Nick Weaver
Spring 2019

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 TODO (170?) minutes. There are 9 questions of varying credit, for a total of 180 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 ***Potpourri*** **(25 points)**

Q1.1 FSE (your favorite search engine) corporation has several datacenters (WSCs) that use an interrupt-based approach to handling network-interface events. The context switch from the current process to the interrupt handler takes 15 microsec- onds of overhead, while the actual handling of the packet takes 45 microseconds. How long does it take us to process one packet given the interrupt model?

> **Solution:** 60 microseconds
>
> To process a packet, we have to trigger an interrupt, which takes 15 microseconds, then process the packet, which takes 45 microseconds. $15 + 45 = 60$.

Q1.2 The engineers have a new solution that uses polling. Polling has only 1 microsecond of overhead, and the polling interval is every 60 microseconds. How long does it take us to process one packet given the new polling model?

> **Solution:** 46 microseconds
>
> To process a packet, we have to wait until the next poll, but the time spent waiting until the next poll isn't spent processing a new packet.
>
> We only start processing a new packet when the next poll starts. At that point, we have to wait 1 microsecond of overhead, then 45 microseconds to process the packet. $1 + 45 = 46$.

Q1.3 If there is no network traffic, what percentage of the CPU will be spent polling?

> **Solution:** 1.66% or $(1/60) \times 100$
>
> There's no network traffic, so we don't need to consider the time to process a packet. We just poll every 60 microseconds, only to discover that there's nothing to process.
>
> In every 60 microseconds, we spend 1 microsecond of overhead polling. That's 1/60 of the total CPU time spent processing. We multiply by 100 to express the fraction of time as a percent.

Q1.4 The engineers have another, slightly more complicated solution that uses interrupts and polling. Here, the first arriving packet generates an interrupt with 15 microseconds of overhead, and the next 4 packets (which are guaranteed to have arrived) can be polled with a 1 microsecond overhead each. Assume processing cannot happen in parallel. What is the average time to process a single packet in this new model? Leave your answer unsimplified.

Solution: $(15 + 4 \times 1 + 5 \times 45)/5 = 48.8$ microseconds

We have to take a weighted average of the different cases.

Case A, interrupt: interrupt overhead time, plus processing time $= 15 + 45$

Case B, polling: polling overhead time, plus processing time $= 1 + 45$

In every set of 5 packets, 1 packet follows Case A, and 4 packets follow Case B. The weighted average looks something like:

$\frac{1}{5}(15 + 45) + \frac{4}{5}(1 + 45) = 48$

The expression in the original solutions follows pretty similar logic. In a set of 5 requests, we have to wait 15 microseconds for one interrupt, 4×1 microseconds for 4 polls, and 5×45 microseconds for 5 packets to be processed. This is the total time for processing 5 packets. We divide by 5 to get the average time of processing 1 packet.

Q1.5 FSE really appreciates your help with fixing their networking stack, but now they are looking to do some hardware upgrades and it is up to you to analyze if the upgrades are worthwhile!

As a datacenter, YFSE spends 4 MW (Mega Watts) on compute, 2 MW on networking, 2 MW on storage to serve their customers. They also spend 2 MW on cooling, and 1 MW on power and other sources.[a]

What is the current PUE? Simplify to a fraction of the from A/B where A and B are single integers.

Solution: 11/8

PUE is the total power used, divided by the power used on actual computing equipment.

This datacenter uses $4 + 2 + 2 + 2 + 1 = 11$ MW in total.

This datacenter uses $4 + 2 + 2 = 8$ MW for computing. (The question notes that networking and storage, in addition to computing, is considered useful work.)

[a] We consider networking and storage as "useful work".

Q1.6 If the new networking hardware uses 1/2 the power of the old networking hardware, what will the new PUE be? Simplify to a fraction of the from A/B where A and B are single integers.

> **Solution:** 10/7
>
> Same calculation as the previous subpart, but we replace 2 MW on networking with 1 MW on networking.
>
> The datacenter uses $4 + 1 + 2 + 2 + 1 = 10$ MW in total.
>
> The datacenter uses $4 + 1 + 2 = 7$ MW for computing.

Q1.7 YFSE runs a test program to see how the new system performs. They see the program spends 3% of its time traversing the network (latency), and 7% of its time actually transferring (transmission delay).

If the new networking hardware speeds up our network traversal by a factor of 1.5 and also speeds up transmission by a factor of 1.75, what is the speedup of the test program? Don't simplify.

> **Solution:** $\frac{1}{(1-0.1)+\frac{0.03}{1.5}+\frac{0.07}{1.75}}$
>
> Assume the test program takes 1 unit of time to run on unoptimized hardware.
>
> On the new networking hardware, 3% of the computation is sped up by a factor of 1.5, and 7% of the computation is sped up by a factor of 1.75. This leaves $100\% - 3\% - 7\% = 90\%$ of the computation time unchanged.
>
> In total, the new networking hardware takes $0.9 + \frac{0.03}{1.5} + \frac{0.07}{1.75}$ units of time to run the test program.
>
> We divide the 3% and 7% of running times by 1.5 and 1.75 respectively to account for speedup. (Intuitively, if something is sped up by a factor of 2, it takes half as long to run, so we would divide the running time by 2.)
>
> To calculate overall speedup, we divide the unoptimized running time (1 unit of time) by the optimized running time (which we just calculated above).

Q1.8 FSW has some FPGA based accelerators, so for this custom hardware they are using a 16 bit floating point scheme which works the same as our in-lecture version, except it has the following breakdown:

Sign: 1 bit, Exponent: 7 bits, Signficand: 8 bits

Represent the given number in our scheme by filling in fields below. If you cannot represent the number **exactly**, you should select NOT REPRESENTABLE

$-3/64$

Solution:

Sign: `0b`1

Exponent: `0b`011 1010

Significand: `0b`1000 0000

SOLUTION-TODO: Format question blanks

Convert the number to scientific notation:

$$-3/64 = -3 \times 2^{-6}$$
$$= -\texttt{0b}1.1 \times 2^{-6} \qquad = -\texttt{0b}1.10000000 \times 2^{-6}$$

Sign bit is `0b1` because the number is negative.

Exponent is -6. We encode this by subtracting the bias. The bias isn't specified in the question, so we default to $-(2^{7-1} - 1) = -63$. Subtracting the bias gives $-6-(-63) = 57$. Encoding 57 in 7-bit unsigned binary gives `0b`0111010. SOLUTION-TODO: Is this wrong?

Significant is `0b`10000000.

$1/6$

Solution: Not representable

One useful observation about floating point numbers is that since you can always write them in binary scientific notation, they can always be written in the form of some integer, multiplied by a power of 2. In other words, if you cannot write a number as an integer multiplied by a power of 2, then it cannot be represented in floating-point. $1/6 = 1/3 \times 2^{-1}$ cannot be represented as an integer multiplied by a power of 2, because there's no way to encode the 1/3 as an integer or a power of 2. Therefore, 1/6 can't be represented in floating-point.

Another way to see this is to note that $1/6 = 0.16666666\ldots$ is an infinitely repeating decimal, and floating-point is limited to finite precision. The question says to select not representable if we cannot exactly represent the number, and floating-point cannot exactly represent numbers with infinite precision.

Q1.9 Find the average memory access time for a system with the following characteristics. For partial credit, write the formula. For full credit, simplify your answer. Assume we check the TLB, then the page table. For this problem, we ignore page faults.

Translation:

TLB Hit Time	0ps*
TLB Miss Rate	20% of accesses
Page Table Hit Time	60 ps

Data Access:

Data Cache Hit Time	10 ps
Data Cache Miss Rate	25% of accesses
Memory Access time	40 ps

Solution: $0 + .2(60) + 10 + .25(40) = 32$ ps

We have to compute two AMATs, one for the page table entry (to translate virtual page number to physical page number), and one for the actual data.

Page table entry AMAT can be read directly off the first table: We always have to hit the TLB, which takes 0 ps. Then, 20% of the time, we have to pay an extra 60 ps to check the page table in memory.

Data access AMAT can be read directly off the second table: We always have to hit the data cache, which takes 10 ps. Then, 25% of the time, we have to pay an extra 40 ps to access memory.

We add the two AMATs together for the total access time.

Question 2 ***Damon-path*** **(21 points)**

SOLUTION-TODO: Attempt to dig up the diagrams.

Q2.1 Which of the options below is the best fit for box I (1) on the previous page? Fill in the multiple choice bubbles below.

Solution: The new `madd` instruction needs to read from 3 registers and write to 1 register. This means that we need three read inputs to the Regfile. This eliminates option (D).

The `WriteAddr` input to the Regfile is used to identify which register we want to write to. Writing to the Regfile happens during the write-back stage, so we should use the a signal from the write-back stage for the `WriteAddr` input. This eliminates option (B).

If we're executing a non-`madd` instruction, we can just ignore the output from `ReadData3`. This is similar to how we ignore the output from `ReadData2` from instructions like `jal` that read from a second register (there's no `rs2`). The AND gate in option (C) may actually cause issues with reading, since during a `madd` instruction, the non-zero 5-bit `rd` gets ANDed with a 1-bit `madd` value of 1, which may zero out the top 4 bits of `rd`. The AND gate also isn't necessary, since as discussed, we can just ignore `ReadData3` on other instructions. This eliminates option (C).

In summary, we want option (A): We need a third read input, which should read from `rd`, and we need a write input that comes from the write-back stage.

Q2.2 Which of the options below is the best fit for box II (2) on the previous page? Fill in the multiple choice bubbles below.

Solution: In the new `madd` instruction, we need the ALU to compute `R[rd] + (R[rs1] * R[rs2])`. This means that one of our inputs to the ALU should be `R[rd]` and the other input should be `(R[rs1] * R[rs2])`.

None of the answer choices in this subpart have `R[rd]`, so this first input to the ALU must be `(R[rs1] * R[rs2])`.

However, we can't break existing datapath functionality. In the regular datapath, this box usually has a mux: if `ASel` is 0, we input `R[rs1]` into the ALU, and if `ASel` is 1, we input `PC` into the ALU.

Now we want to add an extra condition: If `madd` is 1, we should input `(R[rs1] * R[rs2])` into the ALU.

Option (A) and (C) are ruled out: If `ASel` is 1, we input `(R[rs1] * R[rs2])`, which breaks existing instructions. (Note that even if `madd` is 0, we'd still input `(R[rs1] * R[rs2])`, breaking existing instructions).

Options (A) and (B) are ruled out: We're only able to input `(R[rs1] + R[rs2])`, not `(R[rs1] * R[rs2])`.

This leaves Option (D), which is correct. It retains the existing datapath logic of sending in `PC` if `ASel` is 1 and `R[rs1]` if `ASel` is 0 and `madd` is 0. However, it adds the functionality of sending in `(R[rs1] * R[rs2])` if `madd` is 1 (you would then need to set `ASel` to 0 for `madd` instructions).

Q2.3 Which of the options below is the best fit for box III (3) on the previous page? Fill in the multiple choice bubbles below.

Solution: Continuing from the previous part: for `madd` instructions, the second input we need to send to the ALU is `R[rd]`.

Just like before, we cannot break existing datapath functionality. When `madd` is 0 and `BSel` is 0, we should still send `R[rs2]` to the ALU. When `madd` is 0 and `BSel` is 1, we should still send `imm` to the ALU.

Option (A) breaks existing functionality: When `madd` is 0 and `BSel` is 0, we send `R[rd]` to the ALU instead of `R[rs2]`.

Options (B) and (D) will never select `R[rd]` to send to the ALU. `R[rd]` would require the select bits of the mux to be 2 (`0b10`, but the AND/XOR gates only output a 1-bit value (since both of the inputs are 1-bit).

This leaves option (C), which matches the existing functionality described above, and adds one extra case: When `madd` is 1, we caFill in the correct values for the control signals below to correctly execute a madd instruction. Assume other control bits have been done for you and are correct. For ALUSel, please write an operation name (ie. add) not a numeric value. For RegWEn and WBSel, use integer, decimal numbers.n send `R[rd]` to the ALU by also setting `BSel` to 1 for `madd` instructions.

Q2.4 Fill in the correct values for the control signals below to correctly execute a madd instruction. Assume other control bits have been done for you and are correct. For ALUSel, please write an operation name (ie. add) not a numeric value. For RegWEn and WBSel, use integer, decimal numbers.

Solution: ALUSel should be `add`. As described above, we're using the ALU to compute `R[rd] + (R[rs1] * R[rs2])`, and we used Box II and Box III to send `(R[rs1] * R[rs2])` and `R[rd]` to the ALU.

RegWEn should be 1, since we're writing something back to the Regfile.

WBSel should be 1, since we want to write the ALU output back to the Regfile (not data from memory, and not PC+4).

Q2.5 Given the timing specifications below, calculate the critical path of the new pipelined datapath.

Solution: 325 ps

We're looking for the longest path between any two clocked elements. (Start a path at a register output, and end the path at the RegFile write or a register input.)

The longest path in the IF stage starts at the PC register output, passes through IMEM, and ends at the next pipeline registers. This has a combinatorial delay of 250 ps.

The longest path in the ID stage starts at the pipeline registers, passes through Regfile (to read values, not write anything), and ends at the next pipeline registers. This has a combinatorial delay of 150 ps.

The longest path in the EX stage starts at the pipeline registers, passes through Box II (or Box III) and the ALU, and ends at the next pipeline registers. This has a combinatorial delay of $75 + 200 = 275$ ps.

The longest path in the Mem stage starts at the pipeline registers, passes through the DMEM (to read values, not write anything), and ends at the next pipeline registers. This has a combinatorial delay of 250 ps.

The longest path in the write-back stage starts at the pipeline registers, passes through a mux, and ends at the RegFile (for writing). This has a combinatorial delay of 25 ps.

The longest combinatorial delay is in the EX stage, with 275 ps.

The total clock period time for this circuit is the clk-to-q delay, plus the longest combinatorial delay, plus the setup time, which is $30 + 275 + 20 = 325$ ps.

In other words: When the clock has a rising edge, we wait 30 ps (clk-to-q delay) for the signal to appear at the register outputs. Then we wait for the longest combinatorial delay to finish (275 ps EX stage). Then, we have to wait another 20 ps (setup time) for the signal to stabilize at the next register input before the next rising edge can happen.

Question 3 ***Virtual Memory*** **(15 points)**

Assume we're working on a machine which has the following parameters:

- 16GiB of physical memory
- 22 bit virtual addresses
- 128B pages
- Each PTE in our single level table is 4B

Q3.1 How many bits are in the Physical Address Offset?

Solution: 7 bits

Each page (whether virtual or physical) is 128 bytes. We need 7 bits to uniquely find one byte within this page.

Q3.2 How many bits are in the PPN? the VPN?

Solution: VPN = 15 bits, PPN = 27 bits

The number of offset bits (7) is the same for both virtual and physical addresses (since the page size is the same).

Virtual addresses have 22 bits. 7 of those bits are used for the offset, leaving $22-7 = 15$ bits for the VPN.

We have 16 GiB = 2^{34} bytes of physical memory, so we need 34-bit physical addresses to uniquely identify each byte of physical memory. 7 of those bits are used for the offset, leaving $34 - 7 = 27$ bits for the PPN.

Consider the following snippet of code. Assume the following:

- Arrays begin on page boundaries.
- We have a function `initialize` that creates an array containing size integers.
- A starts at 0x20000 and B starts at 0x30000
- `sizeof (int)` == 4

```
int size = 256;
int32 t *A = initialize(size);
int32 t *B = initialize(size);
for (int i = 0; i < size; i += 32) {
    B[i] = B[size - i - 1] * A[i];
}
```

For the following parts, assume "data pages" refers only to pages containing elements of A and B (ie. not pagetable pages). Remember that we have 128B pages and each PTE is 4B.

Q3.3 How many unique DATA pages does this access pattern traverse?

- ○ 0 pages
- ○ 2 pages
- ○ 3 pages
- ○ 8 pages
- ○ 12 pages
- ○ 13 pages
- [✓ **16 pages**]

Solution: `A` is 256 elements, and each element is 4 bytes, for a total of 1024 bytes. `B` is also 1024 bytes by the same logic.

Each page is 128 bytes, so `A` takes up 8 pages of memory. Likewise, `B` takes up 8 pages of memory by the same logic.

Each page is 128 bytes, and each element is 4 bytes, so 32 elements of the array fit on each page.

This loop accesses every 32nd element of `A` (e.g. `A[0]`, `A[32]`, etc.). This loop also accesses every 32nd element of `B`. This means that we eventually need to access every page of both arrays in memory, for a total of $8 + 8 = 16$ pages.

(The accesses to `B[size - i - 1]` don't really change the answer to these questions, because we end up having to access every page of `B` anyway.)

Q3.4 How many unique NON-DATA, NON-CODE pages does this access pattern traverse? (ie. page table pages)

- ○ 0 pages
- [✓ **2 pages**]
- ○ 3 pages
- ○ 8 pages
- ○ 12 pages
- ○ 13 pages
- ○ 16 pages

Solution: Faster intuitive answer: All 8 PTEs for `A` fit on one page, and all 8 PTEs for `B` (whose address is really far away from `A`) fit on a different page.

More comprehensive answer: As mentioned in the previous subpart, array `A` takes up 8 pages of memory. This means that `A` has 8 VPNs that we need to translate to 8 PPNs. This means that we need 8 PTEs to map each of the 8 VPNs to 8 PPNs. The same logic applies to `B`, so we need 16 PTEs in total.

First, note that pages are 128 bytes, and each PTE is 4 bytes, so we can fit 32 PTEs on each page. However, this does not mean that all 16 PTEs are located in the same page in memory.

To find where the 16 PTEs live in memory, we need to check their VPNs (since we use the VPN to index into the page table in memory).

From the previous part, `A` is 1024 = `0x400` bytes long. This means that `A` takes up memory addresses from `0x20000` to `0x20400`.

Converting virtual addresses to binary, we get that `0x20000` = `0b00 0010 0000 0000 0000 0000` and `0x20400` = `0b00 0010 0000 0100 0000 0000`.

The VPN is the top 15 bits, so the virtual page numbers that hold the array `A` are `0b000 0100 0000 0000` and `0b000 0100 0000 1000`. (Aside: These two numbers differ by `0b1000` = 8, which confirms that we needed 8 pages of memory for the array.)

By the same logic, we can extract the virtual page numbers of `B`. Converting to binary, we get that `0x30000` = `0b00 0011 0000 0000 0000 0000` and `0x30400` = `0b00 0011 0000 0100 0000 0000`.

The VPN is the top 15 bits, so the virtual page numbers that hold the array `B` are `0b000 0110 0000 0000` and `0b000 0110 0000 1000`.

Note that the VPNs for `A` and `B` differ by around `0b000 0010 0000 0000` = 2^9. This means that the PTEs for `A` and the PTEs for `B` are around 2^9 PTEs away from each other in memory. This is much further than the 32 PTEs that fit on one page, so the PTEs for `A` and the PTEs for `B` must live on different pages.

In summary: `A` has 8 PTEs that all live on one page in memory. `B` has 8 PTEs that all live on some different page in memory.

Q3.5 Suppose this process has a single-level page table that starts out empty and we run through this access pattern. How much space would the page table take in memory? Give your answer in units of PAGES (*NOT BYTES*).

Solution: Regardless of access pattern, we need to store the entire single-level page table in memory. SOLUTION-TODO: Pretty sure this is fine, but double-check it.

The page table maps every VPN to a PPN. (In other words, each VPN has its own PTE.)

The VPN is 15 bits long, so we have 2^{15} different VPNs. This means we have 2^{15} different PTEs.

Each PTE is 4 bytes, so the page table is $2^{15} \times 4 = 2^{17}$ bytes.

Each page is $128 = 2^7$ bytes, so we need $2^{17}/2^7 = 2^{10}$ pages to store the entire single-level page table.

Question 4 ***Nick Goes Nuclear - Atomics*** **(17 points)**

In class you learned about OpenMP and got to experience speedups on the Hive Machine. However after your time in 61C you developed a deep-seated hatred for X86 and have determined that you want to employ OpenMP on RISC-V machines using atomic instructions.

You decide to start small and you seek to implement the following parallelization of summing a loop.

```
int sum = 0;
#pragma omp parallel for {
for (int i = 0; i < n; i++) {
    #pragma omp critical
    sum += A[i];
}
```

When executing the for loop, each thread holds its local starting and terminating byte offset in t0 and t1 respectively. You store the address of `sum` in s1 and the address of `A` in s2. Now you are tasked with implementing the actual `sum` update. You develop the following code which WORKS:

```
loop start:
    beq t0 t1 end
    add t2 s2 t0
    lw t2 0(t2)
retry:
    lr.w t3 (s1) # Load sum and place our reservation
    add t3 t3 t2
    sc.w t4, t3 (s1)
    bne t4 x0 retry # Check if our store failed
    addi t0 t0 4
    j loop start
```

Q4.1 Your friend, however, took 61C back in Fall 2017, so he only understands amoswap. Your friend asks if you could reimplement the same piece of coding using amoswap instead, without needing any values other than those in t0, t1, s1, and s2. Is this possible? Why or why not?

Yes [✓ **No**]

Solution: To increment the sum atomically using `amoswap` you need to already know the old value of `sum` (otherwise you cannot replace it with the incremented value). This requires another atomic access for the read, most likely through a lock.

Q4.2 SOLUTION-TODO: Typeset the rest of the question, but this isn't in scope for SU22 so who cares for now.

Question 5 *SIMD* (26 points)

Eh too lazy for this one sorry.

Question 6 *C Reading* (16 points)

SOLUTION-TODO: Transcribe stuff.

Q6.1 SOLUTION-TODO: Transcribe stuff. I mainly wanted to get this fucking 3 year old typo fixed.

Solution: Line 6 is wrong. `* arr + i` should be `*(arr+i)`. Without parentheses, the dereference operator takes precedence over addition, so we would first compute `*arr` to get the value of the first element in the array, then add `i` to the value of the first element in the array. However, what we actually want here is the value of the `i`th element of the array. Therefore, we should take the address `arr`, add `i` to find the address of the `i`th element, and then dereference the address `(arr+i)` to get the `i`th element of the array.

Line 7 is wrong. We're comparing `pointer`, which is an address, to `"STOP"`, which is the address of a hard-coded static string in memory. This will only return true if the two addresses are the same (i.e. both are pointing at the same string in memory). However, what we actually want here is to check that the string that `pointer` is pointing at is the string `STOP`. We can replace this line with something like `strcmp(pointer, "STOP")` instead.

Line 9 is wrong. `init_size == i - 1` should be something like `i == init_size - 1`. This expression is checking when we need to reallocate space in the `output` array. If we wait until `init_size == i - 1`, we could have a case where `init_size = 8` (i.e. `output` has space for 8 pointers), but `i = 9` (i.e. we're about to access the 9th element of the `output` array at Line 13).

Line 11 is wrong. `realloc` returns the address of the resized array, which is not necessarily the same as the address of the original array. We need to assign the result of `realloc` back to `output` so that `output` is pointing at the newly-reallocated array. Something like `output = realloc(...)` works, where the arguments to `realloc` stay the same.

Line 13 is wrong. `strlen` doesn't count for the null terminator. When we make space for the string, we need to add 1 to make space for the null terminator. For example, the string `hello` takes 6 bytes in memory, but `strlen("hello")` would return 5. Something like `output[i] = malloc(sizeof(char) * (strlen(pointer) + 1))` would work.

Q6.2 SOLUTION-TODO: Transcribe, are we splitting up this part?

Solution: `arr[0]` is the address of some string. The question doesn't specify where strings are being stored, and strings could be in heap, stack, or static memory. Strings can't be in code, because that's where we put the actual instructions of the program.

`"STOP"` is a hard-coded string stored in static memory.

`output[0]` was assigned to the return value of `malloc` on Line 13, so it's a pointer to the heap.

`output` is a local variable stored on the stack, so `&output` is an address on the stack.

`parse_message` is a function, whose instructions are stored in the code section. `&parse_message` is the address of the instructions in the code section.

Question 7 ***Go Go Power Potpourri*** **(21 points)**

Eh, too lazy for this one, sorry.

Question 8 *Gotta Cache 'em All* (19 points)

Consider a 8-way set associative cache with 64 B blocks, and 64 total blocks as part of a 16 bit physical address

Q8.1 Given the machine specs above, how big is each field?

Tag

Solution: 7 bits

The index and offset use up $3 + 6 = 9$ bits (see next parts). The address has 16 bits. This leaves $16 - 9 = 7$ bits for the tag.

Index

Solution: 3 bits

From the question, there are 64 blocks. The cache is 8-way set associative, so there are 8 blocks in each set. This means there are $64/8 = 8$ sets in the cache.

We need 3 bits to uniquely identify one of the 8 sets in the cache.

Offset

Solution: 6 bits

From the question, each block is 64 bytes. We need 6 bits to uniquely identify one of the 64 bytes.

Now imagine we use the same cache on the following RISC-V code:

```
.data:
    arr: .byte 0, 1, 2 , ... 255 # All values from 0 to 255
.text:
        # ASSUME A WORKING PROLOGUE
    la a0 arr
    li a1 256
        #Scramble randomizes the elems of arr
    jal scramble
        # Assume t0 = 0, t1 = 256, s0 = A, s1 = B, s2 = C
        # START OF HIT RATE
Start:
    beq t0 t1 End # Iterate 256 times
    add t2 a0 t0
    lbu t2 0(t2) # t2 = arr[t0]
    add t3 s0 t2
    lw t3 0(t3) # t3 = A[t2]
    add t4 s1 t2
    lw t4 0(t4) # t4 = B[t2]
    add t3 t3 t4
    add t4 s2 t0
    sw t3 0(t4) # C[t0] = t3 + t4
    addi t0 t0 1
    j Start
        # END OF HIT RATE
End:
        # ASSUME A WORKING EPILOGUE
```

Let `scramble` be a function that randomly sorts the elements of an array. Additionally assume that:

- A is located at 0x1000
- B is located at 0x2000
- C is located at 0x3000
- arr is located at 0x4000
- Our cache is empty when reaching `# START OF HIT RATE`

Q8.2 What is the best case hit rate for this code? Write your answer as a fraction.

Solution: 63/64

Start by thinking about what this code is doing. In the `i`th iteration of the loop, we get the `i`th element of `arr`. In other words, we access `arr` sequentially. We know that `arr` is a scrambled list of numbers from 0 to 255, so we'll get a random number between 0 and 255. Call this random number `k`. Then we access `A[k]` and `B[k]`. In other words, we access a random element of `A` and `B`. Then we access `C[i]`. In other words, we access `C` sequentially.

Now, consider the first iteration of the loop. First, we access `arr[0]`. This brings the first 64 bytes = 64 elements in `arr` into the cache. Now, we make a random access to `A[k]`, which causes us to bring a block of 64 bytes = 64 elements of `A` into the cache. If we access `B[k]`, we bring in another block of 64 bytes = 64 elements into the cache. Finally, we access `C[0]`, which brings the first 64 bytes = 64 elements in `C` into the cache. (Note that all the arrays in this question are byte arrays, because RISC-V indices memory in bytes and we never multiplied offsets by 4 or any other variable size.)

After the first iteration of the loop, we have the first block of `arr`, the first block of `C`, some block of `A`, and some block of `B` in the cache. The cache is 8-way set associative, so there can't be a conflict miss yet (we only brought in 4 blocks so far).

Now, consider how much the cache can fit before we have to start kicking blocks out. Each array is 256 elements = 4 blocks long. Each array starts at a cache-aligned boundary (the addresses `0x1000`, `0x2000`, `0x3000`, `0x4000` all have index bits `0b000` and offset bits `0b0000000`). In other words, we can bring an entire array into the cache by putting 4 blocks at indices 0, 1, 2, and 3.

Each index can fit 8 blocks (since the cache is 8-way set associative), and we have 4 arrays. This means that all 4 arrays can fit entirely in the cache. Index 0 will have block 0 of each array, index 1 will have block 1 of each array, index 2 will have block 2 of each array, and index 3 will have block 3 of each array.

Since all 4 arrays fit entirely in cache, we never have to kick any blocks out. Once a block is brought into the cache, it stays in the cache until the end of the program.

This means that the best-case and worst-case hit rate follow the same pattern; no matter what order you do it, you eventually need to bring all the blocks into the cache. Each block has 64 elements that you access exactly once (again, regardless of order). You miss on the first element and hit on the next 63 elements. This gives the hit rate of 63/64.

Q8.3 What is the worst case hit rate? Write your answer as a fraction.

Solution: 63/64

See solution to the previous part.

Q8.4 Now assume that we can modify the associativity without changing any other property. What is the minimum associativity for which the best case hit rate can equal the best case hit rate with 8 way set associative?

- ○ 1-way (Direct Mapped)
- [✓ **2-way**]
- ○ 4-way
- ○ 8-way
- ○ 16-way
- ○ Fully Associative

Solution: Important observations from the previous part: each array is 4 blocks long, and there are 4 arrays. The best-case hit rate involved a 64-element block being brought in (1 miss) and every element being used exactly once (63 hits). Each element is accessed once, so after the block is entirely used, we can safely kick the block out of the cache without affecting hit rate.

In the first iteration of the loop, we bring in the first block of `arr` (index 0). Also, we have to bring in the first block of `C` (index 0). If the cache were direct-mapped, the `C` block would kick out the `arr` block, even though we haven't used all the elements of the `arr` block yet. This would affect hit rate, so the cache cannot be direct-mapped.

If the cache is 2-way set associative, then we could keep both `arr` and `C` blocks in the cache. Now we can't bring in the first block of `A` or `B`, since that would kick out a block (`arr` or `C`) that we haven't fully used yet. However, in the best case, we could bring in a different block `A` and `B`, putting the blocks in a different index of the cache.

After the first iteration in the best case, we have the first block of `arr` and `C` in the cache. We also have some block (not the first block) of `A` and `B` in the cache.

Consider the next 63 iterations. In the best case, we keep hitting random elements of `A` and `B` that are in the block we brought in, creating the 63 hits. We also access all the elements of the first blocks of `arr` and `C`. Now, we've used up all 4 blocks currently in cache, so we can safely kick them out without affecting hit rate.

In the next set of 64 iterations, we put blocks of `arr` and `C` in index 1 of the cache. In the best-case, the next accesses of `A` and `B` are all at some other index, causing blocks of `A` and `B` to appear at a different index in the cache.

In summary: We can use a 2-way set associative cache, relying on the best-case scenario where the `i`th iteration of the loop never tries to bring in `A[i]` and `B[i]` (as this would kick out `arr[i]` or `C[i]`).

Question 9 ***SDS*** **(20 points)**

Q9.1 SOLUTION-TODO transcribe

> **Solution:** The longest combinatorial logic path is between any two timed elements. Registers are always timed elements, and in this question, we also assume that inputs are timed elements (their values arrive on the rising edge of the clock). (This assumption doesn't change the solution to this subpart though.)
>
> Tracing out paths between registers shows that the longest path starts at Reg1, passes through the XOR gate, goes down and through the AND gate, down and through the OR gate, then up and through the OR gate, before arriving at Reg2.

Q9.2 SOLUTION-TODO transcribe

> **Solution:** 31 ps
>
> The combinatorial path has delay $6 + 5 + 6 + 6 = 23$ ps.
>
> When the clock has a rising edge, we have to wait 5 ps (clk-to-q time) for the register output to change (since the critical path starts at a register, not an input). Then, we wait 23 ps for the combinatorial logic to propagate. Then, we wait 3 ps (setup time) for the signal to stabilize at the next register input.
>
> In total, this is $23 + 5 + 3 = 31$ ps.

Q9.3 SOLUTION-TODO transcribe

> **Solution:** 25 GHz
>
> Remember that frequency and period are inverses, i.e. $f = 1/p$ and $p = 1/f$.
>
> Also recall that a picosecond is 10^{-12} seconds, and a gigahertz is 10^9 hertz.
>
> From the previous subpart, we need to choose a frequency such that the corresponding period is at least 31 ps to meet the timing requirements.
>
> If the frequency is 100 GHz $= 100 \times 10^9 = 10^{11}$ Hz, then the period is 10^{-11} seconds $= 10 \times 10^{-12}$ seconds = 10 ps. This is too short.
>
> If the frequency is 50 GHz, the frequency has halved, so the clock period (the inverse) will double. Intuitively, if the clock ticks half as often, then the time between ticks is twice as long. (You could also replicate the math in the previous subpart if you don't want to use this shortcut.) Therefore, the period is 20 ps, which is still too short.
>
> If the frequency halves again to 25 GHz, the clock period doubles again to 40 ps, which is greater than 31 ps and thus satisfies the timing requirement.

Q9.4 TODO-SOLUTION the NAND logic stuff

Q9.5 TODO-SOLUTION transcribe

Solution: In words: Reg2 and Reg3 start undefined (we don't know their output values). The clock has a rising edge. 25 ps (clk-to-q delay) later, Reg2 gets an output value.

The combinatorial logic delay is 10 ps (2 gates), so after another 10 ps, the input to Reg3 changes and stays there until the next clock rising edge. (We know that this all happens within 1 clock cycle, because the period of 100 ps is more than the clk-to-q + delay + setup time).

The clock has another rising edge. After another 25 ps (clk-to-q delay), Reg3 gets an output value.

Options (A) and (D) are eliminated because Reg3 is not known until after the second rising edge. However, these options draw Reg3 as being known after the first rising edge.

Options (B) and (C) only differ in what value Reg3 takes on on the second rising edge. Checking the waveform in the first clock cycle, Input_D is 0 and Reg2_output is 0.

The circuit computes XOR(AND(Reg2_output, Input_D), OR(Reg2_output, Input_D)).

Plug in values to get XOR(AND(0, 0), OR(0, 0)).

Simplify to get XOR(0, 0) = 0.

On the second rising edge, Reg3 should output 0, not 1. This is option (B), not (C).