

Name:

Seat Number:

Sciper:

# CS-307 Final Exam

## Solutions

**Please do not turn this page until instructed to do so.**

**Please write your seat number at the top of each page.**

You have **150 minutes** in total to answer all questions.

Please write clearly and concisely, using a **blue** or **black** pen. Show all work for full credit.

There is an empty sheet in the end which you can use as scratch paper. Please use that first and only ask for extra sheets if you need more.

Total number of pages: **15**

| Problem                                      | Points |
|--|--------|
| Miscellaneous (24 points)                    |        |
| Synchronization and Lock Elision (24 points) |        |
| GPUs (23 points)                             |        |
| Multithreaded Processors (29 points)         |        |
| Total (100 points)                           |        |

**Miscellaneous**

1. The 4C cache miss model:
  - a. List and briefly explain the four types of cache misses. [4 points]
    - a. **Cold/Compulsory misses:** Misses on the first access to data elements
    - b. **Capacity misses:** Misses because the dataset is bigger than the cache capacity
    - c. **Conflict misses:** Misses because data elements compete for the same position in the cache, while other parts are empty
    - d. **Coherence misses:** Misses because of other cores taking away a cached data element

[0.5 points for miss type, 0.5 points for correct explanation]

- b. In the table below, mark how increasing each of the parameters affects each type individually while other parameters are kept constant. You need to first label the first row based on your answer to the previous part. You should put “+” (increases), “-” (decreases), or “=” (unchanged) to indicate the effect of each parameter. Please note each correct answer gets you 0.5 points and for every two wrong answers we deduct 1 point. [8 points]

|                         | Type 1:<br>Cold | Type 2:<br>Capacity | Type 3:<br>Conflict | Type 4:<br>Coherence |
|-------------------------|-----------------|---------------------|---------------------|----------------------|
|                         | =               | -                   | -                   | =                    |
| <b>Cache capacity</b>   | -               | +                   | +                   | +                    |
| <b>Cache block size</b> | =               | =                   | =                   | +                    |
| <b>Number of cores</b>  | =               | =                   | -                   | =                    |
| <b>Associativity</b>    | =               | =                   | -                   | =                    |

**Seat Number:**

2. We have learnt about the directory-based coherence protocols, along with various hardware designs for directories. Based on the same, briefly explain the following:

a. What do directories track? Why is that useful? [2 points]

Directories track which private cache contains a cache block. This is useful because it allows direct cache-to-cache communication.

b. What are sparse directories? What is the benefit over duplicated tag directories? [2 point]

Sparse directories contain more sets and less ways compared to the aggregated private cache sets and ways respectively. This is done by using bits from the tag. Having less ways reduces the number of comparisons for each access, thus reducing the power requirement.

c. What is the problem with sparse directories? Briefly describe a possible solution. [2 points]

As sparse directories use bits from the tag, this can result in contention as the used bit might not be evenly distributed across all the cached blocks. A possible solution is to overprovision the number of sets or ways.

**Seat Number:**

3. We are entering the era of post-Moore computing. Vendors such as Intel and AMD are building products with accelerators not just for AI but also for various software bottlenecks. Consider a program that is composed of 20% memory operations, 60% arithmetic operations, and 20% logic operations. There are a total of 10000 operations in the program. There are no dependencies between different types of operations.

The latencies of each type of operation are as follows:

Memory operation = 20 clock cycles

Arithmetic operation = 5 clock cycles

Logic operation = 10 clock cycles

a. What is the execution time (in cycles) when the program is executed on a single core? [2 points]

$$(0.2*20 + 0.6*5 + 0.2*10)*10000 = 90000$$

b. Assume you have two separate possible choices to optimize the program:

I. Use a better load-store unit to speed up memory accesses by 4x

II. Use a SIMD unit of width 10 for arithmetic operations

What is the execution time (in cycles) in each case? [2 points]

$$(0.2*20/4 + 0.6*5 + 0.2*10)*10000 = 60000$$

$$(0.2*20 + 0.6*5/10 + 0.2*10)*10000 = 63000$$

c. You are given access to an accelerator containing 1000 arithmetic units. Each arithmetic unit executes arithmetic operations 5x faster. However, you need two new additional “accelerator operations” for each arithmetic operation to use the accelerator. Assume the latency of each accelerator operation is 1 clock cycle, what is the execution time (in cycles)? [2 points]

$$(0.2*20 + 0.6*1/1000 + 0.6*2*1 + 0.2*10)*10000 = 72006$$

## Synchronization and Lock Elision

4. Consider a processor that implements load-linked/store-conditional (LL/SC) functionalities using the following two functions:

```
int ll(int *ptr);
bool sc(int new_val, int *ptr); // returns false if SC fails
```

We provide you with the following `do_lock` function to acquire a lock using the two functions above. Assume that for the `lock`, 0 represents the unlocked state and 1 represents the locked state.

```
Line 1      void do_lock(int *lock) {
Line 2          while (true) {
Line 3              if (ll(lock))
Line 4                  continue;
Line 5              if (sc(1, lock))
Line 6                  break;
Line 7          }
Line 8      }
```

Now consider the case where two cores, i.e., Core0 and Core1, are trying to acquire the same `lock` using the `do_lock` function.

a. What actions are performed by Core0 when it executes the first `if` block (Lines 3 and 4)? [3 points]

Load the latest value of the lock [1 point], set the link register to the address of the lock [1 point], and retry if the lock is already held [1 point].

b. What actions are performed by Core0 when it executes the second `if` block (Lines 5 and 6)? [3 points]

Check if the link register is still set [1 point] and perform the store to hold the lock if it is the case [1 point]. Otherwise retry [1 point].

c. Explain how the `do_lock` function can guarantee that at most one core can hold the lock simultaneously. [4 points]

The first `if` block ensures that a core can only grasp a lock when it is currently unlocked [1 point], while the second `if` block ensures that there is only one core that can bring the lock from the unlocked state to the locked state [1 point]. Even when the two cores are simultaneously updating the lock, due to cache coherence, only one core can perform the store first, which will clear the link register of the other core, making its SC failed without touching the lock [2 points].

**Seat Number:**

5. Consider the following object modification code executing on a multiprocessor with speculative Hardware Lock Elision (HLE) support. Assume that two CPU cores, Core0 and Core1, are running this code in parallel. You may assume that each core has an ROB of eight entries.

```
Modify_Object:
    call    get_obj_addr    # Call get_obj_addr
                                # The return value is
                                # saved in r3
Lock:
    ts      r1, [LCK]        # T&S the global lock
    bnz    Lock             # If lock held, retry
Critical:
    ld      r2, [r3]          # Load object value
    addi   r2, 10            # Modify (addition)
    mult   r2, 5             # Modify (multiplication)
    sub    r2, 2             # Modify (subtraction)
    st      [r3], r2          # Write new value
Unlock:
    st      [LCK], 0          # Free the lock
```

a. Assume that the function `get_object_addr` returns the addresses of two different cache blocks for Core0 and Core1. Does the speculative HLE succeed for both cores? Explain why or why not. [4 points]

Because the critical section does not attempt to write to the same address, speculation for both cores succeeds.

b. Assume that the function `get_object_addr` returns the same address for Core0 and Core1. Does the speculative HLE succeed for both cores? Explain why or why not. [4 points]

Because both cores are attempting to write to the same address, the cores will detect coherence activity for the same memory location, and rollback their state to the `ts` instruction. Thus, the speculation will not succeed.

**Seat Number:**

c. Assume that each core now has an ROB of four entries. Does the speculative HLE succeed in this case? Explain why or why not. [3 points]

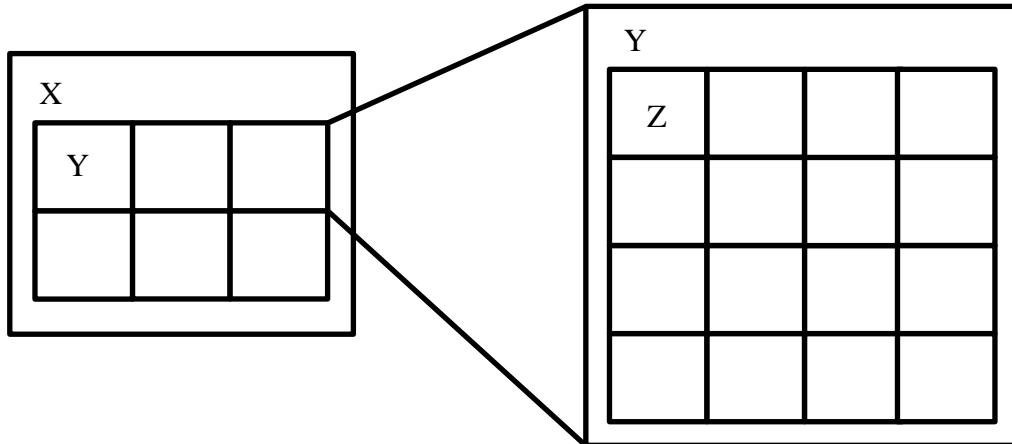
Because the cores need to keep instruction in speculative state in their ROB, the ROB size should be sufficiently large for all instructions in the critical section. Because the ROB size of 4 cannot keep all instructions in the critical section of the given function, the speculation will fail.

d. Explain how processors can ensure forward progress while supporting HLE. [3 points]

When speculation fails multiple times, the cores turn off lock elision and the instruction `ts` is executed as read-modify-write without speculation.

**GPUs**

6. Consider the 2D thread organization in the CUDA programming paradigm as shown below, what are X, Y and Z? [3 points]



**X = Grid, Y = Thread Block, Z = Thread [1 point for each]**

7. For the following vector addition kernel and the corresponding kernel launch code, answer each of the questions below, assuming the code is running on a GPU similar to the ones mentioned in the lectures and n is 10000.

```
1 __global__ void elemWiseMult (float* X, float* Y, float* Z, int n)
2 {
3     int i = threadIdx.x + blockDim.x * blockIdx.x;
4
5     int stride = blockDim.x * gridDim.x;
6     while (i < n) {
7         Z[i] = X[i] + Y[i];
8         i += stride;
9     }
10 }
11
12 int vectMult (float* X, float* Y, float* Z, int n)
13 {
14     // Parameter "n" is the length of arrays X, Y, and Z.
15     int size = n * sizeof (float);
16     float* X_d, Y_d, Z_d;
17     cudaMalloc ((void **) &X_d, size);
18     cudaMalloc ((void **) &Y_d, size);
19     cudaMalloc ((void **) &Z_d, size);
20     cudaMemcpy (X_d, X, size, cudaMemcpyHostToDevice);
21     cudaMemcpy (Y_d, Y, size, cudaMemcpyHostToDevice);
22
23     elemWiseMult<<<8, 1024>>> (X_d, Y_d, Z_d, n);
24     cudaMemcpy (Z, Z_d, size, cudaMemcpyDeviceToHost);
25 }
```

**Seat Number:**

a. How many warps are there in each thread block? [2 points]

$$\frac{1024}{32} = 32$$

b. How many threads will be created for the grid launched in line 23? [2 points]

$$TotalThreads = NumberOfBlocks * BlockSize = 8 * 1024 = 8192$$

c. Is there any control-flow divergence during the execution of the kernel? Explain why or why not. If so, identify the block number(s) and warp number(s) experiencing control-flow divergence. Also, identify the line number(s) at which control-flow diverges for the warp(s) you have identified. [7 points]

Yes.

We have 8,192 threads. All threads will iterate at least one iteration to process 8,192 elements. During the last iteration, only  $10,000 - 8,192 = 1,808$  threads will be active. These threads form 56.5 warps. So, all threads in all the first 56 warps will be active. Warps 0-31 on block 0 have no divergence. The 25th warp in Block 1 will have control divergence. All remaining 199 warps will see all their threads inactive and thus see no control divergence.

[1 point identifying there is divergence;  
2 points for explaining why there is divergence;  
2 points for the correct block and warp number;  
2 points for identifying the correct line at which the divergence occurs]

**Seat Number:**

8. True or false? Briefly explain your reason for each choice. [9 points]

- Cache hierarchies in GPUs are primarily designed to reduce memory access latency.

1 point for each correct choice, and 2 points for each correct explanation.

False. [1 point]

GPUs embrace many-core compute devices which give data-parallel and compute-intensive, or throughput-oriented programs a significant performance benefit compared to CPUs. To run a program on GPU, first the data has to be moved to GPU's memory. This data transfer along with the kernel launch incur a latency overhead which isn't tolerable for latency-critical programs that have tight latency requirements in the order of tens of microseconds to a few milliseconds. However, CPUs have less but more powerful cores and they have fast access to data residing in their caches and memories, making them a better fit for latency-critical programs. [2 points]

- Each thread running on a GPU has its own execution context.

False. [1 point]

On a CPU, a thread has its own execution context: Instruction stream, stack, program counter. However, on a GPU, all threads share an execution context (including stack) and all threads in a program have the same instruction stream.

- Thread divergence between two warps degrades performance.

False [1 point]. Thread divergence within a warp degrades performance because all the threads share the same instruction (SIMT); and thread divergence across warps has no effect on performance. [2 points]

## Multithreaded Processors

9. Consider a 4-way superscalar processor executing two types of workloads A & B repeatedly. Table 1 (in page 13) shows the instruction pattern for each type of workload separately when executed once.

Assume all pipeline slots and consequently, all instructions, have the same functionality. The given scheduling in Table 1 is dictated by dependencies. The naming convention of the instructions in Table 1 is illustrated with this example: 9A (7A) means it is the 9th instruction of a workload of type A and has a dependency on the 7th instruction of the same workload of type A. Idle cycles among instructions are generated by data dependence.

We want to add hardware multithreading to improve performance, measured in total workloads executed per second. Each thread executes one type of workload one after the other. A new workload cannot start executing before the previous workload of the same type finishes. Assume that if the number of instructions that can be executed in a specific clock cycle exceed the number of available pipeline slots, then priority is to be given in ascending order to instructions. Also assume that one clock cycle is 1 ms.

a. Fill in Table 2 and Table 3 (in page 15) to show how the following two scheduling strategies will reschedule instructions:

- I. FGMT with compulsory round robin scheduling at every cycle starting with cycle 0 allocated to the thread executing type A workloads [4 points]
- II. SMT with a 1:1 split in the pipeline slots between the two threads for every cycle (i.e., pipeline slots 0 and 1 are always allocated to the thread executing type A workloads, and pipeline slots 2 and 3 are always allocated to the thread executing type B workloads) [4 points]

b. What is the total number of workloads executed per second in each of the two cases? Explain your solution. [4 points]

FGMT: [2 points]

Approach 1:

There are 1000 cycles in 1 second, 500 each for type A and B workloads  
Type A workloads take 6 cycles to finish and Type B workloads take 4 cycles  
So Workloads per second (WPS) =  $500/6 + 500/4 = 208.3$

Approach 2:

The smallest unit that repeats =  $2 * \text{LCM}(4, 6) = 12 * 2 = 24$  cycles  
So, in 24 cycles, 3 type B workloads finish and 2 type A workloads finish, and this unit of 5 workloads keep repeating, so WPS =  $(5/24) * 1e3 = 208.3$

SMT: [2 points]

Type A and Type B workloads start and finish together in 10 cycles  
So, WPS =  $(2/10) * 1e3 = 200$

Name:

Seat Number:

Sciper:

| Cycle 0 | Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 | Cycle 5 | Cycle 6 | Cycle 7 |
|---------|---------|---------|---------|---------|---------|---------|---------|
|---------|---------|---------|---------|---------|---------|---------|---------|

|                 |
|-----------------|
| Pipeline Slot 0 |
| Pipeline Slot 1 |
| Pipeline Slot 2 |
| Pipeline Slot 3 |

|    |         |         |  |         |  |  |          |
|----|---------|---------|--|---------|--|--|----------|
| 1A | 3A (1A) | 6A (5A) |  | 7A (6A) |  |  | 9A (7A)  |
| 2A | 4A (1A) |         |  | 8A (6A) |  |  | 10A (8A) |
|    | 5A (2A) |         |  |         |  |  | 11A (8A) |
|    |         |         |  |         |  |  |          |

|                 |
|-----------------|
| Pipeline Slot 0 |
| Pipeline Slot 1 |
| Pipeline Slot 2 |
| Pipeline Slot 3 |

|    |  |         |  |         |  |          |
|----|--|---------|--|---------|--|----------|
| 1B |  | 4B (3B) |  | 7B (6B) |  | 8B (7B)  |
| 2B |  | 5B (3B) |  |         |  | 9B (7B)  |
| 3B |  | 6B (3B) |  |         |  | 10B (7B) |
|    |  |         |  |         |  |          |

Time

Table 1: Instruction patterns

**Name:**

**Seat Number:**

**Sciper:**

(This page is intentionally left blank. You can use it as you wish.)

Name:

Seat Number:

Sciper:

| Cycle 0 | Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 | Cycle 5 | Cycle 6 | Cycle 7 | Cycle 8 | Cycle 9 | Cycle 10 |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|----------|
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|----------|

|                 |
|-----------------|
| Pipeline Slot 0 |
| Pipeline Slot 1 |
| Pipeline Slot 2 |
| Pipeline Slot 3 |

|    |    |    |    |    |    |    |     |  |    |     |
|----|----|----|----|----|----|----|-----|--|----|-----|
| 1A | 1B | 3A | 4B | 6A | 7B | 7A | 8B  |  | 1B | 9A  |
| 2A | 2B | 4A | 5B |    |    | 8A | 9B  |  | 2B | 10A |
|    | 3B | 5A | 6B |    |    |    | 10B |  | 3B | 11A |
|    |    |    |    |    |    |    |     |  |    |     |

Table 2: FGMT scheduling.

[-1 for an incorrect cycle, -1 for not following constraints, -0.5 for not showing repeated instructions, -4 for incorrect threading model]

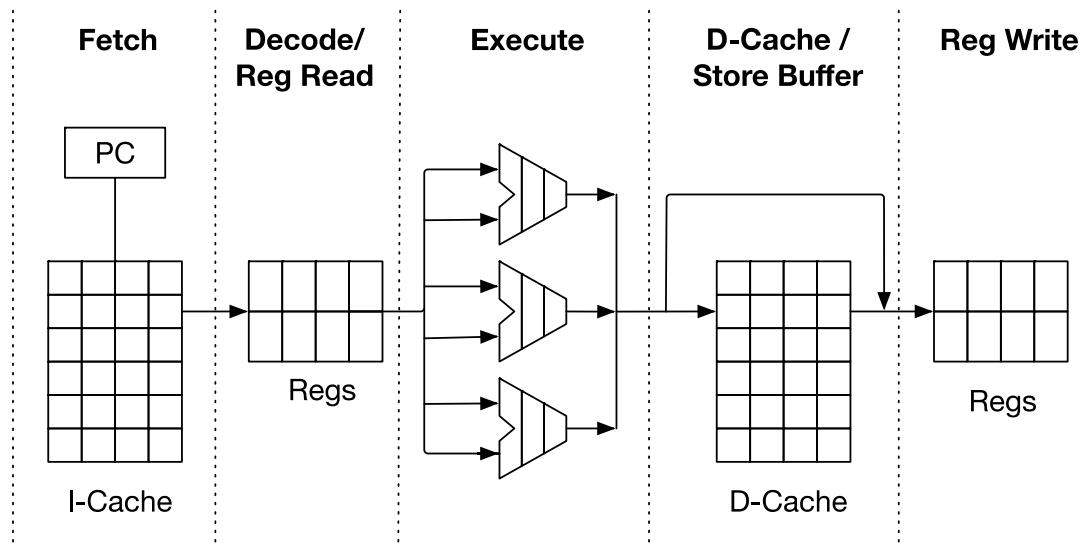
| Cycle 0 | Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 | Cycle 5 | Cycle 6 | Cycle 7 | Cycle 8 | Cycle 9 | Cycle 10 |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|----------|
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|----------|

|                 |
|-----------------|
| Pipeline Slot 0 |
| Pipeline Slot 1 |
| Pipeline Slot 2 |
| Pipeline Slot 3 |

|    |    |    |    |    |    |    |  |     |     |    |
|----|----|----|----|----|----|----|--|-----|-----|----|
| 1A | 3A | 5A | 6A |    | 7A |    |  | 9A  | 11A | 1A |
| 2A | 4A |    |    |    | 8A |    |  | 10A |     | 2A |
| 1B | 3B |    | 4B | 6B |    | 7B |  | 8B  | 10B | 1B |
| 2B |    |    | 5B |    |    |    |  | 9B  |     | 2B |

Table 3: SMT scheduling

10. Consider the single-threaded processor pipeline shown in the figure below.



a. What components **must** be replicated to support fine-grained multithreading correctly? Explain. [4 points]

The register file and the PCs must be replicated to support FGMT. Both these structures hold a thread context and cannot be shared. [2 point for reg file, 2 points for PC, each has 1 point for mentioning and 1 point for the explanation]

b. What components **should** be augmented to efficiently support fine-grained multithreading? Explain. [4 points]

The D-cache and I-caches should be augmented to support FGMT. Running multiple threads on a processor can cause contention for cache space. Thus, an FGMT processor would ideally add more cache to accommodate the working-sets of multiple threads. [2 points for D-, 2 points for I-cache, each has 1 point for mentioning and 1 point for the explanation]

**Seat Number:**

11. A superscalar processor runs at the clock frequency of 2 GHz with an average cache miss latency of 20 ns. The processor implements Coarse-Grained Multithreading (CGMT) with the policy to switch threads on each cache miss. The overhead of the switching is 10 cycles. Assume that the pipeline depth refers to the number of stages between the fetch and the retire stage.

a. Given the above information, what is the upper bound of the pipeline depth for CGMT to provide a performance benefit? [3 points]

$$\begin{aligned} \text{Thread Switch + Pipeline Fill Time} &< \text{Blocking Latency} \\ 10 \text{ cycles + Pipeline Depth} &< 20 \text{ ns} * 2 \text{ GHz} \\ \text{Pipeline Depth} &< 30 \text{ stages} \end{aligned}$$

[2 points: getting blocking latency equation right]  
[1 point: getting the math right]

b. What happens if the processor uses a pipeline deeper than the value calculated in the previous part? [3 points]

If the processor uses a pipeline deeper than the calculated value, then the processor won't be able to hide the latency of the stall events [2 points], leading to overall lower throughput [1 points].

c. An alternative architecture of the processor has a pipeline depth of 10 with a reduced average cache miss latency of 12 ns and reduced clock frequency of 1.5 GHz. Is CGMT still a good choice? Explain. [3 points]

$$\begin{aligned} \text{Thread Switch} &= 10 \text{ cycles} \\ \text{Pipeline Fill Time} &= 10 \text{ cycles} \\ \text{Blocking Latency} &= 12 \text{ ns} * 1.5 \text{ GHz} = 18 \text{ cycles} \end{aligned}$$

Thread Switch + Pipeline Fill Time (20) is larger than Blocking Latency (18), hence CGMT is not worth it.

[2 points: getting the equation and the math right]  
[1 point: final conclusion]

**Seat Number:**

(This page is intentionally left blank. You can use it as you wish.)