CS152/252A
Fall 2026

Memory Hierarchy
Discussion 3 Solutions

1 Exceptions

Exceptions are unexpected events that require jumping to an OS handler. Sources include page faults, illegal instructions, divide by zero, etc.

1.1 Define precise exceptions.

All instructions prior to the faulting instruction in program order have completed/committed, and none of the instructions after (and including) the faulting instruction appear to have started.

1.2 Why are precise exceptions useful?

- Restartable: fix the fault, then resume
- Architectural state matches programmer's model of sequential execution
- Deterministic debugging: you know exactly which instruction faulted
- Clean restartability without exposing microarchitectural state

1.3 Why might one not want to always implement precise exceptions?

- Introduces hardware complexity
- Not always needed: some faults simply kill the program, so there is nothing to recover from

1.4 Describe how program execution time may be affected by adding support for precise exceptions. Recall, execution time = instructions/program * cycles/instruction * time/cycle.

Instructions/Program: Decrease. Without hardware support, you may need more instructions to restore microarchitectural state, so by adding hardware support, you remove this overhead and thus decrease the number of instructions needed

Cycles/Instruction: None[1] (precise exception information carried in parallel)

Time/Cycle: Increase (increased hardware complexity may be on critical path)

1.5 Compare and contrast precise exception handling with branch misprediction.

Exception: update Cause and EPC registers, flush pipeline, inject handler PC into Fetch stage

[1] You can also argue it decreases because it takes extra cycles to restore architectural state after resolving an exception. It's hard to say without making assumptions about the underlying hardware.

Branch mispredict: kill instructions in F and D phases and use calculated target PC

1.6 (Spring 25 MT1, Q1) Answer True or False, and give a 2-3 sentence explanation.

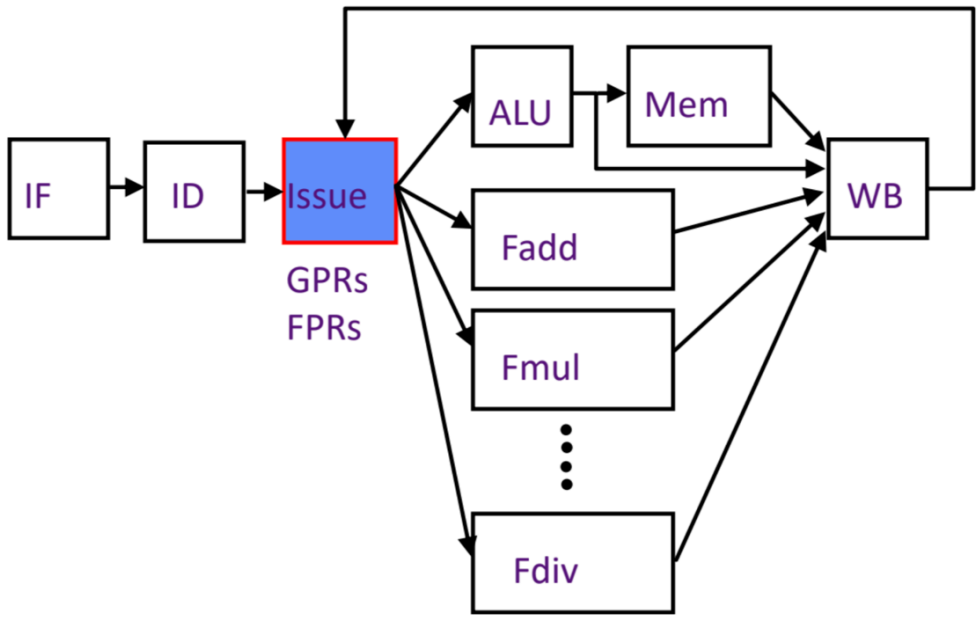
(a) It is possible to enable precise exceptions for stores (whose commit point is the memory stage).

True. If a store instruction raises an exception, don't commit it in the memory stage; instead raise a flag. Once the flag reaches the commit point, allow the instruction in WB to commit, then flush everything that follows the store.

(b) Suppose that instead of waiting for the faulting instruction to reach its commit point, you flush the pipeline and jump to the trap handler the cycle after the exception is detected. This still enables precise exceptions.

False. This change could cause instructions prior to the faulting instruction to be flushed before they commit, so the "everything before has committed" requirement is violated and precise exceptions are no longer supported.

2 More Complex Pipelines



The Issue stage reads operands from the register files, checks hazards, and routes each instruction to the right functional unit (FU). Issue and completion could be out-of-order! Note that *completion* means the result exists and other instructions can use it; *commit* is different and will be made clearer later in the course.

2.1 What data hazards could exist in this pipeline? What mechanism is used to prevent them?

RAW, WAR, and WAW (all of them!). A scoreboard is used to keep track of hazards and holds instructions in Issue to prevent them.

2.2 What are the structural hazards if we have unpipelined multi-cycle FUs vs. pipelined FUs with varying latencies? Assume we can only issue one instruction per cycle.

Unpipelined multi-cycle FUs: the FUs themselves are a structural hazard (a busy FU cannot accept a new instruction). WB can also be a structural hazard if two FUs have outputs ready on the same cycle.

Pipelined FUs: WB is the only structural hazard. Two FUs could have outputs ready on the same cycle given varying latencies. Again, we assume one issue per cycle.

3 Cache Organization

3.1 Why caches?

- Low memory latency accesses (fewer cycles)
- High bandwidth accesses (can deliver multiple bytes per cycle)
- Lower energy cost per access (don't need to go out to DRAM)
- Can be fully hardware managed / transparent to programmer: the cache automatically decides what to keep, evict, and fetch

3.2 Define and give an example of a program that exhibits the following:

(a) Temporal locality:

Recently accessed locations are more likely to be accessed again. Example: repeatedly adding to a counter / running sum, e.g. `sum` in `for (i = 0; i < N; i++) { sum += array[i]; }`

(b) Spatial locality:

Locations near recently accessed locations are more likely to be accessed. Example: looping through an array, e.g. `array[i]` in `for (i = 0; i < N; i++) { sum += array[i]; }`

3.3 Consider a 1 KiB 4-way set-associative cache with 32-byte cache lines. The address is 12 bits wide. How are the address bits partitioned?

Tag: 4 bits, addr[11:8] (12 bits - 3 index bits - 5 offset bits = 4 bits)

Index: 3 bits, addr[7:5] (2^{10} bytes / (2^5 bytes per line * 2^2 ways) = 2^3 sets)

Offset: 5 bits, addr[4:0] (32-byte cache lines = 2^5 bytes)

4 Replacement Policies

4.1 What are some replacement policies? Why is random replacement not as bad as it sounds?

- Random
- Least-Recently Used (LRU)
- Not-Most-Recently Used (NMRU)
- Pseudo Least-Recently Used (PLRU)
- First-In First-Out (FIFO)
- Plus many more possibilities…

Some deterministic policies (like FIFO or PLRU) can be gamed by certain access patterns. Random doesn't consistently evict the same "unlucky" line and breaks regular worst-case patterns, which makes performance more stable across workloads.

Suppose we see the following stream of accesses where `A`, `B`, `C`, `D`, and `E` represent unique addresses from different lines that all map to the same set:

```
A, B, C, D, B, A, E
```

Assume the cache has four ways and all lines in the set are initially invalid. When address `E` is accessed, one of the cache lines must be evicted. For each replacement policy, which line gets evicted?

4.2 First In, First Out (FIFO)

State: the next line to replace. On a replacement, increment the state. Assume the state starts at 0.

Access	Way0	Way1	Way2	Way3	Policy State After Request
A	A				1
B		B			2
C			C		3
D				D	0
B		hit			0
A	hit				0
E	E				1

`A` is evicted.

4.3 Not Most Recently Used (NMRU)

Almost the same as FIFO. State: the next line to replace. On a replacement *or* a hit to the line-to-be-replaced, increment the state. Assume the state starts at 0.

Access	Way0	Way1	Way2	Way3	Policy State After Request
A	A				1
B		B			2
C			C		3
D				D	0
B		hit			0
A	hit				1
E		E			2

B is evicted.

4.4 Least Recently Used (LRU)

State: ways ordered from most recently used to least recently used.

Access	Way0	Way1	Way2	Way3	Policy State After Request
A	A				[0,3,2,1]
B		B			[1,0,3,2]
C			C		[2,1,0,3]
D				D	[3,2,1,0]
B		hit			[1,3,2,0]
A	hit				[0,1,3,2]
E			E		[2,0,1,3]

C is evicted.

4.5 Pseudo-LRU (PLRU)

State: a tree of bits (root bit, then the two child bits) pointing toward the pseudo-least-recently-used way.

Access	Way0	Way1	Way2	Way3	Policy State After Request
A	A				1(1)(0)
B		B			0(1)(1)
C			C		1(0)(1)
D				D	0(0)(0)
B		hit			0(0)(1)
A	hit				1(1)(1)
E		E			0(1)(0)

B is evicted.

5 Causes of Cache Misses

Recall: Average memory access time (AMAT) = Hit time + Miss rate * Miss penalty

5.1 Define the following terms:

(a) Compulsory Misses

First reference to a line. Will always happen
To improve: increase line size (but increased conflict misses and miss penalty)

(a) Capacity Misses

Cache is too small, so requested line was evicted
To improve: increase cache size (but increased hit time)

(a) Conflict Misses

Occur due to placement policy (would not occur in a fully associative cache)
To improve: increase cache size and increase associativity (but increased hit time)

6 Cache Write Policies

6.1 Describe the cache write policies on:

(a) Cache Hit:

Write-through - write to $ + memory
Write-back - write to $ only, evictions spawn writes

(b) Cache Miss:

No-write-allocate - write to memory and do not fetch into $
Write-allocate - fetch into $ and then write

6.2 What combinations of these policies make sense?

Write-through + no-write-allocate (memory always has the most up-to-date data)
Write-back + write-allocate (cache always has the most up-to-date data)

6.3 What issues arise when multiple CPUs access the same shared memory?

Writes may not be visible to other CPUs causing them to read stale data - you'll learn lots more about dealing with this in the Cache Coherency lectures!

7 Cache Optimizations

Recall: Average memory access time (AMAT) = Hit time + Miss rate * Miss penalty

For each technique, indicate whether implementing it will increase, decrease, or have no change on each aspect.

Technique	Hit Time	Miss Penalty	Miss Rate	Hardware Complexity
Smaller, simpler caches	Decrease	No Change	Increase	Decrease
Multi-level caches	No Change[2]	Decrease	No Change	Increase
Smart replacement policy	May Increase[3]	No Change	Decrease	Increase
Pipelined writes	Decrease	No Change	No Change	Increase
Write buffer	No Change	Decrease	No Change	Increase
Sub-blocks (sector cache)	No Change	Decrease	Increase	Increase

Smaller, simpler caches
- Hit Time: Decrease → smaller cache is faster to access
- Miss Penalty: No change → main memory access time unchanged
- Miss Rate: Increase → small cache holds fewer blocks → more misses
- Hardware Complexity: Decrease → simpler design, less circuitry

Multi-level caches (L1, L2, L3)
- Hit Time: No change → no change to the shape of L1
- Miss Penalty: Decrease → if L1 misses but L2 hits, don't go to main memory
- Miss Rate: No change → no changes to the L1
- Hardware Complexity: Increase → need logic for multiple caches, coherence, replacement policies
- Intuition: hierarchy reduces penalty without slowing L1

Smart replacement policy
- Hit Time: May increase → slightly more logic to find which line to replace
- Miss Penalty: No change → memory access time unchanged
- Miss Rate: Decrease → smarter replacement → fewer unnecessary evictions
- Hardware Complexity: Increase → need extra tracking bits
- Intuition: smarter eviction = fewer misses, but hardware cost rises

Pipelined writes
- Hit Time: Decrease → pipeline lets CPU continue after issuing write

[2] The no change in this question refers specifically to the L1$ hit time. The overall hit time of memory would Decrease.
[3] A smarter policy may require more hardware that delays a memory access, but this depends on the implementation

- Miss Penalty: No change → memory access still takes same time
- Miss Rate: No change → doesn't affect what is in cache
- Hardware Complexity: Increase → pipelining adds control logic
- Intuition: hides write latency → faster effective hit time

Write buffer

- Hit Time: No change → CPU can still access cache normally
- Miss Penalty: Decrease → writes buffered → CPU doesn't stall on write
- Miss Rate: No change → cache content not changed
- Hardware Complexity: Increase → need buffer to track pending writes

Sub-blocks (sector cache)

- Hit Time: No change → accessing sub-block in cache about same speed
- Miss Penalty: Decrease → only fetch smaller block, not full line → less memory traffic
- Miss Rate: Increase → smaller blocks = higher chance of compulsory misses
- Hardware Complexity: Increase → need extra tags per sub-block