.2 (0.5 point) RegWEn

○ 0 ● 1 ○ Doesn't matter

Q3.3 (0.5 point) BrUn

○ 0 ○ 1 ● Doesn't matter

Q3.4 (0.5 point) ASel

● PC ○ RegReadData1 ○ Doesn't matter

Q3.5 (0.5 point) BSel

○ RegReadData2 ● Immediate ○ Doesn't matter

Q3.6 (0.5 point) ALUSel

● add ○ sll ○ mul ○ Doesn't matter

○ slt ○ srl ○ bsel ○ Other

Q3.7 (0.5 point) MemRW

● Read ○ Write ○ Doesn't matter

Q3.8 (0.5 point) WBSel

● PC+4 ○ ALU ○ Mem ○ Doesn't matter

After implementing the processor, AJ spends some time writing code for it. However, because the project he is working on has a really large codebase, he encounters the branch and jump range limitations.

Q3.9 (2 points) What is the maximum number of bytes a `bge` instruction can branch by? Write your answer as a sum or difference of unique powers of 2 (e.g. $2^3 - 2^2 + 2^1$).

Solution: Branches have 12-bit immediates, along with an additional implicit 0. Since this immediate is interpreted as a two's complement number, this means that we can only represent a range from -2^{12} to $2^{12} - 2$ (since bit 0 is forced to be 0).

Thus the maximum number of bytes we can branch is 2^{12} (backwards).

Grading: The question was graded as all-or-nothing, except partial credit was awarded for accounting for only the forward branch.

Q3.10 (2 points) What is the maximum number of bytes a `jal` instruction can jump by? Write your answer as a sum or difference of unique powers of 2 (e.g. $2^3 - 2^2 + 2^1$).

Solution: Jumps have 20-bit immediates, along with an additional implicit 0. Since this immediate is interpreted as a two's complement number, this means that we can only represent a range from -2^{20} to $2^{20} - 2$ (since bit 0 is forced to be 0).

Thus the maximum number of bytes we can jump is 2^{20} (backwards).

Grading: The question was graded as all-or-nothing, except partial credit was awarded for accounting for only the forward branch.

AJ decides to deal with this problem by implementing new hardware and a new set of instructions, *long jump* and *long branch*. These instructions will allow offsets of up to 32 bits by storing the immediate in the 4 bytes immediately after the instruction within the IMEM, rather than within the instruction itself.

To accommodate this, the IMEM has been modified to add another output port that contains the four bytes stored at `addr + 4`, where `addr` is the input to the IMEM.

Q3.11 (2 points) What additional changes would we need to make to our datapath in order for us to implement both of these instructions (with as few changes as possible)? Select all that apply.

Assume that each of the options also include any relevant combinational logic and control logic additions or modifications.

■ Add a new input to the PCSel mux

□ Add a new input to the Regfile

□ Add a new output to the Regfile

■ Add a new input to the immediate generator

□ Add a new input to the AMux

□ Add a new input to the BMux

□ Add a new input to the ALU

□ Add a new operation to the ALU

□ Add a new input to DMEM

■ Add a new input to the WBMux

□ None of the above

Solution: We need to add a new input to PCSel because in the event that we have a long branch, and we do not branch, the next instruction we load should be at PC+8, not PC+4. We also need to add an input to the imm gen, which should be the value from the second output port of the IMEM. We need to add a new input to the WBMux since we want to write back PC+8, instead of PC+4, to rd on long jumps.

An alternate solution exists where we can add an additional input to the BMux (the 32-bit "instruction" read from memory), instead of adding an input to the immediate generator. The changes to PCSel mux and WBMux are still required.

After implementing *long jump* and *long branch*, AJ pipelines his CPU using the standard 5-stage pipeline included in the CS 61C Reference Card.

Q3.12 (1.5 points) Assuming the above changes were implemented, what hazards can be caused by a *long jump* instruction?

■ Control ■ Data □ Structural □ None

Solution: The hazards remain the same as a normal jump instruction. There can be a control hazard since it is a jump, and a data hazard since it writes back to rd (which may be needed in the next instruction executed).

Q3.13 (1.5 points) Assuming the above changes were implemented, what hazards can be caused by a *long branch* instruction?

■ Control □ Data □ Structural □ None

Solution: The hazards remain the same as a normal branch instruction, where there are only control hazards.

Q4 *One Bot's Trache is Another Bot's Cache* (11 points)

CoryBot comes from a parallel universe where computers (and thus caches) are based in ternary (base 3). SodaBot want to know how CoryBot's ternary caches (traches) work, and they need your help! Instead of using binary and having 8 bits in a byte, CoryBot's computer uses ternary, where a trit is either 0, 1, or 2, and a tryte is made up of 3 trits.

For convenience, a list of powers of 3 is given below:

$3^1 = 3$	$3^3 = 27$	$3^5 = 243$	$3^7 = 2,187$	$3^9 = 19,683$
$3^2 = 9$	$3^4 = 81$	$3^6 = 729$	$3^8 = 6,561$	$3^{10} = 59,049$

CoryBot has a tryte-addressable memory space with 10-trit memory addresses, and their CPU has a direct-mapped trache with 9 blocks that hold 27 trytes each.

Q4.1 (3 points) Calculate the TIO trits in this setup.

Solution: 5:2:3

Offset: 27 trytes requires $\log_3 27 = 3$ trits.

Index: 9 blocks requires $\log_3 9 = 2$ trits.

Tag: The addresses are 10 trits, which leaves us with 5 tag trits.

Regardless of your answer to the above question, assume that CoryBot has a direct-mapped trache with TIO breakdown of 7:1:2, while SodaBot has a direct-mapped (binary) cache with a TIO breakdown of 7:1:2.

Q4.2 (2 points) For each block in their trache, CoryBot stores the tag and 1 additional trit of metadata (invalid/valid/dirty). What is the total number of trits used by the trache? Express your answer in terms of powers of 2 and 3.

Solution: $3 \cdot \left(2^3 + 3^3\right)$

Each entry must contain 7 trits for the tag, 1 trit for metadata, and 3^2 trytes ($3 \cdot 3^2 = 3^3$ trits) of data. There are 3^1 entries total, for a total of $3 \cdot \left(2^3 + 3^3\right)$ trits.

Q4.3 (2 points) For each block in their cache, SodaBot stores the tag and 2 additional bits of metadata (valid bit, dirty bit). What is the total number of bits used by the cache? Express your answer in terms of powers of 2 and 3.

Solution: $2 \cdot \left(3^2 + 2^5\right)$

Each entry must contain 7 bits for the tag, 2 bits for the metadata, and 2^2 bytes ($2^2 \cdot 2^3 = 2^5$ bits) for the data. There are 2^1 entries total, for a total of $2 \cdot \left(3^2 + 2^5\right)$ bits.

We want to model CoryBot's 7:1:2 trache and memory configuration using binary on SodaBot's CPU. For the below questions, assume that SodaBot creates a binary cache that has the same associativity and uses the same replacement scheme as CoryBot's trache. Recall that CoryBot has a tryte-addressable, 10-trit memory system.

Q4.4 (3 points) Determine the TIO breakdown with the minimum number of TIO bits so that SodaBot's system has:

- At least as many memory addresses as CoryBot's system
- At least as many cache indices as CoryBot's cache
- At least as many cache offsets as CoryBot's cache

Solution: 10:2:4

Offset: CoryBot has $3^2 = 9$ offsets, so we need at least 4 bits to have at least 9 offsets.

Index: CoryBot has $3^1 = 3$ indices, so we need at least 2 bits to have at least 3 indices.

Memory Address Size: CoryBot has 10-trit memory addresses, which means there are $3^1 0 = 59049$ addresses. To have at least least that many memory addresses, we need 16 bits ($2^1 6 = 65536$).

Tag: Given that we have 16-bit addresses, that leaves us with $16 - 2 - 4 = 10$ tag bits.

To model CoryBot's computer programs, SodaBot does the following:

1. Create a binary system with more memory addresses than CoryBot's computer.
2. Create a cache that has more indices and larger blocks than CoryBot's trache.
3. For each memory address in CoryBot's memory, convert the address and the value of the tryte at that address into binary, then load that converted value into that converted memory address.
4. Run CoryBot's program on this system.

Q4.5 (1 point) Supposing that SodaBot splits their memory addresses into the appropriate TIO bits for their cache, which of the following could possibly describe how the hit rate of our code changes after this conversion? You may assume that the program is correctly emulated by SodaBot (e.g. no arithmetic errors occur).

■ The hitrate increases

■ The hitrate remains the same

■ The hitrate decrease

Q5 *Into the Veta-Merse* (12 points)

Suppose we have a 16MiB physical memory, a 256MiB virtual memory space, and 4KiB pages.

For Q5.1 to Q5.9, assume that we are using a one-level page table.

Q5.1 (1 point) How many bits are in the page offset?

Solution: Given 4KiB pages, there are $\log_2 4 \cdot 2^{10} = 12$ bits for the page offset.

Q5.2 (0.5 point) How many bits are in the PPN?

Solution: Given a 16MiB physical address space, physical addresses are $\log_2 16 \cdot 2^{20} = 24$ bits, giving us $24 - 12 = 12$ bit PPNs.

Q5.3 (0.5 point) How many bits are in the VPN?

Solution: Given a 256MiB virtual address space, virtual addresses are $\log_2 256 \cdot 2^{20} = 28$ bits, giving us $28 - 12 = 16$ bit VPNs.

Regardless of your above answers, assume that we have 20-bit physical memory addresses, and 24-bit virtual memory addresses. Assuming that physical pages are assigned sequentially and the following 3 virtual addresses have been accessed, in order:

Virtual Address	PPN
`0xABCDEF`	0
`0xAABBCC`	1
`0x202122`	2

After accessing the previous three addresses, we access the following virtual addresses in order. For each access, fill out the corresponding physical address, and whether the access causes a page hit or a page fault. Assume that if a page fault occurs, then the next sequential physical page number is assigned to the virtual page number.

Q5.4 (1 point) `0xA01243`

○ Page Hit ● Page Fault

Solution: The VPN is `0xA01` and the offset is `0x243`. VPN `0xA01` has not been accessed before, so it is a page fault. Since it's a page fault, we use the next sequential PPN, which is `0x3`, so the physical address is `0x03243`.

Q5.5 (1 point) `0xD12362`

○ Page Hit ● Page Fault

Solution: The VPN is `0xD12` and the offset is `0x362`. VPN `0xD12` has not been accessed before, so it is a page fault. Since it's a page fault, we use the next sequential PPN, which is `0x4`, so the physical address is `0x04362`.

Q5.6 (1 point) `0x61C61C`

○ Page Hit ● Page Fault

Solution: The VPN is `0x61C` and the offset is `0x61C`. VPN `0x61C` has not been accessed before, so it is a page fault. Since it's a page fault, we use the next sequential PPN, which is `0x5`, so the physical address is `0x0561C`.

Q5.7 (1 point) `0xABC61C`

● Page Hit ○ Page Fault

Solution: The VPN is `0xABC` and the offset is `0x61C`. VPN `0xABC` has been accessed before, so it is a page hit. The existing PPN is `0x0`, so the physical address is `0x0061C`.

Q5.8 (1 point) `0x00AA00`

○ Page Hit ● Page Fault

Solution: The VPN is `0x00A` and the offset is `0xA00`. VPN `0x00A` has not been accessed before, so it is a page fault. Since it's a page fault, we use the next sequential PPN, which is `0x6`, so the physical address is `0x06A00`.

Q5.9 (1 point) `0x5E889E`

○ Page Hit ● Page Fault

Solution: The VPN is `0x5E8` and the offset is `0x89E`. VPN `0x5E8` has not been accessed before, so it is a page fault. Since it's a page fault, we use the next sequential PPN, which is `0x7`, so the physical address is `0x0789E`.

Suppose that our machine utilizes a two-level hierarchical page table that has 24-bit memory addresses, and the VPN bits are divided equally between L1 and L2 page tables. VPN1 and VPN2 correspond to the L1 and L2 page tables, respectively.

For each of the below virtual addresses, what are the corresponding VPN1 and VPN2 in hexadecimal?

Q5.10 (1 point) 0x5E889E

Solution: The VPN is `0x5E8`, which gives us a VPN1 of `0x17` (upper 6 bits) and `0x28` (lower 6 bits).

Q5.11 (1 point) 0x61C61C

Solution: The VPN is `0x61C`, which gives us a VPN1 of `0x18` (upper 6 bits) and `0x1C` (lower 6 bits).

Q5.12 (1 point) 0xDE16AB

Solution: The VPN is `0xDE1`, which gives us a VPN1 of `0x37` (upper 6 bits) and `0x21` (lower 6 bits).

Q5.13 (1 point) 0x24EB10

Solution: The VPN is `0x24E`, which gives us a VPN1 of `0x09` (upper 6 bits) and `0x0E` (lower 6 bits).

Q6 *DLPTLPPLTPLD* **(17 points)**

A palindrome is a sequence that reads the same backward as forward. For this question, our sequence will consist of an array of one digit positive integers. For example, `[1, 8, 7, 6, 7, 8, 1]` is a palindrome and `[2, 4, 5, 4, 3]` is not a palindrome

The function `num_palindrome` will take in three arguments:

- `uint32_t** matrix`: A matrix of `uint32_t`'s with dimensions length * width.
- `uint32_t length`: The number of `uint32_t*`s in the matrix.
- `uint32_t width`: The number of `uint32_t`s in each `uint32_t*` in the matrix. You may assume that width is a positive integer greater than or equal to 4 and that each row in the matrix has the same width.

`num_palindrome` should return the number of rows in the matrix that are palindromes.

Here are the SIMD functions that you may use for this question. You may not use any other SIMD functions.

- `_mm128 vecLoad(void* ptr)`: Loads four `uint32_t` from `ptr` into a SIMD vector.
- `_mm128 vecReverse(_mm128 mm)`: Reverses the order of elements in the vector `mm`.
- `void vecStore(void* ptr, _mm128 mm)`: Stores the four `uint32_t`s in `mm` at ptr.
- `_mm128 vecSet0()`: Returns a vector containing only 0s.
- `bool vecEq(_mm128 a, _mm128 b)`: Returns `true` if all elements of `a` are equal to the corresponding elements of `b`, else `false`.

Implement a version of `num_palindrome` that uses both thread-level and data-level parallelism.

```
int num_palindrome(int ** matrix, int length, int width) {
    int count = 0;

    #pragma omp parallel for reduction (+:count)
                  Q6.1         Q6.2

    for (int i = 0; i < length; i ++ ) {
                         Q6.3    Q6.4

        bool is_palindrome = true;

        int left = 0;

        int right = width - 4;
                      Q6.5

        while (left <= right && is_palindrome) {
                    Q6.6           Q6.7

            _mm128 vec1 = vecLoad(matrix[i] + left);
                                  Q6.8
            _mm128 vec2 = vecReverse(vecLoad(matrix[i] + right));
                                            Q6.9

            is_palindrome = vecEq(vec1, vec2);
                                     Q6.10

            left += 4 ;
                   Q6.11

            right -= 4 ;
                    Q6.12
        }

        if (is_palindrome) {
              Q6.13

            count += 1;
              Q6.14
        }
    }
    return count;
}
```

Q7 *Times Are Not to Scale* (7 points)

CodaBot is in charge of final exam logistics for 61C! Consider the following tasks:

Task Number	Task	Time (hours)	Prerequisites
0	Write Exam	5	-
1	Conduct Review Sessions	3	-
2	Send Seating Chart	1	-
3	Print Exam	2	0,1
4	Proctor Exam	3	3
5	Scan Exam	1	4
6	Write Solutions	2	0

Q7.1 (2 points) Suppose CodaBot can only perform one task at a time. How long would it take for CodaBot to complete all these tasks by themselves?

Solution: 17 hours. Each task must be done sequentially, and there are 17 hours' worth of tasks.

Q7.2 (2 points) Now, suppose CodaBot can perform two different tasks at the same time. What is the minimum time it takes for CodaBot to complete all these tasks by themselves?

Solution: 11 hours. Example task list:

Hour	Active Task	Active Task
0	Task 0	Task 1
1	Task 0	Task 1
2	Task 0	Task 1
3	Task 0	Task 2
4	Task 0	
5	Task 3	Task 6
6	Task 3	Task 6
7	Task 4	
8	Task 4	
9	Task 4	
10	Task 5	

CodaBot wants help, and asks their friend EvanBot for help. Here is how long EvanBot takes to perform each task:

Task Number	Time (hours)
0	4
1	3
2	1
3	1
4	3
5	2
6	3

Q7.3 (3 points) Now, suppose both CodaBot and EvanBot can only perform one task at a time, and each task can only be performed by one Bot. What is the minimum amount of time required to complete these set of tasks?

Solution: 9 hours. Example task list:

Hour	CodaBot	EvanBot
0	Task 1	Task 0
1	Task 1	Task 0
2	Task 1	Task 0
3	Task 2	Task 0
4		Task 3
5	Task 4	Task 6
6	Task 4	Task 6
7	Task 4	Task 6
8	Task 5	

Q8 *Non-quantum Computing* (10 points)

Suppose we have a chunk-addressable address space with 64 byte chunks. That is, address 1024 is 64 bytes away from address 1025 and 1 byte away from "address" $1024 + \frac{1}{64}$. To access every byte of this system, we can't use our standard integer binary addressing system, so let's use floating point!

Suppose that our memory "addresses" follow IEEE-754 floating point convention with 1 sign bit, and the number of exponent and mantissa bits that you will determine below.

Q8.1 (2 points) If we had 4KiB of memory, with chunk addresses 0, 1, 2, etc., what is the minimum number of exponent bits in our floating point memory address required to access every byte, assuming that we use a standard bias?

> **Solution:** Given 4KiB memory and 64B chunks, we have 64 chunks total. Therefore, we need 4 exponent bits to be able to represent values 0 through 63. Given 4 exponent bits, the largest non-infinity/NaN exponent value (pre-bias) is $2^4 - 2$, which, after applying the standard bias of -7, is 7, which allows us to represent values up to 63.
>
> If we used 3 exponent bits, the largest exponent value after applying the standard bias of -3 is 3, which cannot represent the value 63.

Q8.2 (1 point) True or False: The number of exponent bits in our floating point memory address needed can be reduced by using a non-standard bias.

● True ○ False

> **Solution:** If we use a bias of 0, we can address values up to 63 with 3 bits of exponent.

Q8.3 (1.5 points) What is the minimum number of mantissa bits in our floating point memory address required to address every byte?

> **Solution:** The largest memory address we need to represent is `0b11 1111.111111` ($63 + \frac{63}{64}$). To represent this value, we need 11 bits of mantissa ($0b1.11111111111 \cdot 2^5$).

Q8.4 (1 point) True or False: The number of bits used to represent the memory address using this floating point chunk-addressed system will always be greater than the number of bits used to represent the memory address in a byte-addressed system.

● True ○ False

> **Solution:** The floating point system we introduced above uses 11 mantissa bits, 4 exponent bits, and 1 sign bit, for a total of 16 bits. A binary system would require $\log_2 4 \cdot 2^{10} = 12$ bits.

Regardless of your answer to the previous subparts, assume that we have 13-bit floating point memory addresses with 1 sign bit, 4 exponent bits with standard bias, and 8 mantissa bits, and we don't use denormalized numbers.

Convert the following memory addresses from unsigned binary (compacted into hex) to floating point. If there is no floating point number that precisely equals this address, write N/A. Your answer should be in hexadecimal.

Q8.5 (1.5 points) `0xC61`

> **Solution:** N/A
>
> Since bits 11 and 0 are set, it cannot fit into a floating point number with 8 mantissa bits, so it is not representable.

Q8.6 (1.5 points) `0x200`

> **Solution:** `0x0A00`
>
> Each chunk is 64 bytes, so that accounts for the lower 6 bits of the address, `0x00`. The chunk address is the upper 6 bits of the address, is `0x08`. We then must translate `0x08.00` to the floating point system described.
>
> $0x08.00 = 0b1 \cdot 2^3$, so the exponent is `0b1010` after applying the bias, and the mantissa is all 0's. This gives us `0x0A00`.

Q8.7 (1.5 points) `0x7D8`

> **Solution:**
>
> Each chunk is 64 bytes, so that accounts for the lower 6 bits of the address, `0x18`. The chunk address is the upper 6 bits of the address, is `0x1F`. We then must translate `0b1 1111.01 1000` to the floating point system described.
>
> $0b11111.011000 = 0b1.1111011 \cdot 2^4$, so the exponent is `0b1011` after applying the bias, and the mantissa is `0xF6`. This gives us `0x0BF6`.

Q9 *The Finish Line* (1 points)

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

Q9.1 (1 point) Which state is Andrew Liu, the 61C TA, from?

Solution: West Virginia$^{\text{Mountain Mama}}$

Q9.2 (0 points) Is there anything you want us to know? Feel free to use this box for doodles!

Solution: ¯_(ツ)_/¯