Weaver
Spring 2020

CS 61C
Machine Structures

Final

Note (August 10, 2022): These are extremely rough drafts of rewritten solutions. They definitely contain errors and unfinished sections, but might have some useful parts for studying.

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

PRINT your student ID: ______________________________

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: ______________________________

You have 170? minutes. There are 8 questions of varying credit, for a total of 121 points.
For questions with **circular bubbles**, you may select only one choice.

○ Unselected option

● Only one selected option

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

■ You can select

■ multiple squares

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, `0xDEAD BEEF` instead of `0xdeadbeef`. You will be graded incorrectly otherwise! Please always add the hex (`0x`) and binary (`0b`) prefix to your answers or you will receive 0 points. For all other bases, do not add the suffix or prefixes.

Do not add units unless the problem explicitly tells you to!

Question 1 ***Bitz are Bitz!*** **(8 points)**

Let's consider the hexadecimal value `0xFA000003`. How is this data interpreted, if we treat this number as...

an array A of unsigned, 8-bit numbers? Please write each number in decimal, assume the machine is little endian. If the value is unknown, write GARBAGE (in all caps).

Q1.1 (0.5 point) `A[0]`

Solution: 3

If we assume that the system is little-endian, then `0x03` is stored at the lowest address in memory, then `0x00` at the next-highest address, then `0x00` at the next-highest address, then `0xFA` at the highest address.

`A[0]` is the first element of the array, which is stored at the lowest address.

Q1.2 (0.5 point) `A[1]`

Solution: 0

See previous solution; this is the byte at the second-lowest memory address.

Q1.3 (0.5 point) `A[2]`

Solution: 0

See previous solution; this is the byte at the third-lowest memory address.

Q1.4 (0.5 point) `A[3]`

Solution: 250 (which is 0xFA in decimal)

See previous solution; this is the byte at the highest memory address. `0xFA` $= 15 \times 16^1 + 10 \times 16^0 = 240 + 10 = 250$.

Q1.5 (2 points) a IEEE-754-style floating point number, but which uses only 8 bits for the exponent with a bias of 64 (where we subtract the bias)? Write out as binary scientific notation, so e.g. an answer that looks like this: $-1.0100100 \cdot 2^{15}$.

Solution: $-1.00000000000000000000011 \cdot 2^{58}$

Q1.6 (2 points) a RISC-V instruction? If there's an immediate, write it in decimal. If it is an invalid instruction, write INVALID INSTRUCTION (in all caps).

Solution: `lb x0, -96(x0)`

Q1.7 (2 points) a `(uint32_t *)` variable `c` in **big-endian** format, and we call `printf("%i", (int) ((uint8_t *) &c) [0])`? If the value is unknown, write GARBAGE (in all caps).

Solution: `&c` is the address of the bits `0xFA000003`. It could also be interpreted as a pointer to the bits `0xFA000003`.

`(uint8_t *) &c)` says that we should treat this pointer as a pointer to an array of `uint8_t` numbers (each number takes up 1 byte).

`(uint8_t *) &c) [0]` dereferences the pointer to the array, and accesses the first element of the array. The first element of the array is stored at the lowest memory address. Since this question assumes big-endian format, the most-significant byte `0xFA` will be stored at the lowest memory address.

`(int) ((uint8_t *) &c) [0])` casts the `uint8_t` number `0xFA` to an integer, then the `printf` call prints out the integer. SOLUTION-TODO: Why does this cast not change the value of the number?

250 (which is 0xFA in decimal).

Question 2 *Cache me Outsize* (24 points)

Given the following looping workload over an array where `N` is a large power of 2. The cache starts out empty, and the `process()` function doesn't introduce any significant cache pressure (so you can discount any hits or misses in the `process()` function).

```
uint32_t arr[N];

for (int j=0; j < 30; j++) {
  for (int i=0; i < N; i += 1) {
      process(arr[i]);
  }
}
```

Express the following answers as a function of `N`. **If you have a fraction, please fully simplify it.** If you believe that an answer is of the form `42 * N`, DO NOT include the multiply. You should format that answer like `42N`. Failure to do so or not capitalizing `N` will result in no points! If you have a fraction answer of the form `1 / 42 * N`, format it like `(1/42)N`. Note if it is NOT a fraction, you MUST not include the parentheses.

Suppose we have a LRU fully associative cache of size `4N` B and a block size of 4B.

Q2.1 (2 points) Number of hits

Solution: `29N`

Each block can hold 4 bytes. The cache is `4N` bytes in total, so there are `N` blocks in the cache.

Note that each element of `arr` is a `uint32_t`, which is 32 bits = 4 bytes. This means that each block holds one element of the array.

Consider the first iteration of the outer loop, when `j=0`. The inner loop accesses `arr[i]` for all `i` between `i=0` and `i=N-1`. Each access is a miss, since this is the first time we're seeing each element of `arr`. There are `N` misses in the first iteration of the outer loop.

At this point, the entire array is in the cache. Note that the cache holds `N` blocks = `N` elements of the array, which is exactly enough space for the entire array. Also, the cache is fully-associative, so we can put each element of the array anywhere we want in the cache.

For the rest of the iterations of the outer loop, the inner loop will cause `N` hits. There are 29 more iterations of the outer loop, for `29N` hits.

Q2.2 (2 points) Number of misses

Solution: `N`

See previous solution.

Q2.3 (2 points) What type of locality is this cache taking advantage of (select all that apply)

☐ Quasi-balistic ☐ Spatial

[✓ **Temporal**] ☐ None

Solution: Quasi-balistic is made-up (not a real cache term).

We're taking advantage of temporal locality by repeatedly accessing elements of `arr` around the same time.

We're not taking advantage of spatial locality, because each block holds only one element of the array, so we never used the fact that the elements of the array are stored side-by-side in memory. (One way to think about this is to note that if every element of the array were stored in a completely random part of memory, we would still have the same number of hits/misses.)

Q2.4 (2 points) Does your answer change if the cache is 2 way set-associative? (Note: The cache size is still the same)

○ Yes [✓ **No**]

Solution: On the first iteration of the outer loop, there are still `N` misses. After the first iteration, the entire array still ends up in the cache. One way to see this is to note that the elements in the array are all at consecutive memory addresses, so the first half of the array fills up one block of each set, then the second half of the array fills up the second block of each set. Since the entire array ends up in the cache, the rest of the iterations of the outer loop still cause `29N` hits.

We're still only exploiting temporal locality, not spatial locality, because we didn't increase the block size (each block still only holds one element of the array).

Suppose we have a LRU fully associative cache of size `2N` B and a block size of 4B.

Q2.5 (2 points) Number of hits

Solution: 0

Note that the size of the cache is now half as large, so we can only fit half of the array in the cache. Consider the first iteration of the outer for loop. The first half of the inner for loop will fill up the cache with the first half of the array, causing `N/2` misses. Then, the second half of the inner for loop will replace all the data in the cache with the second half of the array, causing another `N/2` misses.

At the end of the first iteration of the outer loop, we have the second half of the array in the cache. In the second iteration of the outer loop, the first half of the inner for loop will replace all the data in the cache with the first half of the array again. Then the first half of the array gets replaced in the cache by the second half of the cache again.

This pattern repeats through the entire code, so we get no hits. We access the array `N` times per iteration of the outer loop, for `30N` accesses in total. Every access is a miss.

Q2.6 (2 points) Number of misses

Solution: `30N`
See previous solution.

Q2.7 (2 points) What type of locality is this cache taking advantage of (select all that apply)

☐ Quasi-balistic ☐ Spatial
☐ Temporal [✓ **None**]

Solution: We aren't taking advantage of any sort of locality, since the cache is always missing. (Adding the cache didn't improve the performance of our code.)

Q2.8 (2 points) Does your answer change if the cache is 2 way set-associative? (Note: The cache size is still the same)

○ Yes [✓ **No**]

Solution: In a 2-way set associative cache, the cache will still alternate holding the first half and the second half of the array. One way to see this is to consider each iteration of the inner for loop. The first quarter of memory accesses will fill up one block of each set, then the second quarter of memory accesses will fill up the second block of each set, filling up the entire cache with the first half of the array in all. Then the same pattern occurs with the third and fourth quarter of memory accesses.

Suppose we have a LRU fully associative cache of size `2N` B and a block size of 8B.

Q2.9 (2 points) Number of hits

Solution: `15N`
Since the entire array is `4N` bytes, this cache fits half of the array. Also, note that each block is 8 bytes, so each block fits 2 elements of the array.
Consider the first iteration of the outer for loop. Accessing `arr[0]` is a cache miss, but it brings in a block with `arr[0]` and `arr[1]`. Then accessing `arr[1]` is a hit. Then accessing `arr[2]` is a miss, bringing in `arr[2]` and `arr[3]`. Then accessing `arr[3]` is a hit. This pattern continues through the entire inner loop, causing `N/2` misses and `N/2` hits.
As in the previous part, the first half of the inner loop brings the first half of the array into the cache. Then the second half of the inner loop replaces all the data in the cache with the second half of the array. In the next iteration of the outer loop, the first half of the inner loop replaces all the data in the cache with the first half of the array again. Since the entire cache data is being swapped out on each iteration of the outer for loop, there are no additional hits caused by the cache being filled up with data between iterations of the outer loop.
In total, there are `30N` array accesses. Half are misses, and half are hits.

Q2.10 (2 points) Number of misses

Solution: `15N`
See previous solution.

Q2.11 (2 points) What type of locality is this cache taking advantage of (select all that apply)

☐ Quasi-balistic [✓ **Spatial**]

☐ Temporal ☐ None

Solution: We're not taking advantage of temporal locality, because repeatedly accessing elements of the array in the short amount of time (i.e. across different iterations of the outer loop) don't improve our cache behavior. The cache may as well start cold at the start of each outer loop iteration, and it wouldn't change our hit rate.

We're taking advantage of spatial locality, because each block brings in two adjacent array elements. This gives us an additional cache hit when we access the second of the two elements in the cache block.

Q2.12 (2 points) Does your answer change if the cache is 2 way set-associative? (Note: The cache size is still the same)

○ Yes [✓ **No**]

Solution: As in the previous solutions, the cache still gets filled with the first half or second half of the array, even if the cache is 2-way set associative.

Question 3 *Nick's Parallelism Computer (uh huh) Setup* (8 points)

Nick has several resources at his disposal. The first is a 12 core AMD Ryzen processor running at over 3 GHz for MIMD computation, the second is a massive SIMD computational engine in the form of a high-end graphics card with 10 teraflops of floating point computation (he got it for compute… Uh hu.), and the final is a large map/reduce cluster on campus he has access to. If a problem requires reading as many memory locations as compute operations, mark it as "memory bound" because the speedups from parallelism are going to be minor because it will be limited by the memory subsystem.

Please select a parallelism technique which would benefit the problem the most.

Q3.1 (2 points) 32b floating point matrix multiply of 100M entry matrices with a transposed matrix

- ○ MIMD parallelism
- [✓ **SIMD parallelism**]
- ○ Map/reduce
- ○ Memory bound
- ○ None (sequential)

Solution: SIMD parallelism

Recall that to perform matrix multiplication, we need to take a row of the first matrix and a column of the second matrix, and perform a dot product. The second matrix is transposed, so fetching a column of the second matrix is equivalent to fetching a row of the transposed second matrix.

Assuming that the matrices are stored in row-major order, we have two rows of values being stored at adjacent addresses in memory. SIMD operations are most powerful when we can load and store vectors of values from adjacent addresses in memory.

Then, we need to perform element-wise multiplication on these two rows of values to perform the dot product. This is a single operation that we're repeatedly computing on multiple data, which is another indicator that SIMD parallelism is most useful here.

MIMD parallelism may also help here, but SIMD is more appropriate here because the vast majority of computation is the same instruction (multiplication and some addition).

Map/reduce processes inputs and outputs as key/value pairs, and it's not immediately obvious how you would design the task of matrix multiplication in the map/reduce paradigm.

This task is not memory-bound because for two $n \times n$ matrices, we need to read $2n^2$ elements of the matrices. This is $O(n^2)$ number of reads. However, matrix multiplication requires $O(n^3)$ number of computations (for each of the n^2 elements of the result matrix, we need to perform a dot product that involves multiplying and adding n elements). The number of computations outnumbers the number of memory reads.

Q3.2 (2 points) Find all references to himself in a downloaded corpus of every Internet post ever made (some 5 PB of data)

○ MIMD parallelism
○ SIMD parallelism
[✓ **Map/reduce**]
○ Memory bound
○ None (sequential)

Solution: Counting the number of instances of a word in a large corpus of documents is a classic problem that map/reduce is designed to solve. We can use the map function to produce key-value pairs for each word mapping the word to the number 1. Then we can use the reduce function to combine all key-value pairs with the same key (i.e. all instances of the same word) and add all the corresponding values together (so that the value corresponding to the count is the number of times the word appears).

MIMD parallelism is not the best solution here, because all the threads would eventually need to combine their results, creating a bottleneck.

SIMD parallelism is not the best solution here, because there isn't a clear single instruction that we're executing on multiple pieces of data.

This task is not memory-bound because there is more computation to be done (checking for references for each document and combining counts) than memory accesses (reading each document).

Q3.3 (2 points) Run a program he's written that has 40 threads that cocmmunicate through queues or channels

[✓ **MIMD parallelism**]
○ SIMD parallelism
○ Map/reduce
○ Memory bound
○ None (sequential)

Solution: The main indicator that MIMD parallelism is the best choice here is the fact that the program has 40 threads. Threads are an example of MIMD parallelism, since each thread can work on different instructions and different parts of data.

Q3.4 (2 points) 32b floating point matrix addition of 100M entry matricees

○ MIMD parallelism
○ SIMD parallelism
○ Map/reduce
[✓ **Memory bound**]
○ None (sequential)

Solution: This task is memory-bound, because to add two $n \times n$ matrices together, we would need to make $2n^2$ memory accesses to read both matrices, but we would only need to perform n^2 additions.

Question 4 ***Virtual Reality! I mean Memory...*** **(8 points)**

Consider a system with 2 MiB of physical memory and 4 GiB of virtual memory. Page size is 4KiB. Recall that the single level page table is stored in physical memory and consists of PTE's, or page table entries.

Q4.1 (3 points) If we choose to store seven information bits in each PTE, how big is the page table in bytes?

> **Solution:** 2^{21}
> VPN contains $log(\frac{2^{32}}{2^{12}}) = 20$ bits
> PPN contains $log(\frac{2^{21}}{2^{12}}) = 9$ bits
> 9 bit PPN + 7 info bits = 16 bits per PTE
> 2^4 bit PTE $\cdot 2^{20}$ PTE's $= 2^{24}$ bit page table, or 2^{21} bytes

Q4.2 (3 points) The page table starts off empty, then we make the following accesses: `0x00111999`, `0x00234567`, `0x00555FFF`. If the page table begins at address `0x20000000`, at what address can we find the PTE for the first access? (Your answer should be in hex)

> **Solution:** `0x20000222`. `0x00111999` has a VPN of `0x00111`. Each PTE is 2 bytes, so `0x00111` $\cdot 2 =$ `0x00222`. `0x20000000` + `0x00222` = `0x20000222`.

Q4.3 (2 points) We have a fully associative TLB that also started empty but now contains the three entries from the accesses above. If we access `0x00556000` now, will we get a TLB hit, page hit, or page fault?

○ TLB hit   [✓ **Page fault**]
○ Page hit   ○ None of the other answers

> **Solution:** Page fault: `0x00556000` is a new page.

Question 5 ***Mover your A***** **(27 points)**

The (not turing complete) programming language Mover is defined as follows:

The program stores an 2-D grid of 8-bit integers, initialized to 0, a memory pointer, which starts at $(x, y) = (0, 0)$, and a program flow, which starts at "FORWARD". The program recognizes only the following 8 commands:

- `>` Moves the pointer one step right (+1 to `x`)
- `<` Moves the pointer one step left (−1 to `x`)
- `^` Moves the pointer one step up (+1 to `y`)
- `v` Moves the pointer one step down (−1 to `y`)
- `+` Increments the value at the pointer by 1
- `-` Decrements the value at the pointer by 1
- `]` If the pointer is currently pointing at a 0 and the current program flow is "FORWARD", change the program flow to "BACKWARD". Otherwise do nothing.
- `[` If the pointer is currently pointing at a 0 and the current program flow is "BACKWARD", change the program flow to "FORWARD". Otherwise do nothing.

If the program flow is "FORWARD", then the next instruction to be executed is the one after the current one; if the program flow is "BACKWARD", then the next instruction is the one before the current instruction.

It is undefined behavior for the pointer to go outside the memory array's bounds and likewise the behavior for integer overflow and underflow are undefined. For the C version, the program halts if it reaches the end of the program string in either direction.

The language ignores any other characters in the program, and terminates if the program counter goes past the bounds of the program.

You want to write a C program that interprets this language: The inputs are a valid mover program as a null terminated string, and `memory_grid`, which points to a sequence of pointers, each of which points to a buffer that is a column of the 2D grid.

```
void runMover(char* program, int8_t** array)
{
    uint x = 0;
    uint y = 0;
    uint pc = 0;
    uint dir = 1; /* forward = true */
    uint len = strlen(program);
    while(pc>=0 && pc<length)
    {
        switch(program[pc])
        {
            //Code for each Mover command
        }
        pc += dir ? 1: -1;
    }
}
```

For the following operators, write the C code for the following Mover commands.

Q5.1 (1 point)

Command: ^

Code:

```
case '^':
    <YOUR CODE HERE>
    break;
```

Solution: `y += 1;` (or equivalent)

Note that `x` and `y` are the memory pointer described in the question.

As stated in the question, `^` should add 1 to `y`.

Q5.2 (2 points)

Command: `+`

Code:

```
case '+':
    <YOUR CODE HERE>
    break;
```

Solution: `memory_grid[x][y] += 1;` (or equivalent)

As stated in the question, `+` should increment the value at the pointer by 1.

As stated in the question `memory_grid` points to a sequence of pointers, each of which points to a buffer that is a column of the 2D grid. This means that we should first pick a column of the 2D grid by indexing into the sequence of pointers and choosing a pointer. The column is chosen by `x`. Then, we pick a row within that column by indexing into the buffer. The row is chosen by `y`.

Q5.3 (2 points) Command: `]`

Code:

```
case ']':
    <YOUR CODE HERE>
    break;
```

Solution: `if (!memory_grid[x][y]) programflow = 0;` (or equivalent)

As stated in the question, `]` should check if the pointer is currently pointing at 0. We do this with `if (!memory_grid[x][y])`.

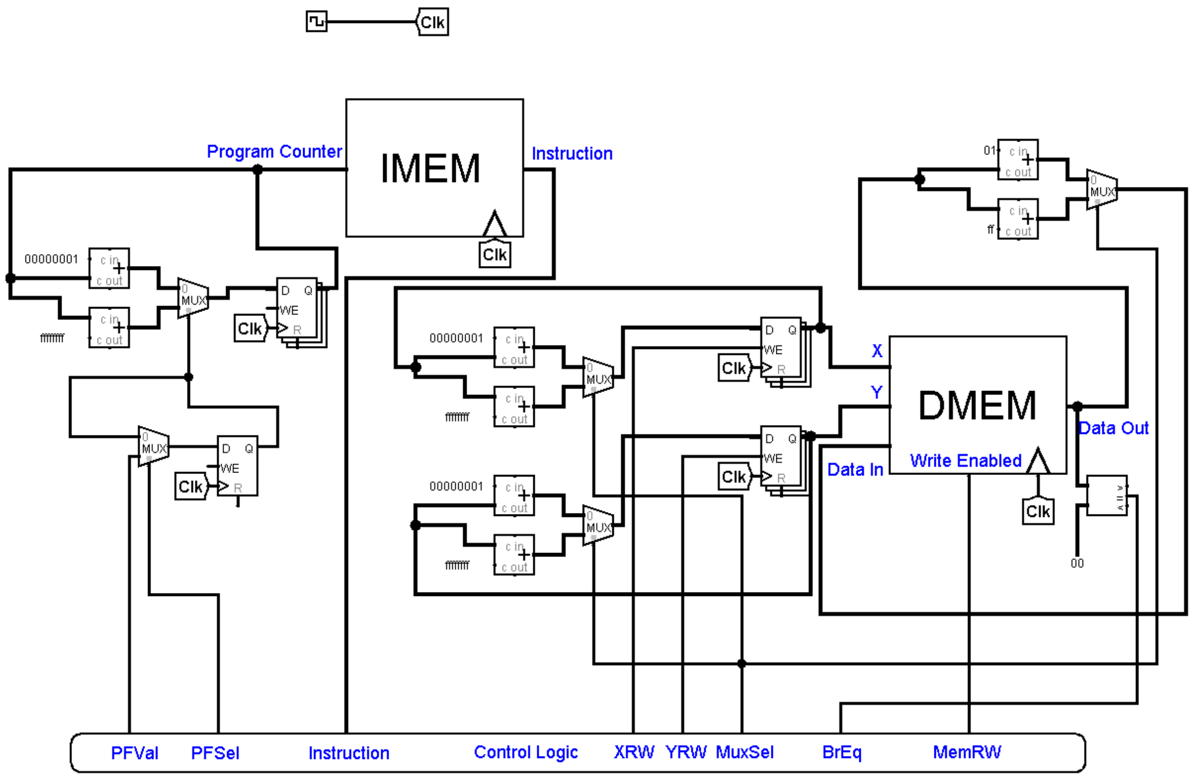
Then if the condition is true, we set the program flow to backward with `programflow = 0`. Note that if the pointer is currently pointing at 0, and the program flow is backward, then the question says we should do nothing.

If the program flow is already backward, setting it to backward again doesn't change anything, so not checking for the program flow in the if condition is fine.

You now want to create a circuit that runs Mover. In order to do that, you assign each Mover command a unique 4-bit code (explanations of each command have been copied for reference):

- (`0001`) `>` Moves the pointer one step right (+1 to `x`)
- (`1001`) `<` Moves the pointer one step left (−1 to `x`)
- (`0101`) `^` Moves the pointer one step up (+1 to `y`)
- (`1101`) `v` Moves the pointer one step down (−1 to `y`)
- (`0011`) `+` Increments the value at the pointer by 1
- (`1011`) `-` Decrements the value at the pointer by 1
- (`0111`) `]` If the pointer is currently pointing at a 0 and the current program flow is "FORWARD", change the program flow to "BACKWARD". Otherwise do nothing.
- (`1111`) `[` If the pointer is currently pointing at a 0 and the current program flow is "BACKWARD", change the program flow to "FORWARD". Otherwise do nothing.

You have finished the general structure of the circuit, and just need to finish writing the control logic. The memories in question are asynchronous read but synchronous write (thus the CLK). During startup the registers are set to 0. Note that the direction register now holds zero (not one) when the program is going forward. We just run forever and it is simply undefined behavior to either overflow or underflow the PC so we aren't worrying about having to check for any of that.



Single Cycle Mover

Let `x3, x2, x1, x0` be the bits of an instruction with `x0` being the least significant bit, and let `BrEq` be the value of `BrEq`. Write the most simplified logical equations for each output of Control Logic.

Please use C syntax when writing out your formulas.

Q5.4 (2 points) YRW

Solution: `~x1 & x2` (or equivalent)

YRW is the write-enable bit for the register holding the value of Y.

We want to write to the Y register in the `^` and `v` instructions, which have opcodes `0101` and `1101`. Thus we want YRW to be `1` when the opcode is either `0101` or `1101`. In all other instructions, we want YRW to be `0`.

We can inspect the pattern of the opcodes to note that `^` and `v` have the only two opcodes where the middle two bits are `10` (`x1` is `0` and `x2` is `1`).

Q5.5 (2 points) XRW

Solution: `~(x1 | x2)` (or equivalent)

XRW is the write-enable bit for the register holding the value of X.

We want to write to the X register in the `>` and `<` instructions, which have opcodes `0001` and `1001`. Thus we want XRW to be `1` when the opcode is either `0001` or `1001`. In all other instructions, we want XRW to be `0`.

We can inspect the pattern of the opcodes to note that `>` and `<` have the only two opcodes where the middle two bits are `00` (`x1` is `0` and `x2` is `0`).

This could be written as `~x1 & ~x2`, but we can use DeMorgan's law to simplify this to `~(x1 | x2)` which requires one fewer logic operator.

Q5.6 (2 points) PFVal

Solution: `~x3` (or equivalent)

PFVal is the second input of a mux whose output is being sent to a one-bit register. We see the memory pointer (X and Y) and the program counter registers already labeled in the circuit, and the grid is stored in memory. The only remaining value we haven't seen in the circuit is the 1-bit program flow, so this 1-bit register must be holding the program flow signal.

The mux before the program flow register is choosing between two signals. The top wire is the output of the program flow register. If we send the output of the program flow register back into the input of the program flow register, we're leaving the program flow unchanged. Then the bottom wire must be used to change the program flow.

The program flow is changed in the `]` and `[` instructions, which have opcodes `0111` and `1111`.

In the `]` instruction (opcode `0111`), we want to change the program flow to backward, so we want the program flow register to hold value `1`. For opcode `0111`, we want PFVal to be `1`.

In the `]` instruction (opcode `1111`), we want to change the program flow to forward, so we want the program flow register to hold value `0`. For opcode `1111`, we want PFVal to be `0`.

For all other instructions, PFVal can be any garbage value, since we won't be selecting the bottom branch of the mux to change the program flow register.

By inspecting the two opcodes and the desired PFVal values, we notice that the left-most bit of the opcode `x3` is the opposite of the desired PFVal value.

Q5.7 (2 points) PFSel

Solution: `x1 & x2 & BrEq` (or equivalent)

Continuing from the previous part, PFSel is being used to choose between the two branches of the mux. The output of the mux is then sent to the program flow register.

When PFSel is 0, we choose the upper input of the mux, which is the output of the program flow register. This will cause the program flow value to stay unchanged. Thus we want PFSel to be 0 when the program flow value should not be changed.

When PFSel is 1, we choose the lower input of the mux, which is PFVal. From the previous part, PFVal forces program flow to change from the `[` or `]` instructions. Thus we want PFSel to be 1 when the instruction is `[` and or `]` instruction, and the pointer is currently pointing at 0.

We can determine when the instruction is `[` or `]` by examining the opcodes and noting that these are the only two instructions where the middle two bits (`x1` and `x2`) are `11`.

We can determine when the pointer is currently pointing at 0 by using the BrEq signal. Note that the BrEq signal comes from a comparator that checks if Data Out (the value in the grid that the pointer is pointing at) is equal to 0.

Thus for PFSel to be 1 (the program flow to change), we need the instruction to be `[` or `]` (`x1 & x2`), and we need BrEq to be 1 (pointer is pointing at 0).

Q5.8 (2 points) MemRW

Solution: `x1 & ~x2` (or equivalent)

MemRW is the write-enable bit for DMEM, which is where we're storing the grid.

We want to write to the grid in the `+` and `-` instructions. For these two instructions, we want MemRW to be `1`. For all other instructions, we aren't writing to the grid, so we want MemRW to be `0`.

Note that `+` and `-` have opcodes `0011` and `1011`. We can inspect the pattern of the opcodes to note that these are the only two opcodes where the middle two bits are `01` (`x1` is `1` and `x2` is `0`).

Q5.9 (2 points) MuxSel

Solution: `x3`

MuxSel is being used as input to two muxes. The top mux chooses between the output of two adders, which are computing `X+1` (top input of mux) and `X-1` (bottom input of mux). The bottom mux chooses between the output of two adders, which are computing `Y+1` (top input of mux) and `Y-1` (bottom input of mux). Then the output of the two muxes are being sent into the X and Y registers.

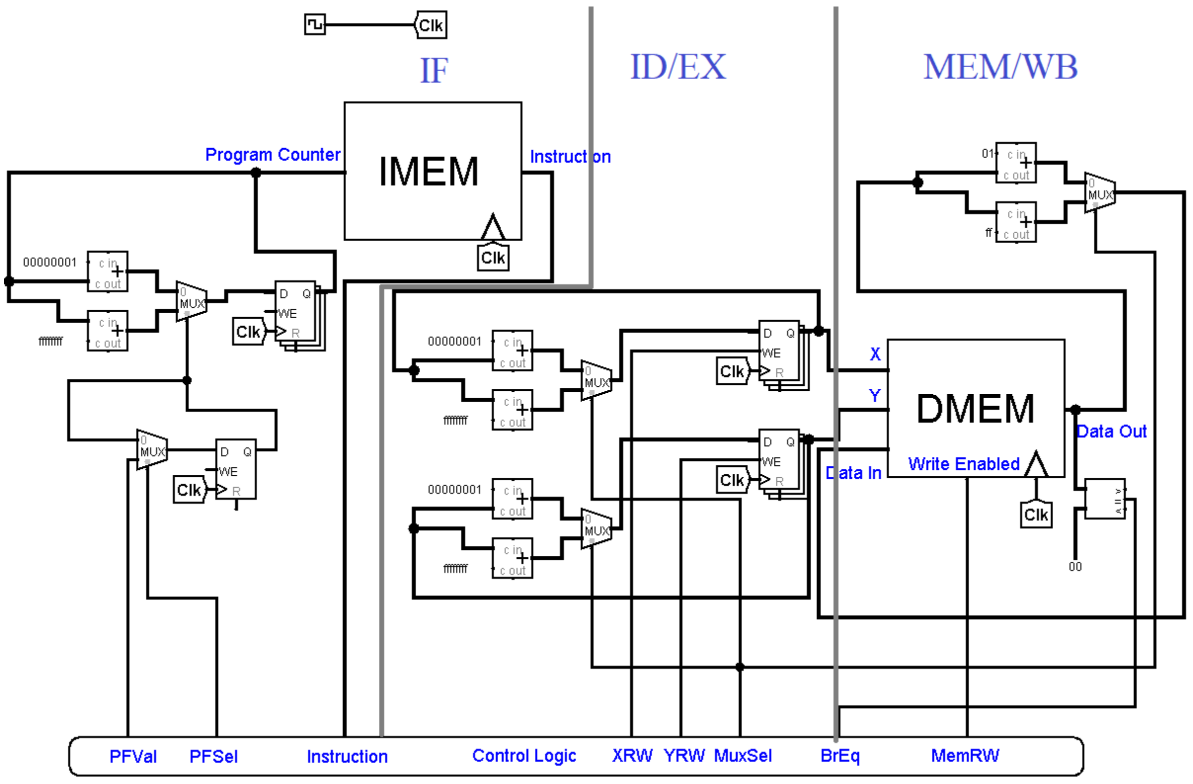
MuxSel should be 0 (top input of mux) when we want to compute `X+1`. This happens in the `>` instruction (opcode `0001`). MuxSel should also be 0 when we want to compute `Y+1`. This happens in the `^` instruction (opcode `0101`).

MuxSel should be 1 (bottom input of mux) when we want to compute `X-1`. This happens in the `>` instruction (opcode `1001`). MuxSel should also be 1 when we want to compute `Y-1`. This happens in the `v` instruction (opcode `1101`).

In all other instructions, we're not updating X or Y, so MuxSel can be garbage (XRW and YRW will ensure that the mux output/register input won't be written to the X and Y registers).

By inspecting the four relevant opcodes and the expected MuxSel value for each opcode, we can notice that the left-most bit `x3` corresponds exactly to the desired value of MuxSel. For example, for opcode `0001`, we want MuxSel to be 0, and the left-most bit is 0. For opcode `1101`, we want MuxSel to be 1, and the left-most bit is 1.

After finishing your circuit, you find that it's a bit slow. To speed things up, you decide to pipeline the above circuit by dividing the circuit into IF, ID/EX, and MEM/WB stages.



Pipelined Mover

Q5.10 (3 points) What hazards can occur? Assume that the single cycle datapath worked as intended (Select all that apply)

■ **Control Hazard** ☐ Data Hazard

☐ Structural Hazard ☐ None of the Other Choices

Solution: Control Hazard: Yes. If our instruction is `[` or `]`, then we need to wait until after MEM to decide whether to reverse flow. If we pipeline, we would need to stall for `[` or `]` instructions.
Structural Hazard: No. In the current system, register reads and writes happen during the same cycle.
Data Hazard: No. Since our writeback doesn't affect X or Y, we can never have a data dependency.
None of the Other Choices: No. While data and structural hazards cannot occur, control hazards can.

Q5.11 (3 points) Please leave your answer in ns and do not add the units.

If all registers (including pipeline registers) have 2ns setup, 0ns hold, and 2ns clk-to-q time, the memories take 4ns to do a read and have a 2ns setup time for writes, and the sequential logic takes 0ns (yes, that's a ridiculous number, we chose it to make the path simple), what is the minimum viable clock period for the resulting datapath?

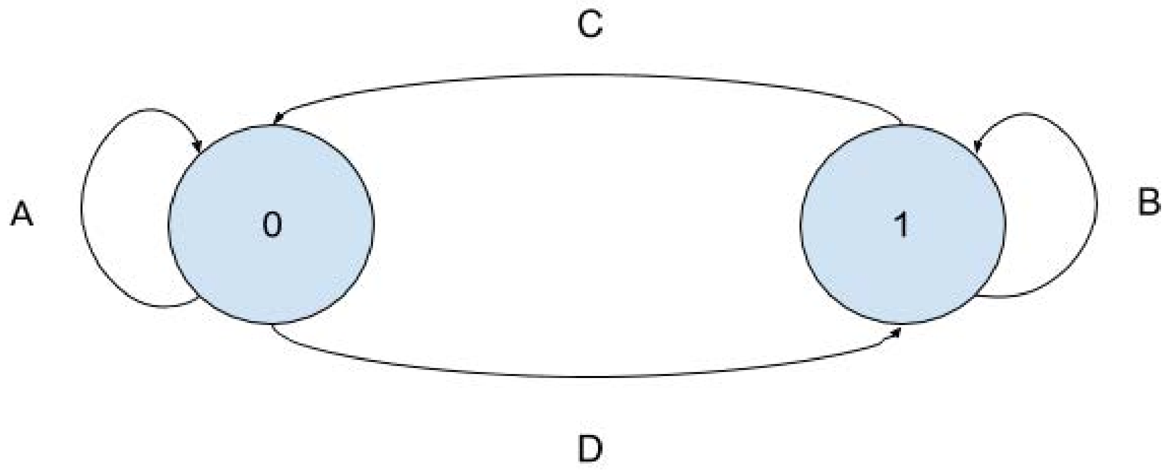
Solution: 8. 2ns (clk-to-q time) + 4ns (IMEM read) + 2ns (setup time) = 8ns

We're looking for the longest path through any two clocked elements in the circuit, where we assume that we've added registers where the circuit is divided into pipeline stages. Note that when writing to IMEM or DMEM, the input of IMEM or DMEM counts as a clocked element (which is why there's a 2ns setup time for writes). However, when reading from IMEM or DMEM, the memory blocks act as combinatorial logic blocks with a 4ns read delay.

Since all sequential logic takes 0 ns, the longest path must be reading from one of the two memories, DMEM or IMEM. There's no path that read from both IMEM and DMEM in the same cycle, so our longest path is any path that reads from one of IMEM or DMEM. This path has a 4ns combinatorial delay to read from memory.

In total, after a rising edge, we need to wait 2ns for values to appear at the register outputs (clk-to-q time). Then we have to wait 4ns for the longest path (reading a memory block) to execute. Then we have to hold the resulting signal at the register input for 2ns before the next rising edge (setup time).

Of course, **direction** tracking can be also implemented as a finite state machine with two states (1 = forward, 0 = backward) and 3 inputs: "CF" which is 1 when the current instruction is "[", "CB" which is 1 when the current instruction is "]", and "Mem" which is 1 if the current memory value is non-zero. For the four transitions on the below state transition diagram, write the most simplified logical equations for each edge.



Q5.12 (1 point) A

Solution: `!(CF & !Mem) = !CF | Mem`
While going backward, we could encounter four different things:
`[` instruction (`CF`) and the pointer is currently pointing at 0 (`!Mem`). Then we should change program flow to forward.
`[` instruction (`CF`) and the pointer is currently not pointing at 0 (`Mem`). Then we should keep program flow as backward.
`]` instruction (`CB`) and the pointer is currently pointing at 0 (`!Mem`). Then we should keep program flow as backward.
`]` instruction (`CB`) and the pointer is currently not pointing at 0 (`Mem`). Then we should keep program flow as backward.
In other words, the only case that doesn't result in arrow A is the first case with `CF & !Mem`.

Q5.13 (1 point) B

Solution: `!(CB & !Mem) = (!CB | Mem)`
Similar logic to the previous subpart. When moving forward, the only case that doesn't result in arrow B is the case with `CB & !Mem` (we would switch to moving backward in this case).

Q5.14 (1 point) C

Solution: `CB & !Mem`
See solution for arrow B. The only case that results in switching directions from backward to forward is the case with `CB & !Mem`.

Q5.15 (1 point) D

Solution: `CF & !Mem`
See solution for arrow A. The only case that results in switching directions from forward to backward is the case with `CF & !Mem`.

Question 6 ***I Forget Where This Data Goes*** **(14 points)**

For each question, select the option which best describes what an operation would evaluate to. If the operation would lead to something which is not a valid address, select "Not an Address". Consider this snipper of a C program.

```
void foo() {
    int64t w = 4;
    int64t* u = malloc(100 * sizeof(w));
    int64t** v = &u;
    ...
}
```

Q6.1 (2 points) What does `sizeof(*u)` evaluate to on a 32-bit system?

Solution: `8`

When we dereference `u`, we are now asking for the size of an `int64_t`. This is 8 bytes as that is the size of this type (64 bits / 8 bits = 8 bytes).

Q6.2 (2 points) What does `sizeof(u)` evaluate to on a 32-bit system?

Solution: `4`

This is because the type of `u` is a pointer to an `int64_t` meaning we are asking for the size of the pointer.

Q6.3 (2 points) What type of value does `v` evaluate to?

[✓ **Stack Address**] ○ Static Address

○ Heap Address ○ Not an Address

Solution: Note that unlike some other exams, we are not asking where the variable `v` is stored. Instead, we're asking where in memory the value in `v` is pointing at.

`v` contains the value `&u`. This is the address of a local variable, and we know that local variables are stored on the stack.

Q6.4 (2 points) What type of value does `*v` evaluate to?

○ Stack Address ○ Static Address

[✓ **Heap Address**] ○ Not an Address

Solution: `*v` evaluates to `*(&u)`, which evaluates to `u`. The value of `u` is an address on the heap, since `u` holds an address returned by `malloc`.

Again, note that unlike some other exam questions, we are not asking where the variable `u` is stored (that would be the stack). We're asking where the pointer `u` is pointing at (the heap).

Q6.5 (2 points) What type of value does `(u + 1)` evaluate to?

○ Stack Address ○ Static Address

[✓ **Heap Address**] ○ Not an Address

Solution: From the previous subpart, we know that `u` holds an address on the heap. If we add 1 to this address (actually, we would add `1 * sizeof(int64_t)`), we still get an address on the heap.

Q6.6 (2 points) What type of value does `*(u + 1)` evaluate to?

○ Stack Address ○ Static Address

○ Heap Address [✓ **Not an Address**]

Solution: From the previous subpart, `u+1` is an address on the heap. If we dereference this address `*(u+1)`, we get some data on the heap. This is space that was allocated by `malloc` and never initialized yet, so the data on the heap is garbage and probably not an address.

Q6.7 (2 points) What type of value does `&w` evaluate to?

[✓ **Stack Address**] ○ Static Address

○ Heap Address ○ Not an Address

Solution: `&w` is the address of the local variable `w`, and we know that local variables are stored on the stack. Thus the value `&w` is a stack address.

Question 7 ***Bloomin Onion*** **(20 points)**

A very clever data structure for efficiently and probabilistically storing a set is called a "bloom filter". It has two functions: `check` and `insert`. The basic idea for checking is that you hash what you are looking for multiple times. Each hash tells you a particular bit you need to set or check. So for checking you see if the bit is set. You repeat this for multiple iteration, with the hash including the iteration count (so each hash is different). If not all bits are set then the element does not exist in the bloom filter. If all bits are set then the element PROBABLY exists in the bloom filter. Similarly, for setting an element as present in a bloom filter you just set all those bits to 1.

We want to make a bloom filter design that is flexible and portable. So we design the following structure.

```
struct BloomFilter {
    uint32t size; /* Size is # of bits, NOT BYTES, in the bloom filter */
    uint16t itercount;
    uint64t (*)(void *data, uint16t iter) hash;
    uint8t *data;
};
```

Q7.1 (2 points) On a 32b architecture that requires word alignment for 32b integers and pointers, what is `sizeof(struct BloomFilter)`?

Solution: 16

`uint32_t size` takes up 32 bits = 4 bytes. Call these bytes 0-3 of the struct.

`uint16_t itercount` takes up 16 bits = 2 bytes. Call these bytes 4-5 of the struct.

The next field in the struct (`hash`) is a function pointer, and the question says we need word alignment for pointers. Word alignment means that the address of the struct field must be a multiple of the word size, which is 4 bytes on a 32-bit system.

Because we need the next field to be word-aligned, we add 2 bytes of padding. Call these bytes 6-7 of the struct. Now the next field will start at byte 8, which is a multiple of 4.

`hash` is a function pointer, which takes up 4 bytes. Call these bytes 8-11 of the struct.

`uint8_t *data` is a pointer, which takes up 4 bytes. Call these bytes 12-15 of the struct.

In total, we have bytes 0-15 of the struct, which is 16 bytes. Alternatively: $4 + 2 + 2 + 4 + 4 = 16$.

And now we have the insert function... For this we need to set the appropriate bit for each iteration.

```
void insert(struct BloomFilter *b, void *element){
    uint64t bitnum; /* which bit we need to set */
    int i;
    for(i = 0; i < (CODE INPUT 1); ++i){
        bitnum = (CODE INPUT 2);
        b->data[bitnum >> 3] = (CODE INPUT 3);
    }
}
```

Q7.2 (1 point) `(CODE INPUT 1)`:

Solution: `b->itercount`

The question says that for inserting, you need to repeat the steps for a certain number of iterations. From the code, we can infer that the `itercount` field of the struct is the number of iterations we need to repeat the steps.

In the `insert` function, we're given a pointer to a struct `struct BloomFilter *b`. We need to dereference this struct pointer to get the actual struct, and then access a field within a struct. Recall that the `->` (arrow) operator dereferences a struct pointer and accesses a struct field. The `.` (dot) operator would not work here, because it accesses a struct field given an actual struct, not a struct pointer.

Q7.3 (3 points) `(CODE INPUT 2)`

Solution: `b->hash(element, (uint16_t) i) % b->size`

According to the question, we need to hash the element we're trying to insert. The question also says that the hash needs to include the iteration count.

The struct has a `hash` field, which we can infer is a pointer to the hash function we need to compute. To access the function pointer, we need to use the arrow syntax (as described in the previous part), i.e. `b->hash`.

Recall that in C, when you have an expression for a function pointer, you can directly call the function pointer. For example, if `x` is a function pointer, then `x(3)` will call the corresponding function with argument 3.

Recall that in C, the syntax for function pointers lists the return value(s) and then the argument(s). In this case, we have `uint64_t (*)(void *data, uint16_t iter) hash` which says that the function takes in arguments `(void *data, uint16_t iter)` and returns a `uint64_t`.

In this case, we want to call the hash function with the element (the argument to `insert`) and the iteration count (we found this in the previous subpart).

The call to the hash looks like this: `b->hash(element, i)`. Casting `i` from an integer to a `uint16_t` is good practice (since the argument to the hash function is a `uint16_t`) but probably not strictly necessary.

Finally, the question says that "each hash tells you a particular bit you need to set or check," but there is no guarantee that the hash output is lower than the number of bits in the Bloom filter. If the hash output is greater than the number of bits in the Bloom filter, then `bitnum` will be too large, and the next line will index out-of-bounds. Therefore, we should use the mod operator on the hash output to ensure that the value in `bitnum` is not greater than the size of the Bloom filter (`b->size`).

Q7.4 (3 points) (CODE INPUT 3)

Solution: `b->data[bitnum >> 3] | (1 << (bitnum & 0x7))` (or equivalent)

The question says that once the hash tells us a particular bit we need to set, we'll need to set that bit to 1 as part of the insertion process.

However, note that C indexes memory in bytes, not bits, so we will need to perform some bit manipulation to set only one bit of memory.

The left side of this expression is `b->data[bitnum >> 3]`. Note that `bitnum >> 3` divides `bitnum` by 8. `bitnum/8` tells us which byte of the data array contains the bit we want to set. We divide by 8 because each byte is 8 bits, and we want to convert from the index measured in bits to the index measured in bytes. Then, we get this byte of the `data` array by indexing into the `data` array, which is an array of bytes (each `uint8_t` is 1 byte).

Once we figure out which byte contains the bit we're trying to set, we can determine exactly which bit to set by checking the bottom 3 bits of `bitnum`. Intuitively: `bitnum >> 3` takes the top bits (all but the bottom 3 bits), which are used to determine the byte containing the bit we want to change. Then, the bottom 3 bits are used to determine the offset within this byte (i.e. where the bit we want to change is located within the byte).

We can extract the bottom 3 bits of `bitnum` by using bitwise AND. Note that ANDing anything with 0 gives 0: `0 AND 0 = 0` and `0 AND 1 = 0`. We can use this property to set all but the bottom 3 bits to 0.

Also, note that ANDing a bit with 1 gives the same bit back: `1 AND 1 = 1` and `0 AND 1 = 0`. We can use this property to preserve the bottom 3 bits.

`bitnum & 0b111` performs a bitwise AND that zeroes out all but the bottom 3 bits of the number, and leaves the bottom 3 bits of the number unchanged. This is the offset within the byte, identifying the bit inside the byte that we want to change. In other words, if we call `bitnum & 0b111 = i`, then we want to change the `i`th bit inside the byte.

We want to set one bit in this byte, leaving the other 7 bits unchanged. We can do this by exploiting the properties of bitwise OR. Note that ORing anything with 1 gives 1: `1 OR 1 = 1` and `0 OR 1 = 1`. We can use this property to set a bit to be 1.

Also, note that ORing a bit with 0 gives the same bit back: `0 OR 0 = 0` and `1 OR 0 = 1`. We can use this property to leave the other bits of the byte unchanged.

In order to change the `i`th bit, we need to perform a bitwise OR where the `i`th bit is OR'd with 1 (which sets the bit to 1), and all other bits are OR'd with 0 (which leaves those bits unchanged). To do this, we need to build a number where the `i`th bit is 1, and all other bits are 0. We can do build this number by taking the bit `1` and left-shifting it by `i` places to put the 1 in the `i`th bit (the left-shift fills in zeros to the right of the 1).

Once we have this number `1 << i`, we take the byte and XOR the byte with `1 << i` to set only the `i`th bit to 1.

In summary:

`bitnum` is the index of the bit we want to set to 1, measured in bits.

`bitnum >> 3` identifies which byte of the data array contains the bit we want to set.

`b->data[bitnum >> 3]` contains 8 bits from the data array. We want to set one of these bits to 1.

`bitnum & 0x7` uses bitwise AND to extract the bottom 3 bits of `bitnum`, which identifies which bit within the selected byte we want to set.

`1 << (bitnum & 0x7)` creates a number with the `(bitnum & 0x7)`'th bit set to 1.

`b->data[bitnum >> 3] | (1 << (bitnum & 0x7))` takes the relevant byte from th

We also have the following function to allocate a new bloom filter

```
struct BloomFilter *alloc(
      uint64t (*)(void *data, uint16t iter) hash,
      uint32t size,
      uint16t itercount){
    struct BloomFilter *ret = malloc(64);
    /* Yes, this is way too big, but we don't want to give you the answer
       to the previous question!  */
    ret->size = size;
    ret->data = calloc(size >> 3, 1);
    ret->hash = hash;
    ret->itercount = itercount;
}
```

Complete the RISC-V translation necessary to allocate this: We will put `ret` in `s0`.

```
alloc: # Prolog
    (CODE INPUT 1)
    sw ra 0(sp)
    sw s0 4(sp)
    sw a0 8(sp)
    sw a1 12(sp)
    sw a2 16(sp)
# body
    addi (CODE INPUT 2)
    jal malloc
    mv s0 a0          # put ret in s0
    (CODE INPUT 3)    # load size into t0
    (CODE INPUT 4)    # store it
    (CODE INPUT 5)    # div size by 8 with a shift
    jal calloc
    sw a0 12(s0)      # store data
    (CODE INPUT 6)    # load hash to t0
    (CODE INPUT 7)    # store it: Use the right type!
    (CODE INPUT 8)    # load itercount to t0
    (CODE INPUT 9)    # store it: Use the right type!
    mv a0 s0
# epilog
    lw ra 0(sp)
    (CODE INPUT 10)
    (CODE INPUT 11)
    jr ra
```

Q7.5 (1 point) `(CODE INPUT 1)`

Solution: `addi sp, sp, -20`
In the lines following this blank, we're storing the values in 5 registers on the stack. Each register contains 4 bytes, so we need to make $5 \times 4 = 20$ bytes of space on the stack.

Q7.6 (1 point) `(CODE INPUT 2)`

Solution: `a0, x0, 64`
The line directly following this blank calls `malloc`, so we should pass an argument into `malloc` first. In the provided C code, we're calling `malloc(64)`, so we should put the number 64 in `a0`, the first argument register.

Q7.7 (1 point) `(CODE INPUT 3)`

Solution: `lw t0, 12(sp)`
`size` is the second argument to the function, passed to the function in `a1`. However, according to calling convention, `a1` is a non-preserved register, so the call to malloc could have modified `a1` (in other words, you must assume that `a1` contains garbage after the function call). Luckily, we saved the value of `a1` on the stack in the prologue, so we can load the value of `a1` from memory and put it back into the `t0` register (as the comment says to do).

Q7.8 (1 point) `(CODE INPUT 4)`

Solution: `sw t0, 0(s0)`
We want to store the value of `size` into the struct that we just allocated memory for. In the struct definition, `size` is the first field, so it's stored at the start of the struct (with offset of 0).
The address of the struct we allocated was in `a0` after the call to `malloc` (since the return value of functions goes in `a0`). Then on the line immediately after the function call, we move the value to `s0`.
In summary, we take the value of `size` (in `t0` from the previous blank), and store it to the address in `s0` (the address of the struct we allocated), with an offset of 0 (because `size` is the first field of the struct).

Q7.9 (1 point) `(CODE INPUT 5)`

Solution: `srli a0, t0, 3`

Recall that shifting a number right by 1 bit divides the number by two. To divide a number by 8, we have to divide the number by two 3 times, which is equivalent to shifting the number right by 3 bits. (The C code also shows that you have to shift right by 3.)

Be careful to use `srli` (shift right logical immediate) and not `srai` (shift right arithmetic immediate), because we want to fill in the 3 empty bits created by the shift with zeros. We don't want to sign-extend the number, since `size` is an unsigned number.

`t0` still holds `size`, so we shift the value in `t0` by 3.

Note that immediately after this line, we're calling `calloc`, so we also need to set up arguments to `calloc`. In particular, `size >> 3`, the value we just calculated, is the first argument to `calloc`, so we should put this value in `a0`.

Q7.10 (1 point) `(CODE INPUT 6)`

Solution: `lw t0, 8(sp)`

`hash` was the first argument to the `alloc` function, so it was placed in `a0` at the start of the function. However, our code has overwritten the value in `a0` a few times, so `hash` is no longer in `a0`. However, in the prologue, we stored the value of `a0` (`hash`) on the stack, so we can load `hash` back from the stack and put the value in `t0`.

Q7.11 (1 point) `(CODE INPUT 7)`

Solution: `sw t0, 8(s0)`

We want to store `hash` in the struct in memory that we allocated. From blank 4, we know that `s0` holds the address of the struct in memory. (The call to `calloc` won't overwrite `s0` because of calling convention.)

`hash` is the third field of this struct, located at bytes 8-12 of the struct (see the first subpart of this question for why bytes 8-12). Thus we need to store at an offset of 8 bytes from the start of the struct.

`hash` is a pointer (4 bytes), so we need to use the store-word instruction.

Q7.12 (1 point) `(CODE INPUT 8)`

Solution: `lw t0, 16(sp)`

Similar logic to blanks 3 and 6. `itercount` is the third argument to the `alloc` function, in `a2` at the start of the function, and stored on the stack in the prologue.

Q7.13 (1 point) `(CODE INPUT 9)`

Solution: `sh t0, 4(s0)`

Similar logic to blanks 4 and 7. `itercount` is the second field of this struct, located at bytes 4-5 of the struct (see the first subpart of this question for why bytes 4-5). Thus we need to store at an offset of 4 bytes from the start of the struct.

`hash` is a `uint16_t` (2 bytes), so we need to use the store half-word instruction.

Q7.14 (1 point) `(CODE INPUT 10)`

Solution: `lw s0, 4(sp)`

According to calling convention, the `alloc` function is allowed to modify all `t` and `a` registers, but not allowed to modify any `s` registers.

Throughout the function, we changed the value in `s0`, so we need to save the value of `s0` at the start of the function and restore the original value of `s0` before the function returns.

Q7.15 (1 point) `(CODE INPUT 11)`

Solution: `addi sp, sp, 20`

In the first blank, we allocated 20 bytes of space on the stack. Before the function returns, we need to free up this space since we're done using that memory.

Question 8 ***CALL me maybe*** **(12 points)**

For the following two parts, please indicate if they always or never need to be relocated.

Q8.1 (2 points) PC-Relative Addressing

[✓ **Never Relocate**] ○ Always Relocate

Solution: TODO

Q8.2 (2 points) Static Data Reference

○ Never Relocate [✓ **Always Relocate**]

Solution: TODO

Q8.3 (3 points) Select all the steps that are done during the Assembler phase of CALL

- [✓ **Producing machine language**]
- ☐ Outputting executable code
- ☐ Generating Assembly code
- ☐ Lexing the C code
- ☐ Semantic Analysis
- [✓ **Pseudo-instruction replacement**]
- ☐ Parsing the C code

Solution: TODO

Q8.4 (3 points) Describe two benefits of using Dynamically Linked Libraries

Solution: There's more than two right answers to this question, but the most common ones we accepted were:

- Saving resources (such as disk space or memory) by sharing data (they do not need the same extra data)
- Easier upgrading as we only need to replace the DLL file (we do not need to recompile the code which used the DLL library)
- Multi-language programming: The DLL may be written in a different language as what you are using.
- System independence/standardized interface: A DLL may offer a standard interface for hardwared of a machine which allows for an abstraction from the main program

Q8.5 (2 points) What are assembler directives (explain what they are used for, don't just give examples of them)?

Solution: They give directions to the assembler, but do not produce machine instructions.