ECE 2300
Digital Logic & Computer Organization

Spring 2025

Performance Tradeoffs

Cornell University

Announcements

- HW 7 due tomorrow
- HW 8 will be released today
  - Some questions tie to lectures next week
- Lab 4c due Monday
  - Download the updated zip file from CMS (see Ed post #266)
- Lab 5 will be released tomorrow

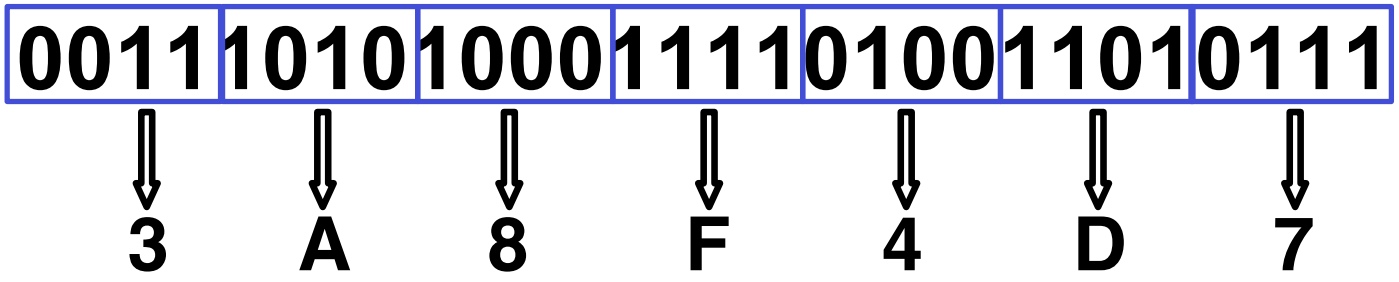
Hexadecimal Notation (used in HW 7&8)

- **Often convenient to write binary (base-2) numbers as hexadecimal (base-16) numbers**
  - **Fewer digits: 4 bits per hex digit**
  - **Less error prone: easy to misread long string of 1's and 0's (such as memory address)**

Binary	Hex	Decimal	Binary	Hex	Decimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

Converting from Binary to Hex

- Every group of four bits is a hex digit
  - Start grouping from right-hand side



We use Hex as a compact way to write binary numbers, not as a new data representation in hardware

Cache Basics (True or False)

- A cold miss is bound to occur on the first access to a memory address
- Fully associative cache does not incur conflict misses
- A capacity miss may occur even when some of the cache blocks are not occupied
- LRU is an optimal block replacement policy

Review: Another LRU Replacement Example

- **Fully associative**

(X) = LRU Age; 2 age bits per block in this case*
**Age saturates* at 3 (an approximation to reduce hardware complexity; multiple blocks can have the same age)

Block address	~~Cache index~~	Hit/miss	Cache contents after access			
0		miss	M[0] (0)			
4		miss	M[0] (1)	M[4] (0)		
2		miss	M[0] (2)	M[4] (1)	M[2] (0)	
6		miss	M[0] (3)	M[4] (2)	M[2] (1)	M[6] (0)
8		miss	M[8] (0)	M[4] (3)	M[2] (2)	M[6] (1)
0		miss	M[8] (1)	M[0] (0)	M[2] (3)	M[6] (2)
4		miss	M[8] (2)	M[0] (1)	M[4] (0)	M[6] (3)
2		miss	M[8] (3)	M[0] (2)	M[4] (1)	M[2] (0)
6		miss	M[6] (0)	M[0] (3)	M[4] (2)	M[2] (1)
8		miss	M[6] (1)	M[8] (0)	M[4] (3)	M[2] (2)
2		hit	M[6] (2)	M[8] (1)	M[4] (3)	M[2] (0)
6		hit	M[6] (0)	M[8] (2)	M[4] (3)	M[2] (1)
2		hit	M[6] (1)	M[8] (3)	M[4] (3)	M[2] (0)

Note: LRU isn't always the best cache replacement policy. Though effective in many cases, its suitability depends on the application characteristics, cache organization, and other factors. Other policies such as LFU (Least Frequently Used), FIFO (First In, First Out), and random may perform better in certain scenarios, determined by access patterns, cache size, etc.

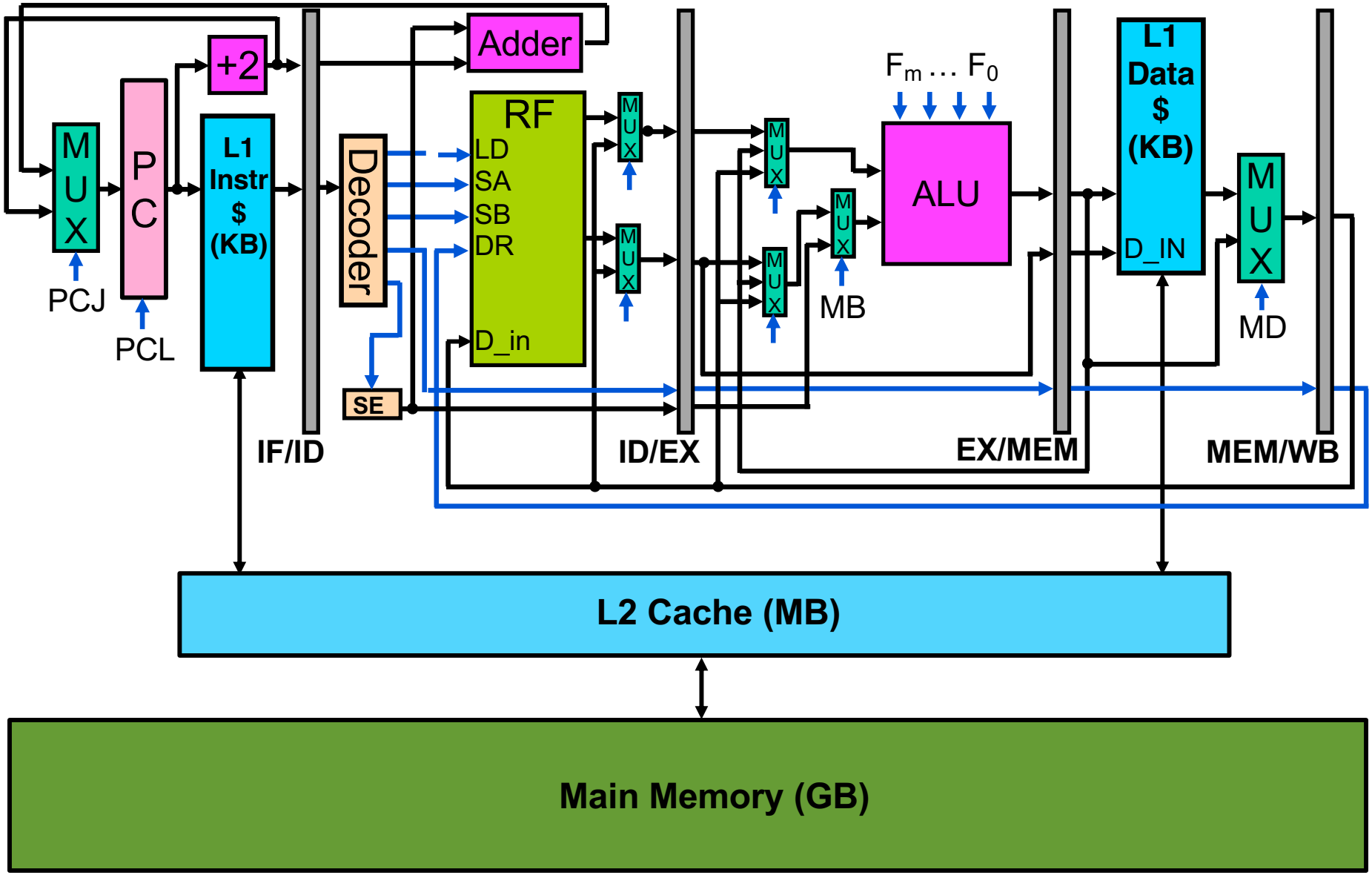
Cache Performance

- Time to get a block from memory is so long that performance suffers even with a low miss rate

- Example: 3% miss rate, 100 cycles to main memory
  - 0.03 × 100 = *3 extra cycles on average* to access instructions or data => low performance

- What's the remedy?

Solution: Add Another Level of Cache



Cache Hierarchy

- Level 1 (L1) instruction and data caches
  - Small, but very fast
- Level 2 (L2) cache handles L1 misses
  - Larger and slower than L1, but much faster than main memory
  - L1 data are also present in L2
- Main memory handles L2 cache misses

- Example: assume 1 cycle to access L1 (3% miss rate), 10 cycles to L2, 10% L2 miss rate, 100 cycles to main memory
  - How many cycles on average for an instruction (or data) access?

$$1 + 0.03 \times (10 + 0.1 \times 100) = 1.6 \text{ cycles}$$

How Do We Measure Performance?

- Execution time: The time between the start and completion of a program (or task)
- Throughput: Total amount of work done in a given time
- Improving performance means
  - Reducing execution time, or
  - Increasing throughput

CPU Execution Time

- Amount of time the CPU takes to run a program

- Derivation

$$\Rightarrow \text{CPU execution time} = \text{I} \times \text{CPI} \times \text{CT}$$

number of instructions in the program

average number of cycles per instruction

clock cycle time (1/frequency)

- Also known as the Iron Law of processor performance
  - The execution time is determined by the product of three factors: instruction count (I), cycles per instruction (CPI), and clock cycle time (CT)
  - Improving performance requires one or more of these factors, while *balancing trade-offs* between them

Instruction Count (I)

- Total number of instructions executed by the processor for a given program

- Factors
  - Instruction set
  - Mix of instructions chosen by the compiler

Cycle Time (CT)

- Clock period (1/frequency)

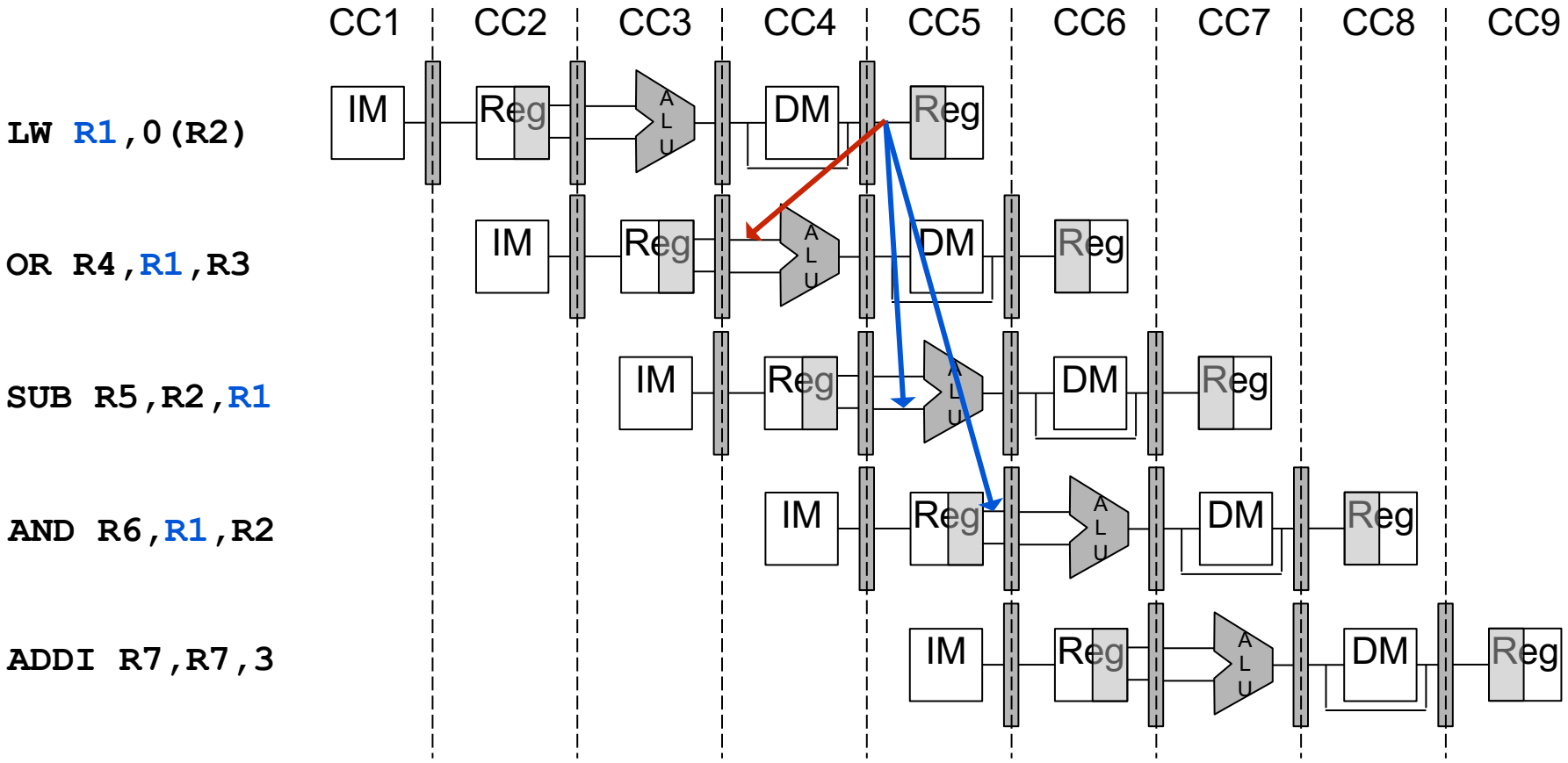
- Factors
  - Instruction set
  - Processor organization and memory hierarchy

Cycles Per Instruction (CPI)

- **Average number of cycles required to execute each instruction**

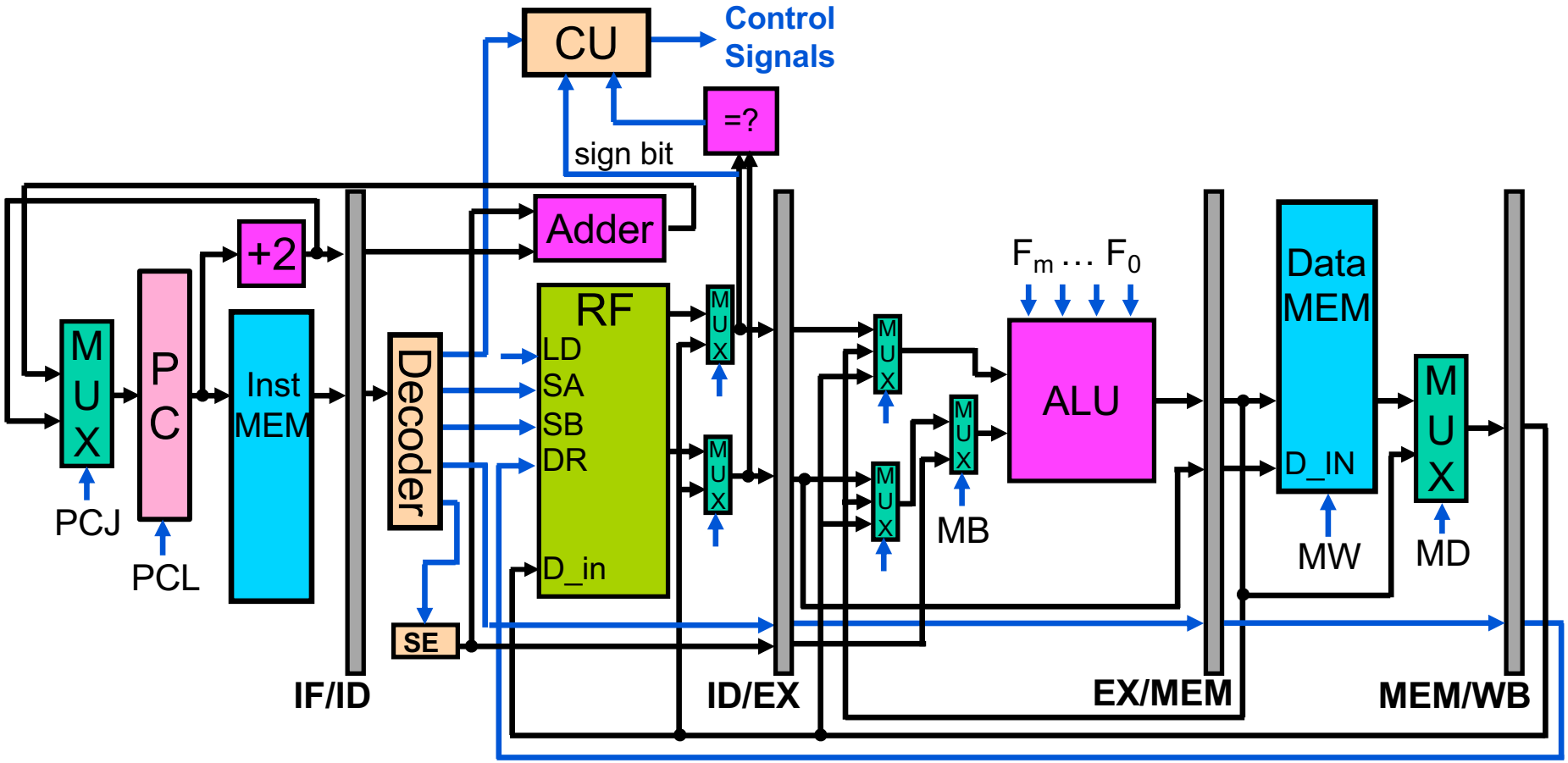
- **Factors**
  - **Instruction set**
  - **Mix of instructions chosen by the compiler**
  - **Ordering of the instructions by the compiler**
  - **Processor organization and memory hierarchy**

Processor Organization
Impact on CPI (Example 1)



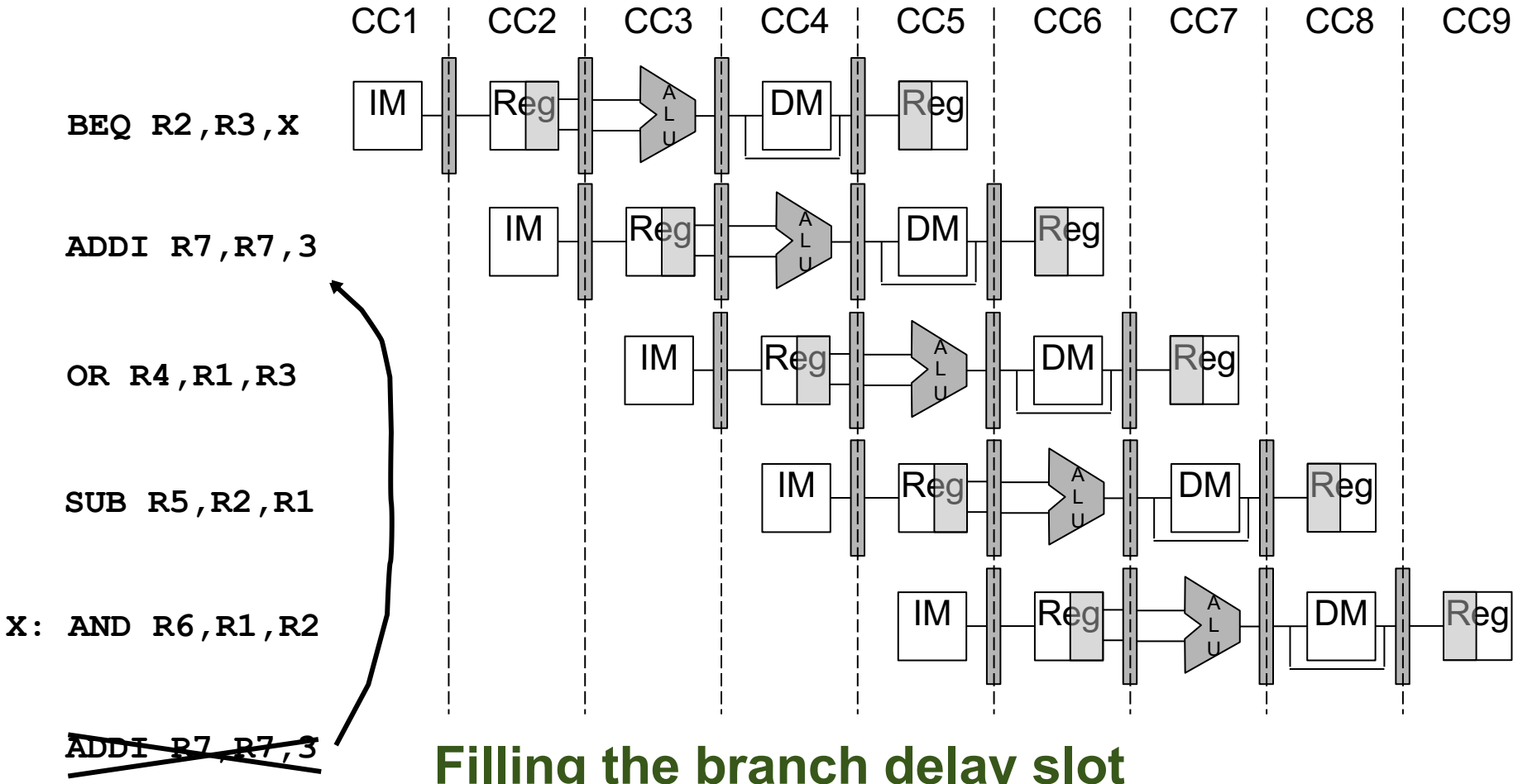
With forwarding: Reduced stall cycles
Lower CPI, potentially reduced execution time

Processor Organization
Impact on CPI (Example 2)



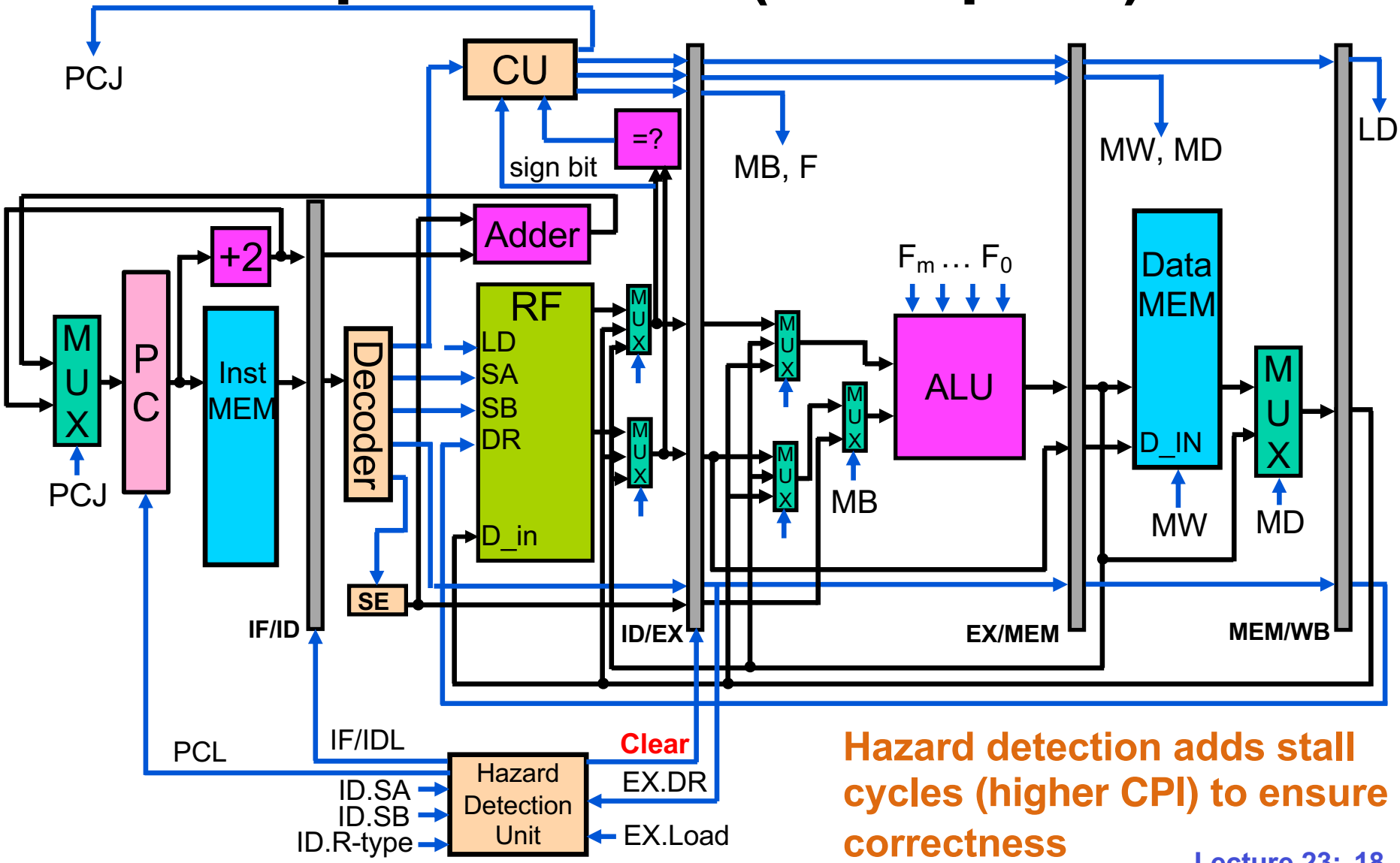
Only one delay slot needed with branch resolved in ID
Lower CPI

Compiler Impact on CPI (Example 3)



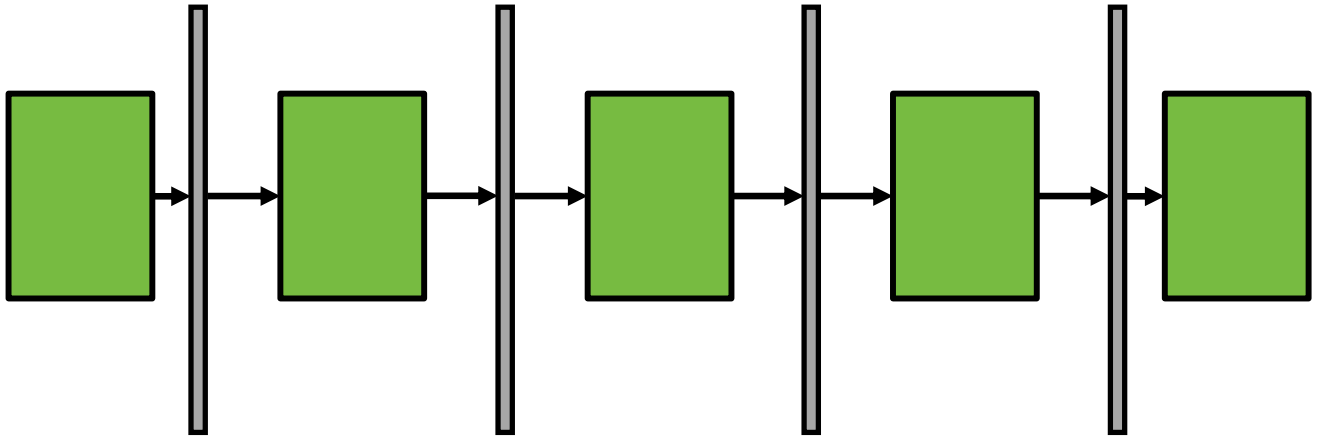
Filling the branch delay slot with a useful instruction

Processor Organization
Impact on CPI (Example 4)

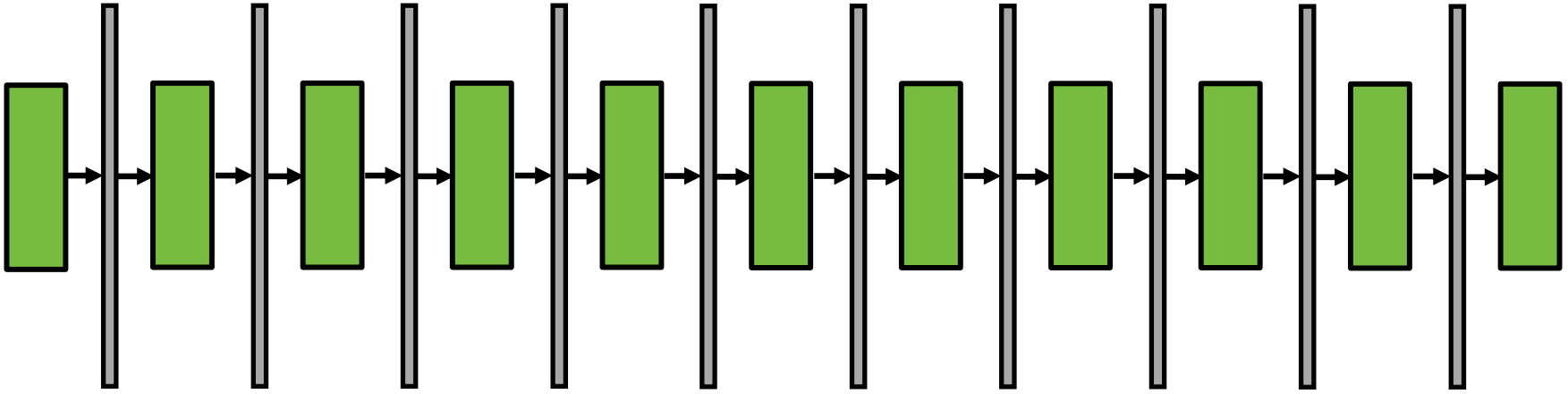


Hazard detection adds stall cycles (higher CPI) to ensure correctness

Discussion 1: Performance Tradeoff



Shallow vs. Deep Pipelining



Impact on CT and CPI?

A Rough Breakdown of CPI

- CPI_{base} is the base CPI in an ideal scenario where instruction fetches and data memory accesses incur no extra delay

- $CPI_{memhier}$ is the (additional) CPI spent for accessing the memory hierarchy when a miss occurs in caches

- CPI_{total} is the overall CPI
  - $CPI_{total} = CPI_{base} + CPI_{memhier}$

Impact of the Memory Hierarchy

- **With NO caches (only main memory)**
  - Every instruction is read from main memory
  - Every load and store instruction accesses main memory
  - Assume
    - 100 cycles to access main memory
    - 25% of all instructions are loads, 10% are stores

$$CPI_{memhier} = \underbrace{100}_{\text{Instruction access}} + \underbrace{(0.25 + 0.1) \times 100}_{\text{Data access}} = 135$$

Impact of L1 Caches

- **With L1 caches**
  - L1 instruction cache miss rate = 2%
  - L1 data cache miss rate = 5%
  - Miss penalty = 100 cycles (access main memory)
  - 20% of all instructions are loads, 10% are stores

$CPI_{memhier}$ =

Impact of L1 Caches

- **With L1 caches**
  - L1 instruction cache miss rate = 2%
  - L1 data cache miss rate = 5%
  - Miss penalty = 100 cycles (access main memory)
  - 20% of all instructions are loads, 10% are stores

$$CPI_{memhier} = \underbrace{0.02 \times 100}_{\text{Instruction access}} + \underbrace{(0.2+0.1) \times 0.05 \times 100}_{\text{Data access}} = 3.5$$

Impact of L1+L2 Caches

- **With L1 and L2 caches**
  - L1 instruction cache miss rate = 2%
  - L1 data cache miss rate = 5%
  - L2 access time = 15 cycles
  - L2 miss rate = 25%
  - L2 miss penalty = 100 cycles (access main memory)
  - 20% of all instructions are loads, 10% are stores

$CPI_{memhier}$ =

Impact of L1+L2 Caches

- **With L1 and L2 caches**
  - L1 instruction cache miss rate = 2%
  - L1 data cache miss rate = 5%
  - L2 access time = 15 cycles
  - L2 miss rate = 25%
  - L2 miss penalty = 100 cycles (access main memory)
  - 20% of all instructions are loads, 10% are stores

- $CPI_{memhier} = 0.02 \times (15 + 0.25 \times 100) +$
  $0.30 \times 0.05 \times (15 + 0.25 \times 100) = 1.4$

Relative Performance

- **Used to compare the performance of machines**
  - **Processor performance and execution time are inversely related**

- **Used to report the performance benefit/loss of adding or subtracting an architectural feature**

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution Time}_Y}{\text{Execution Time}_X}$$

Relative Performance Example: With and Without L2 Cache

- **Assume I and CT stay the same**
- **Execution time is determined by CPI_{total}**
  - **Assume $CPI_{base} = 1.5$**
  - **$CPI_{total} = CPI_{base} + CPI_{memhier}$**

$$\frac{\text{Performance}_{(L1+L2)}}{\text{Performance}_{(L1)}} = \frac{\text{Execution Time}_{(L1)}}{\text{Execution Time}_{(L1+L2)}}$$

$$= \frac{\text{CPI}_{total\,(L1)}}{\text{CPI}_{total\,(L1+L2)}} = \frac{1.5+3.5}{1.5+1.4} = 1.7$$

Amdahl's Law

- **The potential speedup of a program from an enhancement (optimization) is limited by the proportion of the program that is unaffected by the enhancement**

$$\text{Execution Time}_{\text{enhanced}} = \frac{\text{Execution Time affected}}{\text{Amount of improvement}} + \text{Execution Time unaffected}$$

- **Example**
  - **New optimization speeds up multiplication by factor of 10**
  - **Total execution time of a program is 100 sec**
  - **Multiply operations consume 5 sec of the total**

$$\text{Execution Time}_{\text{enhanced}} = \frac{5}{10} + 95 = 95.5\,\text{sec}$$

No matter how fast you make the "optimized" part, the parts you don't speed up will eventually dominate the overall performance

Discussion 2: Performance Tradeoff

Complex Instruction vs. Simple Instruction

ADD 0(R2),0(R1),4(R1)

1 cycle

vs.

```
LW   R3,0(R1)
LW   R4,4(R1)
ADD  R3,R3,R4
SW   R3,0(R2)
```

4 cycles

Does Amdahl's Law apply in this context?

Note: Other advanced computer architecture courses will delve into additional differences and trade-offs between CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer).

Next Class

Virtual Memory
(H&H 8.4)