McMahon, Weaver
Spring 2022

CS 61C

Final

Print your name: ______________________, ______________________
(last) (first)

Print your student ID: ______________________

Solutions last updated: Saturday, March 2, 2024

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: ______________________

This content is protected and may not be shared, uploaded, or distributed.

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

Solution: False. Not all RISC-V instructions are 4 bytes (32 bits) long. The base set, RV32 which is what we work with in this class indeed uses 32-bit instruction lengths, but B-type and J-type instructions under RV32 encode immediates that support addressing of 16-bit addresses, resulting in both immediate encodings to have just **one** implicit 0 each. This is to not only help support legacy code for when memory spaces were small enough to be addressed by 16-bit addresses, but also to help 16-bit compressed RISC-V instructions.

Grading: All-or-nothing.

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

● TRUE ○ FALSE

Solution: True. Typically, computer systems will use little-endian data encoding, which is the way that we typically read it as humans. Networking is where you'll most often encounter big-endian encoding (and maybe 161 exams where they're trying to mess with you). The rationale for computer systems using little-endian is typically such that the processor reads data the same way we do. For networking, there are minor advantages in terms of reliability, but it's likely the current standard because someone decided to encode data like that one day, and the networking architecture supported that, and since then the need to maintain backwards compatibility kept it that way.

Grading: All-or-nothing.

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

Solution: True. The loader is typically a component of the operating system and its job is to load the linked executable into the respective segments of memory (i.e. `.text` into text/code memory, `.data` into static/data memory. Its other important job is to help set up everything the program needs to run, such as clearing/initialising the proper registers, setting up the stack, preparing the program arguments on the stack, and more. Once this is all complete, the loader transfers control to other components of the system to actually execute the program. This last step is the responsibility for correctly starting a program.

Grading: All-or-nothing.

This content is protected and may not be shared, uploaded, or distributed.

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

Solution: False. System calls, more commonly known as syscalls, are entry points by the user program to the operating system, and are **not** executed by the user program. These requests are typically asking the system for resources of some kind or to alter areas of memory the user does not directly have access to. This includes things such as asking for more heap memory (`*alloc()` calls), reading from/writing to I/O (which includes things like reading from and writing to files), amongst others.
Grading: All-or-nothing.

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

Solution: True. Even though the CPU is not directly performing the memory access, as the DMA controller does that, the CPU is still needed to initiate that action, since it's still a form of I/O, and the processor at the very least, always starts the process to handle them.
Grading: All-or-nothing.

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

Solution: True. Water is actually considered to be a better coolant for computation-intensive systems as water conducts heat better than air does. This allows us to remove excess heat from systems in a more efficient way and is typically more common at WSC centers than air is.
Grading: All-or-nothing.

This content is protected and may not be shared, uploaded, or distributed.

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)
```

Solution: `0x01CE0FA3`

From the reference card: `sb rs2 imm(rs1)` is an S-type instruction, its opcode is `0b010 0011`, and its funct3 is `000`.

In addition to the opcode and funct3, an S-type instruction needs a 12-bit immediate, rs1, and rs2.

The 12-bit immediate is 31, which is `0b0000 0001 1111` in two's complement binary. Bits 11-5 are `0b0000000`, and bits 4-0 are `0b11111`.

rs1 and rs2 are both t3, which is register x28. 28 is `0b11100` in unsigned binary.

Combining all these fields according to the S-type instruction structure gives us:

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
0b0000000	11100	11100	000	11111	0100011

Changing spacing and converting to hex:

`0b0000 0001 1100 1110 0000 1111 1010 0011`

`0x01CE0FA3`

Grading: Partial credit was given for correct opcode and funct3 (0.5 points), correct rs1 and rs2 (0.5 points), and correct immediate (1 point).

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

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

Solution: $!B + CD$

First, note that $(A+!A) = 1$, which reduces the expression to $!(B + (CD)) + CD$.

Then, using DeMorgan's law reduces the expression to $(!B)(!(CD)) + CD$.

Finally, using absorption reduces the expression to $!B + CD$.

An intuitive way to see the absorption step: if CD is 0, then the expression reduces to $!B$. If CD is 1, then the expression reduces to 1.

Grading: Partial credit was awarded for expressions that correctly matched the behavior of the given expression, but had more operators than optimal. Each extra operator lost 0.4 points, with the minimum of 0 points at 8 operators (the same number of operators as the original expression).

This content is protected and may not be shared, uploaded, or distributed.

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.

Solution: We can fully parallelize 60 minutes of pre-processing among 5 TAs, so pre-processing is finished in $\frac{60}{5} = 12$ minutes.

We can fully parallelize 4500 minutes of grading among 10 TAs, so grading is finished in $\frac{4500}{10} = 450$ minutes. Note that there is an additional restriction that each TA must fully grade one question, so we should verify that this doesn't change our answer. There are 10 long-answer questions and 10 TAs, so the long-answer questions can be graded in 250 minutes by having each TA grade one long-answer question. Then, there are 40 short-answer questions and 10 TAs, so we can have each TA grade 4 short-answer questions. This means the short-answer questions get graded in $4 \times 50 = 200$ minutes. In total, grading is finished in $250 + 200 = 450$ minutes, confirming our original answer.

In total, pre-processing and grading takes $12 + 450 = 462$ minutes.

Grading: All or nothing

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.

Solution: As in the previous part, it takes $\frac{60}{5} = 12$ minutes for pre-processing.

Because of the restriction that each question must be fully graded by a single TA, no matter how many TAs we have, grading will still take 250 minutes for one TA to fully grade a long-answer question.

In total, pre-processing and grading takes $250 + 12 = 262$ minutes.

Grading: All or nothing

This content is protected and may not be shared, uploaded, or distributed.

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.

Solution: $\frac{97}{100}$
There are 6 failures, and each one takes 12 hours = 0.5 days to repair, so the total downtime is $6 \times 0.5 = 3$ days.
Out of 100 days, the hive machine was up for 97 days, which gives us an availability of $\frac{97}{100}$.
Grading: All or nothing. Half a point was taken off for incorrect format, e.g. $\frac{970}{1000}$ or 97%.

This content is protected and may not be shared, uploaded, or distributed.

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	

Solution: Here's a list of solutions that we came across and considered correct. Due to the open-ended nature of this question, some correct answers may not be included in this list. Note that for credit, all portions of this question required an understandable explanation of an advantage for both sides. Fully correct statements without an explicit advantage provided, or any statement that was factually incorrect, yielded no credit for that section. Note that several of these benefits are considered somewhat outside the scope of this class; unless otherwise stated, all sections had at least one benefit considered within scope.

Solution: A 32-bit IEEE-754 standard floating point number

- Can represent values that are not integers.
- Greater range (including if just restricted to integers).
- Able to represent smaller and larger values than ints.
- Extensions to more bits yields superexponentially increasing range instead of exponentially increasing range.
- Built-in error values allows for division by 0 and graceful overflows.
- Due to the complexity of the data type, it's harder to "spoof" a large value.

This content is protected and may not be shared, uploaded, or distributed.

Solution: A 32-bit two's complement integer

- More values representable due to not having NaNs.
- Way simpler circuitry = faster implementations in general.
- The group structure results in same circuitry as unsigned numbers.
- There is only one 0.
- All valid operations are infinitely precise and fully associative. Bitwise operations cause sensible results.
- Standards for ints are far simpler and reverse-engineerable.
- Spacing between representable values is consistent.
- Larger range of consecutive representable integers.
- Can be used as indices and for pointer arithmetic.

Q3.2 (3 points)	
C	
Python	

Solution: C

- Pass by value makes things quick
- Faster runtime by far
- Lower-level language means it's easier to translate to assembly
- No explicit garbage collection
- Gives you greater freedom over your program (you can do things that Python won't let you)

This content is protected and may not be shared, uploaded, or distributed.

- The memory model is more visible to the programmer, allowing for better optimizations
- Static typing means you can't reference a variable before using it
- Lower resource use
- Compiled languages tend to tell you about typo-style bugs before you start running the code

Solution: Python

- Typically lower barrier to entry when first using it (easier to learn).
- Closer to English.
- Code is portable across systems.
- Automatic data allocation/garbage collection causes this to be more memory safe.
- Dynamic typing means you can do lists of multiple different compatible data types
- Better libraries for ML and NLP
- Better runtime error messages; C will likely let you do something you didn't want to do
- No compiler needed
- Far easier to code in, so it reduces the length of a development cycle.

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

Solution: A dynamically linked library

- Able to be updated to the most recent versions on the fly when compiling.

This content is protected and may not be shared, uploaded, or distributed.

- Results in a smaller code size.
- Does not have to go through the entire CALL process when updating libraries.

Solution: A statically linked library

- Faster to link and load.
- Has fewer security risks.
- Resulting executable is portable.

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

Solution: A RAID-0 array with 5 2TB disks

- Faster access time (reads, writes).
- Can physically store more data.
- No (technically very small, but out of the scope of this course) overhead to set up.
- Lower latency.
- Higher bandwidth.

Solution: A RAID-5 array with 5 2TB disks

- More redundancy as no data is lost when a single drive fails.

This content is protected and may not be shared, uploaded, or distributed.

- Higher reliability because when a disk fails, the data is still accessible (though, the general practice is to let the array rebuild first in case a second drive fails).

Q3.5 (3 points)	
Polling	
Interrupts	

Solution: Polling

- Allows for "always-busy" devices to continuously send data.
- Has an overall average lower latency.
- There is a deterministic response time per event.
- Does not have to redirect CPU resources from processes after they've started to service I/O.

Solution: Interrupts

- Handles input from devices that require less CPU.
- Allows CPU to execute other processes when there is no data.
- When used correctly, typically will use less power as it relies on user input to be triggered.

This content is protected and may not be shared, uploaded, or distributed.

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"`.

<table>
<tr><th>A</th><th>B</th></tr>
<tr><td>

```
char* returnHello() {
    return "Hello";
}
```

</td><td>

```
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;
}
```

</td></tr>
</table>

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

Solution:

As with the first half of this question, answers were graded primarily on understanding; correctly selecting the bubble that matched our solution without a discussion of advantages/disadvantages yielded no credit. Conversely, we accepted solutions that correctly identified issues, even if we classified your advantage as a correctness issue (or vice versa). If only one advantage was provided, we assumed that your answer for the other direction was "None". Unless otherwise stated, all sections had some advantage that we considered within scope.

Advantages of A

- Gets a string stored in the data segment of memory, which allows for faster runtime.
- Direct pointer comparisons are viable if you look only at return values of this function.
- Uses less memory upon multiple function calls.
- (Answer if B is considered incorrect) Repeatedly calling B without freeing will eventually lead to segfault.

Advantages of B

- Stored on the heap, which allows modification of the string.

This content is protected and may not be shared, uploaded, or distributed.

- Return value can point to different pointers.
- (Answer if A is considered incorrect) Trying to modify the returned string will cause a segfault.

Common answers that were NOT given credit

- A does not allocate space for the string/stores the string on the stack: The use of the string literal in the code creates a static string that gets used whenever the literal gets referenced.
- "Hello" is a string, not a char pointer: Strings are char pointers in C; furthermore, the C compiler replaces all instances of the string literal with a pointer to that string in static memory
- strncpy/calloc is incorrect: Per the comments provided, the given lines correctly work as intended

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)

Solution:

Advantages of A

- Able to send in different data types, such as floats and longs, without requiring multiple functions and function names
- Faster to use due to inlining the function; as such, it doesn't create a stack trace
- Shorter code length (barely)
- Gets removed by the preprocessor if you never use abs
- (Answer if B is considered incorrect) Works on non-integers; the given behavior technically specified that it takes the absolute value of a number, which is not necessarily an integer

Advantages of B

- Safer to use due to avoiding the issue of a find and replace. Common examples of code that wouldn't yield expected behavior with A was: `abs(x++)`, `abs(10-20)`, `-3*abs(10)`, `abs(5)+3`
- Forces an integer typecast, which may be useful to prevent errors from sending in wrong datatypes
- Useable as a function pointer in higher-order functions; A does not create a function pointer
- (Answer if A is considered incorrect) The lack of parentheses makes this yield strange behavior when used with certain operands. See above for details.

This content is protected and may not be shared, uploaded, or distributed.

- (Answer if A is considered incorrect) This question asked for a function. Option A is a preprocessor directive, which is technically not a function.

Common answers that were NOT given credit

- A takes in certain data types and doesn't catch a type error: Sending in a pointer will cause a compiler error the same way as with B, because a type check will restrict the types of x to those which have a defined unary minus operator. Bools and chars will yield valid results (though potentially garbage; absolute value of any valid ASCII char/bool is itself), but both have implicit type conversions to int (bool is generally defined as an int), so they will also pass into B properly.
- A/B is in static memory, while B/A is in the code: Both versions end up in code

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)

Solution:
Advantages of A

- (Answer if B is considered incorrect) Follows calling convention.

Advantages of B

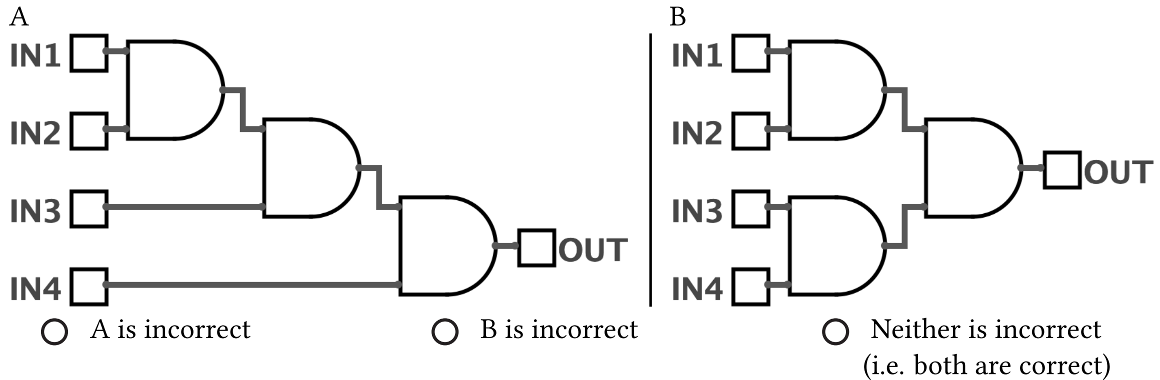
- None

Common answers that were NOT given credit

- Any statement that suggests that calling convention violations are acceptable (including trying to read t0 or s0 after the function returns) was considered incorrect. From an engineering perspective, calling convention errors are the worst possible error, because they appear to work. It is in general preferable to have a program that doesn't work over a program that looks correct, but has a subtle bug that breaks everything three years down the line, since if you have the former, you know that you need to fix something.

This content is protected and may not be shared, uploaded, or distributed.

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



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

Solution:

Advantages of A

- None was accepted as a valid answer, as the advantages here were considered either negligible, or outside of scope
- Can be better if IN4 is on the critical path, but no other inputs are.
- Can be better if you can reuse the intermediate computation values in another part of the circuit
- Depending on the properties of the input stream, A or B might be slightly better in terms of security properties (less channel info from out) and energy use, but this is in general hard to discern.

Advantages of B

- There is a shorter critical path delay, so this component will in general finish computing faster than choice A
- Depending on the input stream, A or B might be slightly better in terms of security properties (less channel info from out) and energy use, but this is in general hard to discern.

Common answers that were NOT given credit

- Incorrect results from either side: Both circuits reduce to the correct results, since AND is associative and commutative.

This content is protected and may not be shared, uploaded, or distributed.

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

<table>
<tr><th>A</th><th>B</th></tr>
<tr><td><pre>#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;
    }
  }
}</pre></td><td><pre>#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;
  }
}</pre></td></tr>
</table>

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

Solution: For this question, the answers that would be obtained from a full understanding of the question were significantly different from answers that would be obtained from a partial understanding. As such, 3 points were awarded for full understanding, while 2 points were awarded for partial understanding.

Advantages of A (Fully correct)

- None was accepted only if cache coherency/data thrashing was mentioned as an advantage of B.
- If the threads diverge sufficiently due to the scheduler separating threads, then this code runs faster than B. This is, however unlikely.

Advantages of B (Fully correct)

- Runs in general faster than A. This is because A's use of parallelism ends up interleaving the writes. The result isn't a data race (since each thread gets to run on its own values), but instead causes coherence misses on the L1 cache. Due to repeated data invalidations, memory accesses in A will tend to be around the same as L3 cache access times or longer; this is often longer than an 8x speedup.
- (Answer if A is considered incorrect) A could yield incorrect results. `omp_set_num_threads` is actually not required by the omp library to set the number of threads to exactly 8. As such, this does not always work. (Note that to get credit for this, you needed to explicitly note that `omp_set_num_threads` could choose not to set the thread count to 8)
- More portable, as not all systems have the omp library

Advantages of A (Partially correct)

This content is protected and may not be shared, uploaded, or distributed.

- A runs faster than B due to the work being divided over 8 threads.

Advantages of B (Partially correct)

- B doesn't need to deal with the overhead of creating new threads, which could make B faster if there is only one hardware core, or if that thread overhead is significant
- More easily readable

Common answers that were NOT given credit

- Issues involving the correctness of A/A is slower due to all threads doing the same work as B's single-threaded code: The code in A is correct. In particular:
- A pragma omp for is unneeded since we manually distribute the iterations among 8 threads.
- The length is specified as a large power of 2, which is therefore a multiple of 8. As such, no tail case is needed
- The variable `i` is defined inside the parallel segment, so it is implicitly private.
- `omp_get_thread_num` is used within the parallel segment, and so will correctly return a value between 0 and 7, inclusive. `omp_get_thread_num` is correct syntax, and does not return the number of threads running. Thread numbers are 0-indexed, so this function will return 0 for some thread, not 8.
- The end of the parallel segment creates an implicit barrier, so it is impossible for the parent thread to return before all child threads finish setting their values to 0.

This content is protected and may not be shared, uploaded, or distributed.

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.

Solution: The system can only represent numbers whose most-significant digit is 1. Numbers whose most-significant digit is 2-9, such as 525600 or 2, cannot be represented.

Alternate answer: 0 cannot be represented.

In binary, the only possible bits are 0 and 1, and a number can't have 0 as its most-significant bit (or else we could truncate it, e.g. 0b0010 = 0b10), so we know that every binary number starts with a 1. This allows us to use the implicit 1 in binary floating point. In decimal floating point, there are 10 possible digits, so the implicit 1 no longer makes sense.

Grading: For full credit, answers needed to explicitly mention the most-significant digit of the number or give a concrete example of an unrepresentable number.

We gave half credit for answers that mentioned numbers in $[-1 \times 10^{-95}, 1 \times 10^{-95}]$ being unrepresentable; this is a true statement, but not really the primary issue with the implicit 1.

Some common incorrect answers that we did not give credit to:

- Cannot represent negative numbers (or other issues with signed numbers). This is not true because the sign bit can still be set to make numbers negative.
- Changes range/step size/precision/number of representable values. This doesn't specify anything about the most-significant digit of the number.
- Adds 1×10^X to the number. This is true, but it would be intended behavior with the implicit 1, and we would account for it when representing numbers in floating point. (You wouldn't say that the implicit 1 in binary is a problem because it adds a power of 2 to the number, for example.)

This content is protected and may not be shared, uploaded, or distributed.

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

Solution: There are six possible correct answers:

- Exponent `1`, significant `610 0000`. This gives $6.100000 \times 10^1 = 61$.
- Exponent `2`, significant `061 0000`. This gives $0.610000 \times 10^2 = 61$.
- Exponent `3`, significant `006 1000`. This gives $0.061000 \times 10^3 = 61$.
- Exponent `4`, significant `000 6100`. This gives $0.006100 \times 10^4 = 61$.
- Exponent `5`, significant `000 0610`. This gives $0.000610 \times 10^5 = 61$.
- Exponent `6`, significant `000 0061`. This gives $0.000061 \times 10^6 = 61$.

Grading: One point for each correct answer. Half-credit if both answers are correct, but the exponent is off by one in the same direction in both answers (e.g. first answer is 2 and 610 0000, second answer is 3 and 061 0000). If the significand provided was fewer than 7 digits, we right-padded with zeros until it was 7 digits (e.g. 1 and 61 would be treated as 1 and 610 0000).

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

Solution: 384

The significand must be all zeros (1 possibility). The exponent can be any integer in $[-95, +96]$, which is a total of $96 + 95 + 1 = 192$ possibilities. The sign bit can be positive or negative (2 possibilities). In total, we have $1 \times 192 \times 2 = 384$ possible representations of zero.

Grading: Half credit was awarded for the following minor errors: Missing the negative zeros (192), and miscounting the number of exponents (382). Two or more minor errors yielded no credit.

This content is protected and may not be shared, uploaded, or distributed.

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

Solution: The key realization to solving some of these answer choices is to note that every IEEE-754 floating-point number can be expressed as an odd integer multiplied or divided by a power of 2. (The odd integer comes from the significand, and the power of 2 comes from the exponent). If a number cannot be represented this way, then it must not be representable as an IEEE-754 floating-point number.

0.1 can be represented by the decimal floating point standard: for example, significand 010 0000 and exponent 1 gives 0.1×10^1. However, $0.1 = \frac{1}{10}$ is not an odd integer multiplied or divided by a power of 2, so it cannot be represented as an IEEE-754 floating-point number.

$\frac{1}{3}$ cannot be represented by either system, as it has infinitely many significand bits, and both floating-point systems have finite precision.

0.5 can be represented by the decimal floating point standard: for example, significand 050 0000 and exponent 1 gives 0.5×10^1. 0.5 can also be represented by the IEEE-754 floating point standard: for example, significand 0 (plus the implicit 1) and exponent -1 gives `0b`1.0×2^{-1}.

0.8 can be represented by the decimal floating point standard: for example, significand 080 0000 and exponent 1 gives 0.8×10^1. However, $0.8 = \frac{4}{5}$ is not an odd integer multiplied or divided by a power of 2, so it cannot be represented as an IEEE-754 floating-point number.

1 can be represented by the decimal floating point standard: for example, significand 100 0000 and exponent 0 gives 1×10^0. 1 can also be represented by the IEEE-754 floating point standard: for example, significand 0 (plus the implicit 1) and exponent 0 gives `0b`1.0×2^0.

10^6 can be represented by the decimal floating point standard: for example, significand 100 0000 and exponent 6 gives 1×10^6. 10^6 can also be represented by the IEEE-754 floating point standard: one way to note this without actually deriving the number is to recall that all integers up to 2^{24} can be represented in floating point (this is where the step size becomes 2), and $10^6 < 2^{24}$.

2^{32} cannot be represented by the decimal floating point standard. The decimal floating point standard gives 7 digits of precision, but $2^{32} = 4{,}294{,}967{,}296$ requires 10 digits of precision. In other words, trying to write this in the decimal floating point standard gives 4.294967296×10^9, but this significand does not fit in 7 digits.

10^{90} can be represented by the decimal floating point standard: for example, significand 100 0000 and exponent 90 gives 1×10^{90}. However, 10^{90} cannot be represented as an IEEE-754 floating-point number. One way to see this is to write $10^{90} = 5^{90} \times 2^{90}$. 2^{90} is the exponent part of the number, which means that 5^{90} must be represented in the significand. However, 5^{90} has many more than 23 bits, so the significand does not have enough precision/bits to represent 5^{90}.

This content is protected and may not be shared, uploaded, or distributed.

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`.

Solution:

```
addi sp sp -4
sw rs2 0(sp)
```

The first instruction decrements the stack pointer by 4 bytes (one word). The second instruction stores the word in `rs2` in the newly-allocated space on the stack.

Note that accessing memory below the stack pointer `sp` is undefined, so we have to decrement the stack first before accessing that memory.

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

Solution:

```
lw rd 0(sp)
addi sp sp 4
```

The first instruction loads the bottom-most word on the stack into `rd`. The second instruction removes that word from the stack by incrementing the stack pointer by 4 bytes (one word).

As in the previous part, accessing memory below the stack pointer `sp` is undefined, so we have to load the word from memory first before incrementing the stack (which effectively deletes that word from the stack).

This content is protected and may not be shared, uploaded, or distributed.

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

This content is protected and may not be shared, uploaded, or distributed.

Solution:

We can reuse one of the existing instruction format types, as the `push` instruction only needs `rs2` encoded in the instruction. For example, we could make `push` an S-type instruction, give it a new opcode (not in use by any other instruction), and leave the funct3, immediate, and `rs1` fields unused (e.g. fill them with all zeros, or hard-code them to a value consistent with how we're using the rest of the datapath, for rs1).

ImmGen will be used to output 4 here (so we can add 4 to `sp`), so a second new immediate type is not needed.

The existing Regfile has two outputs for reading two registers. The `push` instruction only requires reading the values in `sp` and `rs2`, so a third Regfile output is not needed.

The existing Regfile has one writeback input for writing to one register. The `push` instruction only requires writing to `sp`, so a second writeback input is not needed.

The ALU will be used to subtract 4 from `sp`. We can use the first Regfile read output (`rs1`) to read the value in `sp`, so AMux selects the register output, not the PC. BMux selects the immediate (which is always outputting 4), not the `rs2` Regfile read output. Neither AMux nor BMux needs a new input, and the ALU does not need a third register input or a new operation (`sub` already exists).

In the existing datapath, the DMEM address input is the ALU output. In the `push` instruction, the ALU output is `sp-4`, which coincidentally is also the address we want to store to, so an additional DMEM address input is not needed.

In the existing datapath, WBMux supports writing back the ALU output to Regfile. In the `push` instruction, the ALU output is `sp-4`, which is what we want to write back to the `sp` register, so no new input to WBMux is needed.

This content is protected and may not be shared, uploaded, or distributed.

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

This content is protected and may not be shared, uploaded, or distributed.

Solution: We can reuse one of the existing instruction format types, as the `pop` instruction only needs `rd` encoded in the instruction. For example, we could make `pop` an I-type instruction, give it a new opcode (not in use by any other instruction), and leave the funct3, immediate, and `rs1` fields unused (e.g. fill them with all zeros, or hard-code them to a value consistent with how we're using the rest of the datapath).

ImmGen will be used to output 4 here (so we can add 4 to `sp`), so a second new immediate type is not needed.

The existing Regfile has two outputs for reading two registers. The `pop` instruction only requires reading the value in `sp`, so a third Regfile output is not needed.

The existing Regfile has one writeback input for writing to one register. The `pop` instruction requires writing to two registers, `sp` and `rs2`, so a second writeback input is needed.

The ALU will be used to add 4 to `sp`. We can use the first Regfile read output (`rs1`) to read the value in `sp`, so AMux selects the register output, not the PC. BMux selects the immediate (which is always outputting 4), not the `rs2` Regfile read output. Neither AMux nor BMux needs a new input, and the ALU does not need a third register input or a new operation (`add` already exists).

In the existing datapath, the DMEM address input is the ALU output. In the `pop` instruction, the ALU output is `sp+4`, but the address we want to load from is `sp`, so an additional DMEM address input is needed.

In the existing datapath, WBMux supports writing back the ALU output to Regfile. In the `pop` instruction, the ALU output is `sp+4`, which is what we want to write back to the `sp` register, so no new input to WBMux is needed.

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

Solution: `push` can cause data hazards if we write to `sp` in the instruction immediately following a `push` instruction.

`push` does not cause PC to jump/branch to a different place (it just increments PC by 4), so it cannot cause control hazards.

This content is protected and may not be shared, uploaded, or distributed.

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.

```
1   char* primelist(uint32_t n) {
2      char* primes = <BLANK 1>;
3      if (<BLANK 2>) return NULL;
4      primes[0] = 1;
5      primes[1] = 1;
6      for (int i = 2; i * i < n; i++) {
7          if (!primes[i]) { // If i is a prime number, ...
8              for (int j = 2 * i; j < n; j += i) {
9                  primes[j] = 1; // Cross out all multiples of i
10             }
11         }
12     }
13     return primes;
14  }
```

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	~~0~~	~~1~~	2	3	4	5	6	7	8	9	10	11	0 and 1 are nonprime
`i = 2`	~~0~~	~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~	11	Cross out all multiples of 2
`i = 3`	~~0~~	~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~	11	Cross out all multiples of 3
`i = 4`	~~0~~	~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~	11	4 isn't prime, continue
`i = 5`	~~0~~	~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~	11	Cross out all multiples of 5
`i = 6`	~~0~~	~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~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.

This content is protected and may not be shared, uploaded, or distributed.

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

Solution:
`<BLANK 1>`: `calloc(n, sizeof(char))` or equivalent.
`<BLANK 2>`: `primes == NULL` or equivalent.

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?

Solution: 20, 5, 7
Each block is $128 = 2^7$ bytes, so we need 7 offset bits to uniquely address each byte in a block.
The cache is direct-mapped, so each block of the cache has a unique index. The cache contains 4 KiB = 2^{12} bytes, and each block is 2^7 bytes, so there are $2^{12}/2^7 = 2^5$ blocks in the cache. We need 5 bits to uniquely address each block in the cache.
The system uses 32-bit addresses, so we have $32 - 7 - 5 = 20$ bits left over for the tag.

This content is protected and may not be shared, uploaded, or distributed.

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

Solution: 32 compulsory misses, 0 capacity misses, 0 conflict misses

Let's start by considering the first iteration of the outer for loop (`i=2`). At this point, `primes[2]` is 0, so the inner for loop will execute. Its bounds are `j=4; j<4096; j+=2`, and on each iteration, we're accessing `primes[j]`.

`primes[0]` is a miss, because our cache starts cold. It's a compulsory miss because we've never accessed this part of memory before. This miss brings in a 128-byte block containing `primes[0]` to `primes[127]`.

`primes[6]` is a hit because of the block we just brought in. `primes[8]`, `primes[10]`, …, `primes[126]` are also hits.

`primes[128]` is a compulsory miss, since we've never accessed this part of memory before. This miss brings in another 128-byte block containing `primes[128]` to `primes[255]`. Since the cache is direct-mapped, and this block is directly after the previous block in memory, these two blocks occupy different indices in the cache. (You can also verify this by checking that the index bit increments by 1 between these two blocks.)

`primes[130]` is a hit because of the block we just brought in. `primes[132]`, `primes[134]`, …, `primes[254]` are also hits.

This pattern continues for the rest of the for loop. We have one compulsory miss for every 128-byte block. Each block we bring in occupies a different index in the cache. The array is $4096 = 2^{12}$ bytes, and each block is $128 = 2^7$ bytes, so we bring in $2^{12}/2^7 = 2^5 = 32$ blocks. This means we have 32 compulsory misses, 0 capacity misses, and 0 conflict misses from the first iteration of the outer for loop.

After the first iteration of the outer for loop, note that our entire array is now in the cache. The cache fits 32 blocks, and the array is made up of 32 blocks, each with a different index (since each block is stored at an adjacent location in memory; you can verify this by checking the index bit changing as we increment through the entire array). Regardless of how memory gets used after this, there are no more misses in any future iteration of the outer for loop, so in total, we have 32 compulsory misses, 0 capacity misses, and 0 conflict misses.

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.)

This content is protected and may not be shared, uploaded, or distributed.

Solution:

128 compulsory misses, $(\text{pi}(128) - 1) * 128$ capacity misses, 0 conflict misses

As in the previous part, let's start by considering the first iteration of the outer for loop (`i=2`). At this point, `primes[2]` is 0, so the inner for loop will execute. Its bounds are `j=4; j<16384; j+=2`, and on each iteration, we're accessing `primes[j]`.

On this first iteration of the outer for loop, we have the same pattern as the previous part. Each block of 128 bytes forces one compulsory miss, and we iterate through every block of the array. The array is $16384 = 2^{14}$ bytes, and each block is $128 = 2^7$ bytes, so we bring in $2^{14}/2^7 = 2^7 = 128$ blocks. This means we have 128 compulsory misses, 0 capacity misses, and 0 conflict misses from the first iteration of the outer for loop.

Unlike the previous part, where the entire array fits in the cache, the 128-block array is now four times larger than the 32-block cache. Because the cache is direct-mapped, this causes the first 1/4th of the array to fill up the cache, then get completely replaced by the next 1/4th of the array, then the next 1/4th of the array, then the last 1/4 of the array. After the first iteration of the outer for loop, only the last 1/4th of the array is in the cache.

On the second iteration of the outer for loop, `i=3` and `primes[3]` is 0, so the inner for loop will execute. Its bounds are `j=3; j<16384; j+=3`, and as before, we're accessing `primes[j]` on each iteration.

There are two important things to note about this loop. First, it starts back at the beginning of the array. Second, it accesses every 128-byte block of the array, just like the previous `j+=2` loop. This means that the cache, which currently contains the last 1/4th of the array, will be entirely replaced with the first 1/4th of the array as this loop executes. In other words, nothing in the cache at the end of the the previous loop helps us in this loop; the cache effectively starts cold again. Also, because we access every block of the array again, we have another 128 misses, just like in the previous for loop. This time, the 128 misses are capacity misses because the cache is full on every miss. (There are no compulsory or conflict misses during this loop.)

On the third iteration of the outer for loop, `i=4` and `primes[4]` is 1, so we skip the inner for loop. This still causes one miss due to accessing the 0th block of our array.

On the fourth iteration of the outer for loop, `i=5` and `primes[5]` is 1, so the inner for loop executes with `j=5; j<16384; j+=5`. As before, we end up having to access every block of the array, which results in another 127 capacity misses. Note that because we missed on the first block in iteration `i=4`, we don't miss again on the first block.

In general, we miss exactly 128 times for each prime number; once for the first composite number after the previous prime, and 127 times within the inner for loop.

This pattern continues all the way to `i=127`, the last point where `i*i<n`. In all of these iterations of the outer for loop, the inner for loop executes only when `i` is prime (per the comment at line 7), so in total, we have $\text{pi}(128)$ (equivalently, $\text{pi}(127)$) iterations of the inner for loop. The first iteration of the inner for loop causes compulsory misses, and the other $\text{pi}(128) - 1$ iterations of the inner for loop causes capacity misses.

Each iteration of the inner for loop causes 128 misses; note that because we only go up to `i=127`, the inner for loop always has to access every block of the array. In other words, for a block to be

This content is protected and may not be shared, uploaded, or distributed.

skipped, the inner for loop would need to be skipping by at least 128 bytes each time, which never happens here.

Finally, we note that 127 is prime, so our last iteration of the loop is a prime number. As such, we do not have an additional miss to check a composite number after our last prime.

In total, we have $(\text{pi}(128) - 1) * 128$ capacity misses, 128 compulsory misses, and 0 conflict misses.

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

```
1   char* primelist(uint32_t n) {
2      char* primes = <BLANK 1>;
3      if (<BLANK 2>) return NULL;
4      primes[0] = 1;
5      primes[1] = 1;
6      #pragma omp parallel for
7      for (int i = 2; i * i < n; i++) {
8          if (!primes[i]) { // If i is a prime number, ...
9              for (int j = 2 * i; j < n; j += i) {
10                 primes[j] = 1; // Cross out all multiples of i
11             }
12         }
13     }
14     return primes;
15  }
```

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

Solution: A data race will occur, but the program will still return the correct results. Because future iterations of the loop depend on past iterations, there is a chance that a composite number gets through the if statement; for example, if the 6 loop occurs before the 2 and 3 loops, then 6 will go through the for loop. However, this code still outputs the correct result, because all prime numbers are guaranteed to not be set to 1 by any loop (even composite ones), and as such any composite number will be set to 1 by some iteration of the inner for loop.

This content is protected and may not be shared, uploaded, or distributed.

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?

> **Solution:** 11 bits
>
> Each page is 2 KiB = 2^{11} B large, so we need 11 bits to uniquely identify one of the bytes in this page.

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

> **Solution:** 15 bits
>
> PPN = physical page number. Remember that the physical address is split into two parts: the PPN identifies a page of memory, and the page offset identifies a byte within that page.
>
> Physical memory is 64 MiB = 2^{26} B large, so we need 26 bits to address every byte of physical memory. Of the 26 bits, 11 bits are used as the page offset (from the previous part). That leaves $26 - 11 = 15$ bits to be used as the PPN.

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

> **Solution:** 24 bits
>
> VPN = virtual page number. Remember that the virtual address is split into two parts: the VPN identifies a page of memory, and the page offset identifies a byte within that page.
>
> Virtual memory is 32 GiB = 2^{35} B large, so we need 35 bits to address every byte of virtual memory. Of the 35 bits, 11 are used as the page offset (from Q7.1). That leaves $35 - 11 = 24$ bits to be used as the VPN.

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

> **Solution:** 2^{24}
>
> PTE = page table entry. A PTE maps a virtual page number to a physical page number, so we need one PTE per virtual page number. The VPN is 24 bits long, so there are 2^{24} different VPNs, which means there are 2^{24} PTEs.
>
> *Grading*: To avoid double jeopardy, full credit was given if you answered 2 to the power of your answer in the previous part. For example, if you answered 61 in the previous part and 2^{61} in this part, you were given full credit for this part.

This content is protected and may not be shared, uploaded, or distributed.

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?

Solution: 2^{12} PTEs

From Q7.3, we know that there are 24 bits in the VPN. If we split these bits evenly among the two levels, we have 12 bits for uniquely identifying a L1 PTE. This gives us 2^{12} possible PTEs in the L1 page table.

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

Solution: 8 pages

As in the previous part, splitting the bits evenly gives us 12 bits for uniquely identifying a page within a single L2 page table. This means that a single L2 page table contains 2^{12} PTEs. From the top of the question, each PTE is 4 bytes long, so a single L2 page table takes up $2^{12} \times 4 = 2^{14}$ bytes.

One page of memory is 2 KiB = 2^{11} bytes, so a single L2 page table takes up $2^{14}/2^{11} = 2^3 = 8$ pages of memory.

This content is protected and may not be shared, uploaded, or distributed.

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]`

This content is protected and may not be shared, uploaded, or distributed.

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

```
void matmul_kernel(double* result, double* mat1, double* mat2_T, int N) {
    vector* a = vec_set0();
    vector* b = vec_set0();
    vector* c = vec_set0();
    vector* d = vec_set0();
    for (int i = 0; i < N; i += 4) {
        vector* m1r0 = vec_load(mat1 + i);
        vector* m1r1 = vec_load(mat1 + i + N);
        vector* m2c0 = vec_load(mat2_T + i);
        vector* m2c1 = vec_load(mat2_T + i + N);
        a = vec_fma(a, m1r0, m2c0);
        b = vec_fma(b, m1r0, m2c1);
        c = vec_fma(c, m1r1, m2c0);
        d = vec_fma(d, m1r1, m2c1);
    }
    result[0] = vec_sum(a);
    result[1] = vec_sum(b);
    result[2] = vec_sum(c);
    result[3] = vec_sum(d);
}
```

Solution: High-level idea: We need to compute four dot products in the result matrix:

- `result[0]` = row 0 of `mat1` dotted with row 0 of `mat2_T`
- `result[1]` = row 0 of `mat1` dotted with row 1 of `mat2_T`
- `result[2]` = row 1 of `mat1` dotted with row 0 of `mat2_T`
- `result[3]` = row 1 of `mat1` dotted with row 1 of `mat2_T`

On each iteration of the for loop, we'll load a 4-element vector from each row of each matrix. (That's a total of 4 vectors, since we have 2 rows from each of the 2 matrices.) Note that to get elements from the second row of the matrix, we add `N` to the address of the start of the matrix to skip past the first row of `N` elements.

Then we'll element-wise multiply those vectors together and add those products into one of the intermediate sum vectors (`a`, `b`, `c`, `d`).

Finally, we'll sum up the intermediate sums in the vectors to produce one final dot product.

This content is protected and may not be shared, uploaded, or distributed.

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?

Solution: 9

One way to get this answer is to see that in the 2×2 code, we have one vector register per element of the result array. In the 3×3 version, the result array would have 9 elements, so we would need 9 vector registers.

Grading: All-or-nothing.

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

Solution: 6

We need to load a vector from each row of each matrix. There are 2 matrices, each with 3 rows, for a total of $2 \times 3 = 6$ vector registers loaded.

Grading: All-or-nothing.

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).

Solution: 7

Generalizing our answers from the previous two parts, when multiplying a $k \times N$ matrix by an $N \times k$ matrix to get a $k \times k$ matrix, we need k^2 registers for intermediate sums (Q8.2), and $2k$ registers to load k rows from each of the 2 matrices (Q8.3). In total, we need $k^2 + 2k$ vector registers.

We have access to 64 vector registers, so we're looking for the largest k such that $k^2 + 2k \leq 64$. The largest integer k that satisfies this inequality is $k = 7$, where $7^2 + 2(7) = 63$.

Grading: All-or-nothing.

This content is protected and may not be shared, uploaded, or distributed.