CS 61C Summer 2022

Caroline, Justin, Peyrin Midterm

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

PRINT your student ID: _____

You have 110 minutes. There are 5 questions of varying credit (100 points total).

Question:	1	2	3	4	5	Total
Points:	20	25	20	25	10	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* **(20 points)**

Q1.1 (6 points) Translate the following decimal numbers into 8-bit two's complement, unsigned, and sign-magnitude representations in the table below.

If a translation is not possible, please write "N/A". **Write your final answer in hexadecimal format, including the relevant prefix.**

Decimal	Two's Complement	Unsigned	Sign-Magnitude
128	N/A	0x80	N/A
-12	0xF4	N/A	0x8C

Solution:

8-bit two's complement numbers can represent numbers up to $2^8 - 1 = 127$, so 128 cannot be represented.

$128 = 2^7$, so in unsigned representation, we have 0b1000 0000 = 0x80.

Intuitively, an 8-bit sign-magnitude number can only use 7 bits to represent the number (leaving 1 bit for the sign). A 7-bit unsigned number can represent numbers up to $2^8 - 1 = 127$, so 128 cannot be represented.

12 in unsigned 8-bit binary is 0b0000 1100. Since we want to represent -12 in two's complement binary, we'll flip the bits: 0b1111 0011, and add one: 0b1111 0100. Converting to hex, we get 0xF4.

Negative numbers cannot be represented as unsigned.

An 8-bit sign-magnitude number uses 7 bits to represent the magnitude of the number. 12 in unsigned 7-bit binary is 0b000 1100. Then we add the sign bit, which is 0b1 since we want to represent a negative number. In total, we get 0b1000 1100, which is 0x8C in hex.

Q1.2 (2 points) Convert 83 to the following bases as an unsigned number. Remove any leading zeros.

Binary	Hex
`0b101 0011`	`0x53`

Solution:

The nearest power of 2 below 83 is $2^6 = 64$. The remainder is $83 - 64 = 19$. The nearest power of 2 below 19 is $2^4 = 16$. The remainder is $19 - 16 = 3$. The nearest power of 2 below 3 is $2^1 = 2$. The remainder is $3 - 2 = 1 = 2^0$. In total, we have $2^6 + 2^4 + 2^1 + 2^0 = 64 + 16 + 2 + 1 = 83$. Converting this to unsigned binary gives us `0b101 0011`.

In hexadecimal, we can zero-pad to 8 bits (1 hex digit = 4 bits, so we need the bit length of the number to be a multiple of 4), which gives us `0b0101 0011`. In hex, this is `0x53`.

Q1.3 (3 points) Which of the following representations have more than one representation of 0? Select all that apply.

☐ (A) Two's complement

■ (B) Sign-magnitude

■ (C) An IEEE-754 standard double-precision float

☐ (D) Bias notation

☐ (E) None of the above

Solution:

Two's complement has one representation of zero. Intuitively, if you try to get a representation of "negative zero", flipping all the bits of `0b0000...0000` gives you `0b1111...1111`, and adding one gives you `0b0000...0000` again.

Sign-magnitude has two representations of zero: `0b0...0000` (positive zero), and `0b1...0000` (negative zero).

Floating point representation has two representations of zero: set all exponent and mantissa bits to zero. Then you can set the sign bit to either 0 or 1, and both result in 0.

Bias notation is unsigned representation with an offset. Unsigned representation has only one representation of 0, so regardless of bias offset, there cannot be a second representation of 0.

Q1.4 (3 points) Which program resolves pseudoinstructions?

- ● (A) Assembler
- ○ (B) Interpreter
- ○ (C) Linker
- ○ (D) Loader
- ○ (E) None of the above

Solution: Pseudoinstructions are resolved by the assembler in the process of converting instructions into machine code (raw bits).

Q1.5 (3 points) Which program can output pseudoinstructions?

- ○ (A) Assembler
- ● (B) Compiler
- ○ (C) Linker
- ○ (D) Preprocessor
- ○ (E) None of the above

Solution: Pseudoinstructions are part of assembly code. The compiler outputs assembly code.

Q1.6 (3 points) Which program initializes registers to their default value?

- ○ (A) Assembler
- ○ (B) Compiler
- ○ (C) Interpreter
- ○ (D) Linker
- ● (E) None of the above

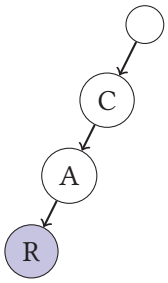
Solution: None of these choices actually start and run the program, so they don't initialize registers.

Q2 *Trie, Trie Again* **(25 points)**

In this problem, we will be implementing a trie. A trie is a data structure that stores strings in a tree-like structure, with each character in the string corresponding to a node. If two strings start with the same characters, the nodes for those characters will be shared between the two strings.

For each character in the string, our trie will create an `AlphaTrieNode` struct with two fields:

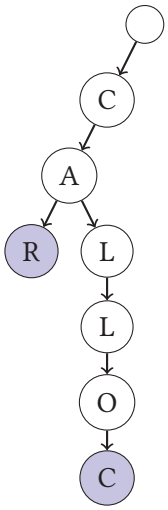
- A boolean `last` that indicates whether or not the character is the last character in a string. In the diagram, nodes with `last` set to true are shaded.
- An array `next` of 26 `AlphaTrieNode` pointers. Each element in the `next` array corresponds to a letter `a-z`. Each array element is either a pointer to the node corresponding to that letter, or `NULL` if there is no node for that letter.



This trie on the left stores one word "car".

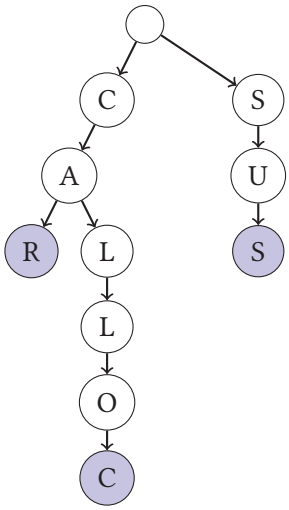
The `last` field is `false` in the C and A nodes, and `true` in the R node.

In the C node, `next[0]` should hold the address of the A node, and `next[1]`, `next[2]`, ..., `next[25]` should all be `NULL`.

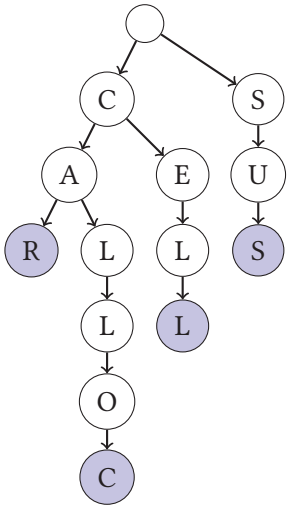


We want to insert "calloc" into the trie. Note that "car" and "calloc" both start with "ca—", so we don't create new nodes for those two characters. However, "—lloc" is different from "—r", so our trie creates new nodes for those four characters.

In the `A` node, `next[11]` (for L) and `next[17]` (for R) contain pointers, and all other pointers in the `next` array are `NULL`.

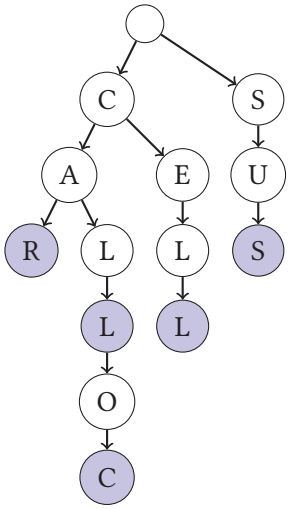


We want to insert "sus" in the trie. "sus" starts with a different character than the words so far, so we create new nodes for "s", "u", and "s", starting from the root node. Note that the second "s" corresponding to the last character of "sus" is shaded (`last` set to `true`).



We want to insert "cell" in the trie. "cell" starts with "c", but does not start with "ca—". We create new nodes for "—ell" and set `last` in the second "l" to be `true`.

Note that the "-ll-" in "calloc" and the "-ll" in "cell" are not shared, because the 2nd character of each word is different.



We want to insert "call" to our trie. "call" starts with the same letters as "calloc", so we don't need to add any nodes. However, even though the nodes are already there, there is nothing to indicate "call" is a word in the trie. We change this by updating the `last` field of the second "l" in "calloc" to be `true`. This indicates that "l" is now the end of a word.

For tries of `AlphaTrieNode`s, assume that:

1. Only lowercase letters (a-z) are supported. The ASCII values for these characters are [97,122], inclusive.
2. If multiple instances of the same string are inserted, it should not affect the trie.
3. The string argument `word` is a well-formatted string composed of only lowercase letters a-z.
4. The `node` argument is the root node of a properly initialized trie.

```
typedef struct AlphaTrieNode {
    bool last;
    struct AlphaTrieNode* next[26];
} AlphaTrieNode;

int main() {
    AlphaTrieNode* root = ... // instantiation of root (code not shown)

    // should insert "crewmate" into the trie
    insert(root, "crewmate");

    // should return false since "imposter" has not been added to the trie
    contains(root, "imposter");

    // should return true since "crewmate" is in the trie
    contains(root, "crewmate");

    return 0;
}
```

Implement the `contains` and `insert` functions below.

Q2.1 (5 points) `contains` takes in a trie root node (`node`), and a pointer to a string (`word`). It should return `true` if `word` is in the trie, and `false` otherwise.

```
1 bool contains(AlphaTrieNode* node, char* word) {
2     for (int i = 0; i < strlen(word); i++) {
3         int char_to_ascii = (int) word[i] - 97;
4         if (node->next[char_to_ascii] == NULL) {
5             return false;
6         }
7         node = node->next[char_to_ascii];
8     }
9     return node->last;
10 }
```

Q2.2 (8 points) `insert` takes in a trie root node (`node`), and a pointer to a string (`word`). It should insert the word into the trie.

```
1 void insert(AlphaTrieNode* node, char* word) {
2   for (int i = 0; i < strlen(word); i++) {
3     int char_to_ascii = (int) word[i] - 97;
4     if (node->next[char_to_ascii] == NULL) {
5       node->next[char_to_ascii] = calloc(1, sizeof(AlphaTrieNode));
6     }
7     node = node->next[char_to_ascii];
8   }
9   node->last = true;
10 }
```

Consider an alternate trie implementation that supports all 256 ASCII characters instead of just 26 lowercase characters. We define a new struct, `ASCIITrieNode`, as follows:

```
typedef struct ASCIITrieNode {
    bool last;
    struct ASCIITrieNode* next[256];
} ASCIITrieNode;
```

We would like to write a function that converts a trie of `AlphaTrieNode`s to a trie of `ASCIITrieNode`s. The function should also free all `AlphaTrieNode`s in the process. You may assume that all `AlphaTrieNode`s are properly initialized.

Below, we have 3 implementations of this conversion function. For each implementation, determine whether or not it is a valid implementation. If the implementation is not valid, please provide a brief explanation (10 words or fewer).

Q2.3 (4 points)

```
ASCIITrieNode* convert(AlphaTrieNode* node) {
    if (node == NULL) {
        return NULL;
    }
    ASCIITrieNode* new_node = malloc(sizeof(ASCIITrieNode));
    for (int i = 0; i < 26; i++) {
        new_node->next[i + 97] = convert(node->next[i]);
    }
    new_node->last = node->last;
    free(node);
    return new_node;
}
```

○ (A) Valid ● (B) Invalid

Solution: This implementation doesn't work because `malloc` allocates memory for the `ASCIITrieNode`, but does not initialize that memory. In the for loop, we only fill in 26 pointers in the `next` array of 256 pointers. The rest of the pointers are uninitialized (garbage), but they should be `NULL` since there are no nodes for the corresponding characters yet.

Q2.4 (4 points)

```
ASCIITrieNode* convert(AlphaTrieNode* node) {
    if (node == NULL) {
        return NULL;
    }
    ASCIITrieNode* new_node = calloc(1, sizeof(ASCIITrieNode));
    for (int i = 0; i < 26; i++) {
        new_node->next[i + 97] = convert(node->next[i]);
    }
    new_node->last = node->last;
    free(node);
    return new_node;
}
```

● (A) Valid ○ (B) Invalid

Solution: This implementation fixes the problem in the previous subpart by using `calloc`. Now, the pointers in the `next` array that aren't set by the for loop are initialized to `NULL` by `calloc`.

Q2.5 (4 points)

```
ASCIITrieNode* convert(AlphaTrieNode* node) {
    if (node == NULL) {
        return NULL;
    }
    ASCIITrieNode* new_node = realloc(node, sizeof(ASCIITrieNode));
    for (int i = 0; i < 256; i++) {
        new_node->next[i] = NULL;
    }
    for (int j = 0; j < 26; j++) {
        new_node->next[j + 97] = convert(node->next[j]);
    }
    return new_node;
}
```

○ (A) Valid ● (B) Invalid

Solution: This implementation is invalid in two different ways, depending on the behavior of `realloc`.

Suppose `realloc` does not change the pointer to `node`. In other words, `new_node` holds the same address as `node`). This means that the `next` array of pointers stays in the same place in memory, but grows in length. In this case, the first for loop sets all the elements in the `next` array to `NULL`. This means that when the second loop tries to read from the same `next` array, it doesn't read the pointers in the original node.

Suppose `realloc` does change the pointer in the process of reallocation. In other words, the memory at `node` is freed, a new, larger block of memory is allocated, and a pointer to this new memory is placed in `new_node`. In this case, we encounter the same problem as the `malloc` case, where not all the pointers in this new block of memory have been allocated.

Q3 *Can You Fix My Float?* **(20 points)**

Consider a 16-bit fixed point system with 1 sign bit, 5 integer bits, and 10 fraction bits. The five integer bits work just like a 5 bit unsigned integer. The 10 fraction bits continue where the integer bits left off, representing 2^{-1}, 2^{-2}, ..., 2^{-10}. For example, the bit representation `0b0 01101 1010000000` represents $(2^3 + 2^2 + 2^0) + (2^{-1} + 2^{-3}) = 13.625$.

In this question, we will compare this fixed-point system to a 16-bit floating point system that follows all conventions of IEEE-754 floating point numbers (including denorms, NaNs, etc.), with 5 bits of exponent and an exponent bias of -15.

Q3.1 (3 points) Write -22.375 in hex using the 16-bit floating point system described above.

Solution:
`0xCD98`

The floating point system in this question has 16 bits total, including 1 sign bit and 5 exponent bits. That leaves 10 bits for the mantissa.

First, write 22.375 in binary: `0b1 0110.011`

Then, move the binary point to create a number of the form `0b1.MM MMMM MMMM` $\times 2^E$. This gives us `0b1.01 1001 1000` $\times 2^4$.

The mantissa bits can be read directly from this form: `0b01 1001 1000`. Remember to drop the implicit 1 to the left of the binary point.

The exponent is -4, which we can convert to bias notation by subtracting the bias: $4-(-15) = 19$. In unsigned 5-bit binary, this is `0b10011`.

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

In total, we have `0b1 10011 01 1001 1000`, which is `0xCD98` in hex.

Q3.2 (3 points) Write -22.375 in hex using the 16-bit fixed point system described above.

Solution: `0xD980`

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

To get the integer bits, we can convert 22 to unsigned binary, which gives us `0b10110`.

To get the fractional bits, we can convert 0.375 to unsigned binary, which gives us `0b0.01 1000 0000`. The fractional bits of this number are `0b01 1000 0000`.

In total, we have `0b1 10110 01 1000 0000`, which is `0xD980`.

Q3.3 (3 points) How many numbers in the range $[16, 64)$ (including 16, excluding 64) can be represented by the floating point system described above?

Solution: One way to solve this question is to consider the step size, or distance between representable numbers, in this range.

Consider the number 16, which can be represented as `0b1.00 0000 0000` $\times 2^4$. The next-largest number is `0b1.00 0000 0001` $\times 2^4$. The distance between these two numbers is `0b0.00 0000 0001` $\times 2^4$ = `0b0.00 0001` = 2^{-6}.

This pattern of representable numbers being 2^{-6} apart continues all the way to `0b1.11 1111 1111` $\times 2^4$ = `0b1 1111.1111 1111` = $2^5 - 2^{-6}$, which is just under 32.

In total, in the range $[16, 32)$, we have a range of 16 with representable numbers spaced 2^{-6} apart. That's $16/2^{-6} = 2^4/2^{-6} = 2^{10}$ numbers that can be represented.

At this point, you can follow the same process to find the step size in the range $[32, 64)$. One shortcut is to note that step size in floating-point always increases by a factor of 2 (intuitively, you're losing one mantissa bit of precision as you reach higher numbers, so you end up skipping every other number that would have been representable). The range $[32, 64)$ is twice as large as $[16, 32)$, but the distance between representable numbers is also twice as large, so we end up getting another 2^{10} representable numbers in this range.

In total, we have $2^{10} + 2^{10} = 2^{11}$ representable numbers in the range $[16, 64)$.

Q3.4 (3 points) How many numbers in the range $[16, 64)$ (including 16, excluding 64) can be represented by the fixed point system described above?

Solution: 16 in the fixed point system is $0b10000.0000000000$. The largest representable number in the fixed point system is $0b11111.1111111111$, which is $2^5 - 2^{-10}$, which is just under 32.

In between these two numbers, we can choose every bit except the most-significant integer bit to be 0 or 1. That gives us 4 integer bits and 10 fractional bits that could all be 0 or 1. Each bit pattern represents a different number, so we have 2^{14} representable numbers in total.

Another way to compute this is to note that we have a range of 16, and the step size in this system is fixed to be 2^{-10} (changing the least-significant fractional bit causes the number to increase by 2^{-10}). This gives us $16/2^{-10} = 2^{14}$ representable numbers in total.

Q3.5 (4 points) What is the smallest positive number representable by the above fixed point system but not the above floating point system?

Express your answer as a sum or difference of powers of two (e.g. $2^4 - 2^2 + 2^{-1}$).

Solution: $2 + 2^{-10}$

As we saw in the previous part, the fixed-point system can represent all numbers from 0 to $2^5 - 2^{-10}$, with representable numbers spaced 2^{-10} apart.

One way to solve this is to note that the step size in floating point numbers (distance between representable numbers) increases as the numbers increase as well. This means that we're looking for the first point where the floating point numbers start skipping by more than 2^{-10} at a time.

Given a floating point number of the form `0b1.MM MMMM MMMM` $\times 2^E$, the step size is 2^{-10} when E is 0. When E increases to 1, the numbers become of the form `0b1.MM MMMM MMMM` $\times 2^1$ = `0b1M.M MMMM MMMM`. At this point, consecutive numbers are 2^{-9} apart, so there will be numbers that fixed-point can represent but floating-point cannot. The smallest such number following this pattern is `0b10.0 0000 0000` = 2. This is representable by floating-point (e.g. `0b1.00 0000 0000` $\times 2^1$. However, the next number in the fixed-point system, $2 + 2^{-10}$, is not representable by floating point. The next number representable by floating point is $2 + 2^{-9}$ (since the step size has increased).

Q3.6 (4 points) What is the largest positive number representable by both systems described above?

Express your answer as a sum or difference of powers of two (e.g. $2^4 - 2^2 + 2^{-1}$).

Solution: $2^5 - 2^{-6}$

We know the largest number representable by the fixed-point system is $2^5 - 2^{-10}$. We also know that the step size for the fixed-point system has a step size of 2^{-10}.

From Q3.3, we know that around 2^5, the step size of the floating-point number is 2^{-6}.

The floating-point number has a larger step size (larger gaps between numbers), so the largest number representable by both systems is going to be limited by floating-point. (In other words, in this area, the fixed-point system can represent some numbers that floating point can't. All floating point numbers in this area can also be represented in fixed-point, though.)

The largest floating-point number under $2^5 - 2^{-10}$ is $2^5 - 2^{-6}$, which is the largest number representable by both systems.

This page intentionally left (mostly) blank.
Please do not tear off any pages from the exam.

Q4 *Even Stevens* **(25 points)**

You are given a list of numbers to add up by your math professor. However, your professor doesn't like odd numbers, so they ask that you only add up the even ones.

The function `add_even_numbers` is defined as follows:

- Inputs:
  - `a0`: the address of the start of an array of 32-bit integers
  - `a1`: the number of integers in the array
- Output:
  - `a0`: the sum of all even numbers in the array

Example: Suppose input `a0` points to `[4, 5, 6, 7]`, and input `a1` holds `4`. Then output `a0` holds `10` $(4 + 6)$.

Q4.1 (15 points) Fill in the blanks in the RISC-V code below. You may not need all the blanks. Each line should contain exactly one instruction or pseudo-instruction.

```
add_even_numbers:
    addi t0, x0, 0      # set t0 to be the running sum
loop:
    beq a1 x0 end
    lw t1 0(a0)         # set t1 to be the number in the array
    andi t2 t1 1
    beq t2 x0 pass
    add t0 t0 t1
pass:
    addi a0 a0 4
    addi a1 a1 -1
    j loop
end:
    ret
```

Alternate Solution:

```
add_even_numbers:
    addi t0, x0, 0      # set t0 to be the running sum
loop:
    beq a1, x0, end
    lw t1 0(a0)         # set t1 to be the number in the array
    srli t2, t1, 1
    slli t2, t2, 1
    bne t1, t2, pass
    add t0, t0, t2
pass:
    addi a1, a1, -1
    addi a0, a0, 4
    j loop
end:
    mv a0, t0
```

Q4.2 (5 points) Translate the `j loop` instruction under the `skip` label to hexadecimal. Assume that every line in the above code is filled with exactly one instruction (or pseudo-instruction that expands to one instruction).

Solution: `0xFDDFF06F`

Optionally, for partial credit, write the offset in bytes as a decimal number in the box below.

Solution: -36
The line of code labeled `loop` is 9 instructions before the `j loop` instruction.

Q4.3 (5 points) Suddenly, your professor starts hating prime numbers, so now they only want you to sum up the non-prime numbers.

Assume you are given a function `is_prime` that follows calling convention. What combination of modifications to the `add_even_numbers` function is needed in order to sum up all the non-prime numbers in the array? Select all that apply.

- ☐ (A) Use another register to track the number of times `is_prime` is called
- ■ (B) Replace the code used to check if the number is even with a call to `is_prime`
- ■ (C) Decrement the stack pointer by some amount at the start of the function, and increment the stack pointer by the same amount at the end of the function
- ☐ (D) Save some values in `a` registers instead of `t` registers
- ■ (E) Save some values in `s` registers instead of `t` registers
- ☐ (F) Save used `a` registers onto the stack at the beginning of the function
- ■ (G) Save used `s` registers onto the stack at the beginning of the function
- ☐ (H) Save used `t` registers onto the stack at the beginning of the function
- ■ (I) Save another register (besides the `a`, `s`, or `t` registers) onto the stack at the beginning of the function
- ■ (J) Restore at least one register from the stack at the end of the function
- ☐ (K) None of the above

Solution: (A): Calling convention doesn't need you to keep track of how many times a function is called. The functionality of this program also doesn't require you to keep track of how many times the `is_prime` function is called.

(B): This is unrelated to calling convention, btu we do need to change the even number check to a prime number check.

For the rest of the answer choices, there are two approaches we can take to follow calling convention. One approach is to save all volatile registers (t and a registers) on the stack before the `is_prime` function call, then restore the values back into those registers after the function returns. However, this question doesn't have any answer choices corresponding to this approach, so we need to take the other approach.

The second approach is to use `s` registers, which don't get changed by the `is_prime` function call, instead of `t` registers, which might get changed by the `is_prime` function call. We can't use `a` registers instead of `t` registers because `a` registers could get modified by the function call too. This is why option (E) is true and option (D) is false.

To use `s` registers, which our own `add_even_numbers` function is not allowed to modify, we need to save the values in the `s` on the stack at the beginning of the function, then restore them at the end of the function. This is why option (G) is true, options (F) and (H) are false, and option (J) is true.

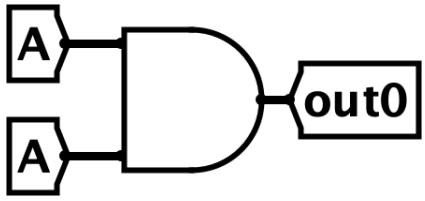
Finally, because we call a function `is_prime` within our function `add_even_numbers`, the return address in `ra` will get modified. This means that at the end of `add_even_numbers`, we won't know where `add_even_numbers` should return to, because `ra` was overwritten to hold the place where `is_prime` should return to. To solve this, we save the value of `ra` on the stack at the beginning of the function, and restore its value at the end of the function. This is why option (I) is true.

Q5 *TF?* **(10 points)**

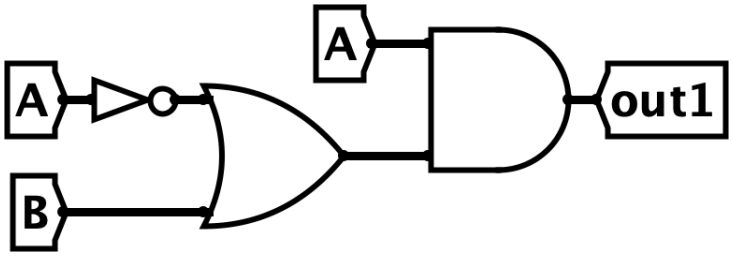
Simplify the Boolean logic for the following circuits. Your answer may only use the following characters:

Character	Description
`A, B, C, D`	Inputs
*	AND
+	OR
^	XOR
~	NOT
`( )`	Parentheses
`0, 1`	Constants

Example: the following circuit, which has Boolean logic `A * A`, can be simplified to `out0 = A`.



Q5.1 (2 points)



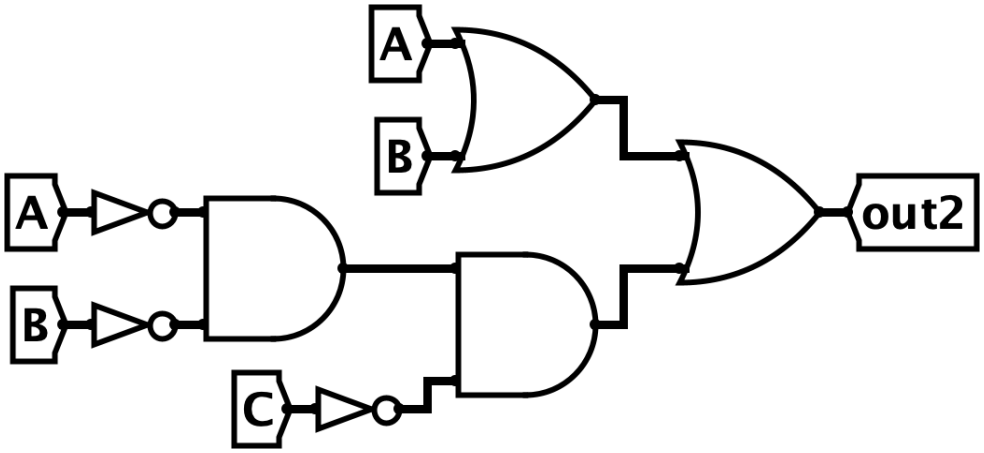
Solution: `A * (~A + B)` (direct translation of circuit)

~~`A * ~A`~~ `+ A * B`

`0 + A * B`

`A * B`

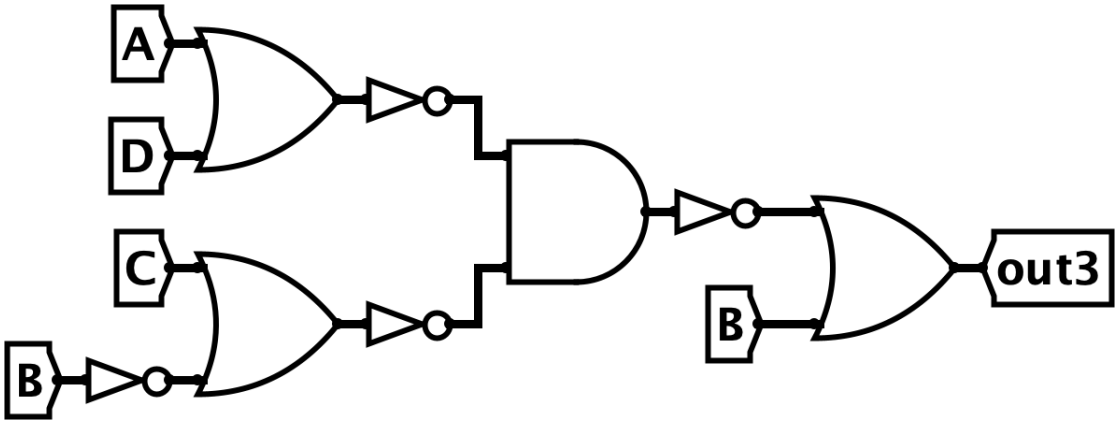
Q5.2 (4 points)



Solution: `(A + B) + ((~A * ~B) * ~C)` (direct translation of circuit)

```
(A + B) + (~(A + B) * ~C)
A + B + ~C
```

Q5.3 (4 points)



Solution: `~(~(A + D) * ~(C + ~B)) + B`

```
(A + D) + (C + ~B) + B
A + D + C + (B + ~B)
1
```

Nothing on this page will be graded.

Is there anything you want us to know?