# EECS 470 *Final Exam*

*Fall 2025*

Name: _____Solution_____ Uniqname: _____

Sign the honor code:

I have neither given nor received aid on this exam nor observed anyone else doing so.

_____

---

| # | Points |
|---|---|
| 1 | **/ 18** |
| 2 | **/ 20** |
| 3 | **/ 24** |
| 4 | **/ 18** |
| 5 | **/ 15** |
| 6 | **/ 5** |
| ***Total*** | **/100** |

# NOTES:

- Closed book and one 8.5x11 page of notes
- Calculators are allowed, but no smart watches, cell phones, earbuds, etc.
- Don't spend too much time on any one problem.
- You have about 120 minutes for this exam.
- There are **11** pages including this one.
- **Be sure to show your work and explain what you've done where appropriate.**

**1) Short Answer - Fill in the correct bubbles or blank [18 Points]:**

1.1) The MSI coherence protocol can suffer from frequent upgrade request broadcasts from a sole sharer attempting to gain the write permission. What simple solution alleviates the need for the upgrading processor to send a message? [2 points]

| **Add Owner state** | **Add Exclusive state** | **Add transient state $SM^A$** | **Use a directory over a bus system** |
|---|---|---|---|
| ◯ | ◯ | ◯ | ◯ |

In multiprocessor memory systems, select the best one: [4 points]

1.2) to be **consistent**, the memory system must **(A ◯ / B ◯ / C ◯ / D ◯)**

1.3) to be **coherent**, the memory system must **(A ◯ / B ◯ / C ◯ / D ◯)**

**Options:**

**A)** never re-order loads with respect to stores

**B)** always have constant global order of thread execution

**C)** maintain predictable interleaved order of all memory transactions

**D)** make all threads observe writes to each individual address in the same order

1.4) VIPT caches can have synonym and homonym problems when the cache is too large. Is this true for fully-associative caches? [2 points]

**Yes** ◯ **No** ◯

1.5) If a GShare branch predictor uses 6 bits of history with 2 bit hysteresis, what would be the size of the pattern history table in **bits**? [2 points]

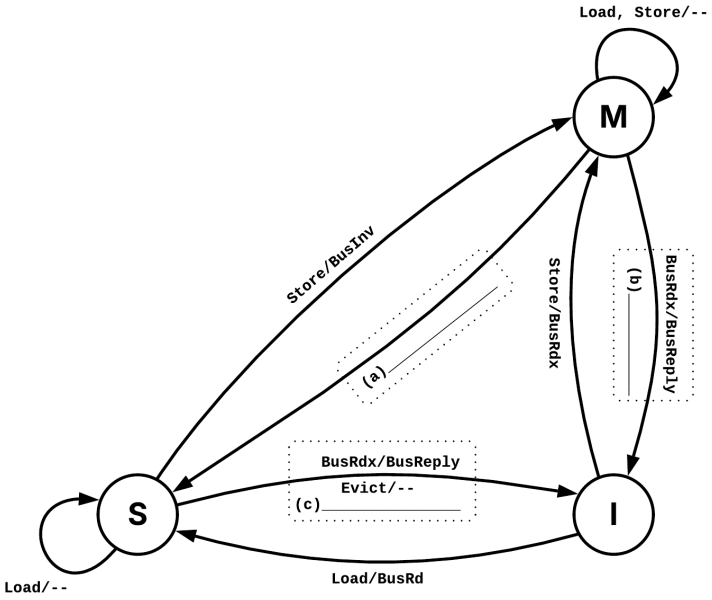
Ans: _2^7*2 = 128_____

1.6) Which kind(s) of caches need to be flushed to memory if the OS switches context between processes? Assume tag bits do not include the process/thread/task ID. [2 points]

◯ **Virtually Indexed Virtually Tagged**
◯ **Virtually Indexed Physically Tagged**
◯ **Physically Indexed Physically Tagged**
◯ **None of the above**

The following diagram depicts the MSI coherence protocol, but is missing the classification for three (3) transition edges. Out of the listed options, match the correct transition to the respective edge. Recall, in a transition there is <action> / <response>, e.g. a **Load** in the **Invalid** state invokes a **BusRd**. A box has been drawn around the missing transitions and its respective arrow for clarity. [6 points]

**E) BusRd/BusReply  F) BusRdx/BusWb  G) BusInv/--  H) Evict/BusWb  I) Evict/--**




1.7) Edge (a) **E** ◯ / **F** ◯ / **G** ◯ / **H** ◯ / **I** ◯

1.8) Edge (b) **E** ◯ / **F** ◯ / **G** ◯ / **H** ◯ / **I** ◯

1.9) Edge (c) **E** ◯ / **F** ◯ / **G** ◯ / **H** ◯ / **I** ◯

**2. Cache Money [20 points]**

A design team at the up-and-coming chip company AAPPLE (Ann Arbor's Performant Processors and Legendary Electronics) is deciding between various different designs and configurations for their new uniprocessor's data cache.

Consider each set of options below individually while keeping the cache size the same. For each section, determine which design decision is better for the given metric and explain why. If you can see benefits to both options listed, include explanations for both.

**Design Decision 1:** 16-byte blocks vs 32-byte blocks

2.1) Miss rate [2 points]

16 byte because there are fewer conflicts OR
32 byte because more spatial locality

2.2) Efficient bandwidth utilization [2 points]

16 byte because smaller blocks mean less potentially useless data brought in, also writebacks are smaller on eviction if you only update part of a line

2.3) Overall storage overhead [2 points]

32-byte because fewer blocks so fewer instances of tag + valid (+dirty?) bits. Note that each individual tag is the same size (1 has 1 more BO bit, the other has 1 more Index bit), so saying there are more/less bits per tag is incorrect.

**Design Decision 2:** 2-way vs 4-way set associative

2.4) Miss rate [2 points]

4-way because more possible locations for a block to be in

2.5) Hit time [2 points]

2-way maybe, because tag is shorter and mux is smaller, so can be faster. Note, number (as opposed to size) of tags compared is not a valid reason itself (unless it references the muxing, as the comparisons are done in parallel)

4-way maybe, because SRAM array is smaller per way, so faster lookup

In real systems, the answer is 4-way, as the smaller SRAM array is usually a much bigger gain than the extra muxing.

2.6) Overall storage overhead [2 points]

2-way because more bits needed for set indexing so fewer stored as tags

**Design Decision 3:** Write-through vs write-back

2.7) Efficient bandwidth utilization [2 points]

Write-back because only utilizing bus when evicting dirty blocks

2.8) Overall storage overhead [2 points]

Write-through because no need to include dirty bits

**Design Decision 4:** Inclusive vs Non-Inclusive L2

2.9) L1 miss rates [2 points]

Non-inclusive because not forced to back-invalidate for inclusion

2.10) Effective bandwidth utilization [2 points]

Non-inclusive because not forced to back-invalidate for inclusion OR
inclusive because reducing calls to memory because L1 evictions are still in L2, or in a multi-core you don't need to send snoops to the L1 if not in the inclusive L2

**3) Load-Store Queues [24 Points]**

As described in lecture, in a system with a non-speculative Load-Store Queue that implements store-load forwarding, the earliest a Store was allowed to be sent to the memory was when it was at the head of the ROB and retired, and the earliest a Load could complete was when all older stores knew their addresses.

3.1) Why are stores allowed to leave the store queue only upon their retirement in the ROB? [2 points]

This guarantees that the store can not be asked to roll-back, so it is safe at this point to write the result into the memory. Note the in-order nature of the SQ prevents re-ordering stores. Waiting for the ROB is to prevent percise state violations if we need to rollback.

3.2) Why are loads only allowed to complete when all older stores knew their address? [2 points]

Without the address of the older store, the store could write anywhere in memory, including the address from which the load reads.

Willy Cocoa thinks this logic can be changed, while still being non-speculative, to increase throughput. Willy believes he can improve either the loads, stores, or both? Is Willy right? If so, how, and in which cases would it increase throughput, or why can't it be improved?

3.3) What can it improve: **Only Stores ◯ / Only Loads ◯ / Both ◯ / Neither ◯**
[2 points]

3.4) Explanation for Stores (how or why not) [4 points]:

Instead of waiting for the ROB to retire, each store in the SQ can be sent to the memory if it is guaranteed not to be on a speculative path (nothing older than it in the ROB can cause an exception)

3.5) Explanation for Loads (how or why not) [4 points]:

If a recent store (or set of stores) know their destination addresses overlap with all of the source addresses for the load with no ambiguous stores in between them and the load, then the load doesn't need to wait for the older ambiguous stores as it is guaranteed to not use those values.

Now, consider the following Loads, Store Queue, and Memory states for the following question. Assume Load-Store forwarding exists at the byte level. Store Queue Index (SQI) is the value of the store queue tail during the load's dispatch. Recall, store tail is the writer pointing to an empty (invalid) entry. For clarity, SQI of 1 has been underlined along with an arrow of where it fits in the store queue order. The memory system is little endian.

**Loads**

| Load Size | Address | Dest Reg | SQI |
|---|---|---|---|
| word | 0x100 | R1 | *1* |
| byte | 0x0FF | R5 | 5 |
| half | 0x0FE | R31 | 3 |
| half | 0x102 | R2 | 2 |
| word | 0x100 | R8 | 4 |

**Store Queue**

| Index | Store Size | Address | Store Data |
|---|---|---|---|
| 0 | byte | 0x101 | 0xBA |
| *1* | *word* | *0x100* | *0xBEEFE470* |
| 2 | byte | 0x0FF | 0x91 |
| 3 | half | 0x102 | 0xE498 |
| 4 | byte | 0x102 | 0x72 |
| 5 | -------- | -------- | |

**Main Memory**

| Base Address | byte 3 | byte 2 | byte 1 | byte 0 |
|---|---|---|---|---|
| **0x0FC** | 0xEE | 0xC5 | 0x47 | 0x00 |
| **0x100** | 0xBA | 0x5E | 0x33 | 0x11 |

**3.6)** Assume all stores have executed and are ready to leave the store queue, and the head in the store queue is pointing to index 0 and the tail is pointing to index 5. Assume all entries in the load queue are valid and are waiting to be issued. After all loads and stores shown execute and retire, what is the final state of memory and listed registers? [10 points]

**Word Addressable Main Memory (Final State)**

| Base Address | byte 3 | byte 2 | byte 1 | byte 0 |
|---|---|---|---|---|
| **0x0FC** | 0x91 | 0xC5 | 0x47 | 0x00 |
| **0x100** | 0xE4 | 0x72 | 0xE4 | 0x70 |

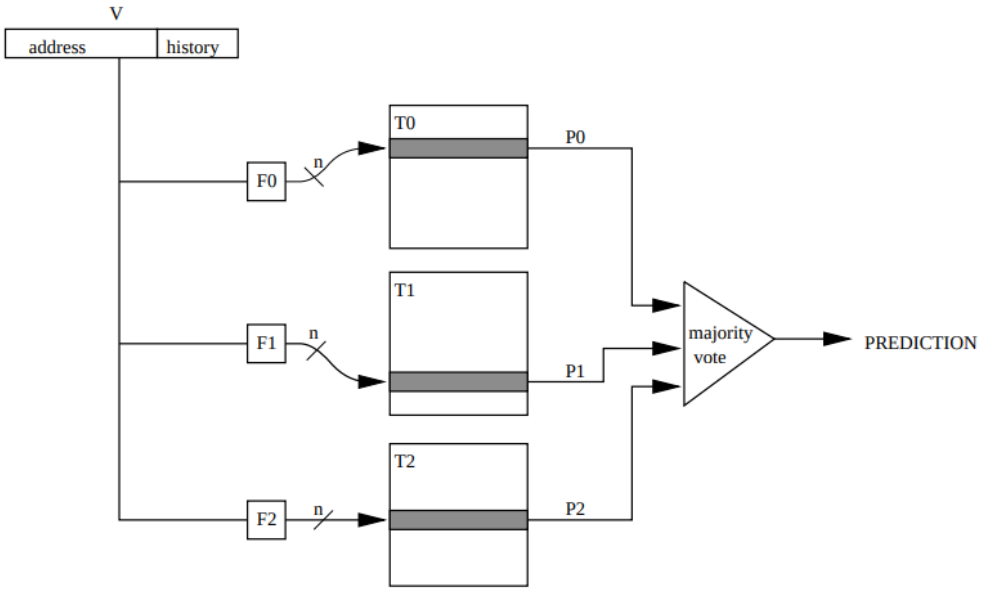
**Registers (Final State)**

| Register | 4-hex value |
|---|---|
| R1 | 0xBA5EBA11 |
| R2 | 0xBEEF |
| R5 | 0x91 |
| R8 | 0xE498E470 |
| R31 | 0x91C5 |

**4) Aliasing Hertz [18 Points]**

A key problem in a gshare predictor is the potential for destructive aliasing between different branches. Aliasing occurs when two history-PC pairs result in the same entry of the PHT when XORed. This causes two different branches to interfere in their predictions while the program is executing and can significantly impact the prediction accuracy.

To mitigate this problem Michaud, Seznec, et. al. introduced Skewed Branch Predictors in 1996. Their proposed predictor *gskewed* (shown below) uses 3 different independent hash functions, on a combination of the PC and the current global history, to index into 3 PHTs and produce 3 predictions. The final prediction is obtained using a majority vote between the outputs of the PHTs.




4.1) Briefly explain why this mechanism can reduce the effect of the aliasing as compared to a gshare predictor. [2 points]

Having three independent hash functions indexing into three tables reduces the overall probability that aliasing causes an incorrect prediction. A PC-history combination would have to alias in at least two of the tables with other branches having different predictions in order for the overall prediction to be made incorrect. This probability is lower compared to aliasing in a single gshare table.

4.2) They also specify a *partial update* policy for the predictor, which means that on a correct overall prediction, a PHT that produced an incorrect prediction is *not* updated. On a misprediction, all 3 tables are updated. Why might this be beneficial? [4 points]

Having a partial update policy is beneficial as it prevents two aliasing branches from repeatedly overwriting each other's predictions. If one of the tables gives an incorrect prediction for a branch due to an aliasing conflict but the other two are correct, *not* updating the incorrect table allows it to act as a correct prediction for the alias.

4.3) Let F0, F1 and F2 be the three hash functions being used where each one outputs *a* 2-bit value as an index into the (tiny) PHTs. Each entry of the PHTs is a 1 bit value initialized to N (not taken). Below is a list of hash values for different PC-history combinations.

| Branch PC | History | F0 | F1 | F2 |
|---|---|---|---|---|
| 0x20 | NNT | 1 | 2 | 0 |
| 0x20 | NNN | 3 | 1 | 1 |
| 0x1C | NTN | 2 | 2 | 3 |
| 0x48 | NNN | 0 | 2 | 1 |
| 0x52 | TNT | 0 | 1 | 3 |

The 3-bit global history is initialized to NNN. Given the following sequence of branches and their directions after resolution, assuming the partial update policy is used, fill out the state of the three PHTs below [10 points]:

```
0x48 -> T   history: NNN -> mispredict, update T0[0], T1[2], T2[1]
0x20 -> N   history: NNT -> correct but don't update T1[2]
0x1C -> T   history: NTN -> mispredict, update T0[2], T1[2], T2[3]
0x52 -> N   history: TNT -> mispredict, update T0[0], T1[1], T2[3]
0x1C -> T   history: NTN -> correct, don't update T2[3]
```

| T0 | T1 | T2 |
|---|---|---|
| N | N | N |
| N | N | T |
| T | T | N |
| N | N | N |

4.4) What is the prediction obtained at this point if the branch at PC 0x52 is encountered? [2 points]

**Prediction: ____N_____**

**5) Multiprocessor Tradeoffs [15 points]**

Consider two cache-coherent shared-memory multiprocessor designs:

System A: Snooping protocol on a shared bus

System B: Directory-based protocol over a point-to-point interconnect

For each of the following scenarios, fill in the appropriate bubble (Bus or Directory) to indicate which system is better suited (or more likely to perform better), assuming both are otherwise well-designed. Provide a **very short** one sentence explanation for each choice.

Supporting hundreds of cores on a single chip

5.1) **Bus** ◯ / **Directory** ◯ [1 point]

5.2) Reason [2 points]:

A bus suffers from bandwidth issues b/c as # of cores scales, contention and arbitration become unmanageable

Minimizing coherence spam traffic for a read-mostly shared data structure

5.3) **Bus** ◯ / **Directory** ◯ [1 point]

5.4) Reason [2 points]:

n a bus system we must broadcast all coherence events to every core, in a directory system traffic is restricted to the relevant sharers

Achieving lowest coherence latency in a very small (e.g., 4-core) multiprocessor

5.5) **Bus** ◯ / **Directory** ◯ [1 point]

5.6) Reason [2 points]:

Bus is just fine for a small system, very simple and fast

Simplifying the implementation and debugging of the coherence mechanism

5.7) **Bus** ◯ / **Directory** ◯ [1 point]

5.8) Reason [2 points]:

On a bus we have one shared medium that you can “look at” for all coherence information, directory protocols have a higher complexity and design overhead

Minimizing overall coherence-related power in a large multicore system

5.9) **Bus** ◯ / **Directory** ◯ [1 point]

5.10) Reason [2 points]:

For a large system, the ability to unicast/multicast rather than broadcast to all cores prevents a lot of unnecessary snooping and tag comparison etc.

**6) Vectorizing Code [5 Points]**

Given the three programs below, which would benefit from running on a vector machine? Select all that apply [2 points]:

6.1) **Code A** ◯ / **Code B** ◯ / **Code C** ◯

```
Code A: void saxpy(int N, float a, float *x, float *y) {
           for (int i = 0; i < N; i++) {
              y[i] = a * x[i] + y[i];
           }
        }

Code B: int very_important_computation(int N, const int *table) {
           unsigned int x = 1;
           int sum = 0;

           for (int i = 0; i < N; i++) {
                x = x * 1664525u + 1013904223u; // very important nums
                if (x & 1u) sum += table[(x >>  0) & 1023];
                else        sum -= table[(x >> 10) & 1023];
           }
           return sum; // very important result
        }

Code C: void agi(int M, int N, int K, const float *A,
                 const float *B, float *C) {
           for (int i = 0; i < M; i++) {
              for (int j = 0; j < N; j++) {
                 float sum = 0.0f;
                 for (int k = 0; k < K; k++) {
                    sum += A[i*K + k] * B[k*N + j]; // generate $5T val
                 }
                 C[i*N + j] = sum;
              }
           }
        }
```

6.2) For those cases **Not Selected,** explain why? [3 points]

This program has a sequential dependency chain through x and is also rather branchy. There is no obvious DLP/TLP, so a regular superscalar OOO machine is probably best to exploit whatever ILP we do have