CS 61C
Fall 2023

Garcia, Yokota
Final

Print your name: ______________________________

Print your student ID: ______________________________

You have 170 minutes. There are 11 questions of varying credit (100 points total).

Question:	1	2	3	4	5	6	7	8	9	10	11	Total
Points:	7	13	6	11	13	7	9	9	8	16	1	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: ______________________________

This page intentionally left (mostly) blank.

The exam begins on the next page.

Q1 *Potpourri* (7 points)

Q1.1 (1.5 points) Using multithreading is guaranteed to speed up all programs.

○ True ○ False

Q1.2 (1.5 points) Virtual memory provides memory isolation across all threads.

○ True ○ False

Q1.3 (2 points) You currently have a memory system with an L1 cache and DRAM with the following hit times and hit rates:

	Hit Time	Local Hit Rate
L1 Cache	5ns	20%
DRAM	250ns	100%

You want to add an L2 Cache to improve the average memory access time of this system. Assume that the hit time for this L2 Cache would be 15ns. What is the local hit rate that the L2 Cache would need to make the average memory access time of this system 25ns? Express your answer as a percentage.

%

Q1.4 (2 points) You have a program that takes 20 seconds to run, but you've found a way to make 80% of the code 4 times faster at the cost of some overhead. After rerunning your program, it takes 13 seconds. What is the overhead (in seconds)?

seconds

Q2 *Eddy Needs a Project 3 Extension* (13 points)

For this problem, assume that we're working with the single-cycle datapath presented on the reference card.

Suppose Eddy wants to support the following instruction:

`baleq rd rs1 rs2 offset` (branch and link if equal)

```
if (rs1 == rs2) {
  rd = PC + 4
  PC = PC + offset
}
```

For each of the following control signals, indicate the value it should have for `baleq`. If the control signal is not constant, select "None of the above".

Q2.1 (1.5 points) PCSel

○ `PC + 4` ○ Doesn't matter

○ `ALUOut` ○ None of the above

Q2.2 (1.5 points) RegWEn

○ Write disabled ○ Doesn't matter

○ Write enabled ○ None of the above

Q2.3 (1.5 points) ASel

○ `rs1` ○ Doesn't matter

○ `PC` ○ None of the above

Q2.4 (1.5 points) BSel

○ `rs2` ○ Doesn't matter

○ `imm` ○ None of the above

Q2.5 (1.5 points) BrUn

○ Signed ○ Doesn't matter

○ Unsigned ○ None of the above

Q2.6 (1.5 points) WBSel

○ `PC + 4` ○ Doesn't matter

○ `ALUOut` ○ None of the above

○ `MemReadData`

Our RISC-V datapath currently does not support an instruction that atomically loads from and stores to memory. Suppose we introduce the following instruction:

`alsw rd rs2 imm(rs1)` (atomic load and store word)

```
rd = *(rs1 + imm)
*(rs1 + imm) = rs2
```

Q2.7 (4 points) What additional changes would we need to make to our datapath in order for us to implement **`alsw`** (with as few changes as possible)? Select all that apply.

- □ Create a new instruction type and update the ImmGen
- □ Add a new output to the RegFile for a third register value
- □ Add another WriteData and WriteIndex input to the RegFile
- □ 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 third input into ALU and update any relevant selector/control logic
- □ Allow the ALU to send out more than 1 output and update any relevant selector/control logic
- □ Allow the DMEM to be able to read and write at the same clock cycle and update any relevant selector/control logic
- □ Add a new input to the WBMux and update any relevant selector/control logic
- □ None of the above

Q3 *π-pelining* (6 points)

For this problem, assume that we're working with the five-stage pipelined datapath presented on the reference card, and that the CPU will always predict that branches are not taken.

Consider the following RISC-V code:

```
1   beq x0 x0 Label
2   addi x0 x0 3
3   addi x0 x0 1
4   addi x0 x0 4
5   addi x0 x0 1
6   addi x0 x0 5
  Label:
7   addi t0 x0 9
8   addi t1 t0 2
9   xori t1 x0 6
```

Suppose that the IF stage of the `beq` on line 1 occurs during cycle 1.

Q3.1 (3 points) If all hazards are resolved through stalling (no double pumping or forwarding paths), during which cycle does the `xori` on line 9 execute its WB stage?

Cycle

For each hazard, write down the hazard, the instructions involved, and the number of stalls required. We will only look at this box if you request a regrade.

Q3.2 (3 points) If we implement double pumping and all forwarding paths, during which cycle does the `xori` on line 9 execute its WB stage?

Cycle

For each hazard, write down the hazard, the instructions involved, and the number of stalls required. We will only look at this box if you request a regrade.

Q4 *Piracy: arr!* (11 points)

For Q4.1, assume that we have a 32-bit address space with a 32KiB, 8-way associative cache with a block size of 512B.

Q4.1 (3 points) Calculate the TIO bits with this cache setup.

T:	I:	O:

For Q4.2 to Q4.5, assume that we have a 16 byte, fully associative cache with 4B blocks. For hit rates, please express your answer as a simplified fraction.

```c
#define ARRAY_SIZE 6
int main() {
    int32_t arr[ARRAY_SIZE];

    for(int i = 0; i < ARRAY_SIZE; i++) {
        arr[i] += arr[0];
        arr[i] += arr[1];
        arr[i] += arr[2];
        arr[i] += arr[3];
    }
}
```

Q4.2 (1.5 points) What is the hit rate for the **first** iteration of the `for` loop, using an **LRU** replacement policy?

Q4.3 (1.5 points) What is the hit rate for the **first** iteration of the `for` loop, using an **MRU** replacement policy?

Q4.4 (2.5 points) What is the hit rate for the **last** iteration of the `for` loop, using an **LRU** replacement policy?

Q4.5 (2.5 points) What is the hit rate for the **last** iteration of the `for` loop, using an **MRU** replacement policy?

Q5 *Even Greater Odds* **(13 points)**

Consider the following C function, which returns `true` if the given array has more even elements than odd elements, or `false` otherwise. You may assume that the array only contains strictly positive integers, that the variables `even_count` and `odd_count` will not overflow, and that all necessary C library header files are included.

```
bool more_even(uint32_t* array, uint32_t n) {
  uint32_t even_count = 0;
  uint32_t odd_count = 0;
  for (uint32_t i = 0; i < n; i++) {
    if (array[i] % 2 == 0) {
      even_count++;
    } else {
      odd_count++;
    }
  }
  return even_count > odd_count;
}
```

You have access to the following SIMD operations. A vector is a 128-bit vector register capable of holding 4 32-bit unsigned integers.

- `vector vec_load(uint32_t* A)`: Loads 4 integers at memory address `A` into a vector
- `vector vec_setnum(uint32_t num)`: Creates 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_add(vector A, vector B)`: Adds A and B together elementwise, and returns a new vector with the result
- `uint32_t vec_sum(vector A)`: Adds all elements of the vector together, and returns the sum

Fill in the blanks below to finish the implementation of `more_even_simd`. Assume that your code for this subpart is only required to work on inputs where `n` (the length of `array`) is a multiple of 4. You may only use up to one SIMD operation per blank.

```
bool more_even_simd(uint32_t* array, uint32_t n) {

  vector counts = vec_setnum(_____________________);
                                     Q5.1

  vector mask = ___________________________________;
                               Q5.2

  for (uint32_t i = 0; i < _____________________; i+=4) {
                                   Q5.3

    vector temp = vec_load(_______________________);
                                    Q5.4

    vector masked = _______________________________;
                                 Q5.5

    counts = ______________________________________;
                              Q5.6
  }

  return (_______________________________________) >
                          Q5.7

         (_______________________________________);
                          Q5.8
}
```

For Q5.9 to Q5.10, consider the following implementations of `more_even` that use thread-level parallelism. Assume that there are no syntax errors.

For each implementation, determine whether the implementation is...

...correct and faster than the naive `more_even`

...correct and slower than the naive `more_even`

...incorrect.

A correct implementation is one that will always return the same value as the naive `more_even` function.

You should evaluate the performance of each implementation as the array size approaches infinity (in other words, when the array is really really large). You may assume that the machine has 16 cores and OpenMP uses 16 threads.

If you choose "correct and slower" or "incorrect", please justify your answer. We will only read the first 15 words of each justification.

Q5.9 (2 points)

```
bool more_even_pragma(uint32_t* array, uint32_t n) {
  uint32_t even_count = 0;
  uint32_t odd_count = 0;
  #pragma omp parallel for
  for (uint32_t i = 0; i < n; i++) {
    if (array[i] % 2 == 0) {
      even_count++;
    } else {
      odd_count++;
    }
  }
  return even_count > odd_count;
}
```

○ Correct and faster than the naive `more_even`

○ Correct and slower than the naive `more_even`

○ Incorrect

Q5.10 (2 points)

```
bool more_even_pragma(uint32_t* array, uint32_t n) {
  uint32_t even_count = 0;
  uint32_t odd_count = 0;
  #pragma omp parallel
  {
    uint32_t evens;
    uint32_t odds;
    for (uint32_t i = 0; i < n; i++) {
      if (array[i] % 2 == 0) {
        evens++;
      } else {
        odds++;
      }
    }
    #pragma omp critical
    {
      even_count = evens;
      odd_count = odds;
    }
  }
  return even_count > odd_count;
}
```

○ Correct and faster than the naive `more_even`

○ Correct and slower than the naive `more_even`

○ Incorrect

Q6 *Mixed Messages* (7 points)

A new data center has to perform `2N` tasks using its `C` cores. These tasks, indexed `0` to `2N - 1` inclusive, are independent, except that each task `N + i` must be done after task `i` has been completed. We decide to use the manager-worker approach with one process per core, where the manager keeps track of a queue `task_queue` containing all unassigned tasks that can be started. We'll use process 0 as the manager.

Throughout this question, you must minimize the resources consumed by the data center:

- Do not send unnecessary messages.
- Terminate unused processes as soon as possible.

Assume that each process enters the main loop simultaneously, and that processes will not crash during execution. We implement this task with the following messages:

- `EXECUTE(x)`, where `0 <= x < 2N`.
- `EXIT`
- `DONE(x)`, where `-1 <= x < 2N`. `DONE(-1)` indicates that a worker is ready but has not performed a task.

A worker should have the following behavior:

Q6.1 (1 point) Before the main loop...

○ Perform task 0

○ Cleanup and exit process

○ Send the `DONE(-1)` message to process 0

○ Do nothing

Q6.2 (1 point) Within the main loop, if we receive an `EXECUTE(x)` message...

○ Perform task `x`

○ Perform task `x`, then cleanup and exit process

○ Perform task `x` and send the `DONE(x)` message to process 0

○ Perform task `x` and send the `DONE(x)` message to process `x`

○ Do nothing

Q6.3 (1 point) Within the main loop, if we receive an `EXIT` message...

○ Perform task 0

○ Cleanup and exit process

○ Send the `DONE(-1)` message to process 0

○ Do nothing

A manager should have the following behavior:

Q6.4 (1 point) Before the main loop, initialize `task_queue` to be...

○ An empty list

○ A list containing tasks `0` through `N - 1` in order

○ A list containing tasks `0` through `2N - 1` in order

Q6.5 (1 point) Within the main loop, if we receive a `DONE(x)` message from process `p`...

○ Perform task `x`

○ Add task `x` to `task_queue`

○ If `0 <= x < N`, add task `N + x` to `task_queue`

○ Do nothing

Q6.6 (1 point) Immediately after Q6.5, if the `task_queue` is not empty...

○ Remove the first task `t` in `task_queue`, send `EXECUTE(t)` to process 0

○ Remove the first task `t` in `task_queue`, send `EXECUTE(t)` to process `p`

○ Remove the first task `t` in `task_queue`, send `EXECUTE(N+t)` to process 0

○ Remove the first task `t` in `task_queue`, send `EXECUTE(N+t)` to process `p`

○ Send `EXIT` to process `p`

○ Do nothing

Q6.7 (1 point) Immediately after Q6.5, if the `task_queue` is empty...

○ Send `EXIT` to process `p`

○ If we have assigned all `2N` tasks, send `EXIT` to process `p`, otherwise do nothing

○ Do nothing

Once all workers have received an `EXIT`, the manager should exit.

Q7 *The Fault in Our Pages* (9 points)

Q7.1 (3 points) Suppose we have a 48-bit address space with 32 GiB of physical memory and a 2 MiB page size. How many bits are in our page offset, physical page number, and virtual page number?

Offset: bits

PPN: bits

VPN: bits

Regardless of your answers to Q7.1, now assume we have a 48-bit address space using 24-bit page offsets, 24-bit virtual page numbers, and 12-bit physical page numbers. The system also has a TLB. The TLB and a subset of the page table are shown below. You may assume that the next physical page to be allocated has PPN `0x123`, and that all accesses are independent of each other.

TLB			
Dirty	Valid	VPN	PPN
1	0	0x00 0000	0x203
0	1	0x00 0003	0x168
0	1	0x00 0002	0x164
1	0	0x61 C002	0x727
1	1	0x61 B001	0xC8E
0	0	0x0F 100F	0xE02
0	1	0x61 C001	0xAF4
1	1	0x00 0001	0x162

Page Table			
Index	Dirty	Valid	PPN
0x00 0000	1	1	0x161
0x00 0001	1	1	0x162
0x00 0002	0	1	0x164
...			
0x61 C000	0	0	0x625
0x61 C001	0	1	0xAF4
0x61 C002	1	0	0x727
...			

For each of the following virtual addresses, translate it into a physical address and determine what will happen if we access this address.

Q7.2 (2 points) `0x0000 00AB ACAB`

0x

○ TLB hit

○ TLB miss and page table hit

○ Page fault

Q7.3 (2 points) `0x61C0 02B1 ADE2`

0x

○ TLB hit

○ TLB miss and page table hit

○ Page fault

Q7.4 (2 points) `0x61B0 01FE 3121`

0x

○ TLB hit

○ TLB miss and page table hit

○ Page fault

Q8 *Cumulative: Unconditional RISC* **(9 points)**

Assume we have a function, `f`, that takes in a 32-bit unsigned integer, `x`, as an argument. `f(x)` is defined as follows:

$$\texttt{f(x)} = \begin{cases} \texttt{x + 9} & \text{if } \texttt{x \% 4 == 0} \\ \texttt{x * 2} & \text{if } \texttt{x \% 4 == 1} \\ \texttt{x} & \text{if } \texttt{x \% 4 == 2} \\ \texttt{x // 8} & \text{if } \texttt{x \% 4 == 3} \end{cases}$$

Jero wants to write this function in RISC-V, but he couldn't get his CS61CPU's branch instructions to work! As a result, you may use any RV32I instruction **except** branch instructions.

Write a function, `f`, which accepts one argument `x` in `a0`, and returns `f(x)`. You may assume that there is no overflow.

```
f:

    ________________________________________
                   Q8.1

    ________________________________________
                   Q8.2

    ________________________________________
                   Q8.3
    add t3 t3 t7

    jalr __________________________________
                     Q8.4

    ________________________________________
                   Q8.5
    j f_end

    ________________________________________
                   Q8.6
    j f_end

    ________________________________________
                   Q8.7
    j f_end

    ________________________________________
                   Q8.8
f_end:
    ret
```

Q9 *Cumulative: Orion's Hell* (8 points)

Having failed to completely tame the CS61Cerberus, Heracles has been stuck in Hades for the past two months. The goddess of knowledge Athena decides to help, by letting Heracles cast a spell on Orion.

Warning: This question is significantly harder than any other problem Heracles has faced so far.

Recall from the midterm the following:

- Orion has a favorite number, represented as an n-bit integer. In order to tame Orion, Heracles must determine Orion's favorite number (through his `solve_orion` function) by calling the `orion` function on various numbers, and observing the results.
- Originally, the `orion` function works by performing a bitwise OR on the input and Orion's favorite number, and returning 1 (true) if the result is 0, and 0 (false) otherwise.

The below is the compiled (RISC-V RV32I) code of Orion's `orion` function:

```
orion: # Input is received in a0, and the result is outputted in a0
  li a1 ORNUM # Orion's favorite number omitted
  or a0 a0 a1
  bne a0 x0 FalseCase
  addi a0 x0 0
  jr ra
FalseCase:
  addi a0 x0 1
  jr ra
```

Athena's spell can change **one bit** in the assembled bytecode of the `orion` function; Heracles can then use the modified `orion` function instead in his `solve_orion` function. The spell only lasts for a short time, so `solve_orion` will now have stricter asymptotic runtime requirements.

After the single bit change, the `orion` function must still be valid RISC-V code that observes calling convention, and must still work regardless of any other code (e.g. you can't load/store to unknown memory, specify the value of any register except `a0`, or jump out of the `orion` function in undefined manners).

Q9.1 Select one bit in the **orion** function to flip, which will allow Heracles to determine Orion's favorite number. In addition, write the **solve_orion** function (in C), which will work given your modified **orion** function.

For full credit, your **solve_orion** must run in $O(1)$ time relative to the number of bits in Orion's favorite number. For 75% credit, your solution may run in $O(n)$ time instead.

Flip bit ☐ of the instruction ☐

to turn it into the instruction ☐ .

```
uint32_t solve_orion(bool(*orion)(uint32_t)) {
    // Your code here




}
```

Q9.2 Briefly explain your solution. We will only look at this box if you request a regrade.

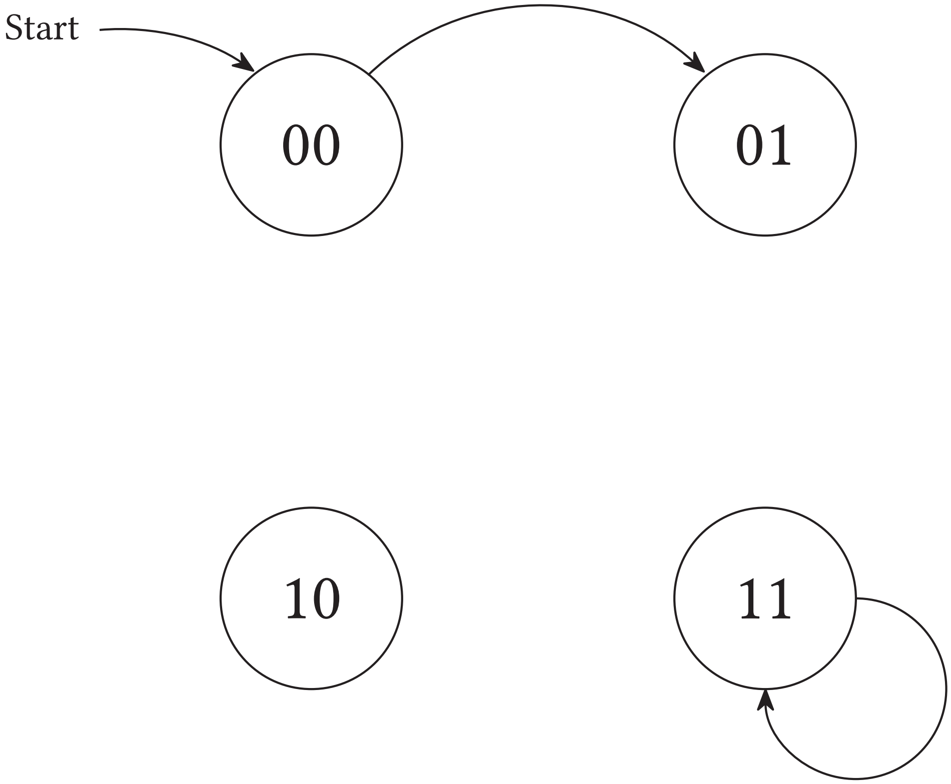
Q10 *Cumulative: Art Class* (16 points)

You wish to create an FSM whose output at time step N is 0 for the first two time steps, and the input of time step $N - 2$ otherwise.

For example, if the input to this FSM was `0b01 1011 1011 1000 1001`,

the output should be `0b00 0110 1110 1110 0010`.

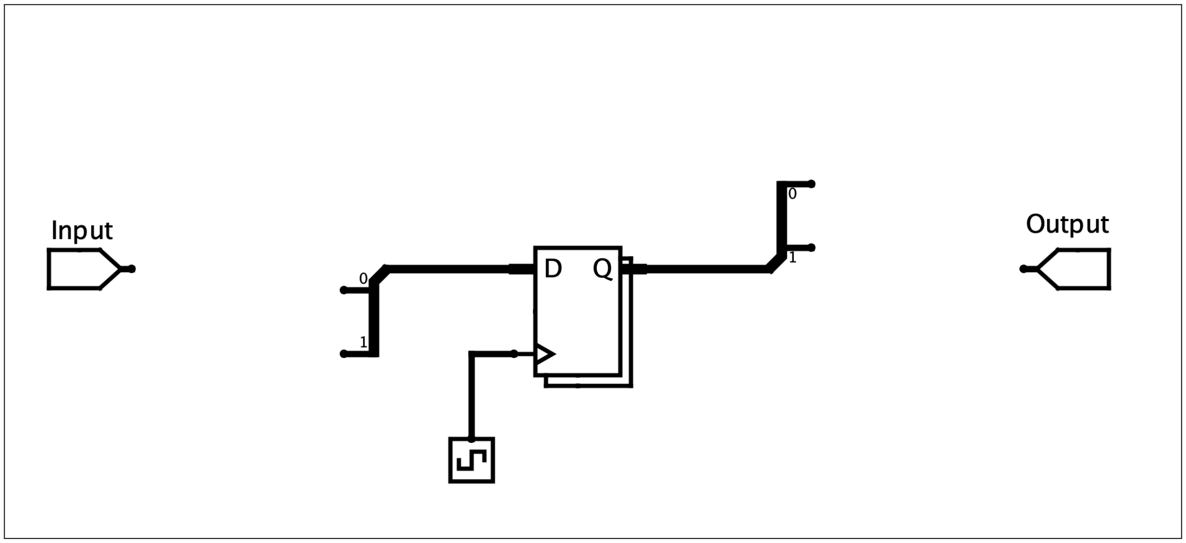
Q10.1 (8 points) Complete the FSM below. You may not add additional states. Note that you must also label the state transitions we have provided for you.



Q10.2 (4 points) Fill in the circuit diagram below to implement this FSM. For full credit, your circuit must have the minimum possible clock period, assuming the following component delays:

$$t_{\text{AND gate}} = 12\text{ps} \qquad t_{\text{Register clk-to-q}} = 10\text{ps}$$
$$t_{\text{OR gate}} = 15\text{ps} \qquad t_{\text{Register setup}} = 15\text{ps}$$
$$t_{\text{NOT gate}} = 4\text{ps} \qquad t_{\text{Bit splitter}} = 0\text{ps}$$
$$t_{\text{XOR gate}} = 31\text{ps} \qquad t_{\text{Wire}} = 0\text{ps}$$

You may not use any other components. You may assume that the input and output connect directly to registers (for the purpose of determining the clock period), and that the register stores 2 bits. Your circuit does not need to "match" the states you use in your answer to Q10.1; it will be considered correct if its behavior matches the intended behavior described above.



Unsatisfied with just delaying the input by 2 cycles, you decide to create an FSM that delays the input by 12 cycles (and outputs 0 for the first 12 cycles).

For example, if the input to the new FSM was `0b01 1011 1011 1000 1001`,

you should output `0b00 0000 0000 0001 1011`.

Q10.3 (2 points) What is the fewest number of states that an FSM solving this problem can have? Your answer must be an exact **integer**.

Q10.4 (2 points) What is the minimum clock period of any circuit that solves this problem (assuming the register is expanded to sufficiently many bits without increasing clk-to-q and setup times)?

Q11 *The Finish Line* (1 points)

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

Q11.1 (1 point) How long does it take to “do nothing”?

Q11.2 (0 points) 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.