CS152/252A
Spring 2026

Memory Hierarchy
Discussion 3 Solutions

1 Exceptions

1.1 Define precise exceptions.

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

1.2 Why are precise exceptions useful?

- Architectural state matches programmer's model of sequential execution
- Deterministic debugging
- Clean restartability without exposing microarchitectural state

1.3 Why might one not want to implement precise exceptions?

- Introduces hardware complexity
- Specific application does not need to or cannot recover from fatal exceptions

1.4 Describe how program execution time may be affected by adding support for precise exceptions.

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)

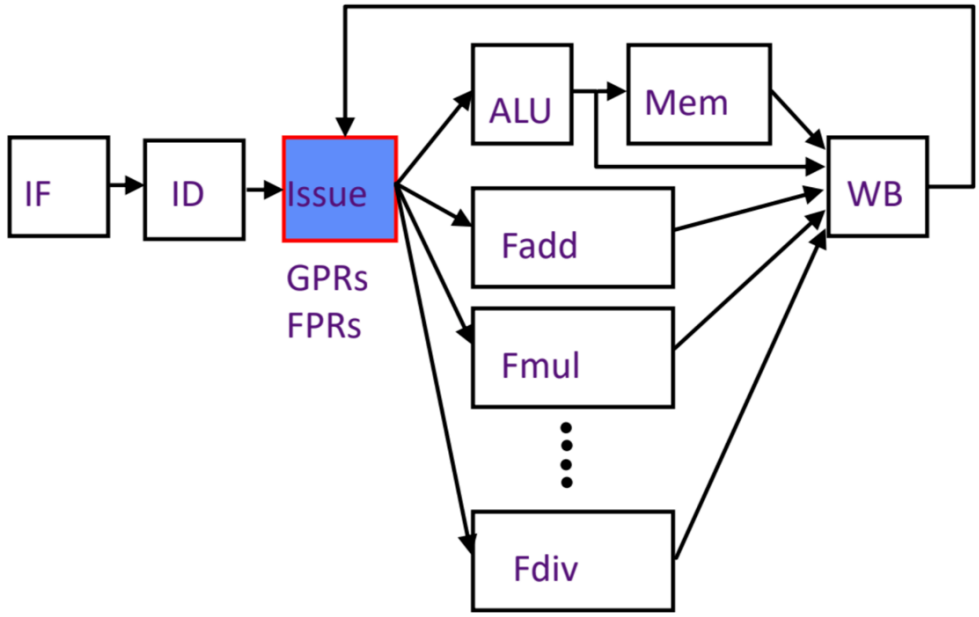
1.5 Compare and contrast precise exception handling with branch misprediction

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

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

[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.

2 More Complex Pipelines



2.1 What additional hazards are introduced by this pipeline?

Unpipelined FU's (FAdd, FMul, FDiv…) cause hazards in E/WB (structural/data)

2.2 How are precise exceptions enforced in this pipeline?

Stall everything into commit point (End of M)

3 Cache Organization

3.1 Why caches?

- Low latency accesses
- High bandwidth accesses
- Lower energy cost per access
- Can be fully hardware managed / transparent to programmer

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. (Adding to counter, Matmul...)

(b) Spatial locality:

Locations near recently accessed locations are more likely to be accessed (Looping through an array)

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 - (10 bits for $ - 2 bits of associativity) = 8 bits)

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

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

4 Replacement Policies

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?

4.1 First In, First Out (FIFO)

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

4.2 Not Most Recently Used (NMRU)

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

4.3 Least Recently Used (LRU)

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]

4.4 Pseudo-LRU (PLRU)

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)

5 Causes of Cache Misses

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 / Write-back + write-allocate

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

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	Decrease	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
Code optimization	No Change	No Change	Decrease	No Change
Compiler prefetching	No Change	No Change	Decrease	No Change
Hardware prefetching (stream buffer)	No Change	Increase	Decrease	Increase
Victim cache	No Change	Decrease	No Change	Increase

[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