CS61C Spring 2025 — Yan Midterm

Solutions last updated: Monday, March 31, 2025

PRINT Your Name: ______________________

PRINT Your Student ID: ______________________

PRINT the Name and Student ID of the person to your left: ______________________

PRINT the Name and Student ID of the person to your right: ______________________

PRINT the Name and Student ID of the person in front of you: ______________________

PRINT the Name and Student ID of the person behind you: ______________________

You have 110 minutes. There are 8 questions of varying credit. (100 points total)

Question:	1	2	3	4	5	6	7	8	Total
Points:	14	11	21	24	13	8	9	0	100

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

○ Unselected option (Completely unfilled)

✓ Don't do this (it will be graded as incorrect)

● Only one selected option (completely filled)

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

■ You can select

■ multiple squares

✓ (Don't do this)

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

If an answer requires hex input, you must only use capitalized letters (`0xB0BACAFE` instead of `0xb0bacafe`). For hex and binary, please include prefixes in your answers unless otherwise specified, and do not truncate any leading 0's. For all other bases, do not add any prefixes or suffixes.

As a member of the UC Berkeley community, I act with honesty, integrity, and respect for others. I will follow the rules of this exam.

Acknowledge that you have read and agree to the honor code above and sign your name below:

____

Clarifications made during the exam:

Q1.5: On line 6, jal ra should be jr ra [fixed]

Q5.4: Main memory access time refers to the L1 cache miss penalty.

Q4.13: On page 9, the example under the table should read "For example, if a0 points to … then **`store_str_as_int`** would store the word `0x12345678` at `0x10000000`" [fixed]

Q1 *Potpourri* 🥗 **(14 points)**

Q1.1 (1.5 points) Convert −49 from decimal to 8-bit binary two's complement representation.

`0b`11001111

Solution: $-49 = -128 + 64 + 8 + 4 + 2 + 1$. Alternatively, you can "flip the bits and add one" to the binary representative of +49, giving: `0b00110001 -> 0b11001110 -> 0b11001111`

Common error(s): `0b11001110` if converted to one's complement instead of two's complement.

Q1.2 (1.5 points) Convert the biased hexadecimal number `0x2A` (with a bias of −31) to decimal.

11

Solution: $42 - 31 = 11$

Common error(s): subtracted the bias instead of adding: $42 - (-31) = 73$

Q1.3 (2 points) Convert the RISC-V instruction `beq a3 s1 12` into 32-bit hexadecimal machine code.

`0x`00968663

Solution: `0x00968663`

The opcode for beq is `0b110 0011`, the funct3 is `0b000`, and the instruction is a B-type instruction with rs1 = a3 (`0b01101`), rs2 = s1 (`01001`), and offset/immediate = 12 (`0b000000001100`). Putting this together according to the B-type format gives `0b0000000 01001 01101 000 01100 1100011`, or `0x00968663`. Note that for branch and jump instructions, the implicit 0th bit is omitted.

Common error(s): Forgot the implicit bit for branch and jump instructions, did not rearrange the immediate bits according to the B-type instruction format, or swapped placement of rs1 and rs2.

Q1.4 (2 points) Convert the following RISC-V machine code into its corresponding instruction. If there is an immediate value, express it in decimal form. Provide the appropriate register names, not numbers, where necessary (e.g., `s5` instead of `x21`).

`0x01685A93`

srli s5, a6, 22

Solution: srli s5, a6, 22

Breakdown: `0b0000 0001 0110 1000 0101 1010 1001 0011` Opcode = 001 0011 -> this is an I-type instruction. Looking at the funct3 (`0b101`) and funct7(`0b000000`), we narrow it down to an srli instruction. Finally we translate rd, rs1, and immediate according to the I-type instruction format chart, and fill in the overall instruction.

Common error(s): swapped positions of rd and rs1, used numbers instead of appropriate register names.

Q1.5 (2 points) In the code below, which registers are used in a way that **violates** calling convention?

```
1 foo:
2   lui a0, 0xFFEE0
3   addi t0, a0, 4
4   jal ra bar
5   lw s0, 0(t0)
6   jr ra
```

☐ `a0`	[✓ `t0`]	☐ `sp`
☐ `a1`	[✓ `s0`]	[✓ `ra`]

Solution: `t0`, a temporary register, is used after the call `jal ra bar`, even though its value may no longer be defined. `s0`, a saved register, is written without saving it's original value. Finally, `ra` must be saved in order to properly return from the function foo after `ra` is modified by the call `jal ra bar`.

Exam clarification: `jal ra` on line 6 should instead be `jr ra`

Q1.6 (1 point) Which of these statements about the Assembly stage of CALL are true? Select all that apply.

[✓ Outputs a file that contains text and data segments]

☐ Converts high-level language code to assembly language code

[✓ Generates a relocation table]

☐ Resolves all label and data references

(4 points) You have started a GDB session with the following code segment:

```
 1    int main(int argc, char *argv[]) {
 2      char *crops[4] = {"parsnip", "melon", "pumpkin", "corn"};
 3      int count = // code omitted;
 4  --> quality(crops, count);
 5    }
 6
 7    void quality(char** crops, int count){
 8      while (count >= 0) {
 9        // code omitted
10        quality_crops = // code omitted
11        count -= 1;
12      }
13      return quality_crops
14    }
```

Your GDB session is about to execute line 4, but has not executed it yet. Write a sequence of GDB commands that would perform the following actions. Assume each command is executed before the next:

```
# 1. print the address of count:

(gdb) p &count
          Q1.7
# 2. set a breakpoint on line 10 if count is even:

(gdb) b 10 if (count % 2) == 0
                  Q1.8
# 3. go to your breakpoint without restarting your session:

(gdb)   c
      Q1.9
# 4. print 1 if quality_crops is 3, otherwise print 0:

(gdb) p quality_crops == 3
              Q1.10
```

Solution: see blanks above. Tip: review the debugging lab!

Common error(s): `p count` instead of `p &count`, you need the `&` to specify the address of a variable.

Q2 *Conditional Eggsecution* 🥚 **(11 points)**

For the entirety of this question, consider the following C code:

`conditional_sum`: Takes the values greater than or equal to `50` in `arr`, and adds their sum to `*dest`.		
Arguments	`int32_t *dest`	Pointer to allocated space for the sum.
	`int32_t *arr`	Array of integers to conditionally sum.
	`size_t length`	The number of elements in `arr`.
Return value	`void`	

```c
void conditional_sum(int32_t *dest, int32_t *arr, size_t length) {
  for (int i = 0; i < length; i++) {
    if (arr[i] >= 50) {
      *dest += arr[i];
    }
  }
}
```

Q2.1 (4 points) We decide to call this function from `main` on a 32-bit **little-endian** system:

```c
int main() {
  int32_t dest1 = 0;
  int32_t dest2 = 0;
  int16_t arr1[4] = {0x0080, 0xA000, 0x0036, 0x0000};
  uint8_t arr2[4] = {0x6C, 0xFF, 0xFF, 0xFF};
  conditional_sum(&dest1, (int32_t *) arr1, 2);
  conditional_sum(&dest2, (int32_t *) arr2, 1);
}
```

What are the values in `dest1` and `dest2` in hexadecimal after `main` is run?

dest1: 0x36

dest2: 0x0

Solution: Given a little endian system, each byte is stored LSB first. This means from highest to lowest memory address, `arr1` looks like this: `0x80 0x00 0x00 0xA0 0x36 0x00 0x00 0x00`, while `arr2` looks like this: `0x6c 0xFF 0xFF 0xFF`. Note that elements in an array are still stored in order with the first element having the lowest address.

dest1: Casting arr1 to an array of `int32_t` of length 2, we get the two values `0xA0000080` and `0x00000036`. Only `0x00000036` is greater than 50 (`0xA0000080` is negative), so that is our overall sum.

dest2: Similarly, treating `arr2` as a `int32_t` gives us `0xFFFFFF6C`, which is negative and less than 50. Thus our overall sum is 0.

Common error(s): Storing bytes in big endian instead of little endian, missing the cast to (`int32_t *`), interpreting as a `uint32_t` instead of `int32_t`.

Q2.2 (1 point) Which section of memory would the variable `dest1` live in?

[✓ Stack] ○ Heap ○ Code ○ Static

Q2.3 (6 points) Write the loop body of `conditional_sum` to use **bit masking** instead of branch statements (`if-else`) or comparisons.

Hint: If `arr[i] < 50`, what is the sign of `arr[i] - 50`?

You may assume that:

- Signed integers are stored in two's complement.
- No overflows occur during addition/subtraction. This means that `arr[i] - 50` does not overflow.
- `dest` is large enough to hold the result.

Your answer may consist of only the following operations:

Allowed	
Addition / Subtraction	`+ -`
Shifts	<< and >> (sign-extended)
Bitwise Operations	`~ & | ^`
Boolean Operations	`! && ||`
Parentheses	`()`
Array Subscripting	`arr[x]`

```
void conditional_sum(int32_t *dest, int32_t *arr, size_t length) {
  uint32_t mask;
  for (int i = 0; i < length; i++) {

    mask = ~((arr[i] - 50) >> 31);
                    Q2.3
    *dest += mask & arr[i];
  }
}
```

Solution: Alternate solution: `(49 - arr[i]) >> 31`

Based on the hint, we know that if `arr[i] < 50`, then `arr[i] - 50 < 0`. This means that `arr[i] - 50` will be negative, and thus have a MSB = 1. We can use this fact to create a mask for each `arr[i]`, where the mask will equal `0xFFFF` if `arr[i] >= 50`, and equal to `0x0000` if `arr[i] < 50`. We do this by right shifting (and sign extending) the MSB by 31 bit positions, which extends the sign bit across all 32 bits of our mask. Because we want all ones if our number is positive (MSB = 0), and all zeroes if our number is negative (MSB = 1), we take the bitwise NOT of this shifted value to be our mask. Finally, we can perform a bitwise AND of the mask with our value, effectively zeroing out any element < 50 before summing.

Q3 *Generic Linked Lists* **(21 points)**

This question defines a linked list with a twist: The linked list uses generics to store data without a data type! Consider the following definition of a linked list node, `node_t`:

```
typedef struct node {
  struct node *next_node; // next_node is a node_t*. If this node is the end
                          // of the list, next_node is NULL
  size_t width;           // size of data, in bytes
  void *data;
} node_t;
```

(9 points) Implement the function `add_head_node`. The `memcpy` function prototype is provided below for your reference:

```
void *memcpy(void *dest, void *src, size_t n);
```

`add_head_node`: Add a new node to the head of `list` whose `data` member points to a heap-allocated copy of `temp`, which is `width` bytes long.		
Arguments	`node_t *list`	Pointer to linked list to add a new head node to
	`size_t width`	The size of `temp`, in bytes
	`void *temp`	A pointer to the data that should be copied into this node
Return value	`node_t *head`	The new head of the linked list

```
node_t* add_head_node(node_t *list, size_t width, void* temp){

  node_t *head = (node_t *)malloc(sizeof(node_t));
                                Q3.1

  head->next_node = list;
                    Q3.2

  head->width = width;
                Q3.3

  head->data = malloc(width);
              Q3.4

  memcpy(head->data, temp, width);
             Q3.5       Q3.6   Q3.7

  return head;
          Q3.8
}
```

Solution: We want to create a new head node that points to the previous head of our linked list, with a copy of the data pointed to by `temp`. First, we need to allocate space for this new node on the heap using `malloc(sizeof(node_t))` or `calloc(1, sizeof(node_t))`. Then we tell the new head node to point to the old list with `head->next_node = list`. We then set the data `width` of this particular node, and use that value to `malloc` space for the data we are about to copy. Finally, we use `memcpy` to copy over the data from `temp` into the newly allocated space pointed to by `head->data`.

Common error(s): Note that both `malloc` and `calloc` return the `void *` type, meaning they return a pointer to the allocated space, not the space itself. When accessing members of a struct given the pointer to the struct, use arrow notation: `head->next_node` or dereference first `(*head).next_node`.

(7 points) Implement the function **`arr_to_ll`** that uses **`add_head_node`** to convert an array of strings into a linked list, preserving the order of elements. In other words, the head of the linked list returned from **`arr_to_ll`** should contain a pointer to a copy of the data in **`arr[0]`** on the heap.

Regardless of what you wrote before, assume that **`add_head_node`** is correctly implemented.

`arr_to_ll`: Given a **`char*`** array, create an equivalent linked list of strings		
Arguments	`char** arr`	Array of strings
	`size_t count`	The number of strings in the array **`arr`**
Return value	`node_t *head`	The head of the linked list

```
node_t* arr_to_ll(char** arr, size_t count){
  node_t *head = NULL;

  for (size_t i = count; i > 0;  i-- ) {
                         Q3.9    Q3.10

    size_t len = strlen(arr[i-1]) + 1;
                         Q3.11

    head = add_head_node( head ,  len , arr[i-1])
                          Q3.12   Q3.13   Q3.14
  }

  return  head ;
          Q3.15
}
```

Solution: In order for the head node to contain the first string in our array, and the last node to contain the last string in our array, we need to iterate backwards across **`arr`** as we build the linked list. We use the **`add_head_node`** function to add nodes for us, but it takes in a **`size_t width`** argument. We can figure out how many bytes we need to store each string using the **`strlen()`** function, which tells us the number of bytes in a string up to but not including the null terminator. To ensure that we copy over well formed strings, we add 1 to this value to get the total number of bytes needed to store each string.

Common error(s): Not considering the null terminator when calculating length, off by one iteration in the for loop (i >= 0), improper array indexing (Ex. **`(*arr)[i-1]`**).

(5 points) Finally, implement the function **`free_ll`** that frees a linked list returned from **`add_head_node`**. Assume that the **`data`** member of each node in the list points to heap-allocated memory.

`free_ll`: Given a linked list of nodes, free the entire linked list		
Arguments	`node_t *list`	Pointer to the linked list to free
Return value	`void`	

(Question 3 continued...)

```
void free_ll(node_t* list){
  node_t *curr = list;

  while (curr != NULL) {
             Q3.16

    node_t *temp = curr;
                   Q3.17

    curr = curr->next_node;
               Q3.18

    free(temp->data);
           Q3.19

    free(temp);
         Q3.20
  }
}
```

Solution: For each node of our linked list, we need to free the space we allocated for the node data as well as the space for the node itself. To do this, we use a temporary `node_t *` to point to the node to be freed, while we move our loop pointer `curr` to the next node. This allows us to free `temp->data` and `temp` while maintaining a reference to the rest of our linked list.

Common error(s): Incorrect free (freeing `temp->width` or `temp->next_node`), incorrect struct member accesses (Ex. `(*temp)->data` or `temp.data`), freeing the node before we free the node's data (this causes us to lose the pointer to allocated memory).

Q4 *Not Like Us* 🗣️ **(24 points)**

Consider the function `store_first_byte` as described below on a 32-bit **little-endian** system:

`store_first_byte`: Reads the first two digits of a string as hexadecimal and stores it in memory.		
Arguments	`a0`	A `char*` consisting of `char`s from `'0'` – `'9'`. `strlen(a0)` is even and at least 2.
	`a1`	A **pointer** to allocated memory to store the first byte of the string pointed to by `a0` read as hexadecimal
Return value	`void`	

For example, if `a0` points to `"12345678"`, and `a1` contains `0x10000000`, then `store_first_byte` would store the byte `0x12` at `0x10000000`.

(10 points) Implement your own version of `store_first_byte` but the only load instruction you can use is `lbu`:

```
store_first_byte:

    lbu t0 0(a0)
              Q4.1

    slli t0 t0 4
          Q4.2

    lbu t1 1(a0)
              Q4.3

    andi t1 t1 15
               Q4.4

    add t0 t0 t1
          Q4.5

    sb t0 0(a1)
             Q4.6
    jr ra
```

Solution: Key ideas: the ascii values of each digit character is different from the value of the digit itself, (Ex. 2 $\neq$ `0x32`). However, for the digits 0 - 9 (`0x30` - `0x39`), we see that the bottom 4 bits (or 1 hex digit) of their ascii value matches the digit value, and is in hexadecimal as desired.

We first load the byte for the first number character using lbu, storing this in `t0`. In the provided example, we would load `0x31` for the character '1'. We then shift this left by 4 bits to position the character in the correct spot (Ex.`0x31`->`0x310`). We then load the second number character into `t1`, which in the example is `0x32` or the character '2'. Because we only want the bottom 4 bits, we perform a bitwise AND of `t1` with `15` or `0b1111` in binary. Finally, we add `t0` and `t1` to get `0x312`, and we can use the `sb` instruction to store just the bottom 1 byte (`0x12`) into memory.

(10 points) Now, re-implement `store_first_byte`, but the only load instruction you can use is `lhu`.

```
store_first_byte:

    lhu t0 0(a0)
           Q4.7
    srli t1 t0 8
            Q4.8
    andi t1 t1 15
            Q4.9
    slli t0 t0 4
       Q4.10
    add t0 t0 t1
       Q4.11
    sb t0 0(a1)
          Q4.12
    jr ra
```

Solution: The difference here is that because we are using `lhu`, we get the first 2 “digit characters” of our string at the same time, albeit in the “wrong” order due to the endinanness of the system (in the provided example, we would load in `0x3231`). To place the first 4 bits in the correct location, we shift this value by 8 bits and store it in `t1` (`0x3231` -> `0x32`). Similar to the previous problem, we perform a bitwise AND to isolate only the bottom 4 bits (`0x2`). For the second 4 bits, we shift the original number left by 4 (`0x3231` -> `0x32310`). Finally, we add these two values (`0x32310 + 0x2` = `0x32312`), and use `sb` to store the byte (`0x12`) into memory.

Now, consider the following function `store_str_as_int`:

`store_str_as_int`: Reads a string of digits as hexadecimal and stores it in memory.		
Arguments	`a0`	A `char*` consisting of `char`s from `'0'` – `'9'`. `strlen(a0)` is exactly 8.
	`a1`	A **pointer** to allocated memory to store the value of the string pointed to by `a0` read as hexadecimal
Return value	`void`	

For example, if `a0` points to `"12345678"`, and `a1` contains `0x10000000`, then `store_str_as_int` would store the word `0x12345678` at `0x10000000`.

Drake implements `store_str_as_int` in the following way:

(Question 4 continued...)

```
store_str_as_int:
    # prologue omitted
loop:
    lb t0 0(a0)
    beqz t0 end
    # save a0 and a1 on the stack
    jal ra store_first_byte
    # restore a0 and a1 from the stack
    addi a0 a0 2
    addi a1 a1 1
    j loop
end:
    # epilogue omitted
    ret
```

Regardless of what you wrote before, assume that `store_first_byte` works according to the specification.

Kendrick doubts that Drake's implementation of `store_str_as_int` works properly. To check Drake's implementation, Kendrick puts a pointer to `"12345678"` in `a0` and `0x10000000` in `a1`, then calls `store_str_as_int`. Assume Kendrick has already allocated sufficient memory at `0x10000000`.

Kendrick then executes `lw a0 0(t0)`, where `t0` holds the memory address `0x10000000`. After this instruction, Kendrick finds that `a0` holds the number `0x78563412`, not `0x12345678`.

Q4.13 (4 points) Why is Drake's implementation of `store_str_as_int` wrong? Give your answer in at most 20 words.

Drake forgot that in RISC-V integers are stored in little-endian order not big endian order.

Q5 *Caches* 💸 **(13 points)**

Q5.1 (3 points) What is the tag-index-offset breakdown for a 32 KiB direct-mapped cache with 64B lines on a system with a 128KiB address space?

T: 2 bit(s)	I: 9 bit(s)	O: 6 bit(s)

Solution: With a 128KiB address space (2^{17} bytes), we know that the total number of address bits should be 17. Given that each line of our direct mapped cache is 64B, we need 6 offset bits to distinguish them all ($2^6 = 64$). For the index, we know that there are (32KiB / 64B = $\frac{2^{15}}{2^6} = 2^9$) cache lines in our cache, so we need 9 index bits to index them all. Finally, 17 total address bits - 6 offset bits - 9 index bits gives us 2 tag bits.

For Q5.2–Q5.3, the code below is executed on a 32-bit system with a 256B, 4-way set-associative cache with 8B cache lines and a first-in-first-out (FIFO) replacement policy. Assume the cache is cold prior to the start of loop 1.

```
#define STRIDE 2

// register int32_t <var> means that <var> is stored in a RISC-V register

void foo(int32_t *arr) {   // Assume arr = 0x1000 0020
  register int32_t index;

  /* LOOP 1 */
  index = 0;
  for (register int32_t j = 0; j < 32; j += 1) {
    arr[index] = j;
    index += STRIDE;
  }

  /* LOOP 2 */
  index = 0;
  for (register int32_t j = 0; j < 32; j += 1) {
    arr[index] = j + 1;
    index += STRIDE;
  }
}
```

Q5.2 (4 points) How many cache hits and misses occur during `LOOP 2`?

Hits: 32	Misses:0

Solution: Loop 1 fully populates our cache with 32 memory accesses, and our cache contains the addresses `0x10000020`, `0x10000028`, `0x10000030`, ... , where each cache line contains 2 4-byte integers. For each access to Loop 2, the cache block containing the desired address is already in our cache (we are accessing the same addresses as in Loop 1), meaning we have 0 misses and 32 hits.

Common error(s): Forgetting to execute Loop 1 before calculating Loop 2.

Q5.3 (2 points) What is the smallest positive integer value for `STRIDE` such that every element accessed in `LOOP 1` and `LOOP 2` share the same index in the cache?

`STRIDE:` 16

Solution: We need to choose a stride such that each of the 32 arr[i] accesses maps to the same index in the cache. We have 3 offset bits (as each cache line is 8B), and we have 3 index bits (as there are 256B / 8B / 4 ways = 8 cache lines). In order to ensure each access maps to the same index, we need a stride between accesses equal to `0x1000000`, or 64. Dividing this by 4 to account for 4 byte integers, we get 64/4 = 16.

Q5.4 (4 points) Our system uses a single L1 cache (L1$). Now, we run two separate programs to measure the performance of the cache and observe the following results:

Program A	
AMAT	200ns
L1$ Miss Rate	35%

Program B	
AMAT	100ns
L1$ Miss Rate	10%

What is our cache hit time and main memory access time? Assume these times remain consistent between Program A and Program B.

L1$ Hit Time: 60ns

Main Memory Access Time: 400ns

Solution: Let X = L1$ hit time, and Y = main memory access time.

Program A: 200ns = X + 0.35(Y) Program B: 100ns = X + 0.10(Y)

We can solve this system of equations to get X = 60ns and Y = 400ns.

Q6 *Ba-lite-tro* ♥♣♦♠ **(8 points)**

Noah is playing the game Ba-lite-tro on his laptop. The game stores his score using the IEEE-754 single-precision floating-point format with 1 sign bit, 8 exponent bits (with a bias of −127), and 23 significand bits.

Q6.1 (2 points) Noah somehow scores −67.125 points?! Convert his score into hexadecimal in this format.

0xC2864000

Solution: 0xC2864000

-67.125 = `0b1000011.001` = `0b1.000011001` $* 2^6$ Exponent bits - 127 = 6, so the exponent bits are 133 or `0b10000101`. The sign bit is 1 for negative, and the significand bits are `0b00001100100000000000000`. Putting this all together gives `0b1 10000101 00001100100000000000000` = `0b1100 0010 1000 0110 0100 0000 0000 0000` or `0xC2864000` in hex.

Common error(s): Improper calculation of exponent bits using the bias, forgetting to omit the implicit 1.

Noah wants to switch to playing Ba-lite-tro on his phone, but it doesn't have enough storage! He decides to modify the game to use a **16-bit floating-point format** instead, following the IEEE-754 standard with 1 sign bit, 7 exponent bits (with a bias of −63), and 8 significand bits. This 16-bit format is used to store his game score.

Q6.2 (3 points) Given this new representation, what is the **maximum game score** (i.e. largest possible non-infinite positive number) in this format?

Express your answer first as 16-bit hexadecimal:

0x7EFF

Now express your answer in terms of powers of two and **simplify as much as possible.**

$2^{64} - 2^{55}$

Solution: 2^64 - 2^55

We want the maximum game score without the number becoming infinite or NaN. To do this we max out the significand bits (all 1s), and max out the exponent by setting it to all 1s with the exception of the last exponent bit (so it is not classified as NaN). In this format, the number would have a sign bit = 0 (positive), exponent bits = `0b1111110`, and significand bits = `0b11111111`.

Finally we can write this value in terms of powers of two. The exponent is equal to $2^7 - 2^1 + (-63) = 128 - 2 - 63 = 63$, while the significand is equal to $2 - 2^{-8}$ (remember the implicit 1. and count the bit positions after the decimal point). Finally, multiplying exponent with significand gives $2^{63} * (2 - 2^{-8}) = 2^{64} - 2^{-55}$

Noah decides to adjust the 16-bit game score format by changing how the 16 bits are allocated to the sign, exponent, and significand. He decides to reallocate bits to optimize for the largest maximum game score.

While adjusting, Noah still follows the IEEE-754 floating-point number format. This means that:

- There must be one sign bit
- The exponent bias is $-(2^{n-1} - 1)$, where n is the number of exponent bits
- Infinities, NaN, and denormalized numbers are still encoded as expected.

Q6.3 (3 points) If Noah allocates his 16 bits optimally, what is the largest possible maximum game score?

Express your answer first as 16-bit hexadecimal:

```
0x7FFE
```

Now express your answer in terms of powers of two and **simplify as much as possible.**

$2^{2^{14}-1}$

Solution: $2^{2^{14}-1}$

The optimal allocation of bits in order to achieve the largest possible maximum game score is 1 sign bit, 15 exponent bits, and 0 significand bits. This means our standard bias is $-(2^{n-1} - 1) = -(2^{15-1} - 1) = -(2^{14} - 1)$.

Similar to the previous problem, in order to max out our exponent we set all bits equal to 1 with the exception of the last bit to get `0x7FFE`. This makes our exponent bits `0b111 1111 1111 1110`, or $2^{15} - 2^1$, so our overall exponent = exponent bits + bias = $2^{15} - 2^1 - (2^{14} - 1) = 2^{14} - 1$. Finally, the value of our number is 2^exponent, or $2^{2^{14}-1}$.

Common error(s): Forgetting the implicit 1. for normalized numbers, setting the exponent to be all 1s, writing just the exponent instead of 2^exponent.

Q7 *Boolean Algebra* 🙆🙅 **(9 points)**

NOT ($\overline{x}$), AND ($\cdot$), OR ($+$) each count as one operator. We will assume standard C operator precedence, so use parentheses when uncertain.

Your answers may consist of the following operators and symbols:

Operators			**Symbols**		
NOT	AND	OR	Inputs	Constants	Parentheses
$\overline{x}$	$x \cdot y$	$x + y$	A, B, C	0, 1	()

Q7.1 (4 points) Simplify the following boolean algebra expression

$$(\mathrm{A} + \mathrm{C}) \cdot \left(\overline{\mathrm{A}} + \mathrm{B}\right) \cdot \overline{\mathrm{C}}$$

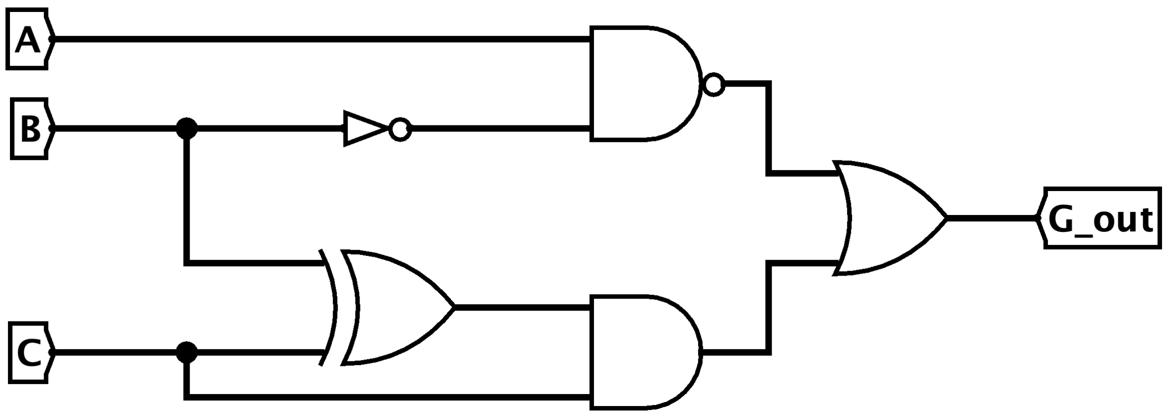
For full credit, reduce the above to an expression that uses at most 3 operators.

$$\mathrm{AB}\overline{\mathrm{C}}$$

Solution:

$$\begin{aligned}
&= (\mathrm{A} + \mathrm{C})\left(\overline{\mathrm{A}} + \mathrm{B}\right)\overline{\mathrm{C}} \\
&= \left(\mathrm{A}\overline{\mathrm{C}} + \mathrm{C}\overline{\mathrm{C}}\right)\left(\overline{\mathrm{A}} + \mathrm{B}\right) \\
&= \left(\mathrm{A}\overline{\mathrm{C}}\right)\left(\overline{\mathrm{A}} + \mathrm{B}\right) \\
&= \mathrm{A}\overline{\mathrm{C}}\left(\overline{\mathrm{A}}\right) + \mathrm{A}\overline{\mathrm{C}}(\mathrm{B}) \\
&= \mathrm{AB}\overline{\mathrm{C}}
\end{aligned}$$

Q7.2 (5 points) Give a simplified boolean algebra expression for `G_out` in terms of the inputs `A`, `B` and `C`



For full credit, reduce the circuit above to an expression that uses at most 3 operators.

$$\overline{A} + B + C$$

Solution:

$$\begin{aligned}
G_{\text{out}} &= \overline{A\,\overline{B}} + \left(\overline{B}\,C + B\overline{C}\right)C \\
&= \overline{A} + B + \left(\overline{B}\,CC + B\overline{C}C\right) \\
&= \overline{A} + B + \left(\overline{B}C\right) \\
&= \overline{A} + B(1 + C) + \left(\overline{B}C\right) \\
&= \overline{A} + B + BC + \overline{B}C \\
&= \overline{A} + B + C\left(B + \overline{B}\right) \\
&= \overline{A} + B + C
\end{aligned}$$

Q8 *Eggcellence* 🍳 **(0 points)**

These questions will not be assigned credit. Feel free to leave them blank.

Q8.1 Given an egg carton with 12 eggs, what are the first two eggs you pick? (Fill in the square with a 1 for the first egg, and a 2 for the second egg)

LID					
☐ 🐣 A1	☐ 🐣 A2	☐ 🐣 A3	☐ 🐣 A4	☐ 🐣 A5	☐ 🐣 A6
☐ 🐣 B1	☐ 🐣 B2	☐ 🐣 B3	☐ 🐣 B4	☐ 🐣 B5	☐ 🐣 B6

...and why?

Q8.2 If there's anything else you want us to know, or you feel like there was an ambiguity in the exam, please put it in the box below.

For ambiguities, you must qualify your answer and provide an answer for both interpretations. For example, "if the question is asking about A, then my answer is X, but if the question is asking about B, then my answer is Y". You will only receive credit if it is a genuine ambiguity and both of your answers are correct. We will only look at ambiguities if you request a regrade.

Alternatively, draw something egg-cellent!