CS 61C Spring 2024

Yan, Yokota Final

Solutions last updated: Monday, May 13th, 2024

Print Your Name: _____

Print Your Student ID: _____

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

Question:	1	2	3	4	5	6	7	8	9	10	Total
Points:	12	12	13	10	13	10	9	8	13	0	100

For questions with **circular bubbles**, you may select only one choice.

- ○ Unselected option (completely unfilled)
- ● Only one selected option (completely filled)
- ◍ Don't do this (it will be graded as incorrect)

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

- ☐ You can select
- ■ multiple squares
- ■ (completely filled)

Anything you write outside the answer boxes or you ~~cross out~~ will not be graded. If you write multiple answers, your answer is ambiguous, or the bubble/checkbox is not entirely filled in, we will grade the worst interpretation. For coding questions with blanks, you may write at most one statement per blank and you may not use more blanks than provided.

If an answer requires hex input, you must only use capitalized letters (`0xDEADBEEF` instead of `0xdeadbeef`). For hex and binary, please include prefixes in your answers unless otherwise specified, and do not truncate any leading 0's. For all other bases, do not add any prefixes or suffixes.

Write the statement below in the same handwriting you will use on the rest of the exam.

> I have neither given nor received help on this exam (or quiz), and have rejected any attempt to cheat; if these answers are not my own work, I may be deducted up to 0x0123 4567 89AB CDEF points.

Sign your name: _____

Clarifications made during the exam:
Q3.5: Assume CPU will always predict branches not taken.
Q8: All instances of `uint32_t` and `int` should be `int32_t`.
Q2: You may also use the registers `x0`, `ra`, and `sp`.
Q9: You may assume that all elements in matrices W and X are positive integers.

Q1 *61Collection (Potpourri)* **(12 points)**

Q1.1 (1 point) Convert the **8-bit** unsigned binary `0b0001 1011` to decimal, treating it as an unsigned integer.

Solution: 1 + 2 + 8 + 16 = 27

Q1.2 (1 point) Convert the decimal -510 to a **12-bit** two's complement hexadecimal. If it cannot be represented, write "None".

Solution: -510 = 0b1110 0000 0010 = 0xE02

Q1.3 (1.5 points) What value does `0x61C0 0000` represent, if interpreted as an IEEE-754 single precision floating point number? Express your answer as $x \times 2^y$, where x is a base-10 number such that $1 \leq |x| < 2$ and y is a base-10 integer. For `NaN`, infinities, or zeros, please leave the boxes blank and fill in the corresponding bubble. Otherwise, leave the bubbles blank.

○ $+\infty$ ○ $+0$ ○ `NaN`

○ $-\infty$ ○ -0

Solution: 0x61C0 = 0x0110 0001 1100 ... => Mantissa = 0, Exponent = 11000011 = 195 - 127 = 68 -> 2^{68}, Mantissa = 1 -> 0b1.1 = 1.5. $x = 1.5, y = 68$

Q1.4 (0.5 points) A user program can request an operating system service, such as printing a string or reading a file, by issuing a system call.

● True ○ False

Solution: System calls provide a way for user-level processes to interact with the kernel or operating system services.

For Q1.5 – Q1.7 indicate the stage of CALL that...

Q1.5 (0.5 points) ...produces pseudoinstructions.

● Compiler ○ Assembler ○ Linker ○ Loader

Q1.6 (0.5 points) ...produces machine code for the RISC-V instruction `bne x0 x0 8`.

○ Compiler ● Assembler ○ Linker ○ Loader

Q1.7 (0.5 points) ...produces machine code for the RISC-V instruction `la t0 magic_num`, assuming that `magic_num` is a label that points to the data segment.

○ Compiler ○ Assembler ● Linker ○ Loader

Q1.8 (1.5 points) Suppose we have two 16-bit integer arrays, each with 73 elements, and we want to compute their element-wise product. If our RISC-V CPU has 128-bit registers and supports `vmul`, an instruction that performs 16-bit elementwise vector multiplication, what is the minimum number of `mul` and `vmul` instructions required to multiply these two vectors?

Solution: 128 / 16 = 8 numbers per addition. 73 elements / 8 numbers per addition = 9.125 additions. We need to do 9 `vmul` and 1 `mul` in total, which sums to 10.

Q1.9 (1.5 points) Suppose we have a program that takes 20 minutes to complete on a system with one core and takes 10 minutes to complete on a system with four cores. What fraction of this program is parallelizable? Express your answer as a simplified fraction.

Solution: Given Amdahl's Law

$$e = \frac{1}{(1-f) + \frac{f}{s}},$$

we know the terms $s = 4$ and $e = 20/10$. We need to solve for f.

$$\begin{aligned} \frac{10}{20} &= \frac{1}{(1-f) + \frac{f}{4}} \\ \implies \frac{10}{20} &= (1-f) + \frac{f}{4} \\ \implies \frac{40}{20} &= 4(1-f) + f \\ \implies 2 &= 4 - 4f + f \\ \implies -2 &= -3f \end{aligned}$$

So $f = \frac{2}{3}$

Q1.10 (1.5 points) Given a cache with a hit rate of 80%, a hit time of 5 cycles, and a miss penalty of 20 cycles, what is the Average Memory Access Time (AMAT) for the system? Express your answer as an integer, rounding up if necessary.

Solution: $5 + (0.2 \cdot 20) = 5 + 4 = 9$

For Q1.11 – Q1.14, choose which section of memory the value would live in. Assume this program compiles successfully.

```
1 int ada_mango(int adam) { return adam * adam; }
2 int main(int argc, char *argv[])  {
3   int *facade = malloc(sizeof(int) * 23);
4   char *brun = "lebrun";
5   return 0;
6 }
```

Q1.11 (0.5 points) `facade`

● Stack ○ Heap ○ Static ○ Code

Q1.12 (0.5 points) `*facade`

○ Stack ● Heap ○ Static ○ Code

Q1.13 (0.5 points) `*brun`

○ Stack ○ Heap ● Static ○ Code

Q1.14 (0.5 points) `adam`

● Stack ○ Heap ○ Static ○ Code

Q2 *61C16 (RISCV)* **(12 points)**

A palindrome is a sequence of characters that reads the same backward and forward. For example, "civic" and "redder" are palindromes, but "wave" and "canal" are not palindromes.

Implement the RISC-V function `find_palindrome` that takes as input a nonempty null-terminated string in `a0` and its length (excluding the null-terminator) in `a1`. The function should return 1 in `a0` if the string is a palindrome and 0 in `a0` otherwise. Assume the input string contains only lowercase letters.

You may only use registers `a0`, `a1`, `t5` and `t6`.

```
1  find_palindrome:

2      add a1 a0 a1
              Q2.1
3  loop:

4      addi a1 a1 -1
              Q2.2

5       lb  t5 0(a0)
            Q2.3

6       lb  t6 0(a1)
            Q2.4

7      bne t5 t6 not_palindrome
           Q2.5

8      addi a0 a0 1
              Q2.6

9      blt a0 a1 loop
              Q2.7

10     addi a0 x0 1
              Q2.8

11     jr ra
          Q2.9
12 not_palindrome:

13     mv a0 x0
             Q2.10
14     jr ra
```

Q3 *61Control Logic* **(13 points)**

Consider the standard 5-stage pipeline included in the CS 61C Reference Card. For each of the control logic signals below, indicate which stage the control signal would be used.

Q3.1 (1 point) `BrUn`

○ IF ○ ID ● EX ○ M ○ WB

Q3.2 (1 point) `RegWEn`

○ IF ○ ID ○ EX ○ M ● WB

Q3.3 (1 point) `MemRW`

○ IF ○ ID ○ EX ● M ○ WB

Q3.4 (1 point) `WBSel`

○ IF ○ ID ○ EX ○ M ● WB

Q3.5 (3 points) Consider the following RISC-V code:

```
1     addi t3 x0 8
2     addi t4 x0 6
3     lw t1 0(t3)
4     bne t3 t4 label
5     addi t3 t3 4
6 label:
7     addi x0 x0 0
```

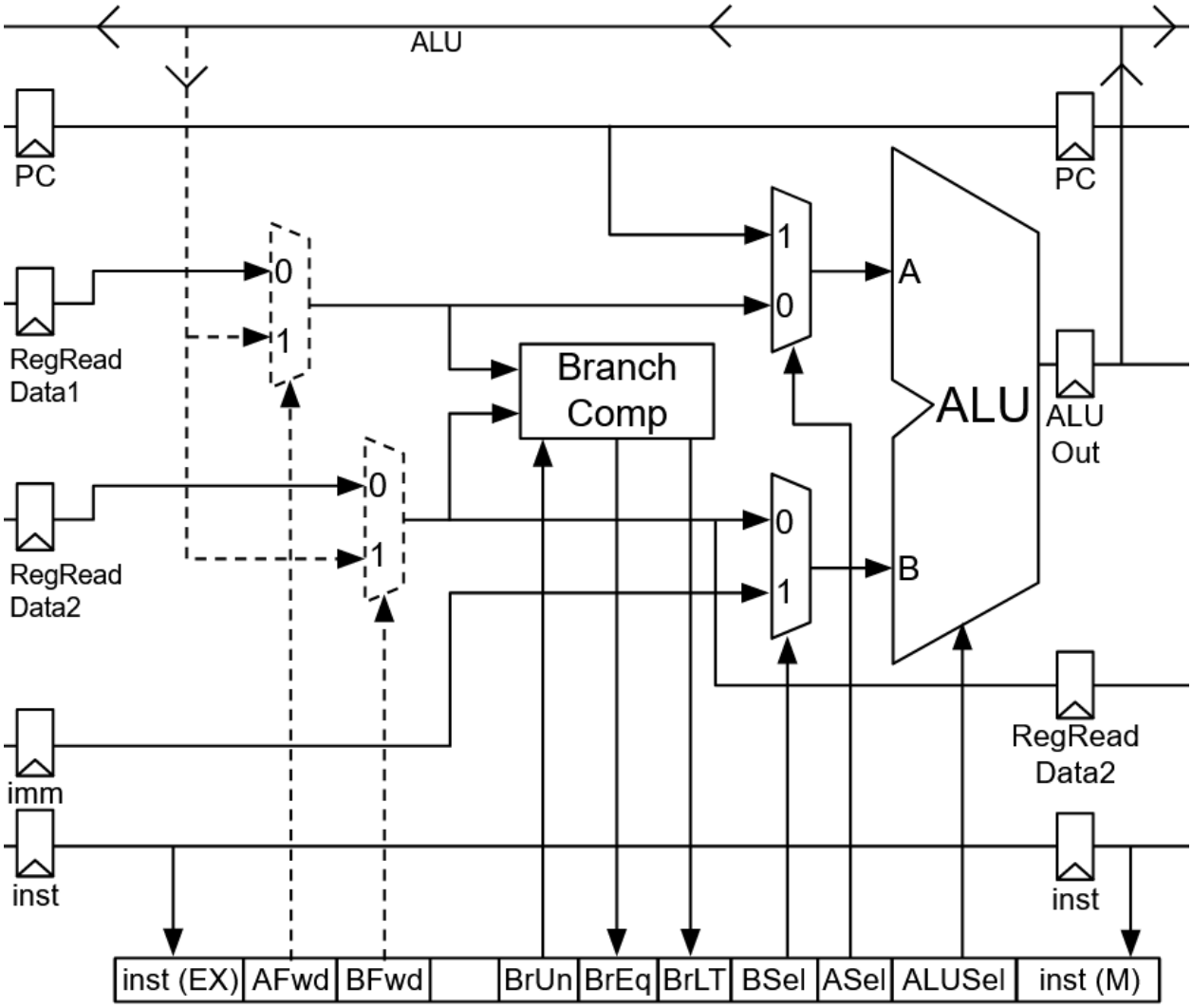
Identify the **first four hazards** that occur when the above program is run on the standard 5-stage pipeline included in the CS 61C Reference Card. Assume that the register file implements write-then-read (i.e. it allows reading out the data being written in the same cycle).

For each hazard, indicate the type of hazard and the line number(s) of the instruction(s) involved. If there are fewer than four hazards, select "None" for the unused rows.

Hazard 1:	○ Structural ● Data	○ Control ○ None	line 1 to 3
Hazard 2:	○ Structural ● Data	○ Control ○ None	line 2 to 4
Hazard 3:	○ Structural ○ Data	● Control ○ None	line 4
Hazard 4:	○ Structural ○ Data	○ Control ● None	

We can mitigate some hazards by adding a forwarding path from the ALU result directly back into the execute stage.

For Q3.6 – Q3.12 on the following page, assume we are working with the standard 5-stage pipeline included in the CS 61C Reference Card with the modifications shown below to implement this forwarding path. The diagram shows a subset of the datapath (specifically the EX stage), and modifications are shown in dotted lines.



Q3.6 (1 point) What type of hazard does this forwarding path attempt to mitigate?

○ Structural ● Data ○ Control

The described implementation introduces two new control signals: **`AFwd`** and **`BFwd`**. These control signals determine which, if any, forwarding path should be used. If a stall is unavoidable, **`AFwd`** and **`BFwd`** may be either 0 or 1. Assume instruction bits are zero-indexed.

For the two boxed statements below, fill in the blanks such that they correctly describe **`AFwd`** and **`BFwd`**.

`AFwd` should be 1 only when
- bits Q3.7 (inclusive) of the instruction in EX are nonzero and equal to bits Q3.8 (inclusive) of the instruction in M **and**
- all of the selected conditions in Q3.9 are true.

Q3.7 (0.5 points) bits 15 to 19

Q3.8 (0.5 points) bits 7 to 11

Q3.9 (1.5 points) Select as few conditions as possible to maintain the correct behavior.

■ The instruction in M writes to **`rd`**
■ The instruction in M is not a load
☐ The instruction in EX is not a store
■ The instruction in EX uses **`rs1`**
☐ The instruction in EX uses **`rs2`**
○ None of the above

`BFwd` should be 1 only when
- bits Q3.10 (inclusive) of the instruction in EX are nonzero and equal to bits Q3.11 (inclusive) of the instruction in M **and**
- all of the selected conditions in Q3.12 are true.

Q3.10 (0.5 points) bits 20 to 24

Q3.11 (0.5 points) bits 7 to 11

Q3.12 (1.5 points) Select as few conditions as possible to maintain the correct behavior.

■ The instruction in M writes to **`rd`**
■ The instruction in M is not a load
☐ The instruction in EX is not a store
☐ The instruction in EX uses **`rs1`**
■ The instruction in EX uses **`rs2`**
○ None of the above

Q4 *61CO (FSM)* **(10 points)**

Q4.1 (4 points) We are designing an FSM for a carbon monoxide (CO) detector that receives input every half-second. A 0 indicates normal CO levels, and a 1 indicates dangerous levels. When at least two of the last three inputs are 1, the detector activates and outputs 1's indefinitely. When not yet activated, it outputs 0's. The FSM starts in state `00`. **Complete the transition table below to match this behavior, filling in the next state and output for each row.** Some boxes are pre-filled for you.

For example, if the input to this FSM was `0b01 0010 1011 1000 1001`,
the output should be `0b00 0000 1111 1111 1111`.

- **CS** and **NS** represent the Current State and Next State of the FSM respectively
- **In** represents the input to the FSM (whether or not the CO level is dangerous)
- **Out** represents the output of the FSM (whether or not the detector is activated)

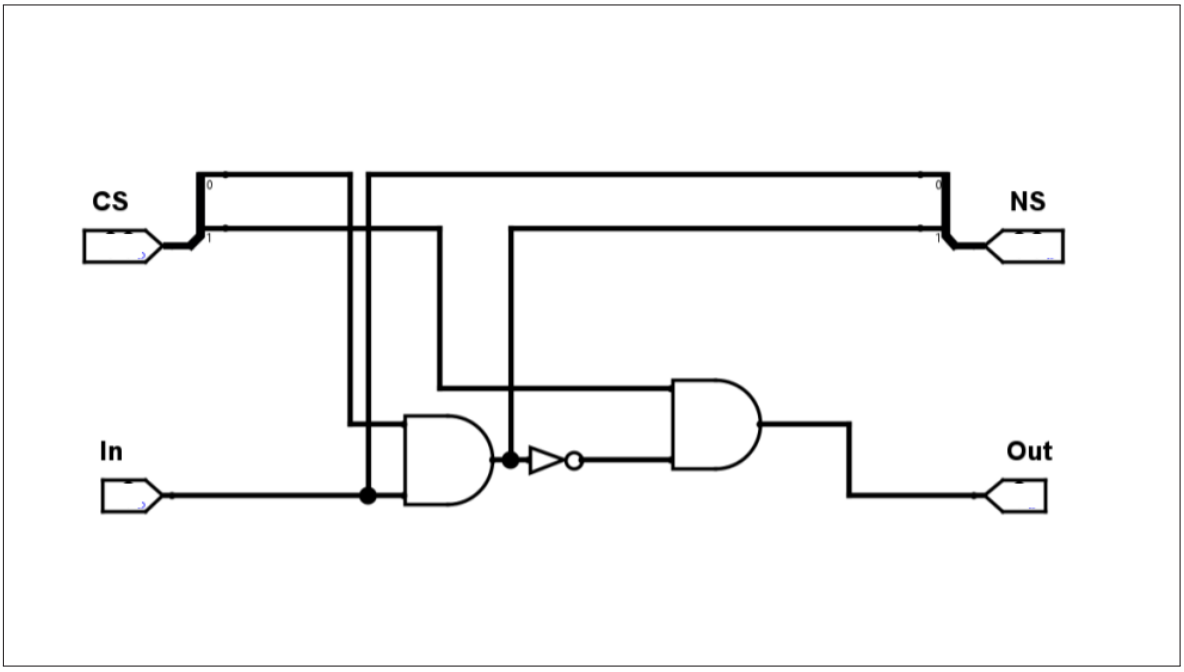
CS	In	NS	Out
00	0	00	0
00	1	01	0
01	0	10	0
01	1	11	1
10	0	00	0
10	1	11	1
11	0	11	1
11	1	11	1

Q4.2 (2 points) We want to make our detector more sensitive. Now, the detector should output 1's indefinitely if **two of the last 12 samples** are 1's. What is the minimum number of states required to implement this updated detector as an FSM?

Solution: `13`. One way to do this is to have a state per time step, e.g. "The last sample was 1 time step ago", all the way up to "The last sample was 11 time steps ago," as well as a state for "No samples detected," and "On." The key idea is that since we only care about 2 samples being detected, we can shortcut storing all possible configurations of the past 12 timesteps, since only the time steps that look like $0 \cdots 010 \cdots 0$ are valid inputs that don't lock the FSM into the "On" state.

Q4.3 (4 points) A state transition table for a **different** FSM is shown below. Complete the circuit below to implement this FSM using only AND, OR, and NOT gates. Your implementation should minimize the critical path delay. Assume wires have negligible delay and each logic gate has the same delay.

CS	In	NS	Out
00	0	00	0
00	1	01	0
01	0	00	0
01	1	11	0
10	0	00	1
10	1	01	1
11	0	00	1
11	1	11	0



Q5 *61Caches* **(13 points)**

Suppose our computer has 64KiB of memory and a 32B direct-mapped cache with 8B blocks.

Q5.1 (1.5 points) How many tag, index, and offset bits are in each address?

T: 11

Solution: 11 tag bits, 2 index bits, 3 offset bits

For Q5.2 – Q5.10, consider running the following program on our computer, assuming the cache starts off cold.

```
1 #define NUM_INTS 24
2 int32_t arr[NUM_INTS + 2]; // arr is located at address 0x2000
3 for (register int32_t i = 0; i < NUM_INTS; i += 1) {
4   arr[i + 2] = arr[i + 1] + arr[i]; // arr[i + 1] is accessed before arr[i]
5 }
```

Q5.2 (1 point) For each iteration of the `for` loop, how many **memory accesses** are there?

Solution: 3 (read arr[i+1], read arr[i], write to arr[i+2])

Q5.3 (1 point) For the **first** iteration of the `for` loop (`i = 0`), how many **hits** are there?

Solution: 1 hit. Accessing arr[i+1] results in a compulsory miss. The corresponding block (data stored at addresses with the same tag + index) has index = 0 and is loaded into the cache. The block contains 8 bytes of data corresponding to the 4 byte integers stored at &arr[i] and &arr[i+1]. Thus, accessing arr[i] results in a hit. Address &arr[i+2] has a different index (increased by 1, index = 1) and the access results in a compulsory miss.

Q5.4 (1.5 points) Which of the following iterations would have the same number of hits as the **first** iteration of the `for` loop (`i = 0`)?

- ☐ `i = 3`
- ☐ `i = 4`
- ☐ `i = 5`
- ☐ `i = 6`
- ☐ `i = 7`
- ☐ `i = 21`
- ☐ `i = 22`
- ☐ `i = 23`
- ■ None of the above

Solution: None of the above
arr[0] and arr[1] belong to the same block, arr[2] and arr[3] belong to the same block...
For odd iterations (i = 1, 3, 5...), the block arr[i] belongs to and the block arr[i+1] and arr[i+2] belong to are already in the cache due to the previous iteration(s). The hit rate is 3/3. For positive, even iterations (i = 2, 4, 6...), the block arr[i] and arr[i+1] belong to are already in the cache. The block arr[i+2] belongs to is not initially in the cache (resulting in a compulsory miss). Therefore, the hit rate is 2/3. Only iteration i = 0 has hit rate = 1/3.

Q5.5 (1 point) For the **second** iteration of the `for` loop (`i = 1`), how many **hits** are there?

Solution: 3

Q5.6 (1.5 points) Which of the following iterations would have the same number of hits as the **second** iteration of the `for` loop (`i = 1`)?

- ■ `i = 3`
- ☐ `i = 4`
- ■ `i = 5`
- ☐ `i = 6`
- ■ `i = 7`
- ■ `i = 21`
- ☐ `i = 22`
- ■ `i = 23`
- ○ None of the above

Solution: Odd iterations

Q5.7 (2.5 points) Considering all memory accesses for this program, how many compulsory misses, non-compulsory misses, and hits are there?

Solution: 13 compulsory misses, 0 non-compulsory misses. 24*3 memory accesses - 13 misses = 59 total hits. Every miss that occurs is compulsory. Each block that is accessed and not currently in the cache has never been loaded into the cache before. 2 misses occur when i = 0. 0 misses occur when i is odd. 1 miss occurs when i is positive and even.

The program on the previous page has been copied below for your convenience:

```
1 #define NUM_INTS 24
2 int32_t arr[NUM_INTS + 2]; // arr is located at address 0x2000
3 for (register int32_t i = 0; i < NUM_INTS; i += 1) {
4   arr[i + 2] = arr[i + 1] + arr[i]; // arr[i + 1] is accessed before arr[i]
5 }
```

For Q5.8 – Q5.10, assume each subpart is independent. As a reminder, our original cache is a **32B direct-mapped cache with 8B blocks**.

Q5.8 (1 point) If we change our original cache to a **24B** direct-mapped cache with 8B blocks, the hit rate for this program...

○ increases. ● remains the same. ○ decreases.

Q5.9 (1 point) If we change our original cache to a 32B direct-mapped cache with **16B** blocks, the hit rate for this program...

● increases. ○ remains the same. ○ decreases.

Q5.10 (1 point) If we change our original cache to a 32B **fully associative** cache with 8B blocks **and a LRU replacement policy**, the hit rate for this program...

○ increases. ● remains the same. ○ decreases.

Q6 *61Confusing (Virtual Memory)* **(10 points)**

For Q6.1 to Q6.3, suppose we have a 4GiB virtual memory space, a 64KiB physical memory space, 128B pages, and a 4-entry fully associative TLB with a most-recently used (MRU) replacement policy.

Q6.1 (2.5 points) How many bits are in the offset, virtual page number (VPN), and physical page number (PPN)?

Solution: 7 offset bits. 32 total virtual address bits - 7 offset bits = 25 VPN bits. 16 total physical address bits - 7 offset bits = 9 PPN bits

Q6.2 (1.5 points) How many entries are in the page table? Write your answer as a power of 2 (e.g. 2^5).

Solution: 2^{25}

Q6.3 (1.5 points) Consider the following program executed by a particular process.

```
1 #define NUM_INTS 256
2 int32_t arr[NUM_INTS];
3 arr[0] = 0;
4 for (register int32_t i = 16; i < NUM_INTS; i += 16) {
5   arr[i] = i;
6 }
```

Suppose the address of `arr` is `0x1000 0000` and the execution of line 3 results in a TLB hit. When executing the `for` loop, what is the **minimum** number of TLB hits that may occur?

Solution: 8. Virtual addresses arr + 0 and arr + 16*4*1 have the same VPN, addresses arr + 16*4*2 and arr + 16*4*3 have the same VPN, ... In total, there are 256/16/2 unique VPNs accessed. Regardless of whether accessing a new VPN results in a TLB hit or miss (during iterations i = even multiples of 16), the corresponding entry of this VPN will be loaded into the TLB before the next iteration. As a result, a TLB hit must occur during iterations i = odd multiples of 16.

For Q6.4 – Q6.6, suppose we have a system as follows:

- 32-bit virtual addresses and 16-bit physical addresses
- 8-bit offsets
- One free physical page with PPN `0x49`
- 4-entry fully associative TLB

The current state of the TLB and first 5 entries of the page table are shown below.

TLB

Valid	VPN	PPN
1	0x00 0005	0x12
0	0x00 0001	0x18
1	0x00 0002	0x45
1	0x00 0003	0x78

Page table (first 5 entries)

PTE
0xB256 0670
0xC490 8924
0x9845 6745
0xA158 9078
0x7256 0645
...

Each page table entry (PTE) is 4 bytes:

For each of the following virtual addresses, translate it into its corresponding physical address and determine what will happen during address translation. Assume each access occurs independently (not sequentially).

Q6.4 (1.5 points) `0x0000 01C9`

Solution: VPN = 0x00 0001. TLB valid bit = 0 -> TLB miss. 2nd row of the page table: valid bit = 1 (page table hit) and PPN = 0x24. Physical address = 0x24C9.

- ○ TLB hit
- ● TLB miss and page table hit
- ○ Page fault

Q6.5 (1.5 points) `0x0000 0340`

Solution: VPN = 0x00 0003. TLB valid bit = 1 -> TLB hit. PPN = 0x78. Physical address = 0x7840.

- ● TLB hit
- ○ TLB miss and page table hit
- ○ Page fault

Q6.6 (1.5 points) `0x0000 0424`

Solution: VPN = 0x00 0004 is not in the TLB -> TLB miss. 5th row of the page table: valid bit = 0 (page fault). VPN = 0x00 0004 is mapped to the free physical page with PPN = 0x49. Physical address = 0x4924.

- ○ TLB hit
- ○ TLB miss and page table hit
- ● Page fault

Q7 *61Calculating π (OpenMP)* **(9 points)**

To estimate the value of π in C, we can generate random points within a unit square $[0, 1] \times [0, 1]$, count how many are within the unit circle, and use the following equation:

$$\pi \approx 4 \times \frac{\text{Points in Circle}}{\text{Total Points}}$$

A working implementation of `estimate_pi` that uses exactly `tot_points` points to estimate π is shown below. You may assume that `random_01` is a function that returns a number uniformly at random from the range $[0, 1)$ and can be called by multiple threads without changing its behavior. Note that a commented out OpenMP directive does nothing.

```
 1 double estimate_pi(int tot_points) {
 2   int points_inside = 0;
 3   // #pragma omp parallel
 4   {
 5     // #pragma omp parallel for
 6     for (int i = 0; i < tot_points; i++) {
 7       double x = random_01();
 8       double y = random_01();
 9       if (x * x + y * y <= 1) { // Check if point is inside
10         // #pragma omp critical
11         points_inside++;
12       }
13     }
14   }
15   return 4 * ((double) points_inside / (double) tot_points);
16 }
```

For Q7.1 – Q7.4, select the below behavior that **best** describes the new behavior of the code if we include OpenMP directives by uncommenting the specified line(s). Each subpart is independent.

Behavior A: Incorrectly estimates π, or does not generate `tot_points` points.
Behavior B: Correctly estimates π using `tot_points` points **faster** than the original implementation.
Behavior C: Correctly estimates π using `tot_points` points **slower** than the original implementation.

Q7.1 (1 point) Uncomment line 3.

● Behavior A ○ Behavior B ○ Behavior C

Q7.2 (1 point) Uncomment line 5.

● Behavior A ○ Behavior B ○ Behavior C

Q7.3 (1 point) Uncomment line 3 and line 10.

● Behavior A ○ Behavior B ○ Behavior C

Q7.4 (1 point) Uncomment line 5 and line 10.

○ Behavior A ● Behavior B ○ Behavior C

Implement the function `estimate_pi_fixed` such that it correctly estimates π using **exactly** `tot_points` points, and is faster than the original `estimate_pi` above (i.e. without any lines commented out). If you do not need a blank, then leave it empty.

```
 1 double estimate_pi_fixed(int tot_points) {
 2   int points_inside = 0;
 3   #pragma omp parallel
 4   {
 5     int local_points = 0;
 6     int thread_num = omp_get_thread_num();
 7     int num_threads = omp_get_num_threads();

 8     for (int i = thread_num;
                    Q7.5

 9              i < tot_points;
                    Q7.6

10              i += num_threads) {
                     Q7.7
11       double x = random_01();
12       double y = random_01();
13       if (x * x + y * y <= 1) {

14         // blank
              Q7.8

15         local_points++;
                Q7.9
16       }
17     }

18     #pragma omp critical
                Q7.10

19     points_inside += local_points;
                      Q7.11
20   }
21   return 4 * ((double) points_inside / (double) tot_points);
22 }
```

Q8 *61Cool C Question (SIMD)* **(8 points)**

Implement `abs_sum`, a function that takes in a large array of integers `arr` of length `arr_size` and calculates the sum of the absolute values of each element.

For example, the `abs_sum` of the array `[1, -2, -3, 4, 5, -6, 7, -8, 9]` is `45`.

For full credit, your implementation must run as fast as possible. You may assume `abs` is a function that correctly returns the absolute value of the argument.

- `vector vec_load(uint32_t *A)`: Loads 4 integers from memory address `A` into a vector
- `vector vec_store(vector *mem_addr, vector A)`: Stores vector `A` to `mem_addr`
- `vector vec_setnum(uint32_t num)`: Returns a vector where every element is equal to `num`
- `vector vec_and(vector A, vector B)`: Computes the bitwise AND between each pair of corresponding vector elements in `A` and `B`, and returns a new vector with the result
- `vector vec_or(vector A, vector B)`: Computes the bitwise OR between each pair of corresponding vector elements in `A` and `B`, and returns a new vector with the result
- `vector vec_xor(vector A, vector B)`: Computes the bitwise XOR between each pair of corresponding vector elements in `A` and `B`, and returns a new vector with the result
- `vector vec_sra(vector A, vector count)`: Shifts each pair of corresponding vector elements in `A` to the right by `count` with sign extension, and returns a new vector with the result
- `vector vec_add(vector A, vector B)`: Adds `A` and `B` together elementwise, and returns a new vector with the result
- `vector vec_sub(vector A, vector B)`: Subtracts `B` from `A` elementwise, and returns a new vector with the result

```
 1 int abs_sum(const int *arr, int arr_size) {
 2   vector sum = vec_setnum(0);

 3   vector shift = vec_setnum( 31 );
                                Q8.1

 4   for (int i = 0; i < arr_size / 4 * 4;  i += 4) {
                         Q8.2                Q8.3
 5     vector vec = vec_load((vector *)(arr + i));
 6     vector mask = vec_sra(vec, shift);
 7     vec = vec_xor(vec, mask);

 8     vec = vec_sub(vec, mask);
             Q8.4

 9     sum = vec_add(sum, vec);
             Q8.5
10   }
11   int result[4];

12   vec_store((vector *)result, sum);
       Q8.6
13   for (int i = /* Blank Q8.2 */; i < arr_size; i += 1) {
14     result[0] += abs(arr[i]);
15   }
16   return result[0] + result[1] + result[2] + result[3];
17 }
```

Q9 *61Core of the Multi Variety Matrix Multiplication (PLP)* **(13 points)**

Note: This question was inspired by the game TIS-100, written by Zach Barth.

Multiprocess programming is great, but communication is so expensive... Taking inspiration from neural networks, we create new *micro-CPUs*, which work as follows:

- They have a reduced instruction set of RISC-V, where there are no load or store instructions. Assume DMEM does not exist in our micro-CPUs because data memory accesses are "too slow".
- They also have built-in communication systems to "connect" (i.e. wire) each micro-CPU to up to four other micro-CPUs:
  - For each of the four directions (`up`, `down`, `left`, and `right`), at most one micro-CPU is connected.
  - All micro-CPUs share a clock but have separate IMEMs, PCs, and RegFiles.
  - To communicate, two additional 32-bit RISC-V instructions are added:
    - `send rs1 direction` starts a "send operation" to the micro-CPU in the specified direction. This operation will stall until the destination micro-CPU runs a corresponding `recv`. Once both sender and receiver are ready, sends the value in `rs1` to the receiver.
    - `recv rd direction` starts a "receive operation" from the micro-CPU in the specified direction. This operation will stall until the source micro-CPU runs a corresponding `send`. Once both sender and receiver are ready, saves the received value in `rd`.

Warmup: Suppose we connect two micro-CPUs that run programs `A` and `B` as follows:

Micro-CPU Layout

A	B

- In the layout above, each cell represents one micro-CPU, and adjacent cells represent a connection. For example, the micro-CPU labeled `A` has `right` connected to the micro-CPU labeled `B` and `left`, `up`, and `down` connected to nothing.
- The label on each micro-CPU represents the program it will execute. For example, the micro-CPU labeled A will execute the program named `A`.

We would like to pass the value stored in the left micro-CPU's `a0` register to the right micro-CPU's `a0` register. If the value in `a0` of the left CPU is `10`, then the following pair of diagrams shows each micro-CPU's `a0` registers before and after running their respective programs. A value of `?` indicates that it can be any value.

`a0` Before Program Execution

10	?

`a0` After Program Execution

?	10

Implement programs `A` and `B` such that the micro-CPUs match the expected behavior described above. After each program is done, it should jump (or branch) to the label `exit`.

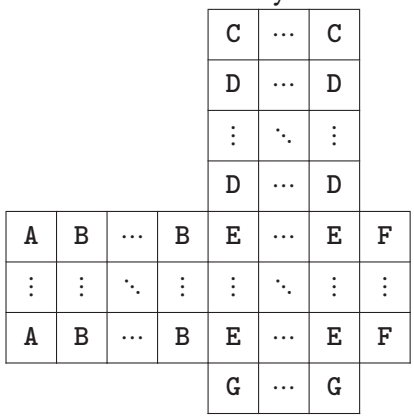
```
A:
    send a0 right
            Q9.1
    j exit
```

```
B:
    recv a0 left
            Q9.2
    j exit
```

We now decide to implement parallel matrix multiplication by running seven programs, `A` through `G`, on micro-CPUs as follows:

- Input in `a0` of the micro-CPUs
  - Micro-CPUs labeled `B` contain elements of the matrix W.
  - Micro-CPUs labeled `D` contain elements of the matrix X.
  - Micro-CPUs labeled `A` and `C` contain the value 0.
  - All other micro-CPUs may contain any value.
- Output in `a0` of the micro-CPUs
  - Micro-CPUs labeled `E` should contain elements of the output matrix.
  - All other micro-CPUs may contain any value.

Micro-CPU Layout

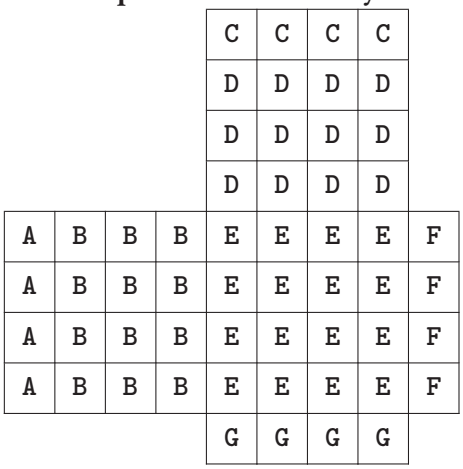


For example, let's consider the following matrix multiplication operation along with the corresponding micro-CPU layout. The values in the `a0` registers in each of the micro-CPUs before execution and the expected values after execution are also shown below.

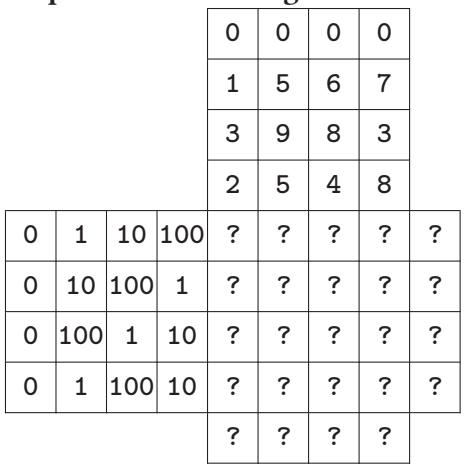
Example: Matrix Multiplication Operation

$$\underbrace{\begin{bmatrix} 1 & 10 & 100 \\ 10 & 100 & 1 \\ 100 & 1 & 10 \\ 1 & 100 & 10 \end{bmatrix}}_{\text{Matrix W}} \times \underbrace{\begin{bmatrix} 1 & 5 & 6 & 7 \\ 3 & 9 & 8 & 3 \\ 2 & 5 & 4 & 8 \end{bmatrix}}_{\text{Matrix X}} = \underbrace{\begin{bmatrix} 231 & 595 & 486 & 837 \\ 312 & 955 & 864 & 378 \\ 123 & 559 & 648 & 783 \\ 321 & 955 & 846 & 387 \end{bmatrix}}_{\text{Output Matrix}}$$

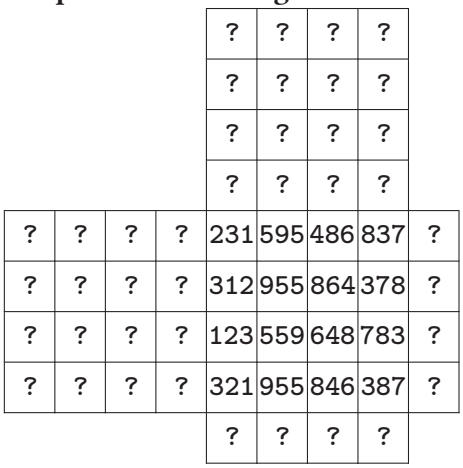
Example: Micro-CPU Layout



Example: `a0` Before Program Execution



Example: `a0` After Program Execution



Implement programs `A` through `G` such that the micro-CPUs match the expected behavior described above and follows calling convention. After each program is done, it should jump (or branch) to the label `exit`.

```
A:
    send a0 right
    j exit
```

```
B:
    send a0 right
        Q9.3
    beq a0 x0 exit
        Q9.4
    recv a0 left
        Q9.5
    j B
```

```
C:
    send a0 down
        Q9.6
    j exit
```

```
D:
    send a0 down
        Q9.7
    beq a0 x0 exit
        Q9.8
    recv a0 up
        Q9.9
    j D
        Q9.10
```

```
E:
    mv a0 x0
Loop:
    recv a1 left
        Q9.11
    recv a2 up
        Q9.12
    send a1 right
        Q9.13
    mul a3 a2 a1
        Q9.14
    send a2 down
        Q9.15
    beq a1 x0 exit
        Q9.16
    add a0 a0 a3
        Q9.17
    j Loop
```

```
F:
    recv a0 left
        Q9.18
    beq a0 x0 exit
        Q9.19
    j F
        Q9.20
```

```
G:
    recv a0 up
        Q9.21
    beq a0 x0 exit
        Q9.22
    j G
        Q9.23
```

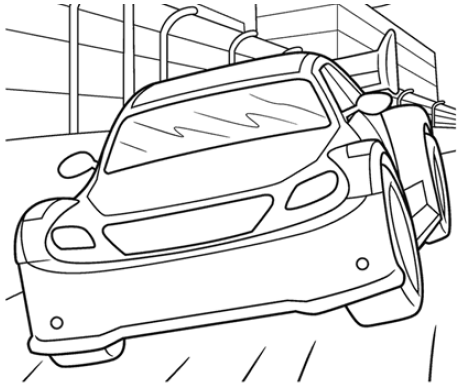
Q10 *61Colorless Epilogue* **(0 points)**

These questions will not be assigned credit; feel free to leave them blank.

Q10.1 The Coppersmith-Winograd algorithm to perform matrix multiplication has a time complexity of $O(n^{2.3737})$. What is the runtime of the matrix multiplication performed in Q9?

Solution: $O(N)$

Q10.2 Decorate the 61Car.



Q10.3 If there's anything else you want us to know, or you feel like there was an ambiguity in the exam, please put it in the box below.

For ambiguities, you must qualify your answer and provide an answer for both interpretations. For example, "if the question is asking about A, then my answer is X, but if the question is asking about B, then my answer is Y". You will only receive credit if it is a genuine ambiguity and both of your answers are correct. We will only look at ambiguities if you request a regrade.

Solution: ¯_(ツ)_/¯