# CS 61C Spring 2023 — Garcia, Yokota — Final

Solutions last updated: Tuesday, May 16, 2023

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

PRINT your student ID: ______________________________

You have 170 minutes. There are 10 questions of varying credit (100 points total).

| Question: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Points: | 10 | 10 | 23 | 13 | 10 | 5 | 14 | 7 | 7 | 1 | 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 you write multiple answers or your answer is ambiguous, we will grade the worst interpretation. For coding questions, you may write at most one statement and you may not use more blanks than provided.

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.

SIGN your name: ____________________________

## Q1 *RISC-y Array Arrchitecture* (10 points)

Writing code to access integer arrays can be really annoying in RISC-V! Suppose we come up with new instructions, `readArr` to read from integer arrays and `writeArr` to write to integer arrays. For this question, you may assume integers are 32 bits.

`readArr rd, rs1, rs2` will read the array that `rs1` points to at the index stored in `rs2`, and put that value in register `rd`. In C pseudocode: `rd = ((int *) rs1)[rs2]`.

Q1.1 (3.5 points) What changes would we need to make to our datapath in order for us to implement the `readArr` instruction with as few changes as possible? Select all that apply.

- ☐ Add a new immediate type for ImmGen
- ☐ Add a new output to Regfile for a third register value
- ☐ Add a new input to the AMux and update any relevant selector/control logic
- ☐ Add a new input to the BMux and update any relevant selector/control logic
- ■ Add a new ALU operation and update any relevant selector/control logic
- ☐ Add a new DMEM mux which feeds into the data input of the DMEM, and any relevant selector/control logic
- ☐ Add a new input to WBMux and update any relevant selector/control logic
- ☐ None of the above

**Solution:** The `readArr` instruction is similar to any other R-type instruction, except that the ALU operation performed must be `rs1 + rs2 * 4`. Hence, we need to add a new ALU operation, and there are no other changes needed to the standard datapath.

**Grading:** Each checkbox was graded as it's own true/false question, and selecting "None of the above" was treated as not selecting any of the other choices.

`writeArr rs3, rs1, rs2` will take the value in register `rs3`, and write that value to the array that `rs1` points to at index `rs2`. In C pseudocode: `((int *) rs1)[rs2] = rs3`.

Q1.2 (3.5 points) Assume that the changes, if any, for `readArr` have **not** been implemented for this subpart. What changes would we need to make to our datapath in order for us to implement the `writeArr` instruction with as few changes as possible? Select all that apply.

- ☐ Add a new immediate type for ImmGen
- ■ Add a new output to Regfile for a third register value
- ☐ Add a new input to the AMux and update any relevant selector/control logic
- ☐ Add a new input to the BMux and update any relevant selector/control logic
- ■ Add a new ALU operation and update any relevant selector/control logic
- ■ Add a new DMEM mux which feeds into the data input of the DMEM, and any relevant selector/control logic
- ☐ Add a new input to WBMux and update any relevant selector/control logic
- ☐ None of the above

**Solution:** The `writeArr` instruction is not similar to any existing RISC-V instructions, so there are a couple of modifications needed. First, similar ot `readArr`, it requires the ALU to compute the address at which to store `rs3`, which requires a new ALU operation (`rs1 + rs2 * 4`). Unline existing S-type instructions, where the data stored in DMEM comes from `rs2`, here, the data comes from `rs3`. As a result, we also need to add a new DMEM mux which can switch between passing `rs2` and `rs3` into the data input port of the DMEM. Finally, `writeArr` is the only instruction that reads three register values, so we need to also add a new output to Regfile.

**Grading:** Each checkbox was graded as it's own true/false question, and selecting "None of the above" was treated as not selecting any of the other choices.

Q1.3 (3 points) Eddy noticed that the structure of `writeArr` is similar to an R-type instruction. However, when he tried to use the control signals for an R-type instruction, it didn't work. Which of the following control signals does he need to change to correctly implement `writeArr`? Select all that apply.

| | |
|---|---|
| ☐ PCSel | ■ RegWEn |
| ☐ ASel | ■ MemRW |
| ☐ BSel | ☐ None of the above |

**Solution:**

- PCSel: always 0 for R-types, 0 for `writeArr`
- ASel: always `rs1` for R-types, also `rs1` for `writeArr` because the ALU performs "`rs1 + rs2 * 4`"
- BSel: always `rs2` for R-types, also `rs2` for `writeArr` because the ALU performs "`rs1 + rs2 * 4`"
- RegWEn: always 1 for R-types, but must be 0 for `writeArr` since it does not write to any register
- MemRW: always 0 for R-types, but must be 1 for `writeArr` since it needs to write to DMEM

**Grading:** Each checkbox was graded as it's own true/false question, and selecting "None of the above" was treated as not selecting any of the other choices.

## Q2 *IF Only ID Pipelined Better* **(10 points)**

In Project 3, we implemented a RISC-V CPU with two stages; stage 1 included IF and stage 2 included ID/EX/MEM/WB. For this question, imagine instead that we implement a two-stage pipeline with a different split; stage 1 will include IF/ID and stage 2 will include EX/MEM/WB (IF/ID/EX/MEM/WB are defined equivalently to the pipelined CPU on the reference card).

For Q2.1 and Q2.2, assume the following delays for each component. Any component not listed is assumed to have a negligible delay.

| **Component** | **Delay** |
|---|---|
| $\tau_{\text{clk-to-q}}$ | 35ps |
| $\tau_{\text{setup}}$ | 20ps |
| Mux | 75ps |
| Regfile Setup | 20ps |
| Regfile Read | 175ps |
| Immediate Generator | 150ps |
| Branch Comparator | 200ps |
| ALU | 200ps |
| Memory Read | 300ps |

Q2.1 (3 points) What is the minimum clock period of this circuit, in picoseconds, to achieve correct behavior?

**Solution:** 855ps ($\tau_{\text{clk-to-q}}$ (35) + Immediate Generator (150) + Mux (75) + ALU (200) + Memory Read (300) + Mux (75) + $\tau_{\text{setup}}$ (20)).

The critical path occurs in stage 2 for a load instruction.

**Grading:** Partial credit was given for errors that showed conceptual understanding of what the critical path is, but excluded or included an extra component's timing. We did not give partial credit for excluding some components since we cannot clearly distinguish between a conceptual misunderstanding or a mechanical error.

Q2.2 (2 points) Which component in stage 2 can we move to stage 1 to decrease the minimum clock period of this circuit the most, while maintaining the same behavior? If a decrease is not possible, write "Not Possible".

**Solution:** Immediate generator

If we move the immediate generator from stage 2 into stage 1, the immediate generator is no longer part of the critical path since it can occur in parallel with Regfile Read (which takes 175ps).

**Grading:** All or nothing.

For the remainder of this question, assume that the changes made in Q2.2, if any, have **not** been implemented.

Q2.3 (3.5 points) In the CPU, which of the following values must have a pipeline register? Select all that apply.

- ■ Instruction
- ■ RegReadData2
- ☐ MemReadData
- ■ Program Counter
- ☐ Immediate
- ☐ None of the above
- ■ RegReadData1
- ☐ ALUOut

**Solution:** The values that require a pipeline register are all values generated in stage 1, which are Instruction (from IMEM), Program Counter (from the PC register), RegReadData1 (from Regfile) and RegReadData2 (also from Regfile). The remaining three values are all outputs of components in the second stage of our pipeline.

**Grading:** Each checkbox was graded as it's own true/false question, and selecting "None of the above" was treated as not selecting any of the other choices.

Q2.4 (1.5 points) Assume that the pipeline has been correctly implemented. Which types of hazards could a program experience? Assume that you cannot read from and write to the Regfile in the same clock cycle.

- ■ Control
- ■ Data
- ☐ Structural
- ☐ None

**Solution:** Similar to the pipeline implemented in project 3B, there could be control hazards if a branch is taken (the instruction in stage 1 must be flushed). There can also be data hazards if the instruction in stage two writes to the Regfile, and the instruction from stage 1 attempts to read from the same register (since you cannot read and write to the Regfile in the same clock cycle). This is not a structural hazard because it cannot be solved by adding another Regfile (since they won't share data).

**Grading:** Each checkbox was graded as it's own true/false question, and selecting "None of the above" was treated as not selecting any of the other choices.

This page intentionally left (mostly) blank.

The exam continues on the next page.

## Q3 *Faster Than Average* **(23 points)**

In this question, you will parallelize a function to compute the average of all values in a matrix. Below is a single-threaded implementation of this function.

```
double matrix_average(double** matrix, int num_rows, int num_cols) {
    double global_sum = 0.0;
    for (int i = 0; i < num_rows; i++) {
        for (int j = 0; j < num_cols; j++) {
            global_sum += matrix[i][j];
        }
    }
    return global_sum / (num_rows * num_cols);
}
```

Using the SIMD operations provided, optimize `matrix_average`. You have access to the following SIMD operations. A `vector` is a 256-bit vector register capable of holding 4 doubles.

- `vector vec_load(double* A)`: Loads 4 doubles at memory address `A` into a vector
- `void vec_store(double* A, vector B)`: Stores the 4 doubles in vector `B` at memory address `A`
- `vector vec_set0()`: Puts all 0s into a vector
- `double vec_sum(vector A)`: Adds all elements of the vector together: `return A[0] + A[1] + A[2] + A[3]`
- `vector vec_add(vector A, vector B)`: Adds `A` and `B` together elementwise

```
double matrix_average(double** matrix, int num_rows, int num_cols) {
    double global_sum = 0.0;

    vector sum_vec = ______________________________________________;
                                         Q3.1
    for (int i = 0; i < num_rows; i++) {

        for (int j = 0; j < __________________; _________________) {
                                   Q3.2                Q3.3

            vector values = ________________________________________;
                                               Q3.4

            sum_vec = ______________________________________________;
                                             Q3.5
        }
        for (int j = _____________; j < _____________; _____________) {
                          Q3.6               Q3.7            Q3.8
            global_sum += matrix[i][j];
        }
    }
    global_sum += ___________________________________________________;
                                           Q3.9
    return global_sum / (num_rows * num_cols);
}
```

**Solution:**

Q3.1: `vec_set0()`

Q3.2: `num_cols / 4 * 4`

Q3.3: `j += 4`

Q3.4: `vec_load(matrix[i] + j)`

Q3.5: `vec_add(values, sum_vec)`

Q3.6: `num_cols / 4 * 4`

Q3.7: `num_cols`

Q3.8: `j += 1`

Q3.9: `vec_sum(sum_vec)`

Parallelize `matrix_average` using OpenMP without using `#pragma omp parallel for` or `reduction`. Each thread should work on one or more rows of the matrix. Assume `num_rows` is a multiple of `num_threads`.

```
double matrix_average(double** matrix, int num_rows, int num_cols) {
    double global_sum = 0.0;

    ________________________________________
                     Q3.10
    {
        int num_threads = omp_get_num_threads();
        int thread_num = omp_get_thread_num();

        int chunk_size = _________________________________;
                                       Q3.11

        int start_row = _________________________________;
                                      Q3.12

        int end_row = ___________________________________;
                                     Q3.13

        for (int i = start_row; i < end_row; i++) {
            double row_sum = 0.0;
            for (int j = 0; j < num_cols; j++) {
                row_sum += matrix[i][j];
            }

            ________________________________________
                             Q3.14

            ________________________________________
                             Q3.15
        }
    }
    return global_sum / (num_rows * num_cols);
}
```

**Solution:**

Q3.10: `#pragma omp parallel`

Q3.11: `num_rows / num_threads`

Q3.12: `thread_num * chunk_size`

Q3.13: `(thread_num + 1) * chunk_size`

Q3.14: `#pragma omp critical`

Q3.15: `global_sum += row_sum`

## Q4 *Convoluted Caching* **(13 points)**

Consider the following function that takes in two integer arrays, `a` (of length `a_len`) and `b` (of length `b_len`), and returns the 1D convolution of `a` and `b`. Assume `results` is properly allocated.

Let `a=0x1000`, `b=0x2000`, `results=0x3030`, `a_len=4`, and `b_len=2`.

```
void convolve_1d(int* a, int a_len, int* b, int b_len, int* results) {
    for (int i = 0; i < a_len - b_len + 1; i++) {
        register int sum = 0;
        for (int j = 0; j < b_len; j++) {
            sum += b[j] * a[i + j];
        }
        results[i] = sum;
    }
}
```

For Q4.1 and Q4.2, we have a single-level, *direct-mapped* 64B cache with 16B blocks and 16-bit addresses.

Q4.1 (3 points) What are the tag, index, and offset bits of the address `0x3037`?

**Solution:** Since the block size is 16B, there are 4 offset bits. The cache is direct mapped, and there are 4 cache lines, there are 2 index bits. This leaves us with $16 - 4 - 2 = 10$ tag bits.

Tag: `0b00_1100_0000`

Index: `0b11`

Offset: `0b0111`

Q4.2 (2.5 points) What is the overall hit rate for a call to `convolve_1d`? No need to simplify the fraction.

**Solution:** $\frac{2}{15}$

There are a total of three iterations of the outer loop, and two iterations of the inner loop per iteration of the outer loop. For each iteration of the inner loop, there are two memory accesses: reading `b[j]` and reading `a[i + j]`. Each outer loop has an additional memory access: writing to `results[i]`. In total, there are 5 accesses per iteration of the outer loop, and 15 accesses overall.

For each set of accesses to `a` and `b`, the program experiences thrashing since the index of these blocks conflict. As a result, the accesses to `a` and `b` will always be misses (first two compulsory, then the rest are conflict).

The accesses to `results` do not cause any thrashing, so out of the three accesses (one per iteration of the outer loop), there is one compulsory miss and two hits, giving us a hit rate of $\frac{2}{15}$.

Q4.3 (2.5 points) We change to a *2-way set associative* cache of the same size with a LRU replacement policy. What is the overall hit rate for a call to `convolve_1d`? No need to simplify the fraction.

**Solution:** $\frac{12}{15}$
Since there can now be two caches lines with the same index, the accesses to `a` and `b` no longer thrash. In total, there are three compulsory misses (one for each array), and the rest are hits, giving us a hit rate of $\frac{12}{15}$.

Q4.4 (2.5 points) We change to a *fully associative* cache of the same size with a LRU replacement policy. What is the overall hit rate for a call to `convolve_1d`? No need to simplify the fraction.

**Solution:** $\frac{12}{15}$
Similar to Q4.3, there is no thrashing here either, and all three arrays fit entirely in the cache. As a result, our hit rate is once again $\frac{2}{15}$.

Q4.5 (2.5 points) We discover that accessing physical memory will take 400 cycles, so we decide to add an L2 cache. The hit rate of the L1 cache is 75%, and the hit rate of the L2 cache is 99%. With an access time of 6 cycles to fetch from the L1 cache, and an access time of 36 cycles to fetch from the L2 cache, what would our memory access time be for this system, on average?

**Solution:** 16 cycles

$$t_{\text{avg}} = t_{\text{L1\_access}} + \text{L1\_miss\_rate} \cdot (t_{\text{L2\_access}} + \text{L2\_miss\_rate} \cdot t_{\text{men\_access}})$$
$$t_{\text{avg}} = 6 + 0.25 \cdot (36 + 0.01 \cdot 400)$$
$$t_{\text{avg}} = 16 \text{ cycles}$$

## Q5 *The Lookup Box* (10 points)

Consider a system with a 32-bit virtual address space, 256B pages, and 16 MiB of DRAM as main memory. Provided below is the TLB and a portion of the page table. The TLB is fully associative and there is no data cache. The next free physical pages in main memory start at physical addresses `0x61DE00` and `0x61EF00`, respectively.

Each PTE is 32 bits. Bit 31 is the valid bit, bit 30 is the dirty bit, bits 16 through 29 hold other metadata (not relevant for this question), and bits 0 through 15 hold the PPN.

Initial TLB State:

| Tag (VPN) | PPN | Valid | Dirty |
|---|---|---|---|
| 0x000000 | 0x23EF | 1 | 0 |
| 0x000001 | 0xFFFF | 0 | 0 |

Page Table:

| Index | PTE |
|---|---|
| 0x0 | 0x80AB23EF |
| 0x1 | 0x80EE00C0 |
| 0x2 | 0x8123200A |
| 0x3 | 0x3561CBA8 |
| ... | ... |
| 0xA | 0xCAFFEEE0 |

For each question, determine what the memory address access results in, and calculate its physical address. Note that each memory access is executed in sequence, so they are **not** independent of each other.

Q5.1 (2.5 points) Virtual Address: `0x000000FF`

[✓ TLB hit] TLB miss, no page fault TLB miss, page fault

Physical Address:

**Solution:** Physical Address: `0x23EFFF`

The VPN is `0x000000` and the offset is `0xFF`. The VPN exists in the TLB and is valid. The corresponding PPN is `0x23EF`, so the physical address is `0x23EFFF`.

Q5.2 (2.5 points) Virtual Address: `0x00000283`

TLB hit [✓ TLB miss, no page fault] TLB miss, page fault

Physical Address:

**Solution:** Physical Address: `0x200A83`

The VPN is `0x000002` and the offset is `0x83`. The VPN does not exist in the TLB, so we look for the corresponding PTE. The PTE tells us that the PPN is `0x200A`, so the physical address is `0x200A83`.

Q5.3 (2.5 points) Virtual Address: `0x00000AAA`

○ TLB hit [✓ ● TLB miss, no page fault] ○ TLB miss, page fault

Physical Address:

**Solution:** Physical Address: `0xEEE0AA`

The VPN is `0x00000A` and the offset is `0xAA`. The VPN does not exist in the TLB, so we look for the corresponding PTE. The PTE tells us that the PPN is `0xEEE0`, so the physical address is `0xEEE0AA`.

Q5.4 (2.5 points) Virtual Address: `0x00000360`

○ TLB hit ○ TLB miss, no page fault [✓ ● TLB miss, page fault]

Physical Address:

**Solution:** Physical Address: `0x61DE60`

The VPN is `0x000003` and the offset is `0x60`. The VPN does not exist in the TLB, so we look for the corresponding PTE. There are no valid PTEs for this VPN, therefore, it is a page fault. The next available page starts at `0x61DE00`, so our PPN becomes `0x61DE`. The physical address is `0x61DE60`.

**Solution:**

**Grading:** 1 point was awarded for the correct access result. 0.5 points was awarded for the correct offset, 0.5 points was awarded for the including the PPN in the address (recognizing the correct PTE), and 0.5 points was awarded for the correct physical address size and no extraneous bits other than the PPN and the offset.

## Q6 *Cumulative: Potpourri* (5 points)

Q6.1 (1 point) The OS running on a cluster of computers in a datacenter allows a single machine to read and write the memory and local disk of a remote machine in the same rack or array.

According to lecture, which of the following are true? Select all that apply.

("farther away" means that the distance the data travels increases in steps, first to our local machine, then to a machine in our same rack, then to a machine in our same array.)

- ☐ As our CPU sends data to DRAM farther away, bandwidth increases
- ☐ As our CPU sends data to Disk farther away, bandwidth increases
- ■ As our CPU sends data to DRAM farther away, latency increases
- ■ As our CPU sends data to Disk farther away, latency increases
- ☐ We have higher latency to DRAM on an Array computer than to our own disk
- ☐ We have higher bandwidth to DRAM on an Array computer than to our own disk
- ☐ None of the above

**Solution:** See lecture 38, slide 20.

**Grading:** Each checkbox was graded as it's own true/false question, and selecting "None of the above" was treated as not selecting any of the other choices.

Q6.2 (1 point) After a single machine `M` finishes its assigned portion of a map task in a MapReduce cluster, which of the following can happen immediately, regardless of the overall program state? Select all that apply.

- ■ `M` can shut down without risking the success of the overall computation
- ■ `M` can be assigned a new map task
- ■ Data shuffling of the workload `M` just finished can begin
- ☐ The reduce task for the workload `M` just finished can begin
- ☐ None of the above

**Solution:** If `M` shuts down, the map task can be reassigned to a different machine and no data is lost. After `M` finishes a task, it can be assigned another map task, regardless of the program state. Data shuffling of the workload that `M` just finished can also begin, since it does not depend on anything else except the task `M` just completed. However, the reduce task for the workload that `M` just finished cannot begin, since reduce tasks aggregate outputs from multiple map tasks, and we don't know if the other map tasks have finished yet.

**Grading:** Each checkbox was graded as it's own true/false question, and selecting "None of the above" was treated as not selecting any of the other choices.

Q6.3 (1 point) We want to send **one** bit using a Hamming error correcting code. What are the valid bit patterns you could send that correspond to `0b0` and `0b1`?

`0b0`: `0b1`:

**Solution:** Each Hamming error correcting code must be at least 3 bits (two parity bits, one data bit). The ECC for `0b0` is `0b000` and for `0b1` is `0b111`.

**Grading:** All or nothing for each answer box.

Your network card just received a packet with an incorrect checksum.

Q6.4 (1 point) According to lecture, which of the following is true?

- ○ There was a guaranteed error in the payload but not the checksum
- ○ There was a guaranteed error in the checksum but not the payload
- [✓ ● There was a guaranteed error in either the payload or checksum]
- ○ None of the above

**Solution:** See lecture 35, slide 28.
**Grading:** All or nothing.

Q6.5 (1 point) According to lecture, what should you do?

- ○ Send back a traditional data packet with information about which packet had the problem
- ○ Send back an "ACK"
- ○ Send back a "NO-ACK"
- [✓ ● Send back nothing and delete the packet]

**Solution:** See lecture 35, slide 28.
**Grading:** All or nothing.

## Q7 *Cumulative: RV32tok* (14 points)

Write a program `splitCode`, which will split a RISC-V program into blocks of code with no branches or jumps (`jal` or `jalr`). Specifically, `splitCode` will have the following function signature:

- **Input:** `int* code`, an array of RISC-V instructions. Each RISC-V instruction is stored as a 32-bit integer, equal to its translation. You may assume that all instructions are valid RISC-V base instructions, and that there are no pseudoinstructions, `ecalls`, or `ebreaks`.
- **Input:** `n`, the number of instructions in `code`.
- **Input:** `int*** result`, a pointer to store your result. Your result should be an array of `int*`s, where each `int*` points to the beginning of a sequence of consecutive instructions with no branches or jumps. Each of these arrays should be "null-terminated"; that is, the last element of each array should be the number 0, to signify the end of the array. Every non-branch/non-jump instruction must be represented in exactly one subarray of your result. No branch/jump instruction should be in any subarray of your result.
- **Output:** `int`, the length of your result.

For example, for the following RISC-V code:

```
 1 beq x0 x0 pass
 2 beq x0 x0 pass
 3 add a0 t0 t1
 4 add t0 a0 a1
 5 add t0 a1 a2
 6 xor a0 t0 t1
 7 j pass
 8 addi t0 x0 1
 9 addi t0 x0 2
10 beq x0 x0 pass
```

`result` should point to the following array, and the return value should be 5.

```
[
    // The instructions before line 1
    [0],
    // The instructions between lines 1 and 2
    [0],
    // The instructions between lines 2 and 7
    [add a0 t0 t1, add t0 a0 a1, add t0 a1 a2, xor a0 t0 t1, 0],
    // The instructions between lines 7 and 10
    [addi t0 x0 1, addi t0 x0 2, 0],
    // The instructions after line 10
    [0]
]
```

Useful C function prototypes:

```
void* malloc(size_t size);
void* calloc(size_t num_elements, size_t size);
void* memcpy(void* dest, void* source, size_t num_bytes);
```

```
// Returns true if instruction is a branch or jump instruction
bool isBranchJump(int instruction) {

    return ____________________________________________________;
                                     Q7.1
}

int splitCode(int* code, int n, int*** result) {
    int num = 0; // total number of branches and jumps

    for(int i = 0; ______________________________________; i++) {
                                     Q7.2

        num += ________________________________________________;
                                     Q7.3
    }
    int** data = malloc(_______________________________________);
                                     Q7.4

    int* codecopy = calloc(n+1, _______________________________);
                                         Q7.5
    // Hint: You should not need any more memory allocations

    memcpy(codecopy, code, ____________________________________);
                                         Q7.6

    for(int i = 0; ______________________________________; i++) {
                                     Q7.7

        data[i] = _____________________________________________;
                                     Q7.8

        while(____________________ && ____________________ != 0) {
                     Q7.9                     Q7.10

            _____________________________________________;
                                 Q7.11
        }

        _____________________________________________;
                             Q7.12
        codecopy++;
    }

    _____________________________________________;
                         Q7.13

    return _____________________________________________;
                                 Q7.14
}
```

**Solution:**

Q7.1: `instruction & 64`

Q7.2: `i < n`

Q7.3: `isBranchJump(code[i])`

Q7.4: `sizeof(int*) * (num + 1)`

Q7.5: `sizeof(int)`

Q7.6: `sizeof(int) * n`

Q7.7: `num + 1`

Q7.8: `codecopy`

Q7.9: `!isBranchJump(*codecopy)`

Q7.10: `*codecopy`

Q7.11: `codecopy++`

Q7.12: `*codecopy = 0`

Q7.13: `*result = data`

Q7.14: `num + 1`

## Q8 *Cumulative: Chips Ahoy* (7 points)

Consider the following set of tasks:

| Task ID | Time (minutes) | Prerequisites | Time Breakdown |
|---|---|---|---|
| 0 | 100 | - | 90% memory, 10% math |
| 1 | 100 | - | 90% memory, 10% math |
| 2 | 100 | 0 | 30% memory, 70% math |
| 3 | 100 | 0,1 | 30% memory, 70% math |
| 4 | 100 | 1 | 30% memory, 70% math |
| Total | 500 | - | 54% memory, 46% math |

After running this set of tasks on your local, single-threaded CPU (referred to as Chip A) in 500 minutes, you decide that it's too slow and decide to upgrade to a new chip.

At the store, you find two options:

- Chip B: A single-threaded CPU optimized for memory accesses. It can do memory operations 3 times as fast as Chip A, but it takes twice as long to do math operations.
- Chip C: A single-threaded GPU optimized for fast math. It can do math operations practically instantly (infinite times speedup), but it takes twice as long as Chip A to do memory operations.

Q8.1 (2 points) Using Chip B alone, how many minutes would all 5 tasks take?

**Solution:** 550 minutes. 270 minutes of memory becomes 90 minutes, 230 minutes of math becomes 460 minutes.

**Grading:** All or nothing.

Q8.2 (2 points) Using Chip C alone, how many minutes would all 5 tasks take?

**Solution:** 540 minutes. 270 minutes of memory becomes 540 minutes, 230 minutes of math becomes 0 minutes.

**Grading:** All or nothing.

Q8.3 (3 points) Using one chip was still taking too long, so you buy both Chip B and Chip C from the store, and connect them to Chip A in a new multicore machine with negligible overhead. Using all three chips, what is the minimum amount of time required to complete this set of tasks? Each task must be completed entirely on one chip.

Please also provide the list of tasks each chip will complete, in order of completion, or write "None" if a chip does not complete any task. For example, if you decide to have Chip A complete all of the tasks, your answer should be "0, 1, 2, 3, 4" for Chip A, and "None" for Chip B and Chip C.

**Solution:** 200 minutes. There are many task lists. For example: Chip A completes tasks 1 and 3, Chip B completes task 0, Chip C completes tasks 2 and 4.

**Grading:** Partial credit was awarded with a task list that would run in 200 minutes but the wrong time was provided, or a task list that runs in <250 minutes and has the correct time.

## Q9 *Cumulative: The Magnus Effect* **(7 points)**

Q9.1 (7 points) Write a Boolean expression that determines if a 19-bit unsigned integer can be expressed exactly as a 19-bit floating point number. For full credit, you may use at most 8 Boolean operators (|, &, ∼).

**Inputs:** Bits `A` through `S`

**Output:** One bit. Output 1 if `0b ABC DEFG HIJK LMNO PQRS` is an unsigned number that can be exactly represented as a 19-bit float which follows all IEEE-754 conventions, with 5 exponent bits (and a standard bias of -15). Output 0 otherwise.

Hint: both the exponent and the mantissa provide nontrivial constraints.

For partial credit, describe in English how you can determine if a 19-bit unsigned integer can be expressed exactly as a 19-bit floating point number (using the float representation described above).

**Solution:** $\overline{A|B|C|(D\&(R|S))|(E\&S)}$

As noted in the hint, two distinct constraints exist, provided by the exponent and mantissa, respectively.

In order for an unsigned number to be representable, it needs to be small enough that its exponent fits within five bits. The maximum exponent of a 19 bit floating point number is $30 - 15 = 15$, taking into account that exponent 31 is reserved for infinities and NaNs. Including the implicit one and a maximized mantissa, our range is slightly less than double this exponent; thus we cannot represent any number that is $2^{16}$ or greater. This means that we cannot represent any number with any of bits $ABC$ set to one.

In order for an unsigned number to be representable, it needs to be divisible by enough powers of two that its mantissa fits within thirteen bits. The smallest level of precision we can represent is $0b1.0000000000001$, which has a total of fourteen bits between the largest and smallest one bit. Thus, we cannot represent any number that has more than fourteen bits between the largest and smallest one bit. Since $ABC$ must be zero by the above, we only have 16 bits that can be nonzero. Thus, we need at least two bits on the outside of this to be zero; one of $DE$, $DS$, or $RS$ must be zero, or alternatively, none of the pairs $DR$, $DS$, $ES$ can be both ones.

Putting this together, we get the equation $(A|B|C)$ for the exponent constraint and $((D\&R)|(D\&S)|(E\&S)) = ((D\&(R|S))|(E\&S))$ for the mantissa constraint. Putting this together and using De Morgan's law provides our final answer.

## Q10 *The Finish Line* **(1 points)**

Everyone will receive credit for this question, even if you leave it blank.

Q10.1 (1 point) On a scale of 1 to 10, rate Eddy's weekly announcement puns.

**Solution:** ¯_(ツ)_/¯

Q10.2 (0 points) Is there anything you want us to know?