McMahon, Weaver
Spring 2022

CS 61C

Midterm

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

PRINT your student ID: ____

Solutions last updated: Sunday, March 6, 2022

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

Question:	1	2	3	4	5	Total
Points:	12	18	18	12	**????**	60

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 and a corresponding notch on Nick's Stanley Fubar demolition tool.

SIGN your name: ____

Q1 *True/False* **(12 points)**

Q1.1 (1.5 points) True or False: If you wanted to store the integer `0xDEADBEEF` in a little-endian system in C, you would have to write `int x = 0xEFBEADDE;`

○ True ● False

Solution: False; You'd write `int x = 0xDEADBEEF;`. One way to see this is that we write `int x = 1;` to store the value meaning 1.

Q1.2 (1.5 points) True or False: When possible, the C compiler by default attempts to store data at aligned addresses (ex. 4 byte objects stored at an address that is a multiple of 4), even if it creates "gaps" of unused memory.

● True ○ False

Solution: True; this allows for faster memory accesses (we'll discuss this in further detail in the caches section), which tends to be worth the tradeoff of using slightly more memory.

Q1.3 (1.5 points) True or False: The compiler converts code written in a higher-level language like C into a lower-level language like RISC-V.

● True ○ False

Solution: True; the job of the compiler is to translate low-level code into even lower level assembly to prepare the rest of CALL in generating an executable.

Q1.4 (1.5 points) True or False: The symbol and relocation tables are discarded after the assembler runs, since all labels get converted into byte offsets.

○ True ● False

Solution: False; The linker needs the symbol and relocation table to link labels and functions from different files. It gets discarded after that and no longer exists in the executable.

Q1.5 (1.5 points) True or False: It is possible to use 9 bits to represent 513 unique values.

○ True ● False

Solution: False; $2^9 = 512$, so only 512 unique bitstrings exist. If we had a system that represented 513 unique values, then by the pigeonhole principle, at least one bitstring would have to represent two different values. This is not possible.

Q1.6 (1.5 points) TRUE or FALSE: Typically, signed integers are stored in sign-magnitude representation in order to simplify arithmetic operations performed on these numbers.

○ TRUE ● FALSE

Solution: False; Signed numbers are stored with two's complement, because it makes addition and multiplication simpler.

Q1.7 (1.5 points) TRUE or FALSE: All base RISC-V 32-bit instructions share the same two least significant bits.

● TRUE ○ FALSE

Solution: True; This is actually a design decision in RISC-V opcodes, and is used to signify 32-bit instructions; this also helps add a nice checksum that the random data you're looking at is indeed RISC-V code. This fact can be verified by checking the RISC-V reference card.

Q1.8 (1.5 points) TRUE or FALSE: Branch instructions can represent a larger immediate value than I-type instructions.

● TRUE ○ FALSE

Solution: True; Branch instructions encode 12 bits worth of immediate, but we include an implicit 0th index bit of 0, bringing up the immediate to be 13 bits. I-type instructions encode only 12 bits, without any implicit bits.

Q2 *Short Answer* (18 points)

Q2.1 (3 points) Convert -12 to an 8-bit two's complement representation.

Express your answer in binary, including the relevant prefix.

Solution: `0b1111 0100`; $12 \longrightarrow$ `0b0000 1100`. To convert to two's complement, we flip the bits, resulting in `0b1111 0011`, then add one, which gets us `0b1111 0100`.

Q2.2 (3 points) Convert $2^{32} - 15$ to a 32-bit unsigned representation.

Express your answer in hexadecimal, including the relevant prefix.

Solution: `0xFFFF FFF1`. Note that $2^{32} - 1$ is `0xFFFF FFFF`, so we subtract 14 more from this to get $2^{32} - 15$.

Alternatively, we can use the equivalence in 2's complement to note that the binary for unsigned $2^{32} - 15$ is the same as the binary for 2's complement -15.

For the following three subparts, assume that we are working with a binary floating point representation, which follows IEEE-754 standard conventions, but which has 3 exponent bits (and a standard exponent bias of -3) and 4 significand bits.

Q2.3 (3 points) Convert -12 to its floating point representation under this floating point system.

Express your answer in binary, including the relevant prefix.

Solution:

$12 =$ `0b1100` = `0b`1.1000×2^3

Significand = `0b`1000

Exponent = $3 - (-3) = 6 =$`0b`110

Sign bit = 1 (negative)

`0b`11101000 $\longrightarrow$ `0b`11101000

Q2.4 (3 points) What is the largest non-infinite number that can be represented by this system?

Express your answer in decimal.

Solution:

Largest significand = 0`b`1111

Largest exponent = 0`b`$110 = 6 + (-3) = 3$

`0b`$1.1111 \times 2^3 =$`0b`$1111.1 = 15.5$

Q2.5 (3 points) What is the smallest positive number that can be represented by this system?

Express your answer as an odd integer multiplied by a power of 2.

Solution:

Smallest significand = `0b0001`

Smallest exponent = `0b`$000 = 0 + (-3) + 1 = -2$

$0.0001 \times 2^{-2} = 1 \times 2^{-6}$

Q2.6 (3 points) Translate the following RISC-V instruction into its corresponding hexadecimal value.
`ori t6 s0 -12`

Solution: `FF446793`

`opcode` = `0b001 0011`

`funct3` = `0b110`

`rd` = `t6` = `x31` = `0b11111`

`rs1` = `s0` = `x8` = `0b01000`

`imm` = -12 = `0b1111 1111 0100`

`0b111111110100 01000 110 11111 0010011`

`0b1111 1111 0100 0100 0110 1111 1001 0011`

`0xFF44 6F93`

Q3 *Trouble With Definitions* (18 points)

Note: we think this is the trickiest question on the exam.

Define statements can be useful, but it's important to be careful when using them.

```c
#include <stdio.h>
#include <stdlib.h>
#define abs(x) ((x) < 0 ? -(x) : (x))
#define f(a,b) a*b/4
int main() {
    int a = 10;
    printf("Question 3.1: %d\n", a^2);
    int i = 0xA6004F4E;

    printf("Question 3.2: 0x%X\n", i|(i<<4));
    printf("Question 3.3: 0x%X\n", abs(i));

    int b = 10;
    printf("Question 3.4: %d\n", f(0+1, b));
    printf("Question 3.5: %d\n", f(1+0, b));

    int k = 100;
    int* kptr = &k;
    printf("Question 3.6: %d\n", f(k+,kptr));
    return 0;
}
```

The `%d` format modifier outputs an integer in decimal. The `%X` format modifier outputs an integer as a hexadecimal string, using capital letters for `A-F`.

This code compiles. What is printed by this code? Please write your answers in the answer boxes provided on the next page.

Each line is worth 3 points.

Solution: Question 3.1: 8

Note that ˆ in C is XOR, not exponentiation!

10 ˆ 2 = `0b1010` ˆ `0b0010` = `0b1000` = 8

Solution: Question 3.2: `0xE604 FFEE`

First, compute the left-shift by 4:

`i = 0b1010 0110 0000 0000 0100 1111 0100 1110`

`i << 4 = 0b0110 0000 0000 0100 1111 0100 1110 0000 = 0x6004 F4E0`

Another way to perform this left-shift is to note that 4 bits = 1 nibble = 1 hex digit, so we can shift the hex number left by 1 digit.

Next, compute the bitwise OR:

```
0xA600 4F4E = 0b1010 0110 0000 0000 0100 1111 0100 1110

0x6004 F4E0 = 0b0110 0000 0000 0100 1111 0100 1110 0000

              0b1110 0110 0000 0100 1111 1111 1110 1110 = 0xE604 FFEE
```

Solution: Question 3.3: `0x59FFB0B2`

`A6004F4E = 0b1010 0110 0000 0000 0100 1111 0100 1110`

This is a negative number, so the absolute value negates it into a positive number. We can negate the number by flipping the bits and adding 1.

```
0b1010 0110 0000 0000 0100 1111 0100 1110
0b0101 1001 1111 1111 1011 0000 1011 0001
0b0101 1001 1111 1111 1011 0000 1011 0010
0x59FFB0B2
```

Solution: Question 3.4: 2

`#define` statements are effectively find-and-replaces. That causes the equation to become `0+1*b/4`, with `b = 10` which evaluates to `0+1*10/4` $= 10/4 = 2$. The result is rounded down because we're working with integers;

Solution: Question 3.5: 1

`#define` statements are effectively find-and-replaces. That causes the equation to become `1 + 0 * b / 4`, with $b = 10$ when substituting, which evaluates to `1 + 0 * 10 / 4` $= 1$.

Solution: Question 3.6: 125

The realization here is that `*`, previously used as the multiply operator, is now used as the dereference operator. After substitution, we get `k+*kptr/4`, which evaluates to $k + k/4 = 100 + 100/4 = 125$.

Question author's note: When writing this question, we discovered that defines aren't actually pure find-and-replaces; for example, when doing `abs(-j)`, a pure find-and-replace would yield `-j < 0 ? --j : -j`; the two negative signs would become the preincrement operator. The original version of this question tried to use this, but when tested on `gcc`, that line got treated as two unary negatives instead. This suggests that the preprocessor works after the lexer of the compiler (after the code has been divided into tokens). This is beyond 61C's scope, so if this comment doesn't make sense, that's totally okay; you're not expected to know it.

Q4 *Lost in Translation* **(12 points)**

Consider the following Python class:

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def transform(self, f):
        return Vector(f(self.x), f(self.y))
```

Q4.1 (20 points) We want to translate this code to C. Fill in the following C code. Assume all allocations succeed. For full credit, your solution must use the minimum amount of memory required.

```c
#include <stdlib.h>

typedef struct Vector {
    int x;
    int y;
} Vector;

____________________ *transform(Vector *self, int (*f)(int)) {

    ________________ newVector = ______________________________;

    newVector ______ x = _________________________;

    newVector ______ y = _________________________;

    return _________________________;
}
```

Solution:

```c
Vector *transform(Vector *self, int (*f)(int)) {
    Vector* newVector = malloc(sizeof(Vector));
    newVector -> x = f(self->x);
    newVector -> y = f(self->y);
    return newVector;
}
```

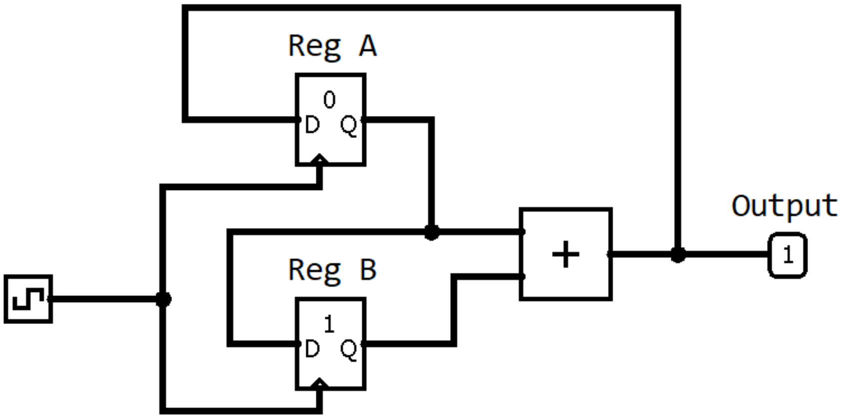
Q4.2 (20 points) Translate the `transform` function to RISC-V. The function takes inputs `self` in `a0` and `f` in `a1`, and returns output in `a0`. You may assume that `Vector` is as defined in the C code. You may also assume that you have access to `malloc`, and that `malloc` and `f` each receive their argument in `a0`, and return their result in `a0`. Your solution MUST fit within the lines provided.

```
transform:
   addi sp sp ____________________

   ______________________________

   ______________________________

   ______________________________

   ______________________________

   ______________________________

   ______________________________

   ______________________________

   jal ___________________ malloc

   ______________________________

   ______________________________

   jalr _________________________

   ______________________________

   ______________________________

   jalr _________________________

   ______________________________

   ______________________________

   ______________________________

   ______________________________

   ______________________________

   ______________________________

   addi sp sp ____________________
   ret
```

Solution:

```
transform:
    addi sp sp -16
    sw ra, 0(sp)
    sw s0, 4(sp)
    sw s1, 8(sp)
    sw s2, 12(sp)
    mv s0, a0
    mv s1, a1
    li a0, 8
    jal ra malloc
    mv s2, a0
    lw a0, 0(s0)
    jalr ra, s1, 0
    sw a0, 0(s2)
    lw a0, 4(s0)
    jalr ra, s1, 0
    sw a0, 4(s2)
    mv a0, s2
    lw ra, 0(sp)
    lw s0, 4(sp)
    lw s1, 8(sp)
    lw s2, 12(sp)
    addi sp sp 16
    ret
```

Q4.3 Consider the following circuit:



All data wires (wires not connected to the clock) are 8 bits wide.

Q4.1 (8 points) Assume that the circuit is in the above state at clock cycle 0; register A is currently storing 0, register B is currently storing 1, and the circuit is outputting 1. **For this part only**, assume that the clock period is significantly longer than any propagation delays and register setup/hold/clk-to-q time. Write the outputted values (in decimal) from clock cycles 1 to 8.

Cycle 1 Cycle 2 Cycle 3 Cycle 4

Cycle 5 Cycle 6 Cycle 7 Cycle 8

Solution: 1, 2, 3, 5, 8, 13, 21, 34

After the first clk-to-q time during clock cycle 0, Q of A is 0, and Q of B is 1. The sum outputted is 1, which gets fed back to RegA to be used for clock cycle 1. The next value taken in for RegB is the **previous** value outputted from Q by RegA. For the next clock cycle, the value of RegA becomes the value of the output from the previous cycle (1) and the value of RegB becomes the output of RegA from the previous cycle (0), so at clock cycle 1, the adder adds together values 0 (from RegA) and 1 from (RegB) and outputs 1. This cycle continues:

Clock	RegA	RegB	Output
0	0	1	1
1	1	0	1
2	1	1	2
3	2	1	3
4	3	2	5
5	5	3	8
6	8	5	13
7	13	8	21
8	21	13	34

Q4.2 (4 points) Assume that the circuit has the following delays:

Register clk-to-q time	3ns
Register setup time	2ns
Register hold time	1ns
Adder propagation delay	4ns

Wires are assumed to have no propagation delay. What is the minimum clock period needed for this circuit to have the same behavior as in Q5.1?

Solution: 9 ns
The longest path between sequential logic blocks (blocks that depend on the clock; in this case, just the registers) is the path from the output of Register B, through the adder gate, and into the input of Register A.
How long does it take for a signal to travel through this longest path? From the positive edge of the clock, we have to wait 3 ns (clk-to-q time) for Register B's input to appear at its output. Then, we have to wait 4 ns (adder delay) for the signal to travel through the adder. Finally, when the signal arrives at the input Register A, we have to wait 2 ns (setup time) before the next positive edge of the clock. In total, our shortest clock period is 3 + 4 + 2 = 9 ns.

(Optional) The Finish Line

(0 points) You've reached the end of the exam! If there's anything you'd like to tell course staff, let us know here!

(0 points) What are their names?

Their names are EvanBot (from 161) and CodaBot!

(0 points) What else are they selling? (fill in the sale table)

This page intentionally left with only one sentence.