McMahon, Weaver
Spring 2022

CS 61C Final

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

PRINT your student ID: ____

You have 170 minutes. There are 8 questions of varying credit (100 points total).

Question:	1	2	3	4	5	6	7	8	Total
Points:	6	8	30	10	11	11	9	15	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 and a corresponding notch on Nick's Stanley Fubar demolition tool.

SIGN your name: ____

Q1 *True/False* (6 points)

Q1.1 (1 point) True or False: All RISC-V instructions are 4 bytes long, which is why the immediates for B-type and J-type instructions each have two implicit 0s.

○ True ○ False

Q1.2 (1 point) True or False: Most computers natively use a little-endian data encoding.

○ True ○ False

Q1.3 (1 point) True or False: The last step of the CALL process, the loader, is part of the operating system and is responsible for correctly starting a program.

○ True ○ False

Q1.4 (1 point) True or False: System calls (such as the `ecall` instruction in RISC-V) are executed by the program itself.

○ True ○ False

Q1.5 (1 point) True or False: Even with a direct memory access (DMA) controller, the CPU is required to initiate a memory transfer.

○ True ○ False

Q1.6 (1 point) True or False: In warehouse scale computing, water can be used as an alternative to air to cool machines and infrastructure.

○ True ○ False

Q2 *Short Answer* (8 points)

Q2.1 (2 points) Translate the following RISC-V instruction into its corresponding 32-bit hexadecimal value.

`sb t3, 31(t3)`

0x

Q2.2 (2 points) Simplify the following Boolean expression. For full credit, your solution must use at most 3 boolean operators (OR, AND, NOT).

$!((A+!A)B + (CD)) + CD$

The DOGGO 61C midterm consists of short-answer and long-answer questions. Grading all submissions for the midterm is done in two steps.

First, the midterm is preprocessed by the **exam TAs**. It would take 1 hour to sequentially preprocess all submissions, but this is infinitely parallelizable.

Then, once preprocessing is done, all TAs (including **exam TAs**) participate in grading. Each question must be graded fully by a single TA (to maintain consistent grading over all submissions). For example, if Justin grades Q1 for a submission, he must grade Q1 for all submissions; no other TA may grade Q1 for any other submission.

Question type	Number of questions	Time to grade one question across all submissions
Short-answer	40	50 minutes
Long-answer	10	250 minutes
Total: 4500 TA-minutes		

Q2.3 (1 point) This class has 10 TAs, 5 of whom are exam TAs.

In the best-case scenario, how many minutes would it take us to grade all submissions? Round your answer to the nearest minute, if needed.

Q2.4 (1 point) Noting that this would take too long, we decide instead to hire all 7 billion people on Earth as TAs to help grade the midterm. As before, we still have only 5 exam TAs.

In the best-case scenario, how many minutes would it now take us to grade all submissions? Round your answer to the nearest minute, if needed.

Q2.5 (2 points) One of the hive machines shuts down unexpectedly. After investigating, the TAs conclude that during the past 100 days, this hive machine encountered 6 failures, and each failure took 12 hours to repair. What is the availability of this hive machine over the last 100 days? Express your answer as a reduced fraction.

Q3 *Decisions, Decisions* **(30 points)**

For each of the following pairs of alternatives, write one advantage that each alternative has over the other, or "none" if none exists.

These questions are open-ended and don't necessarily have one correct answer. Credit will be awarded for reasonable responses, even if it doesn't match the staff solution. One sentence per box is sufficient for full credit.

Q3.1 (3 points)	
A 32-bit IEEE-754 standard floating point number	
A 32-bit two's complement integer	

Q3.2 (3 points)	
C	
Python	

Q3.3 (3 points)	
A dynamically linked library	
A statically linked library	

Q3.4 (3 points)	
A RAID-0 array with 5 2TB disks	
A RAID-5 array with 5 2TB disks	

Q3.5 (3 points)	
Polling	
Interrupts	

In each of the following subparts, two implementations for the given behavior are provided. For each pair of implementations, select which implementation is **incorrect** (does not match the given behavior), or if **neither is incorrec**t (both match the given behavior). It is guaranteed that at least one of each pair is correct.

- If you select that one implementation is **incorrect**, provide an example where it fails to produce the expected behavior.
- If you select that **neither is incorrect**, write one advantage for each implementation. If an implementation has no advantage over the other, write "none".

As before, these questions don't necessarily have one correct answer. Credit will be awarded for reasonable responses, even if it doesn't match the staff solution. 1-2 sentences per box is sufficient for full credit.

Q3.6 (3 points) Given behavior: A function that returns a string `"Hello"`.

A	B

```
char* returnHello() {
    return "Hello";
}
```
	```
	char* returnHello() {
	    // Assume malloc succeeds.
	    char* data = malloc(sizeof(char) * 6);
	    // Correctly copies the string to data,
	    // including NULL terminator.
	    strncpy(data, "Hello", 6);
	    return data;
	}
	```

○ A is incorrect ○ B is incorrect ○ Neither is incorrect (i.e. both are correct)

Q3.7 (3 points) Given behavior: A function that takes the absolute value of a number.

A	B

`#define abs(x) x < 0 ? -x : x`	`int abs(int x) {`
	`    return x < 0 ? -x : x;`
	`}`

○ A is incorrect ○ B is incorrect ○ Neither is incorrect (i.e. both are correct)

Q3.8 (3 points) Given behavior: A function to return the square of a number.

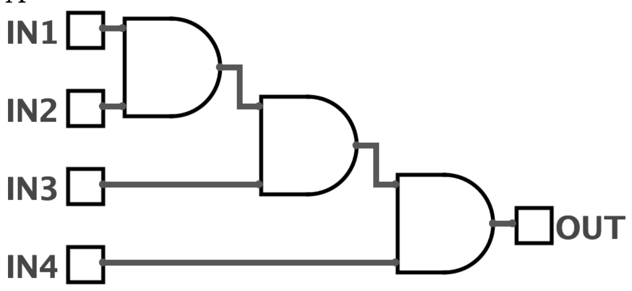
A	B

`square:`
`    mul t0 a0 a0`
`    mv a0 t0`
`    ret`
	`square:`
	`    mul s0 a0 a0`
	`    mv a0 s0`
	`    ret`

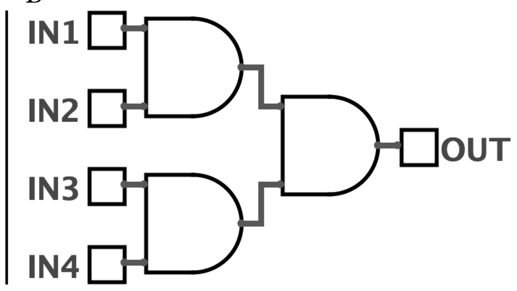
○ A is incorrect ○ B is incorrect ○ Neither is incorrect (i.e. both are correct)

Q3.9 (3 points) Given behavior: A circuit that takes the logical AND of 4 inputs.

A



B



○ A is incorrect ○ B is incorrect ○ Neither is incorrect (i.e. both are correct)

Q3.10 (3 points) Given behavior: A function that sets an array (of the given length) to all 0s.

A	B

```
#define length (1 << 20)
// Assume correct imports
void setToZero(int* data) {
  omp_set_num_threads(8);
  #pragma omp parallel
  {
    int i = omp_get_thread_num();
    for (; i < length; i += 8) {
      data[i] = 0;
    }
  }
}
```
	```
	#define length (1 << 20)
	// Assume correct imports
	void setToZero(int* data) {
	  // The following whitespace is
	  // intentional.

	  int i = 0;
	  for (; i < length; i++) {
	    data[i] = 0;
	  }
	}
	```

○ A is incorrect ○ B is incorrect ○ Neither is incorrect (i.e. both are correct)

Q4 ***Deci-sion*** **(10 points)**

So far in this class, we have discussed IEEE-754 standard binary floating point numbers. IEEE-754 also specifies a separate decimal floating point standard, often used in financial calculations. The exact storage method of this standard is fairly complicated, so for now, we will consider a simplified version of this standard, without regards to the underlying binary value:

A decimal floating point number stores 1 sign bit, an exponent, and a significand:

The exponent is an integer in $[-95, +96]$. The significand is a 7 digit decimal integer. If the stored exponent is X and the significand is $ABCDEFG$ (where A-G are digits between 0 and 9 inclusive), then the represented value is:

$(-1)^{\text{sign\_bit}} \times A.BCDEFG \times 10^X$

For example, if the sign is positive, the exponent is 8, and the significand is 0005256, the represented value is $0.005256 \times 10^8 = 525600$. This system does **not** have an implicit one or denormalized numbers. For the purposes of this question, we will not consider representations of infinities and NaNs.

Q4.1 (3 points) What problem would occur if our floating point system used an implicit one (i.e. represented $1.ABCDEFG \times 10^X$ instead)? Write your answer in 10 words or fewer.

Q4.2 (2 points) Find two distinct representations of the number 61 in this decimal floating point system (give the exponent and significand).

Exponent 1:	Significand 1:
Exponent 2:	Significand 2:

Q4.3 (3 points) How many distinct representations exist for 0 in this system?

Q4.4 (2 points) Which of the following numbers can be represented exactly by our decimal floating point standard, but not by the IEEE-754 standard single precision (32 bit) floating point standard? Select all that apply.

☐ 0.1 ☐ 0.8 ☐ $2^{32} = 4{,}294{,}967{,}296$

☐ $\frac{1}{3} = 0.333333...$ ☐ 1 ☐ 10^{90}

☐ 0.5 ☐ 10^6 ☐ None of these

Q5 *Push-Pops* **(11 points)**

We want to implement two new instructions:

`push rs2`: Put the word stored in `rs2` on the stack, decrementing the stack pointer as necessary.

`pop rd`: Set `rd` to the bottom-most word of the stack, and remove that element from the stack.

It is undefined behavior to call `push sp` or `pop sp`.

We first decide to implement `push` and `pop` as pseudoinstructions.

Q5.1 (2 points) Write a two-instruction sequence that implements `push rs2`.

Your instructions can use `rs2`.

Q5.2 (2 points) Write a two-instruction sequence that implements `pop rd`.

Your instructions can use `rd`.

Instead of implementing them as pseudoinstructions, we decide to modify our datapath to make them proper instructions that run in one cycle of a single-cycle datapath.

For this question, we will make modifications to the RISC-V datapath included in the CS 61C Reference Card. In order to implement `push` and `pop`, we decide to modify our immediate generator so that setting the `ImmSel` control signal to 7 always outputs an immediate with value 4.

Q5.3 (3 points) What further changes would we need to make to our datapath in order for us to implement the **`push`** instruction with as few changes as possible? Select all that apply:

- ☐ Add a new instruction format
- ☐ Add a second new immediate type for the ImmGen
- ☐ Add a new output to Regfile for a third register value
- ☐ Add a second writeback input to Regfile, along with a second `RegWEn`
- ☐ Add a new input to AMux, with the relevant selectors
- ☐ Add a new input to BMux, with the relevant selectors
- ☐ Add a new ALU input for a third register input
- ☐ Add a new ALU operation, with a corresponding `ALUSel`
- ☐ Add a mux before the DMEM address input, including relevant inputs and selectors
- ☐ Add a new input to WBMux, with the relevant selectors
- ☐ No further changes are needed to the datapath, beyond adding additional control logic

Q5.4 (3 points) What further changes would we need to make to our datapath in order for us to implement the **`pop`** instruction with as few changes as possible? Select all that apply:

- ☐ Add a new instruction format
- ☐ Add a second new immediate type for the ImmGen
- ☐ Add a new output to Regfile for a third register value
- ☐ Add a second writeback input to Regfile, along with a second `RegWEn`
- ☐ Add a new input to AMux, with the relevant selectors
- ☐ Add a new input to BMux, with the relevant selectors
- ☐ Add a new ALU input for a third register input
- ☐ Add a new ALU operation, with a corresponding `ALUSel`
- ☐ Add a mux before the DMEM address input, including relevant inputs and selectors
- ☐ Add a new input to WBMux, with the relevant selectors
- ☐ No further changes are needed to the datapath, beyond adding additional control logic

Q5.5 (1 point) Assuming the above changes were implemented, what hazards can be caused by `push` if we use the standard 5-stage pipeline included in the CS 61C Reference Card?

- ○ Data hazard
- ○ Control hazard
- ○ Both data and control hazards
- ○ Neither data nor control hazard

This page intentionally left (mostly) blank.

Please do not tear off any pages from the exam.

Q6 *Fun with Primes* **(11 points)**

The Sieve of Eratosthenes is an algorithm that can be used to determine a list of prime numbers. A full explanation of the algorithm is provided in the Detailed Explanation box below.

The below function implements this Sieve, and is defined as follows:

Input: An integer n. You may assume that $n > 0$.

Output: An array of n 1-byte integers. The ith element of this list is 0 if i is prime, and 1 otherwise.

```
char* primelist(uint32_t n) {
    char* primes = <BLANK 1>;
    if (<BLANK 2>) return NULL;
    primes[0] = 1;
    primes[1] = 1;
    for (int i = 2; i * i < n; i++) {
        if (!primes[i]) { // If i is a prime number, ...
            for (int j = 2 * i; j < n; j += i) {
                primes[j] = 1; // Cross out all multiples of i
            }
        }
    }
    return primes;
}
```

Detailed Explanation:

For each index of the array, we check to see if this index is a prime number. If it is not a prime number, we move to the next index. If it is a prime number, we know that every number that is a multiple of this number is not prime, so we mark the corresponding indices as not prime. An example of running this algorithm with input $n = 12$ is shown below:

Start	$\cancel{0}$	$\cancel{1}$	2	3	4	5	6	7	8	9	10	11	0 and 1 are nonprime
`i = 2`	$\cancel{0}$	$\cancel{1}$	2	3	$\cancel{4}$	5	$\cancel{6}$	7	$\cancel{8}$	9	$\cancel{10}$	11	Cross out all multiples of 2
`i = 3`	$\cancel{0}$	$\cancel{1}$	2	3	$\cancel{4}$	5	$\cancel{6}$	7	$\cancel{8}$	$\cancel{9}$	$\cancel{10}$	11	Cross out all multiples of 3
`i = 4`	$\cancel{0}$	$\cancel{1}$	2	3	$\cancel{4}$	5	$\cancel{6}$	7	$\cancel{8}$	$\cancel{9}$	$\cancel{10}$	11	4 isn't prime, continue
`i = 5`	$\cancel{0}$	$\cancel{1}$	2	3	$\cancel{4}$	5	$\cancel{6}$	7	$\cancel{8}$	$\cancel{9}$	$\cancel{10}$	11	Cross out all multiples of 5
`i = 6`	$\cancel{0}$	$\cancel{1}$	2	3	$\cancel{4}$	5	$\cancel{6}$	7	$\cancel{8}$	$\cancel{9}$	$\cancel{10}$	11	6 isn't prime, continue
…													

When we actually implement it in C code, we use 0 to represent prime and 1 as nonprime, so the array itself looks like this after running the full program:

`i`	0	1	2	3	4	5	6	7	8	9	10	11
`primes[i]`	1	1	0	0	1	0	1	0	1	1	1	0

Note that in the above example, no changes occur after the `i = 3` iteration. In the code above, we only iterate up to the $\sqrt{n}$th iteration of the loop (because every composite number has at least one factor less than or equal to its square root).

This final array shows that the numbers 2, 3, 5, 7, and 11 are prime.

Q6.1 (4 points) What lines of code should be written in the given blanks in the C code?

`<BLANK 1>`:

`<BLANK 2>`:

We run this program on a 32-bit system with a direct-mapped, 4 KiB cache, with 128B blocks.

Q6.2 (1 point) What is the T:I:O breakdown of this cache?

Tag:	Index:	Offset:

Q6.3 (3 points) How many misses of each type occur if we call `primelist(4096)`?

Compulsory:

Capacity:

Conflict:

Q6.4 (1 point) Disclaimer: We think this subpart is especially challenging, so feel free to skip it and come back later.

How many misses of each type occur if we call `primelist(16384)`?

You may use the function `pi(n)` to denote the number of primes less than or equal to `n`. (So `pi(2) = 1`, `pi(10) = 4`, etc.)

Compulsory:

Capacity:

Conflict:

We run this code with `pragma omp` using 4 threads as follows:

```
char* primelist(uint32_t n) {
   char* primes = <BLANK 1>;
   if (<BLANK 2>) return NULL;
   primes[0] = 1;
   primes[1] = 1;
   #pragma omp parallel for
   for (int i = 2; i * i < n; i++) {
       if (!primes[i]) { // If i is a prime number, ...
           for (int j = 2 * i; j < n; j += i) {
               primes[j] = 1; // Cross out all multiples of i
           }
       }
   }
   return primes;
}
```

Q6.5 (2 points) Does this program have data races?

○ Yes, and it could change the output of the function

○ Yes, but it does not change the output of the function

○ No

Q7 *A Page on Page Tables* **(9 points)**

For this question, assume that we are working with a byte-addressed system with 32 GiB of virtual memory, 64 MiB of physical memory, and a 2 KiB page size. Including metadata, each page table entry is set to be 4 bytes long.

For Q7.1 through Q7.4, assume we have a single level page table with no TLBs.

Q7.1 (1.5 points) How many bits long is our page offset?

Q7.2 (1.5 points) How many bits are there in the PPN?

Q7.3 (1.5 points) How many bits are there in the VPN?

Q7.4 (1.5 points) How many PTEs will we have in our page table?

Noticing that this is inefficient, we decide to use a two-level hierarchical page table with no TLBs. Our VPN bits are split evenly among the two levels.

Q7.5 (1.5 points) How many PTEs are in the L1 page table?

Q7.6 (1.5 points) How many pages worth of memory does a single L2 page table take?

Q8 *Kernel Panic* (15 points)

In Project 4, one of the main components was writing a parallel matrix multiplication algorithm. One interesting optimization that we didn't cover is the idea of a register-based kernel; in a similar vein to how cache blocking works, we create a small enough block that our working data gets stored within registers instead.

For this question, we will consider the use of a 2×2 kernel; that is, we compute a 2×2 block of the result matrix at a time. Complete the following function:

Signature: `void matmul_kernel(double* result, double* mat1, double* mat2_T, int N);`

Arguments:

- `double* result`: A section of memory storing at least 2×2 doubles worth of data
- `double* mat1`: A $2 \times N$ matrix stored in row-major order
- `double* mat2_T`: An $N \times 2$ matrix stored in column-major order (in other words, this matrix has already been transposed from row-major order for you)
- `int N`: The value N as above. You may assume that N is a multiple of 4.

Effect: Set `result` to equal `mat1 * mat2_T`, stored in row-major order.

You have access to the following 4-double SIMD operations:

- `vector* vec_load(double* A)`: Load 4 doubles at the given memory address `A` into a vector
- `vector* vec_set0()`: Load all 0s into a vector
- `vector* vec_fma(vector* A, vector* B, vector* C)`: Performs a element-wise fused multiply-add: `result = A + B * C` in a single operation
- `double vec_sum(vector* A)`: Adds all elements of the vector together: `result = A[0] + A[1] + A[2] + A[3]`

Q8.1 (10 points) Fill in the code below.

```
void matmul_kernel(double* result, double* mat1, double* mat2_T, int N) {

    vector* a = ______________________________________________;

    vector* b = ______________________________________________;

    vector* c = ______________________________________________;

    vector* d = ______________________________________________;

    for (int i = 0; ___________________; ___________________) {
        vector* m1r0 = vec_load(mat1 + i);

        vector* m1r1 = vec_load(mat1 + ______________________);
        vector* m2c0 = vec_load(mat2_T + i);

        vector* m2c1 = vec_load(mat2_T + ____________________);

        _______________________________________________________;

        _______________________________________________________;

        _______________________________________________________;

        _______________________________________________________;
    }

    result[0] = ___________________________________________(a);

    result[1] = ___________________________________________(b);

    result[2] = ___________________________________________(c);

    result[3] = ___________________________________________(d);
}
```

Instead of a 2×2 kernel, we realize that we can improve our efficiency by using a 3×3 kernel instead.

Q8.2 (1 point) For a 3×3 kernel, how many vector registers would we initialize outside the `for` loop?

Q8.3 (1 point) For a 3×3 kernel, how many vector registers get loaded inside each iteration of the `for` loop?

Q8.4 (3 points) If we run this on a system which has access to 64 vector registers, what is the largest square kernel size we can use while still maintaining speedup? Submit the side length (so if you believe that the 2×2 kernel shown above is the largest kernel size, submit 2).

Nothing on this page will be graded.

Is there anything you want us to know?

Got a message/doodle for Cece or Nick?

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