ECE411 Computer Organization and Design
Mid-Term Exam 2
4/9/2024

NetID: _____
First Name:_____
Last Name:_____

Exam Guidelines:

1. This exam has **6 problems**. Make sure you have a complete exam before you begin **[18 pages + the cover page]**.
2. Write your NetID on every page in case pages become separated during grading.
3. You will have **3 hours** to complete this exam.
4. Write all of your answers on the exam itself. If you need more space to answer a given problem, continue on the back of the page, but clearly indicate that you have done so.
5. This exam is closed-book. You may use one sheet of notes. You may use a calculator.
6. **DO NOT** do anything that might be perceived as cheating. The minimum penalty will be a grade of zero.
7. Show all of your work on all problems. Correct answers that do not include work demonstrating how they were generated may not receive full credit, and answers that show no work cannot receive partial credit.

Good luck!

Question	Part 1	Part 2	Part 3	Part 4		Total
1	/4	/4	/4	/2		/14
2	/8					/8
3	/6	/6	/10			/22
4	/4	/4	/4	/4		/16
5	/6	/5	/6			/17
6	/4	/4	/4	/2	/2	/16
Total						/93

Question 1: Energy Efficiency (14 points)

You and your friends love to play an online video game called BitCraft, where players are placed into their own virtual world. In the game, players can move around the world to interact with each other and modify their own virtual world however they wish. The status of this virtual world is hosted on another server far away from your computer.

When you and your friends are playing the game, your computer must continuously communicate with the server to fetch information about how other players have modified the world and send information about how you have modified the world. After communicating data with the main server, your computer has to update the display frame and other game data local to the computer. This is done by always running 2 processes throughout the runtime of the game:

- Process 1: One process is responsible for communicating with the main server.
- Process 2: Another process is responsible for updating the display frame and local game data.
- One iteration of process 2 runs many more instructions than process 1, so one iteration of process 2 is much slower when running on a single thread. Also, the instructions in process 2 can be significantly accelerated with multithreading, compared to that of process 1.

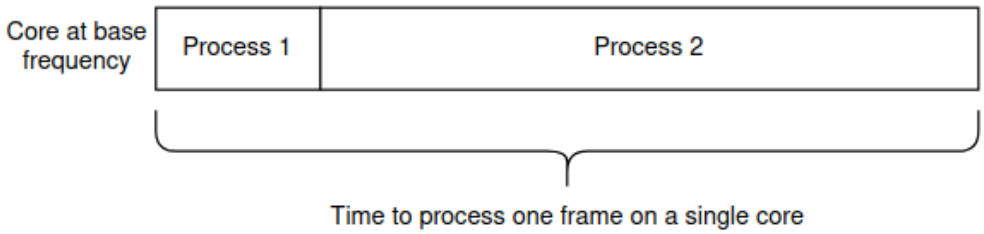
Assume that there are no synchronization issues between the two processes, and **the time it takes to process one frame is equal to the amount of time it takes to go through one iteration of process 1 and process 2**.

The processor chip for your computer has 5 cores that each run a single thread, where each core has a base frequency of 3.5GHz and a base power of 20W. The maximum power your computer can draw is 210W.

We assume:

- No other processes are running on your computer while you are playing the game.
- The performance of each process on a core is linear with the core's frequency
- The frequency of each core can be as high as possible, as long as the total power of the computer does not exceed 210W.
- The non-processor parts of your computer draw 50W constantly while the computer is running.
- When a core is not running, it is **power-gated**.
- The non power-gated cores can run at different frequencies.
- $P = \frac{1}{2}CV^2f$, and V **is proportional to** f.
    - P is power
    - f is frequency

While playing the game, your computer tells you the frames processed per second (FPS) is 50. In other words, it takes 20ms to process one frame (20ms = 1/50s). 50 FPS is considerably low for pro-gamers, so you start looking for ways to increase your FPS by changing your core's frequency and changing the game's settings. You look into the game settings, and you see that the single-core option is turned on, and the single core is running at the base frequency.



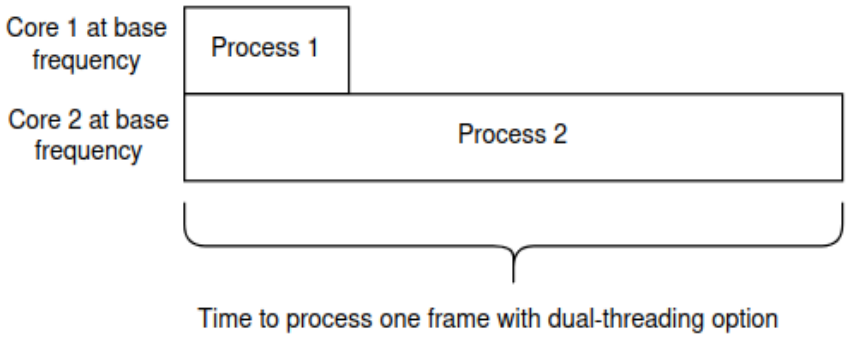
1) (4 points) What is the maximum expected FPS when the program is running sequentially on a single core given the earlier power constraints?

Correct answer: 100 FPS
(10ms to process one frame, 160W on one core)

-1 pt for not subtracting 50 W
-1 pt for math error (global, won't be taken off anywhere else again)
-1.5 pts for not accounting for V proportional to f (cubic relationship)
-1 pts for not converting from time to process one frame to FPS

-2 to -4 pts for completely incorrect direction and misinterpretation of problem

Another setting for the game allows you to enable a dual-threading option such that process 1 runs on one core and process 2 runs on another core, so process 1 and process 2 can run at the same time. When both the cores are running at the base frequency, the video game's FPS jumps to 62.5.



2) (4 points) What is the maximum expected FPS when the dual-threaded option is selected?

Correct answer: 124.356 FPS
(~8.04 ms to process one frame, ~157W on process 2 core, ~3W on process 2 core)

-1 pt for not subtracting 50W
-1 pt for math error (global, won't be taken off anywhere else again)
-1.5 pts for not accounting for V proportional to f (cubic relationship)
-1.5 pts for not calculating/considering work difference between process 1 and process 1 is significantly smaller than process 2
-1 pt for assuming min core power is 20W

-2 to -4 pts for completely incorrect direction and misinterpretation of problem

3) (4 points) Since process 2 can be significantly accelerated by multithreading, the game also offers a multi-threading setting. This setting splits process 2 among the remaining available cores in the system. What is the maximum expected FPS when the multi-threaded option is selected?

Correct answer: 293.255 FPS
(~3.41 ms to process one frame, 32W for each core)

-1 pt for not subtracting 50W
-1 pt for math error (global, won't be taken off anywhere else again)

-1.5 pts for not accounting for V proportional to f (cubic relationship)
-1.5 pts for not calculating/considering work difference between process 1 and process 1 is significantly smaller than process 2

-2 to -4 pts for completely incorrect direction and misinterpretation of problem

4) (2 points) Propose a **hardware/architectural modification** to the processor chip that would allow you to accelerate the instructions in process 2 and use less power.

-2 pts for software/non-hardware change
-1 pts for clock gating mid process (PMS change not architectural modification)
-1 pts for improving memory related systems
-0.5 pts for transistor level improvements

Question 2: nvidia stocks go brrr (8 points)

While researching a specific program, you notice that the majority of your code executes on the GPU while the CPU remains idle. You wonder if it's possible to utilize the CPU idling compute power alongside the GPU. You decide to use data partitioning to distribute portions of tasks between the CPU and GPU.

Data partitioning lets the CPU/GPU handle a subset of the data. It may be tempting to let the GPU do all of the work since the execution time is much shorter, however the data needs to be transferred from main memory to the GPU, which imposes overhead.

You profile the code on the CPU and the GPU and obtain the following results:

Task	A	B	C	D
Time on CPU	40 ms	160 ms	20 ms	15 ms
Time on GPU	20 ms	40 ms	25 ms	10 ms
Data size	10 GB	60 GB	5 GB	15 GB

Assumptions:

- The work distribution between CPU/GPU can differ between tasks
- The amount of data needed to transfer to the GPU is proportional to the percentage of work you assign to it. (If the GPU works on 50% of the task, it only needs 50% of the data)
- Data transfer happens only once; assume no data needs to be transferred back after the computation is done
- Data transfer happens at a rate of 1GB/ms
- Data transfer **cannot** overlap with **GPU** computation but **can** overlap with **CPU** computation
- The tasks are executed in order - A, B, C, D
- Each task must be completed on **both** the CPU and GPU before each device can start on the next task (i.e. Both devices must work on the same task or be idle at any given time)

1) (8 points) Under what distributions is the minimum total execution time achieved?

Time per task is max(t_cpu, t_transfer+t_gpu), minimum total execution is achieved when t_cpu = t_transfer+t_gpu
cpu work distrib.: A - 3/7, B - 5/13, C - 3/5, D- ⅝
-2 for each incorrect work distrib.

Question 3: Load'dib (22 points)

You are designing an out-of-order processor and want to implement load and store logic. Your memory unit contains a load structure and a store structure. Below is the initial high-level description of your system.

The load structure follows these rules:

- Loads are held in a fully associative structure.
  - Entries that are occupied have valid == 1.
- Loads can be removed (sent to memory) out of order with respect to other loads.
  - A load can leave the queue (and be sent to the memory subsystem) once:
    - All stores older than the load have committed
  - At most **one** load will be removed per cycle.
- Insertion requests into the load structure occur in program order
  - New loads can be inserted into any empty entry in the associative structure.
    - Although insertion requests come in order, loads can leave out of order.
      - Thus, the relative ordering of loads within the structure is not guaranteed to be in program order.
  - At most **one** load will be inserted per cycle.

The store structure follows these rules:

- Stores are held in a FIFO queue structure
  - Entries that are occupied have valid == 1. Entries that are empty have valid == 0.
  - The FIFO is implemented via shift registers.
    - Thus, the oldest uncommitted store is always located at index 0.
    - Similarly, the youngest uncommitted store in the store queue is located at the highest index with a valid entry.
  - New stores are inserted at the highest index that is currently invalid (empty)
  - A *signed* tail pointer points to the most recently inserted entry
    - When the FIFO queue is empty, the tail has index == -1.
- Stores can dequeue (and be sent to the memory subsystem) on the same cycle that they commit from the ROB.
  - At most **one** store will dequeue per cycle
- Enqueue requests to the store queue occur in program order
  - At most **one** store will enqueue per cycle

Stores and loads enter the memory unit in program order.

This is a snippet of the running program. These instructions are all in your load/store structures. For simplicity, addresses are disambiguated and stores have their values.

Hints: The processor memory is **little endian** and all operations shown are word (4-byte)
SW X, Y -> Mem[Y] = X
LW X, Y -> RegX = Mem[Y]

```
LW      x1,           0x60000000
SW      0x12345678,   0x6000000C
SW      0x23456789,   0x60000000
SW      0xDEADBEEF,   0x60000008
SW      0x9ABCDEF0,   0x6000000C
LW      x3,           0x60000008
LW      x4,           0x6000000C
LW      x2,           0x60000000
```

1) (6 points) Augment the functional block diagram on the following page with additional signals and/or Load Structure entry information to implement the logic for determining whether a load can be sent to memory.

Requirements:

- You may **not** add additional information to the store queue entries.
- You **can** add arrows between the load and store structures.
    - If you do, label those arrows with the information they carry.
- There is one unlabeled column in the load queue.
    - This column can store information from the store queue.
    - **Derive the bit width of the entries in this column**
    - **Fill in the title of this column and the entries for each valid load instruction.**
    - **Explain how the data in this column would be used to make decisions about when a load can be sent to memory.**

Store Queue

Tail ptr →

Idx	Valid	Address	Value	Size
0	1	0x6000000C	0x12345678	W
1	1	0x60000000	0x23456789	W
2	1	0x60000008	0xDEADBEEF	W
3	1	0x6000000C	0x9ABCDEF0	W
4	0			
5	0			
6	0			

Load Structure

Idx	Valid	Address	Dest Reg	Size	
0	1	0x60000000	x2	W	
1	1	0x6000000C	x4	W	
2	0				
3	0				
4	1	0x60000000	x1	W	
5	0				
6	1	0x60000008	x3	W	

General Rubric

+2 pts: Student draws/mentions need for tail ptr in load structure column in any capacity

+2 pts: verbal/pictorial explanation for how to correctly use the tail ptr (decrement whenever store commits)

+1 pt: Correct bit width for column

+1 pt: Correctly filled in column based on described approach (propagated errors ok)

2) (6 points) Assume:

- The 4 SW instructions commit on cycles X, X+10, X+20, and X+30, respectively.
- If the last SW older than a particular LW commits on cycle Y, that LW will repeatedly attempt to leave the load structure from cycle Y+1 onward.
    - Remember that only one LW can leave the structure per cycle.
    - If multiple LWs are ready to leave on a given cycle, give priority to the one with the smaller index. Stall the other(s).

a) Identify the cycle each LW leaves the Load Structure.

Instruction	Cycle
LW x1, 0x60000000	0, 1, X-1 all accepted
LW x3, 0x60000008	x+31, x+32, x+33 all accepted
LW x4, 0x6000000C	x+31, x+32, x+33 all accepted
LW x2, 0x60000000	x+31, x+32, x+33 all accepted

+1 point per correct value

-1 point if they added optimization for loads leaving if no conflicting stores because that violates the above specification

b) Identify the final values of each of the registers. Assume memory is initialized to all 0s and that no store instructions have run prior to those shown in the program snippet.

Register	Value
x1	0
x2	0x23456789
x3	0xDEADBEEF
x4	0x9ABCDEF0

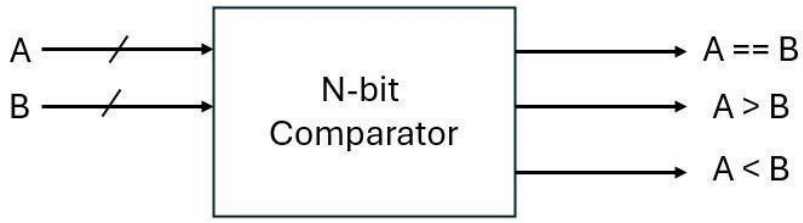
+0.5 points per correct value

3) (10 points) You are interested in store-to-load forwarding. With store-to-load forwarding, stores *older* than a load that target the same address as the load have their values passed directly to the load.

- If multiple stores clobber one another (overwrite the same address), the load should take the value from the *youngest* store that is *older* than the load.
- All stores are compared in parallel against all loads simultaneously.

a) Draw a circuit diagram for the control logic that determines if (and which) store should have its value forwarded to a load. For simplicity, you can draw the circuit for a single load with 4 prospective stores, and can assume all of the stores are older than the load. You may use only standard 2-input logic gates (AND, OR, NOT, etc.), 2:1 n-bit MUXes, and n-bit comparators.

Example comparator:



Any solution that produces correct results without requiring information/gates other than those allowed receive 7 points.

Solutions that were mostly correct received 5 points.

Solutions with large errors received 3 points.

Attempts with some work that is almost entirely incorrect received 1 or 0 points.

0 points if no or barely any attempt

b) How many total MUXes, comparators, and standard logic gates would you need for a design with 8 load structure entries and 8 store queue entries? Report the count for each. You can once again assume all stores are older than the load.

1 point if the solution correctly identifies the need to multiply everything by 8 to account for 8 loads in parallel

1 point for correct comparator count, 0.5 points for correct MUX count and 0.5 points for correct logic gate count.

Cannot receive points if there was no drawing on the previous page. 0 points given for answers with no explanation or clear evidence based on the previous drawing why it is correct.

Question 4: Bodacious Bottlenecks (16 points)

Congratulations! Due to circumstances related to poor benchmarking, Udit has been fired from Raw-Cache, Inc. and you've been selected to be his replacement. Mr. Wu has assigned your first task: Analyzing the team's current out-of-order processor. The following processor is an explicit register renaming architecture and has the following specifications:

- 256 ROB entries
- 128 reservation stations
- 64 physical registers
- One RAT and one RRF
- Four ALUs, four FPUs (with multipliers), and one unified memory access unit (loads and stores)
- All instructions (including branches, jumps, etc.) are only calculated in functional units after being issued
- 16-stage pipeline
- Static not-taken branch predictor
- Unified CDB with a priority arbitration scheme: ALU -> FPU -> memory unit
- 4-way set-associative D-cache and I-cache

Four benchmarks have been run on this processor with a variety of results. For each benchmark, explain what could be hindering performance and suggest a way to redesign the processor. Each part is worth two points - two for correctly determining the bottleneck, and two for proposing a valid solution.

Note: Each design should be independent. In other words, suggest changes for each benchmark independent of other benchmarks. **Make design choices as specific as possible** (include numbers if necessary). Additionally, your design choices should be to the core itself. Do not suggest changes to the caching architecture.

NOTE: The following solutions are only examples of correct answers. There may be more.

Benchmark 1: Float Muls

1) (4 points) This benchmark consists of many floating-point multiplies sequentially with some dependencies. The program works OK at first, but slows down significantly after a short while. You notice that I-cache accesses slow down after some time as well. Additionally, you notice that not all FPUs are used once the processor starts slowing down.

- Bottleneck: You eventually run out of physical registers since there are many more ROB entries than physical registers.
- Solution: Add more physical registers (at least 128 in total)

Benchmark 2: Branch Heavy

2) (4 points) In this benchmark, there are many conditional instructions (branches) that are almost randomly taken or not taken. Performance is generally poor throughout the entire execution.

- Bottleneck: Large misprediction penalty with deep pipeline
- Solution: Decrease the number of pipeline stages or use multiple RATs/RRFs and ROB index tagging to allow for partial flushing

Benchmark 3: Function Calls

3) (4 points) This benchmark consists of a large piece of code making use of many small helper functions. These are called frequently and are rarely more than a few lines of code. Unfortunately, sections that call these helper functions slow down much more than anticipated. Note: do not say that the code is a bottleneck. We have to be nice to software developers.

- Bottleneck: Many unconditional jumps that can't be calculated until they reach the functional unit.
- Solution: Utilize a return address stack (RAS) to improve jump performance

Benchmark 4: Interleaving

4) (4 points) The final benchmark attempts to interleave instructions from two functions - one with mostly integer (ALU) instructions, and another with mostly floating-point (FPU) instructions. The benchmark has been compiled in such a way that the instructions from one function will be independent of the other. For this specific benchmark, the processor has been modified to allow for out-of-order commits between functions. The intended behavior is then that the functions can commit their instructions as soon as they are done without relying on the other function. However, you notice that very few floating-point instructions get committed until nearly the entire integer-based function has completed.

- Bottleneck: CDB priority scheme causes FPU starvation
- Solution: Design a different kind of arbitration scheme - most options work, even a static priority of FPU/mem unit -> ALU.

Question 5: The Cached of Us (17 points)

Cache timing attacks allow attackers to use the timing characteristics of CPU caches to retrieve information from a victim, specifically about the access pattern of shared memory lines. One such attack is known as Flush + Reload, which you will explore and try to mitigate.

Consider a 2-core system with cores P1 and P2. Each core has a write-back L1 cache, and both processors share a write-through L2 cache, which is then connected to DRAM. A modified version of the MSI protocol is used to maintain cache coherence. The CPU supports an instruction `clflush(void *addr)` which will invalidate this cache line in **all** caches (including L2!) Additionally, each core can measure the time of its own memory operations.

Memory operation times on lines in state:

- **M**/**S**: 1 ns
- **I**: 2ns (line is valid in L2) / 100 ns (line not valid in L2)
    - All requests from an invalid line must be handled by the L2 cache.
    - The L2 Cache must write through to DRAM before handling read requests.
    - With the DRAM writeback, an operation will take 100 ns.
- All `clflush` requests take 1 ns

1) (6 points) You are an attacker controlling P2 of this system, trying to spy on the access pattern of a victim on P1. Assume that P1 takes **at most** one action on cacheline A and B each during the wait. As the attacker, you issue and time the following operations:

Time	P2 Operation	Operation Latency
t1	`clflush` (A)	1 ns
t2	`clflush` (B)	1 ns
Wait	-	(No operation for 1 ms)
t3	Read A	100 ns
t4	Write B	2 ns

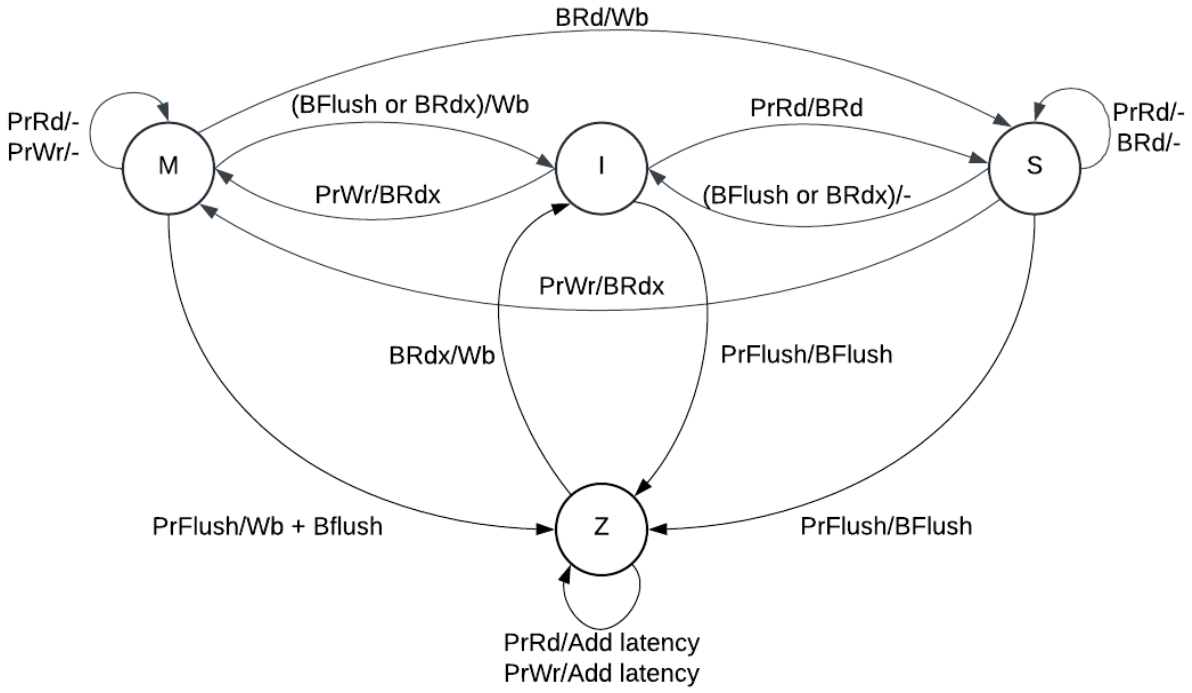
i) During the wait time, which action could the victim have performed on A?
[ ] Read [X] Write [X] No action

ii) During the wait time, which action could the victim have performed on B?
[X] Read [ ] Write [ ] No action

iii) Indicate all possible coherence states for these lines in each processor after t4:

Line	P1	P2
A	I/S	S
B	I	M

As demonstrated above, the attacker can learn some actions that a victim performs. If a victim process has an access pattern dependent on a secret value like a password, the attacker can determine the secret if they can learn the access pattern! To resolve this issue, your friend developed an extension to the MSI protocol, called the MZSI protocol. The new Z state indicates that a line is a **zombie** – it might be the target of a cache timing attack, so it should be handled differently. Any hit on a zombie has artificial latency added to it, so attackers can't infer access patterns from timing. The MZSI protocol that your friend developed is shown below:



PrRd - Local core read **PrWr** - Local core write **PrFlush** - Local core `clflush`
BRd - Bus read miss **BRdx** - Bus write hit/miss **Wb** - Writeback to L2+DRAM
BFlush - Indicate to other caches that a flush has happened, and to invalidate the line
Add latency - Add time to make the operation look like an L2 miss. Does not interact with L2/DRAM.

Transitions are marked as (Trigger conditions/bus actions)

Some things to note:

- Only the L1 cache line whose processor issued a flush is marked as a zombie, to ensure performance for other cores
- Writing to a zombie keeps it a zombie since this operation can be timed as well
- If the BusRdx signal is snooped by a zombie line, it indicates a valid use of `clflush`, (since this instruction has non-malicious uses for coherence, like when expecting another processor to write to a line) and the line can lose the zombie marking

2) (5 points) Fill in the table when using your friend's MZSI Protocol:

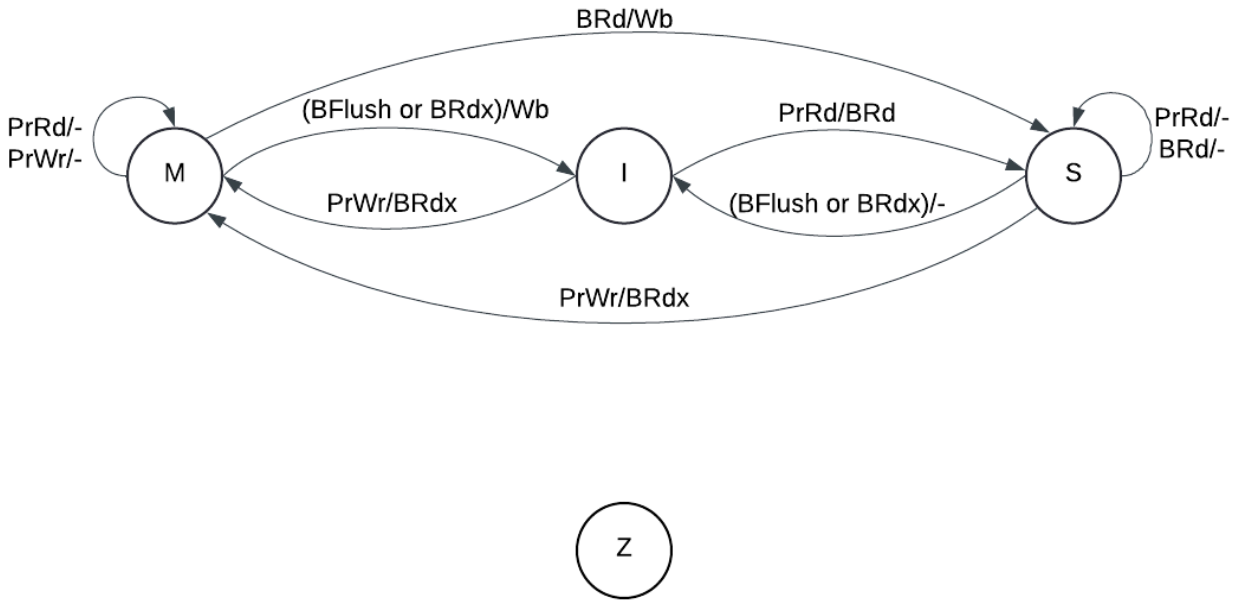
Event	P1 State of line X	P2 State of line X	P1 State of line Y	P2 State of line Y	Operation Time
P2: Flush X	I	Z	I	I	1ns
P2: Flush Y	I	Z	I	Z	1ns
P1: Read X	S	Z	I	Z	100ns
P1: Write Y	S	Z	M	I	100ns
P2: Read X	S	Z	M	I	100ns
P2: Read Y	S	Z	S	S	100ns

3) (6 points) After implementing the new cache protocol, you run some benchmarks. Even benign multithreaded code you run on the dual-core system doesn't work anymore, and you determine the issue is that the new protocol does not maintain coherence.

i) When is coherence not maintained, and why?

When there is a write to a zombie line, coherence is lost. In the event of a flush to a zombie address or a read from a zombie address from another core, there is no writeback to pass the updated memory to the other cores.

ii) Revise your friend's MZSI protocol to preserve coherence, **without** adding any states. Your revision should make no assumptions about the attacker or victim access pattern. Your change must preserve the security advantage. (state diagram on next page):



Solutions need to maintain security and coherence.

For security, any read/write operation should look like an L2 miss, and targeted lines should not leave the zombie state prematurely.

For coherence, zombie lines should have an added write back action, either on PrWr or on BRd.

Question 6: Coherence (16 points)

You are an ECE 411 student preparing to take midterm 2. However, in order to properly get into the study mindset, you want to listen to your favorite music. You've recently discovered that some popular songs (such as Taylor Swift's "Fearless") sound *even better* when sped up + pitched up, so you decide to "nightcore-ify" your entire music library. To make this process quick, you write a multithreaded program to parallelize the work. Some code is shown below.

```
typedef struct {
  uint8_t audio_data [32];
  char title [16];
  char artist [16];
} song_t;

// one thread per core
#define NUM_THREADS 4
#define NUM_SONGS 64

song_t songs [NUM_SONGS];

void mp3_processing_thread(unsigned int thread_id) {
  for (int i = thread_id; i < NUM_SONGS; i += NUM_THREADS) {
      song_t &curr_song = songs[i];
      for(int j = 0; j < 32; ++j) {
            uint8_t b = curr_song.audio_data[j];
            curr_song.audio_data[j] = nightcorify(b);
      }
   }
}
```

1. (4 points) Considering only the data contained in the `song_t` structure, how many cache misses would each of the four `mp3_processing_thread` incur running on a single core system connected to a cache with 256 byte cache lines? Assume no capacity or conflict misses occur.

   core 1: 16
   core 2: 0
   core 3: 0
   core 4: 0

2. (4 points) You instead run the code on a multi-core system with 4 identical cores, where each core is connected to an L1 cache with 256 byte lines. Coherence is maintained via the MESI protocol. How many cache misses would a `mp3_processing_thread` see, assuming all threads start at the same instant, and the bus is fair? (here fair means that the bus will always service pending requests before new ones in a queue-like fashion)

   an exact answer was not required, all answers that noticed the constant invalidation due to the MESI protocol and cache line sharing were accepted
   ex: 16 songs / thread * 32 RAW ops / song = approx. 512 misses per thread

3. (4 points) Assume the nightcoreify function takes 150 clock cycles to compute, and a bus transaction (such as an invalidation, retrieval of data from another core, or DRAM access) takes 25 clock cycles. Will this program run faster in the single core or multi core (parallelized) case?

   faster on single core, constant bus transactions due to invalidation on multi-core system causes it to be slow

4. (2 points) Propose a change to the `mp3_processing_thread` function seen above that will reduce the number of cache misses for the multi-core case.

   change the indexing of the threads so cache lines are not shared b/w cores (no false sharing)
   ex. for(int i = thread_id*(NUM_SONGS / NUM_THREADS); i < (thread_id+1)*(NUM_SONGS / NUM_THREADS); ++i) {}

5. (2 points) Describe a change to the data structure that could reduce the number of cache misses for the multi-core case (without your modification from part 4).
   insert padding such that each element in the array occupies a whole cache line by itself