6.5930/1
Hardware Architecture for Deep Learning

Computer Architecture Basics

Zoey Song
MIT

Legacy slides adapted from 6.191/6.5900

Memory Technologies

	Capacity	Latency	Cost/GB
Register	100s of bits	20 ps	$$$$
SRAM	~10 KB-10 MB	1-10 ns	$$$
DRAM	~10 GB	80 ns	$$
Hard disk	~1 TB	10 ms	$

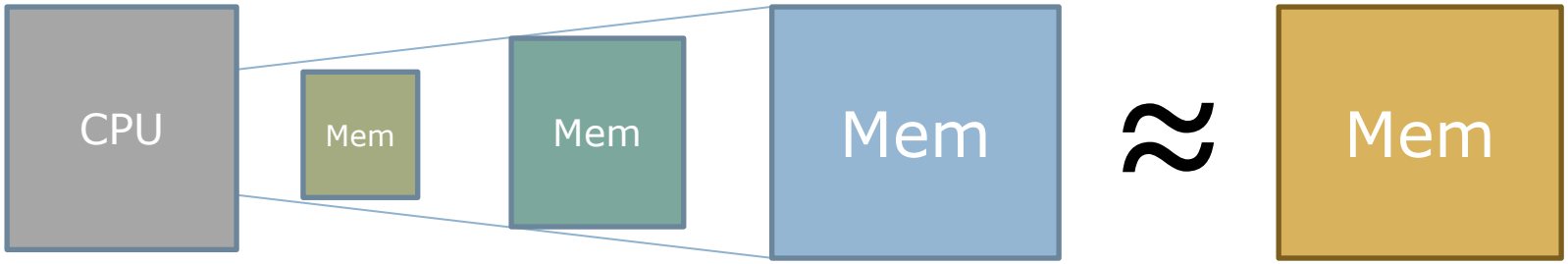
- Different technologies have vastly different tradeoffs
- Size is a fundamental limit, even setting cost aside:
  - Small + low latency, high bandwidth, low energy, or
  - Large + high-latency, low bandwidth, high energy
- Can we get best of both worlds? (large, fast, cheap)

The Memory Hierarchy

Want large, fast, and cheap memory, but…

- Large memories are slow
- Fast memories are expensive

Solution: Use a hierarchy of memories with different tradeoffs to fake a large, fast, cheap memory

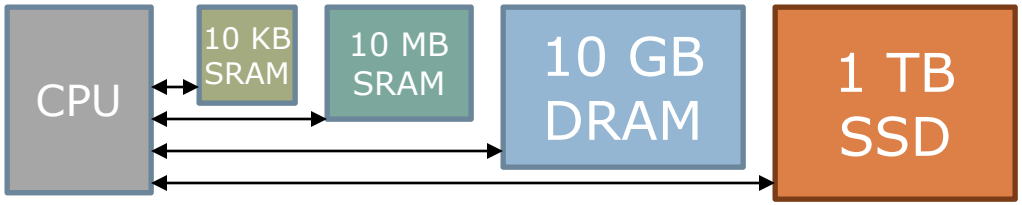


Speed:	Fastest		Slowest	Fast
Capacity:	Smallest		Largest	Large
Cost:	Highest		Lowest	Cheap

Memory Hierarchy Interface

Approach 1: Expose Hierarchy

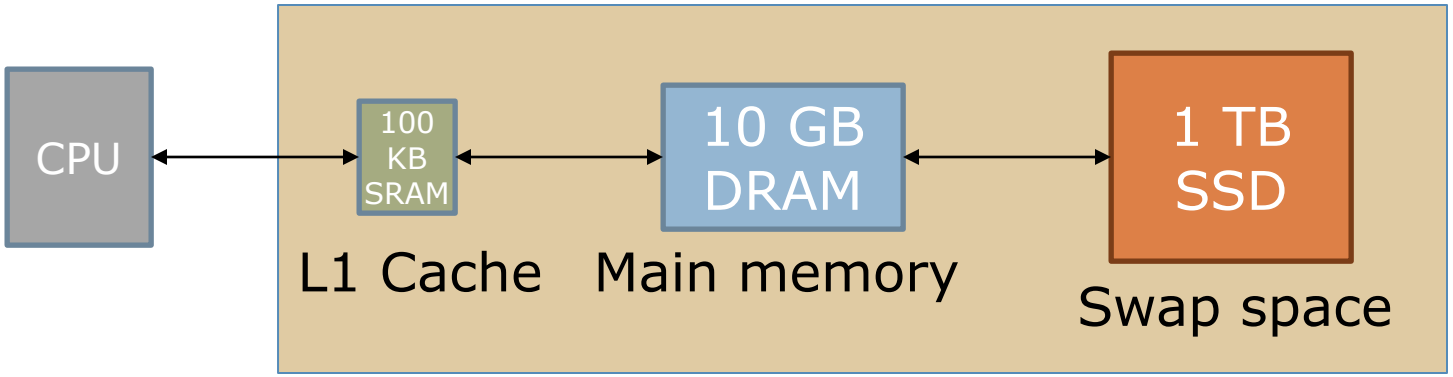
- All memories are exposed to Users (Programmers).



- Tell programmers: "Use them cleverly"

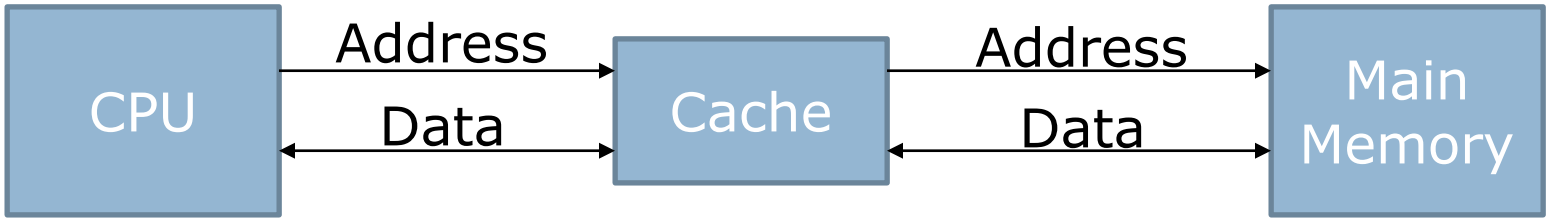
Approach 2: Hide Hierarchy

- Programming model: Single memory, single address space
- Machine stores data in fast or slow memory depending on usage patterns



Caches

- Cache: A small memory component in the hierarchy retains data from recently accessed addrs



- Processor requests data accesses. Two options:
  - Cache hit: Data for this address in cache, returned quickly
  - Cache miss: Data not in cache
    - 1. Fetch data from memory to cache (may replace some data)
    - 2. Deliver the data to CPU
  - Processor must deal with variable memory access time

Cache Metrics

- Hit Ratio: $HR = \frac{hits}{hits + misses} = 1 - MR$
- Miss Ratio: $MR = \frac{misses}{hits + misses} = 1 - HR$
- Average Memory Access Time (AMAT):

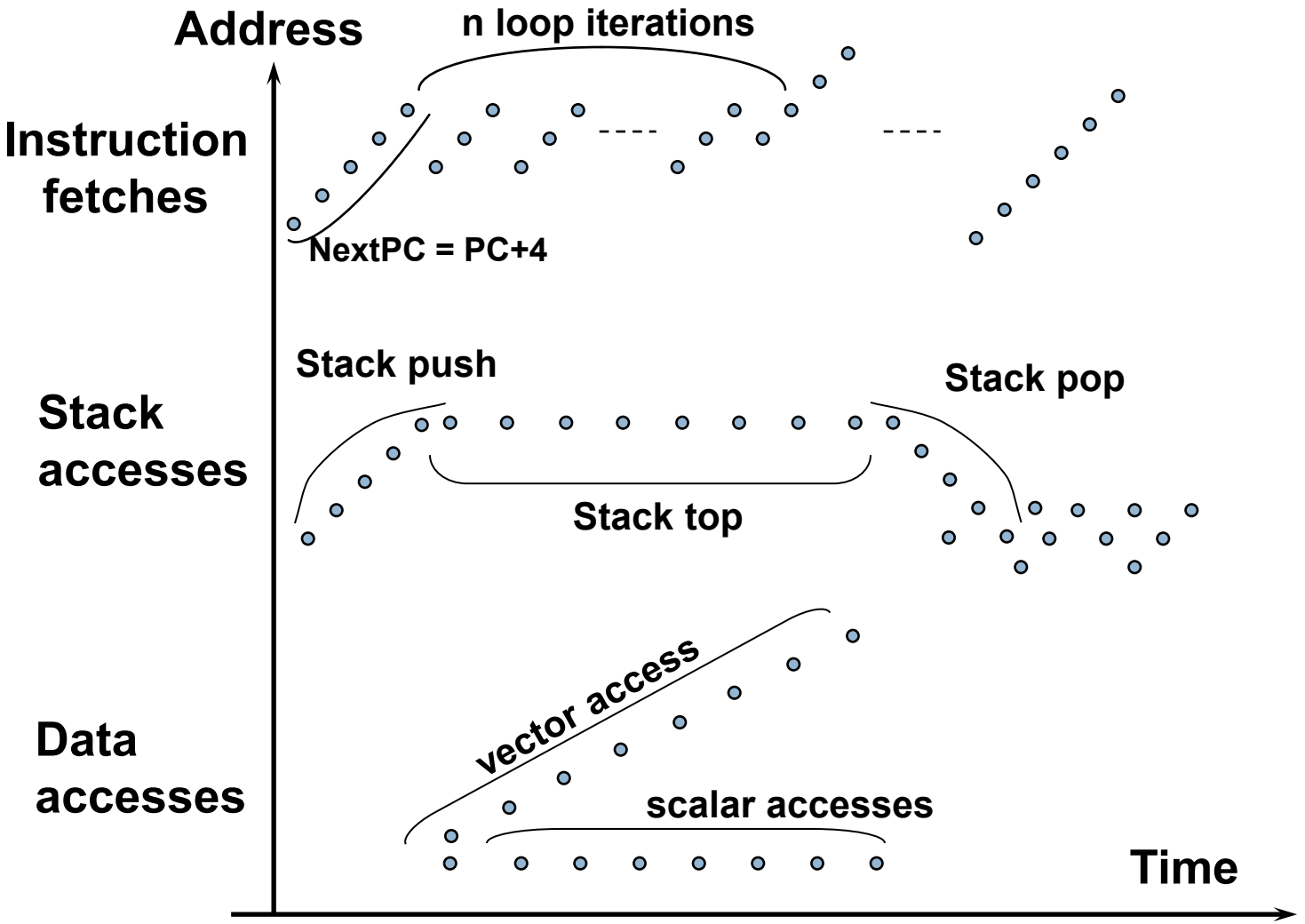
  AMAT = HitTime + MissRatio × MissPenalty

  - Goal of caching is to improve AMAT
  - Formula can be applied recursively in multi-level hierarchies:

$$AMAT = HitTime_{L1} + MissRatio_{L1} \times AMAT_{L2} =$$
$$AMAT = HitTime_{L1} + MissRatio_{L1} \times (HitTime_{L2} + MissRatio_{L2} \times AMAT_{L3}) = \ldots$$

- Key to improve AMAT: Decrease the MissRatio!
  - HitTime and MissPenalty is constant

Typical Memory Reference Patterns

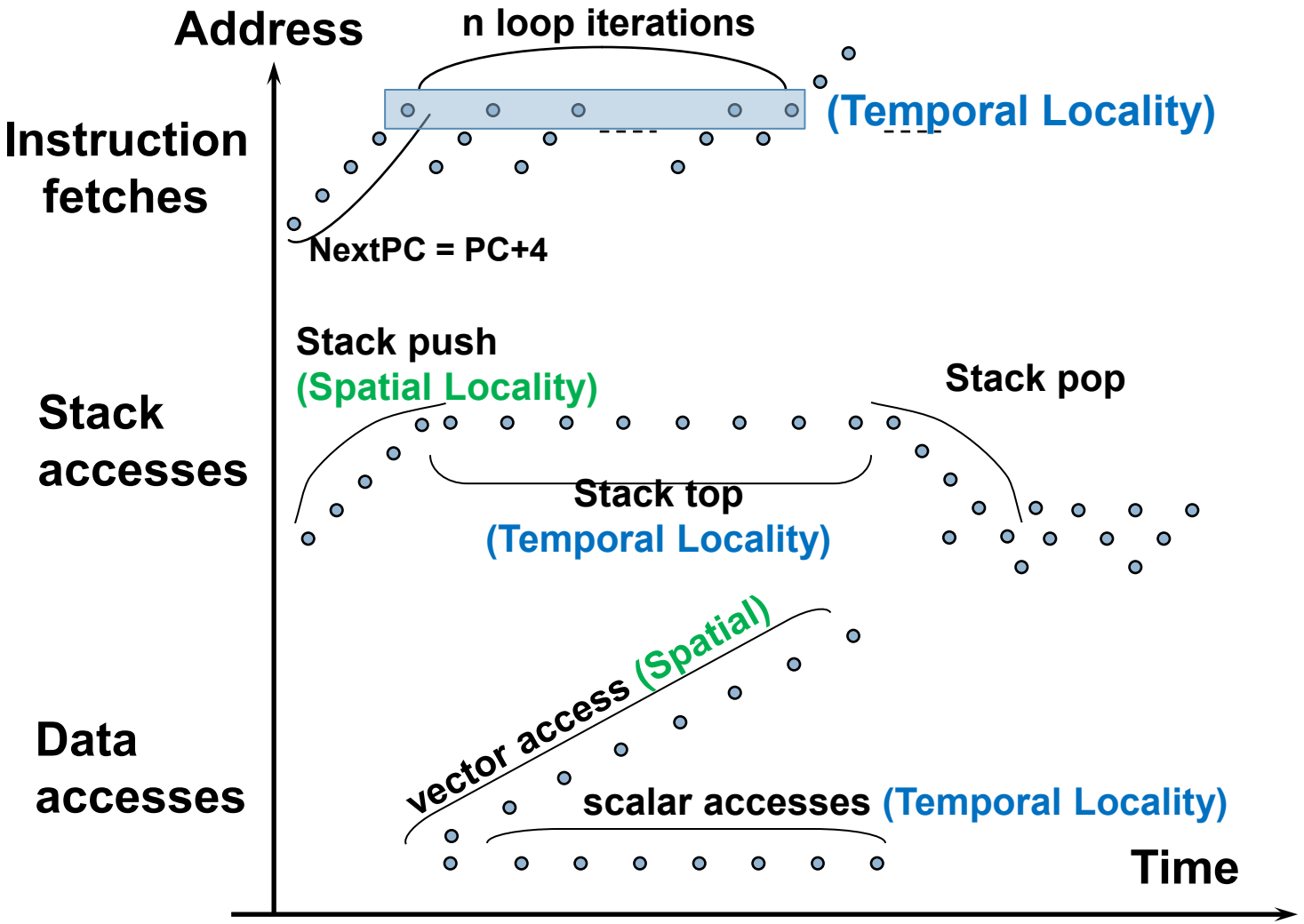


Why Caches Work

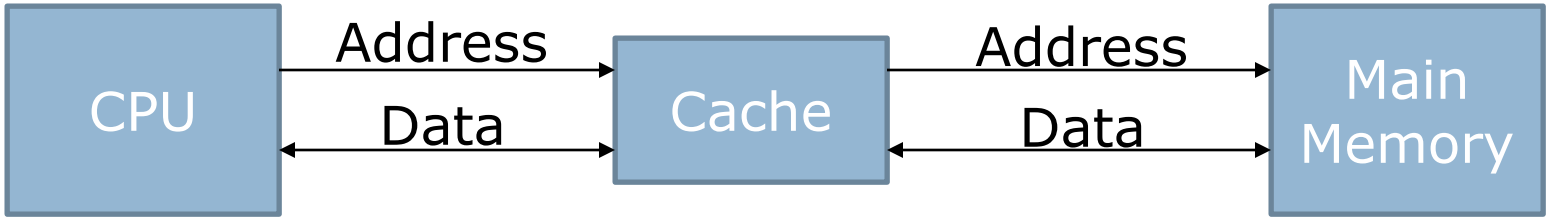
- Two predictable properties of memory accesses:
  - Temporal locality: If a location has been accessed recently, it is likely to be accessed (reused) soon
  - Spatial locality: If a location has been accessed recently, it is likely that nearby locations will be accessed soon
- Result:
  - High hit rate (low miss ratio)
  - Reduced Average Memory Access Time (AMAT):

$$AMAT = HitTime + MissRatio \times MissPenalty$$

Typical Memory Reference Patterns

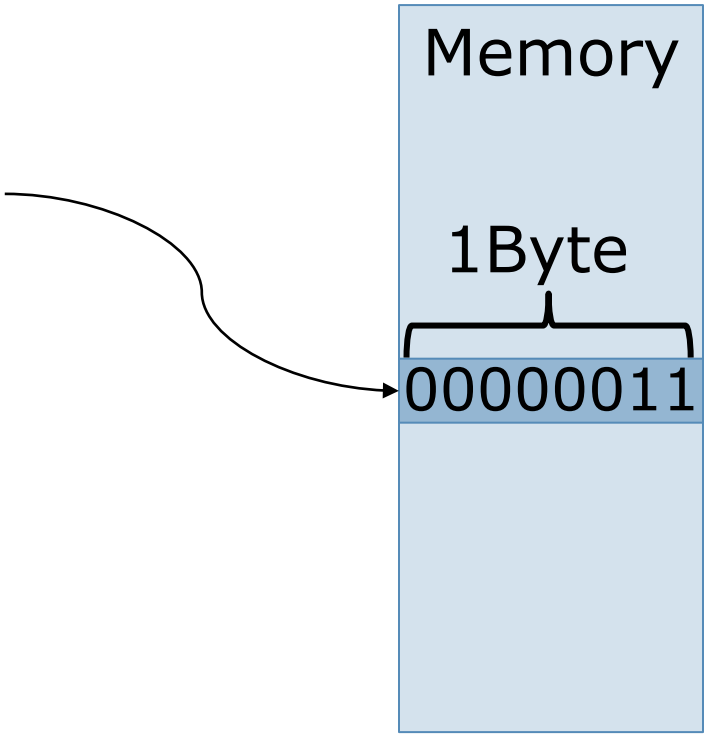


Caches



32-bit BYTE address

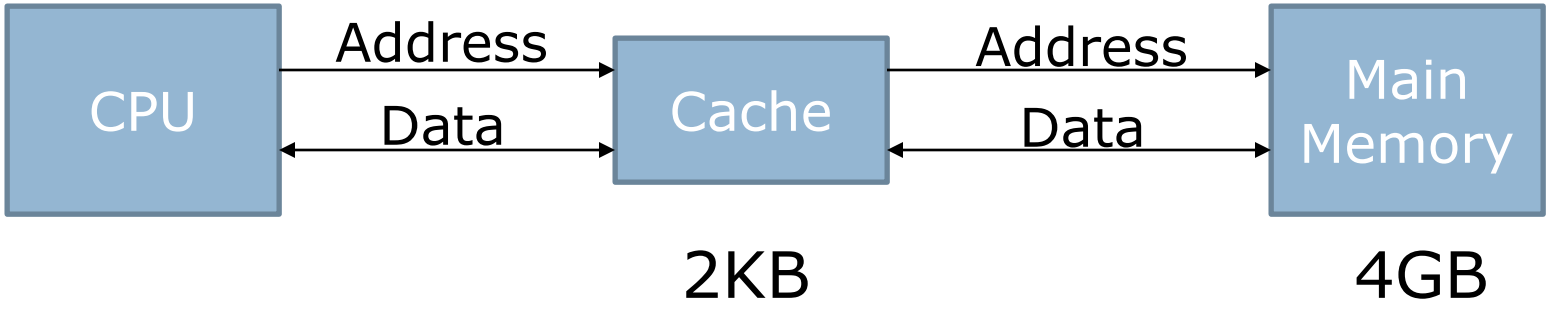
00000000000000000000000011101000



Amount of memory that
32-bit addr can represent

= 2^32 * 1Byte = 4GB

Caches

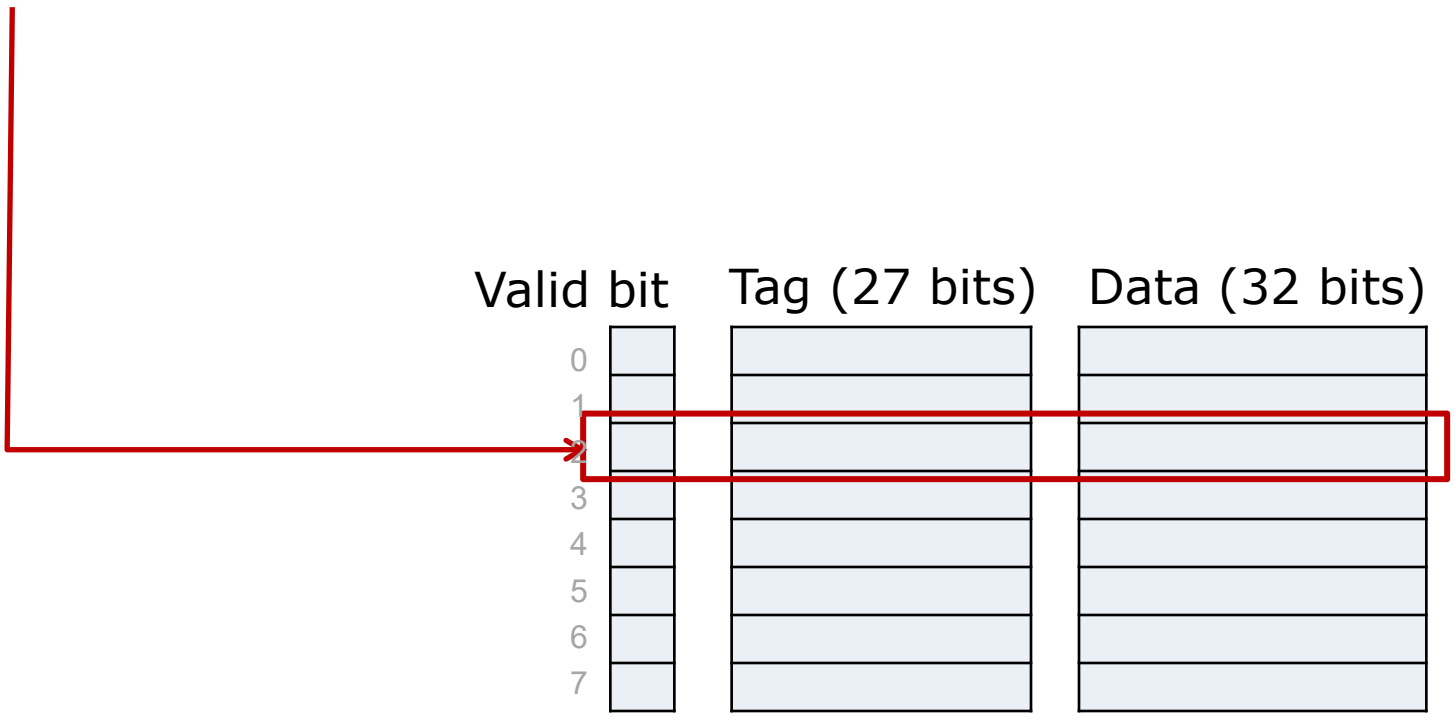


- Cache = Key-Value Store
  - Key : Address (32-bit)
  - Value : Data (1Byte)

- However, cache is smaller than the main memory
  - 2KB = 2048 Byte << 4294967296 Byte
  - Multiple addresses can be mapped into cache line.
    - You need compare a full address to ensure correctness.

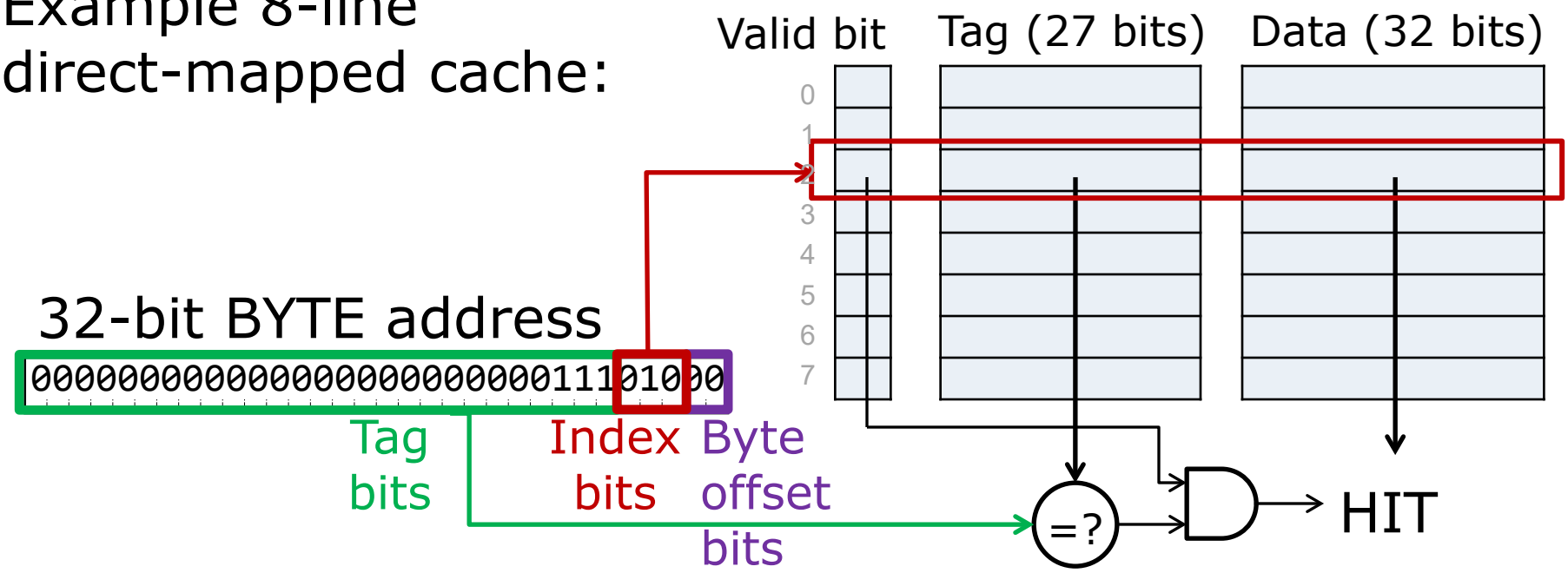
Direct-Mapped Caches

- Terminology
  - Word = 4Byte
  - CacheLine



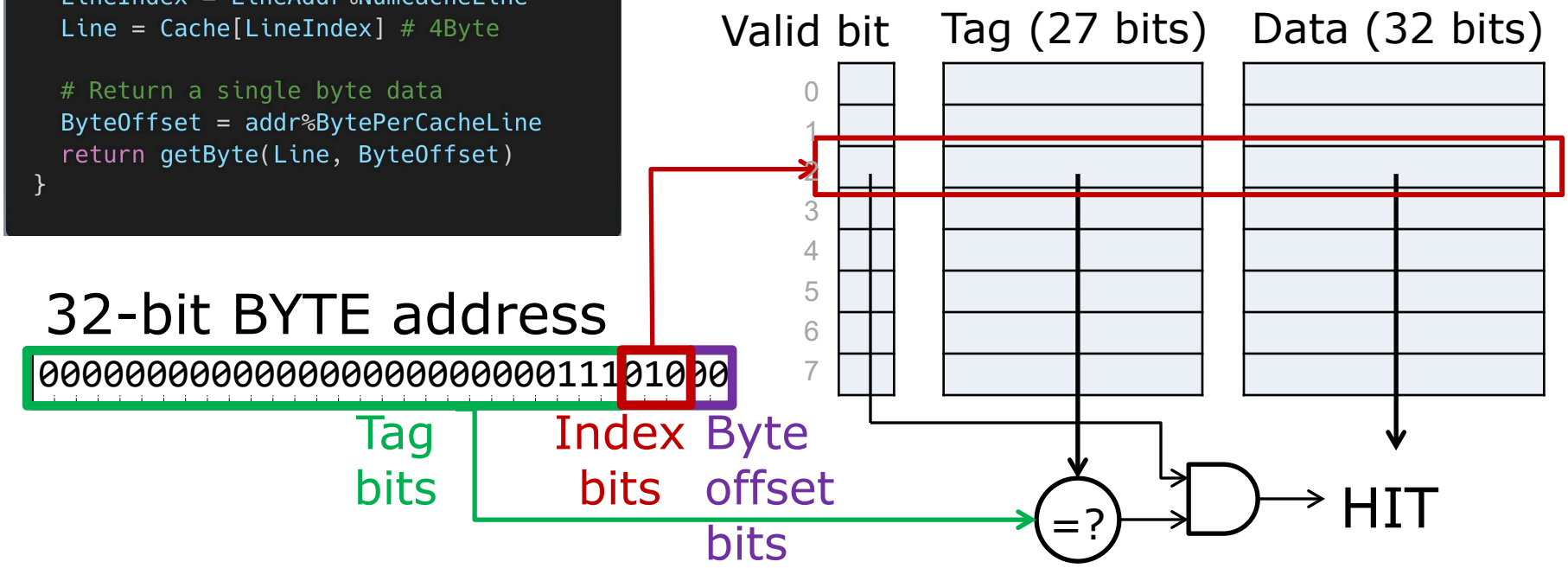
Direct-Mapped Caches

- Each word(4B) in memory maps into a cache line
- Access (for cache with 8 lines):
  - Index into cache line with 3-bits (the index bits)
  - Read out valid bit, tag, and data(line).
  - If valid bit == 1 and tag matches upper address bits, HIT
- Example 8-line direct-mapped cache:



Direct-Mapped Caches

```
# Definition of DM Cache
NumCacheLine = 8
BytePerCacheLine = 4
Cache = [BytePerCacheLine]*NumCacheLine

# Load data
def Load(addr) :
  # Byte Address -> Line Address
  LineAddr = addr/BytePerCacheLine
  LineIndex = LineAddr%NumCacheLine
  Line = Cache[LineIndex] # 4Byte

  # Return a single byte data
  ByteOffset = addr%BytePerCacheLine
  return getByte(Line, ByteOffset)
}
```



Example: Direct-Mapped Caches

64-line direct-mapped cache → 64 indices → 6 index bits (2^6)

Read Mem[0x400C]

0100 0000 0000 1100

TAG: 0x40
INDEX: 0x3
BYTE OFFSET: 0x0

HIT, DATA 0x42424242

Would 0x4008 hit?

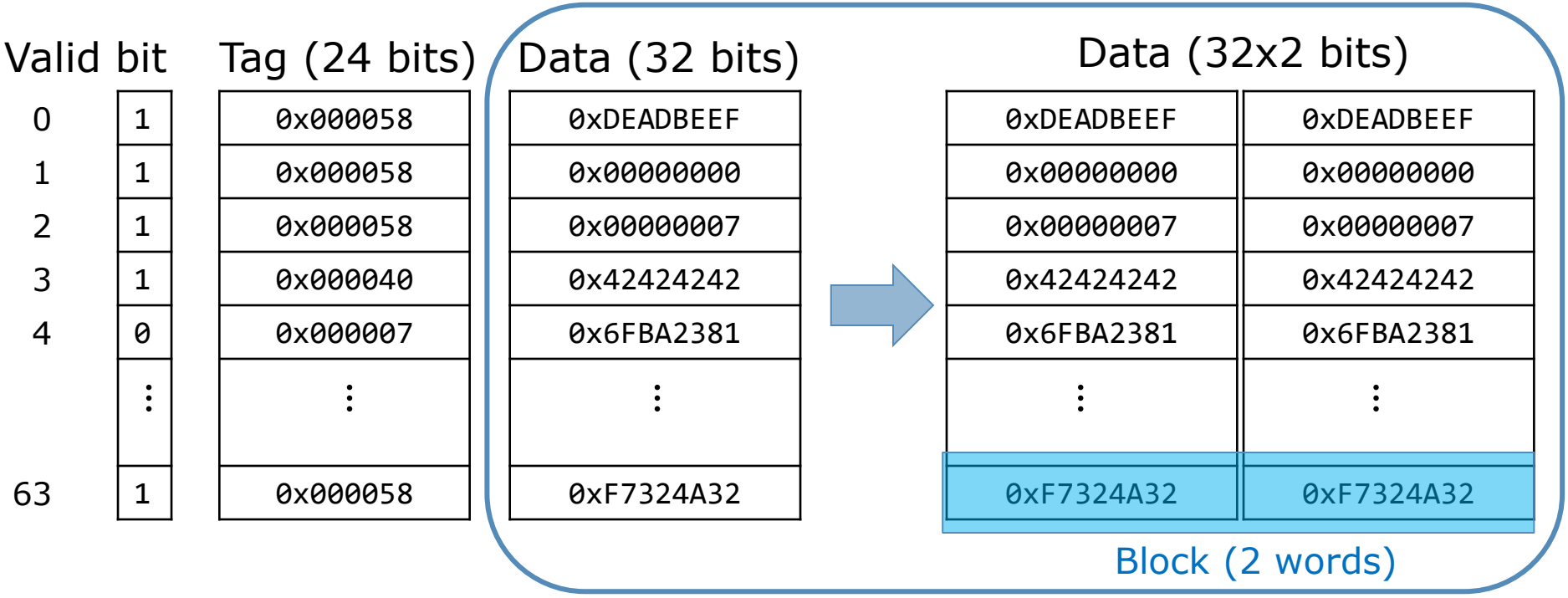
INDEX: 0x2 → tag mismatch
→ MISS

	Valid bit	Tag (24 bits)	Data (32 bits)
0	1	0x000058	0xDEADBEEF
1	1	0x000058	0x00000000
2	1	0x000058	0x00000007
3	1	0x000040	0x42424242
4	0	0x000007	0x6FBA2381
	⋮	⋮	⋮
63	1	0x000058	0xF7324A32

Part of the address (index bits) is encoded in the location
Tag + Index bits unambiguously identify the data's address

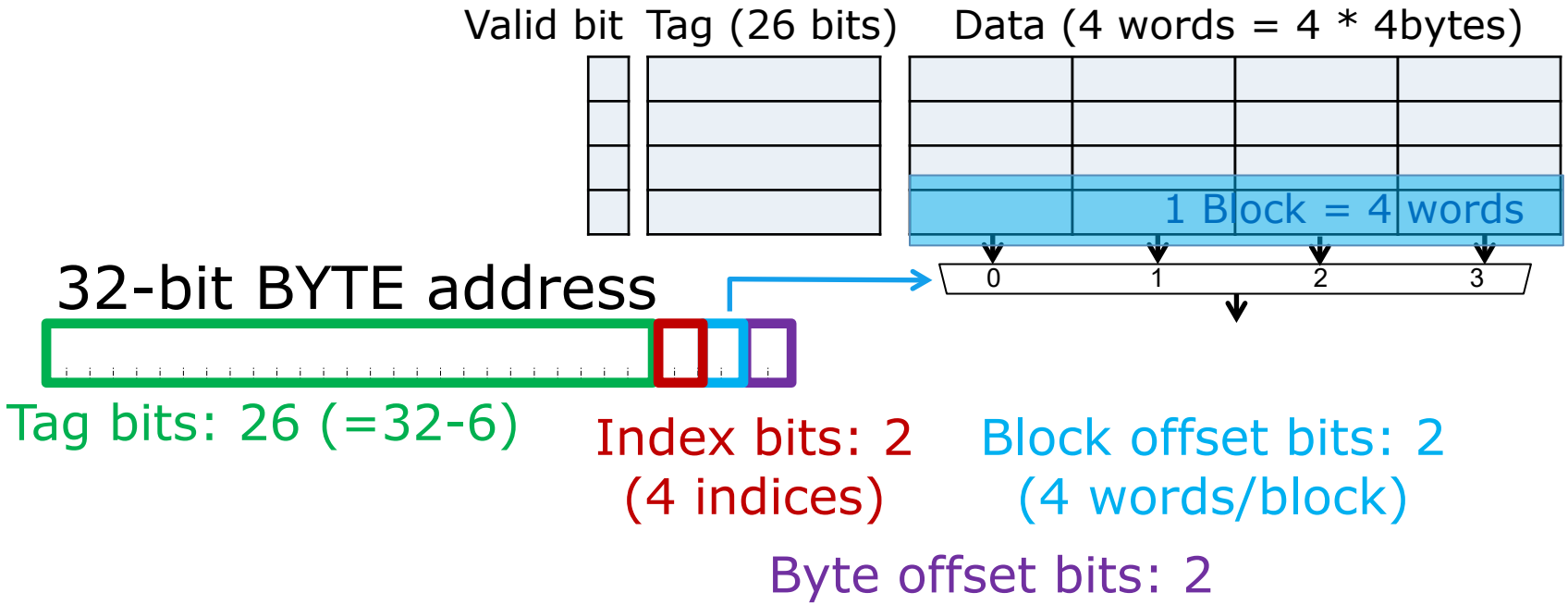
Block Size

- Take advantage of spatial locality: Store multiple words per cache line
  - Always fetch entire block (multiple words) from memory



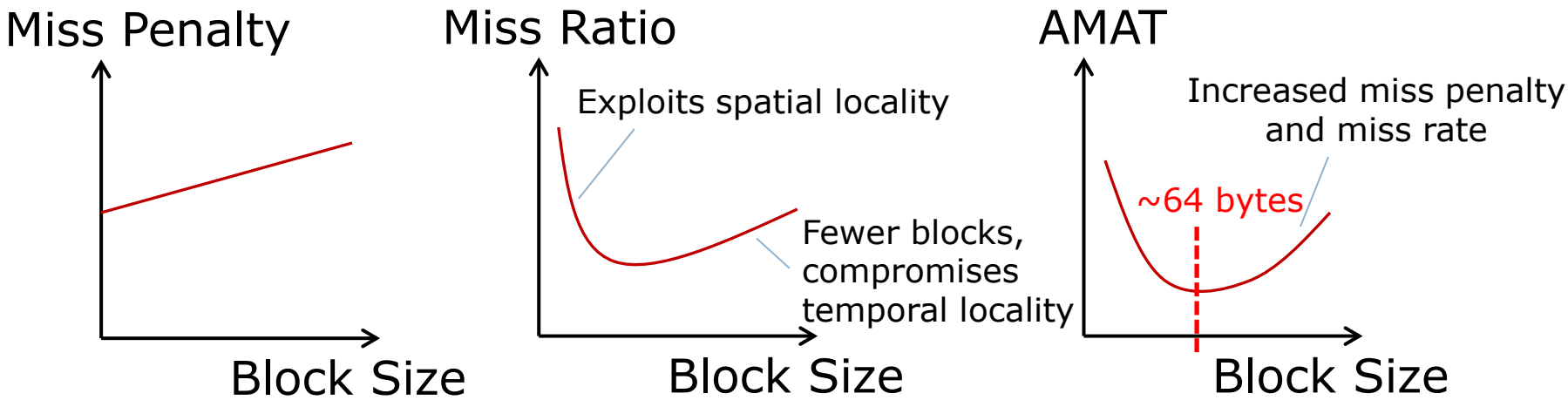
Block Size

- Take advantage of spatial locality: Store multiple words per cache line
  - Always fetch entire block (multiple words) from memory
  - Another advantage: Reduces size of tag memory!
  - Potential disadvantage: Fewer indices in the cache
- Example: 4-block * 4-words/block direct-mapped cache



Block Size Tradeoffs

- Larger block sizes…
  - Take advantage of spatial locality
  - Incur larger miss penalty since it takes longer to transfer the block from memory
  - Can increase the average hit time and miss ratio
- AMAT = HitTime + MissPenalty*MissRatio



Block Size Tradeoffs

- Larger block sizes…
  - Take advantage of spatial locality
  - Incur larger miss penalty since it takes longer to transfer the block from memory
  - Can increase the average hit time and miss ratio
- AMAT = HitTime + MissPenalty*MissRatio

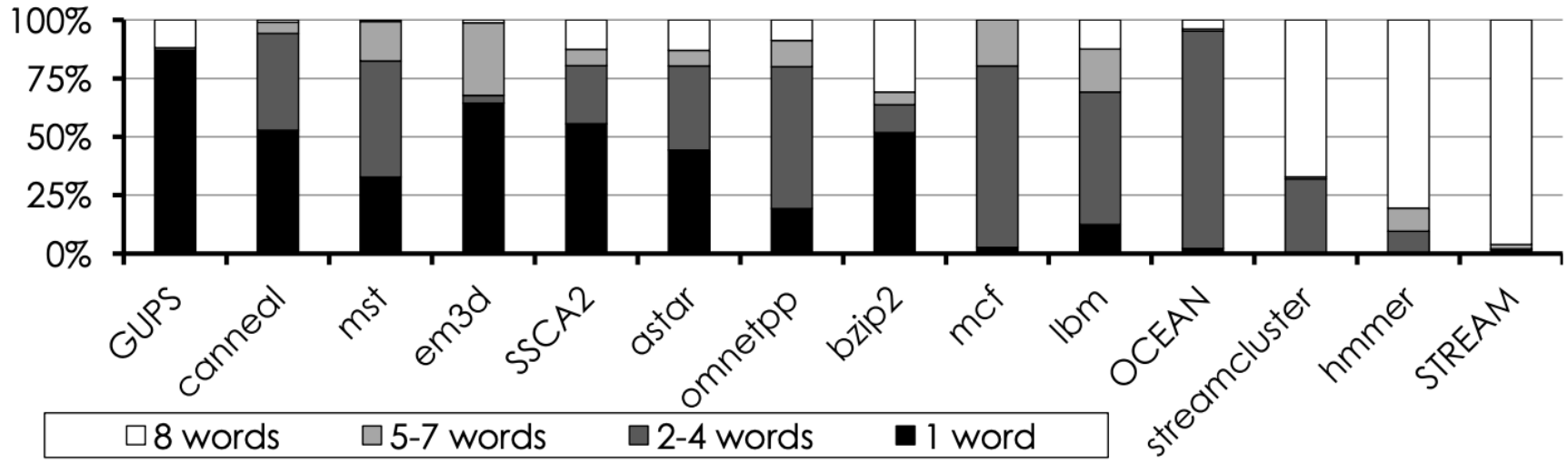


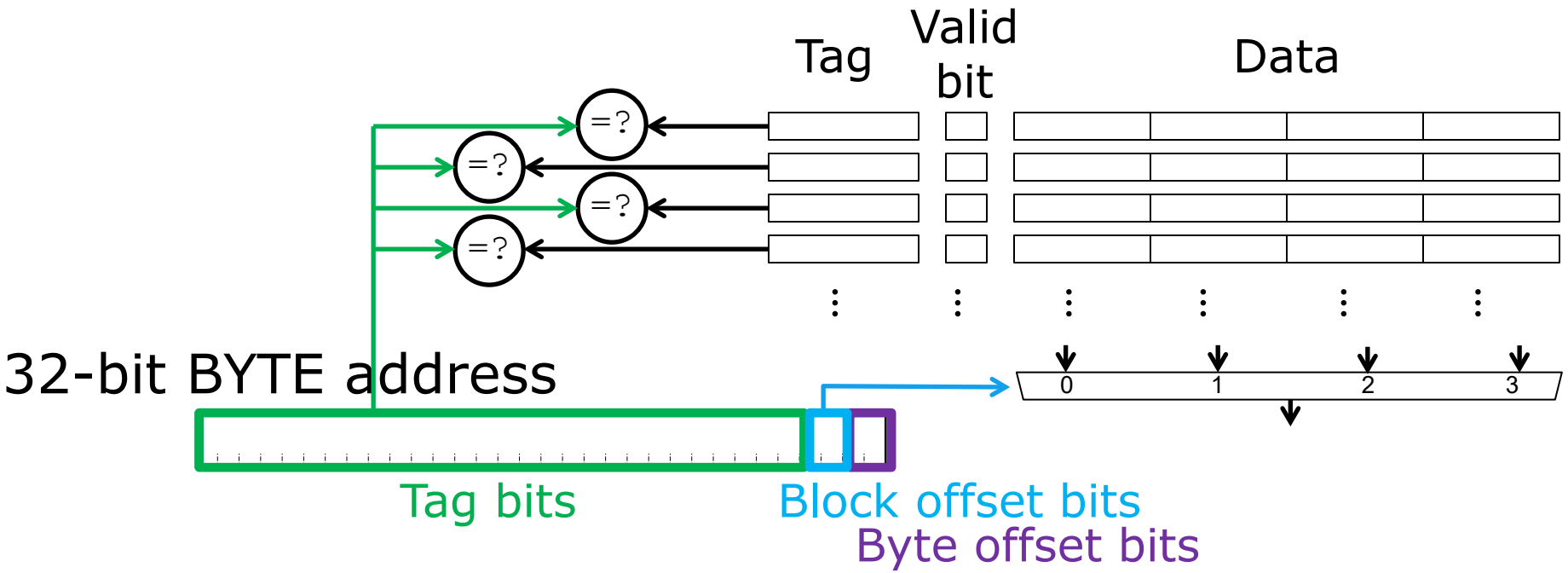
Figure 1: Number of touched 8B words in a 64B cache line before the line is evicted.

Yoon and Sullivan, "The Dynamic Granularity Memory System" [ISCA,2012]

Fully-Associative Cache

Opposite extreme: Any address can be in any location

- No cache index!
- Flexible (no conflict misses)
- Expensive: Must compare tags of all entries in parallel to find matching one

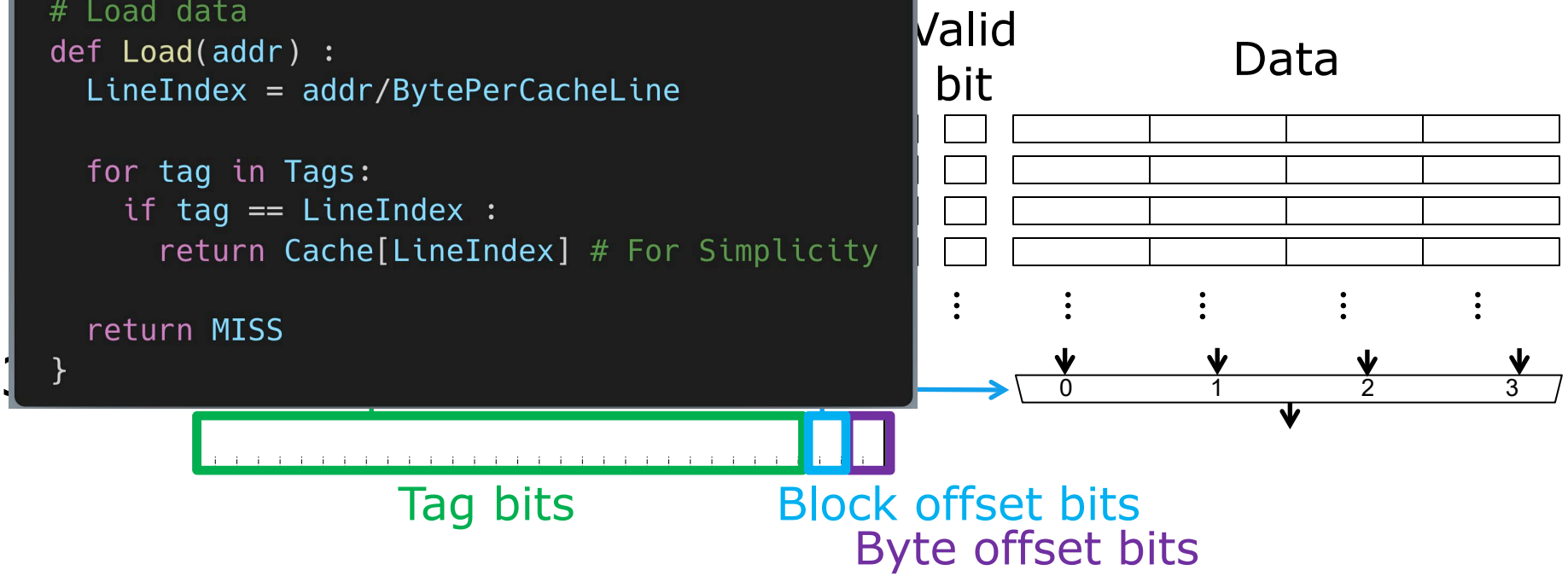


Fully-Associative Cache

```
# Definition of Associtative Cache
NumCacheLine = 4
BytePerCacheLine = 4*4
Cache = [BytePerCacheLine]*NumCacheLine
Tags = [32-log2(BytePerCacheLine)]*NumCacheLine

# Load data
def Load(addr) :
  LineIndex = addr/BytePerCacheLine

  for tag in Tags:
    if tag == LineIndex :
      return Cache[LineIndex] # For Simplicity

  return MISS
}
```

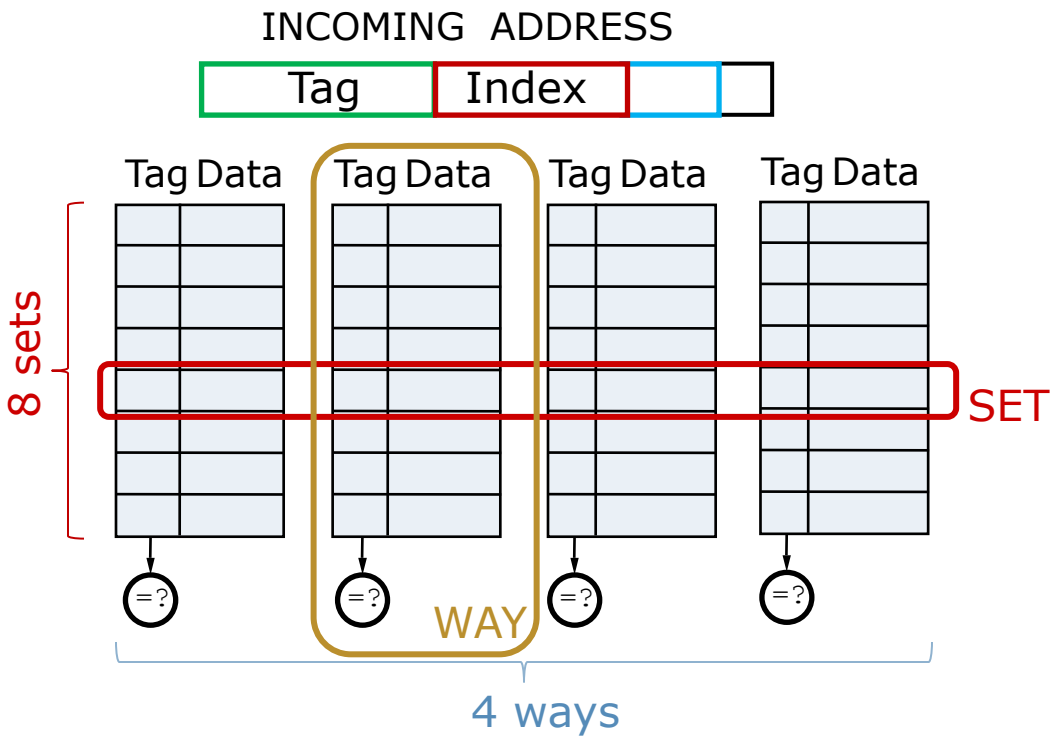
can be in any location

entries in parallel to find



N-way Set-Associative Cache

- Use multiple direct-mapped caches in parallel to reduce conflict misses
  - Nomenclature:
    - # Rows = # Sets
    - # Columns = # Ways
    - "set associativity" (e.g., 4-way → 4 lines/set)
  - Each address maps to only one set, but can be in any way within the set
  - Tags from all ways are checked in parallel



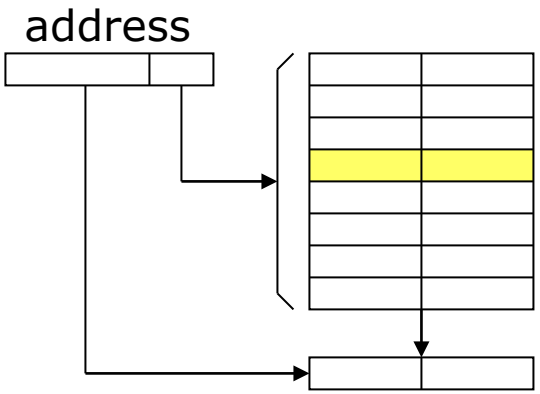
- Fully-associative cache: Extreme case with a single set and as many ways as cache lines

Question: A N-way associative cache with (64/N) sets has how many comparators?

Associativity Implies Choices

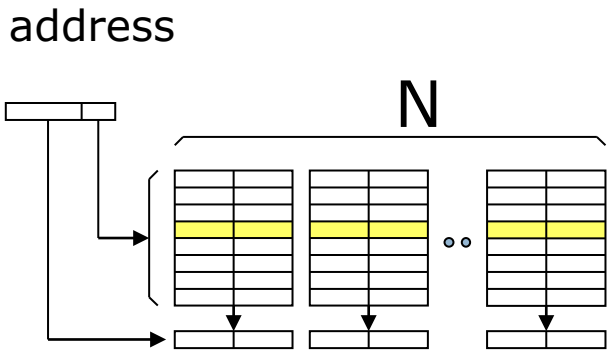
Issue: Replacement Policy

Direct-mapped



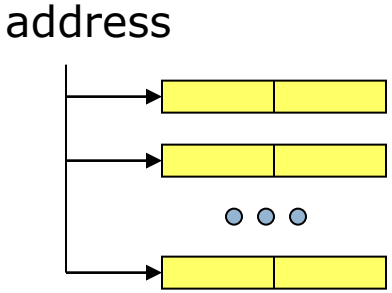
- Compare addr with only one tag
- Location A can be stored in exactly one cache line

N-way set-associative



- Compare addr with N tags simultaneously
- Location A can be stored in exactly one set, but in any of the N cache lines belonging to that set

Fully associative



- Compare addr with each tag simultaneously
- Location A can be stored in any cache line

Replacement Policies

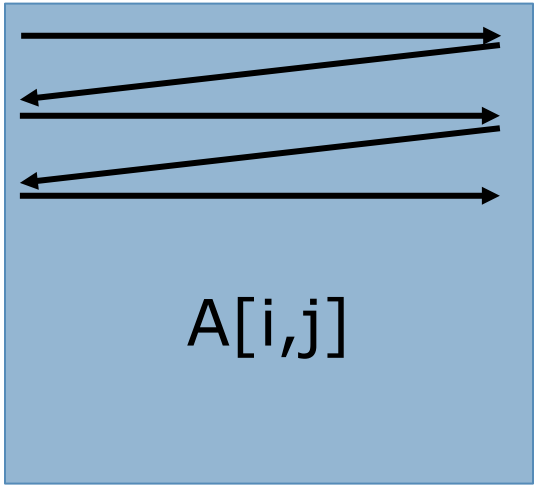
- Optimal policy: Replace the line that is accessed furthest in the future
  - Requires knowing the future…
- Idea: Predict the future from looking at the past
  - If a line has not been used recently, it's often less likely to be accessed in the near future
- Least Recently Used (LRU): Replace the line that was accessed furthest in the past
  - Works well in practice
  - Need to keep ordered list of N items → N! orderings
    → $O(\log_2 N!) = O(N \log_2 N)$ "LRU bits" + complex logic
  - Caches often implement cheaper approximations of LRU
- Other policies:
  - First-In, First-Out (least recently replaced)
  - Random: Choose a candidate at random
    - Not very good, but has better worst case performance

Summary: Cache Tradeoffs

$$AMAT = HitTime + MissRatio \times MissPenalty$$

- Cache size
- Block size
- Associativity
- Replacement policy

Lab2 Foreshadowing



A is stored in row-major order.

Address(i,j)=i*N+j

```
16      // Row Major Traversal (i->j order)
17      for (int i=0; i<1024; i++) {
18        for (int j=0; j<1024; j++) {
19          sum1 += A[i][j];
20        }
21      }
```

```
30      // Col Major Traversal (j->i order)
31      for (int j=0; j<1024; j++) {
32        for (int i=0; i<1024; i++) {
33          sum2 += A[i][j];
34        }
35      }
```

Fine-grain Multithreading

Resolving Hazards

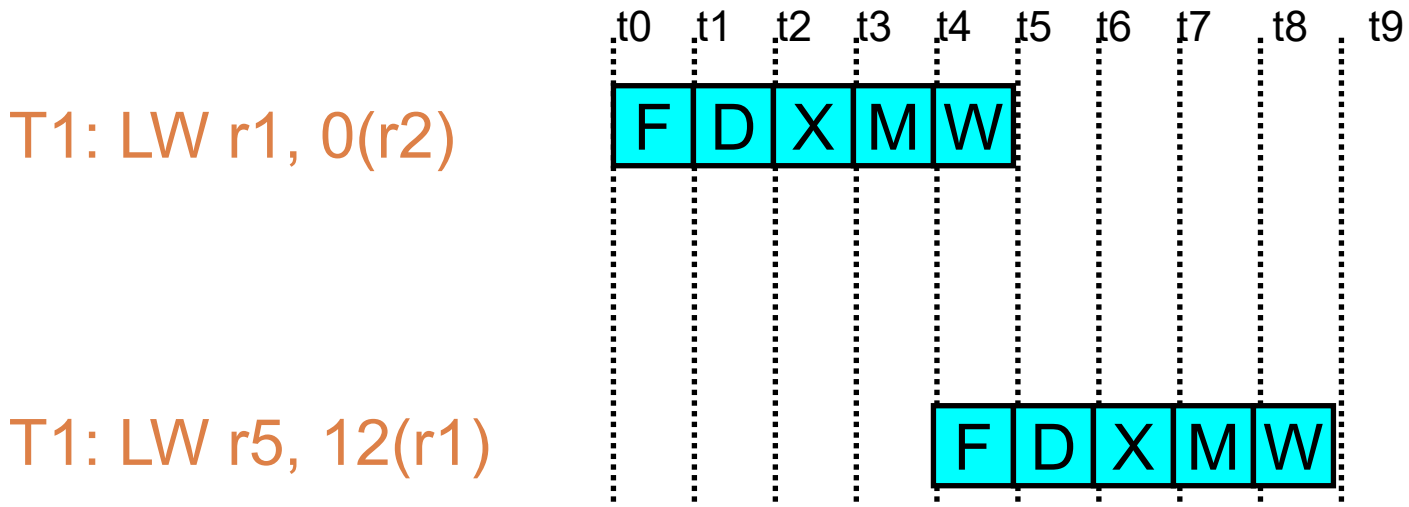
- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
- Strategy 2: Bypass (aka Forward). Route data to the earlier pipeline stage as soon as it is calculated
- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value
- Strategy 4: Find something else to do

Multithreading

How can we guarantee no dependencies between instructions in a pipeline?

Take instructions from different programs

Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe



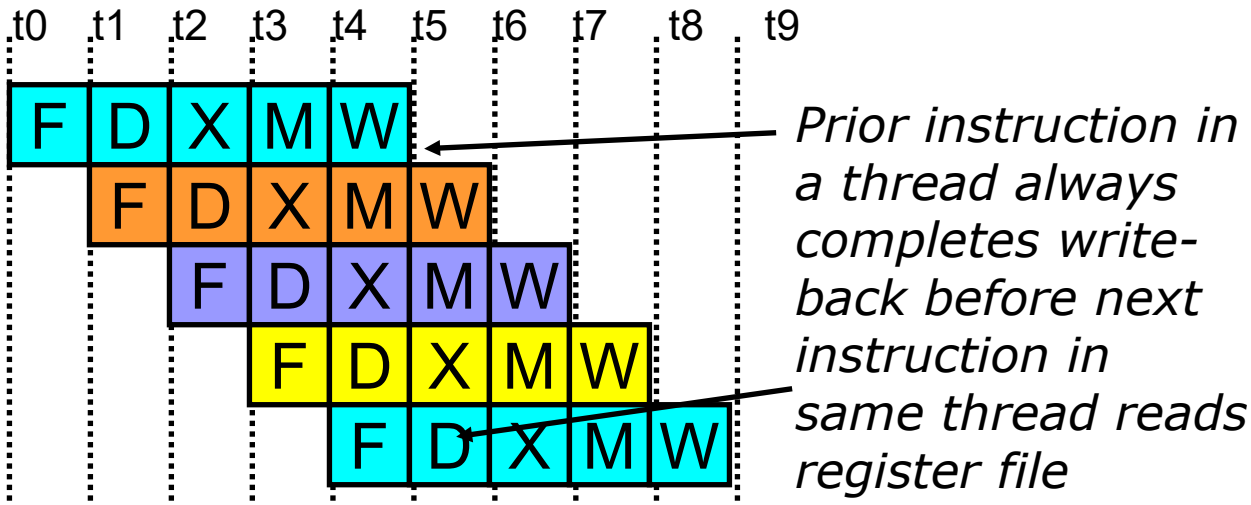
Multithreading

How can we guarantee no dependencies between instructions in a pipeline?

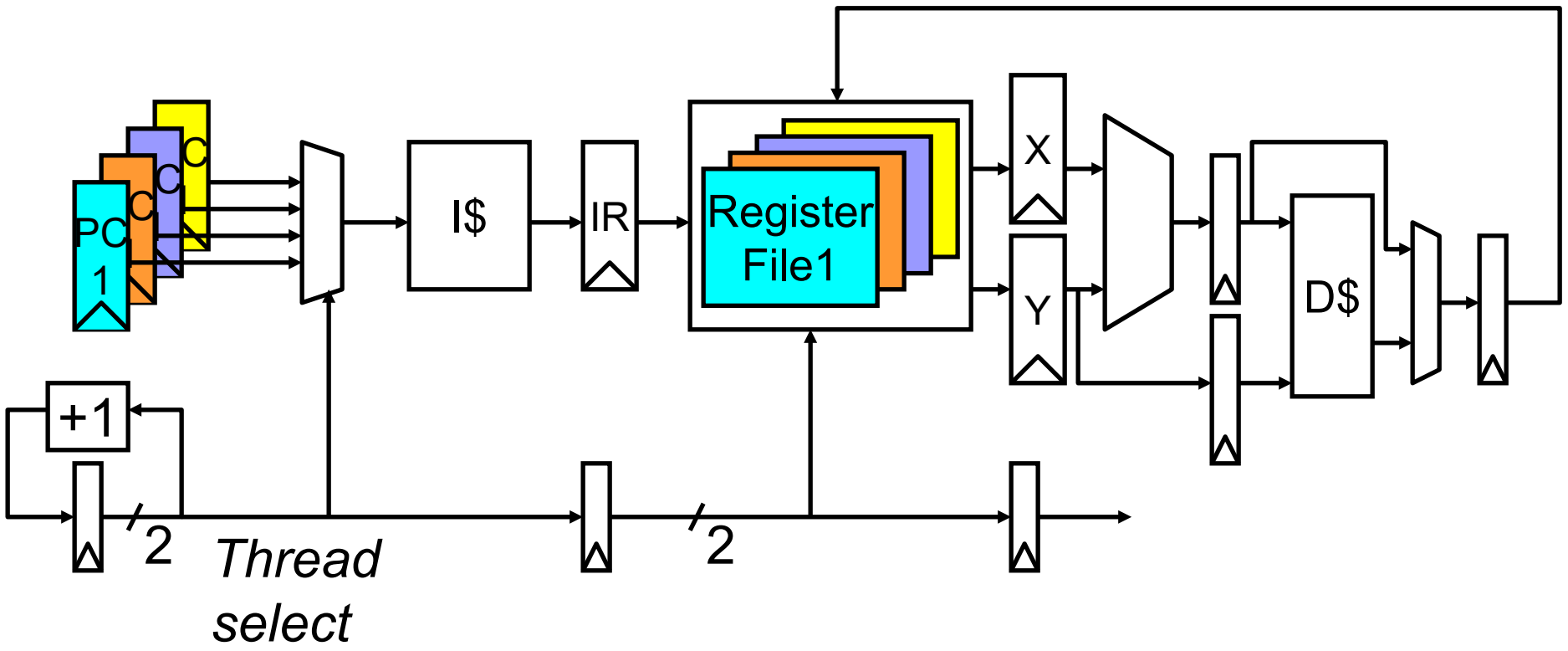
Take instructions from different programs

Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe

T1: LW r1, 0(r2)
T2: ADD r7, r1, r4
T3: XORI r5, r4, #12
T4: SW 0(r7), r5
T1: LW r5, 12(r1)



Fine-grain Multithreaded Pipeline



Have to carry thread select down pipeline to ensure correct state bits read/written at each pipe stage

- Each thread needs its own user architectural state
  - PC, Register Files

Thank you!