Lecture 11

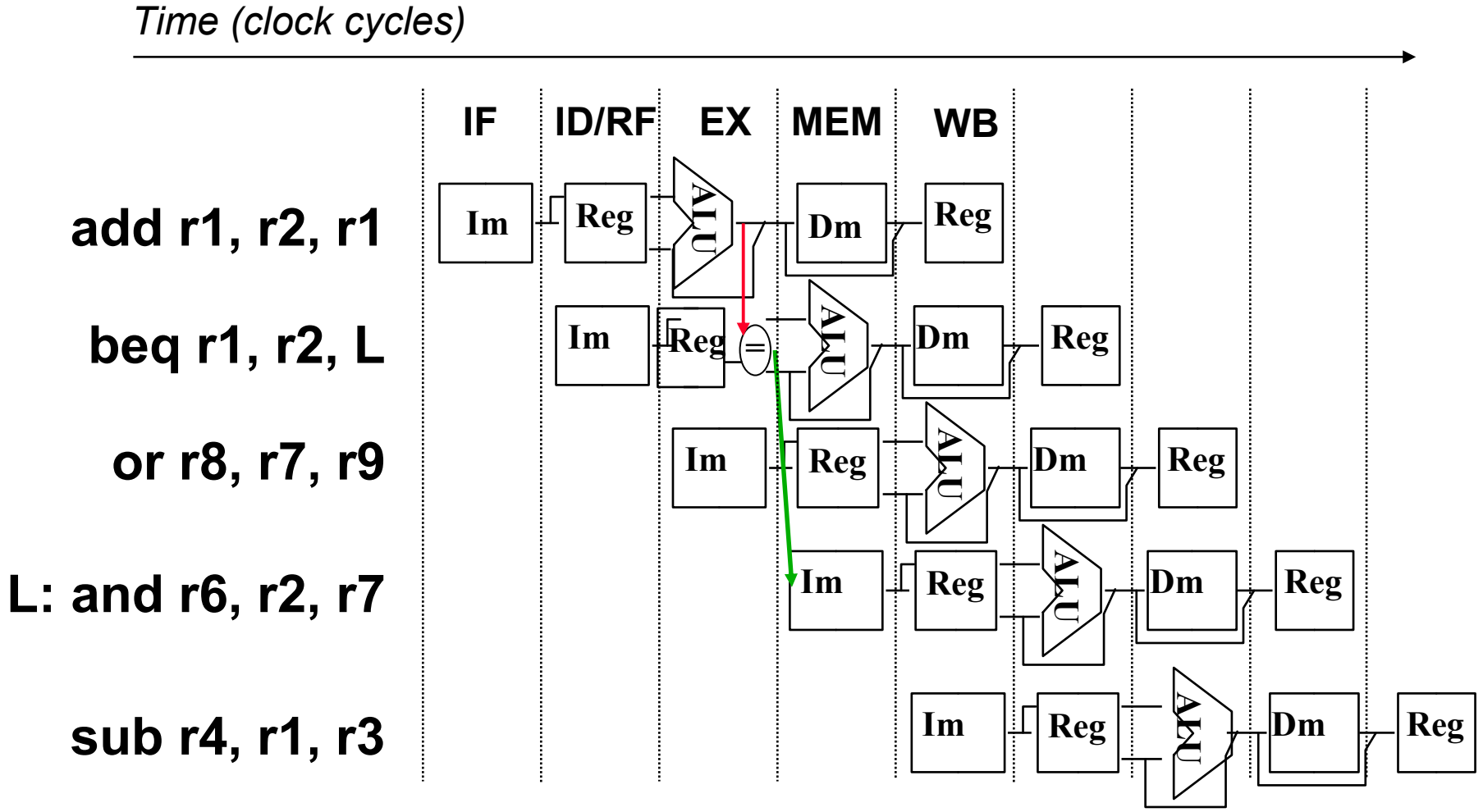
Memory Design

Christos Kozyrakis
Stanford University
http://eeclass.stanford.edu/ee108b

Announcements

- TBD

Review: Reducing Branch Control Hazard



Branch CPI

65% branches are taken

50% of single delay slots are filled usefully

Control point	Branch strategy	Branch CPI
MEM	stall	
ID	stall	
ID	Predict taken	
ID	Predict Not taken	
ID (target) / MEM (=)	Predict taken	
ID	Delayed branch	

Review: Advanced Pipelining

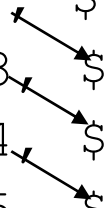
- Where have all the transistors gone?
  - MIPS R3000 : 120 thousand transistors
  - Intel Pentium 4 : 160 Million transistors
  - Many transistors in the cache
- Fancy techniques to decrease CPI and increase clock frequency
  - Superscalar (multiple instruction execution)
  - Deep pipelining
  - Dynamic scheduling (out-of-order execution)
  - Dynamic branch prediction
  - Register renaming
- All pipelining techniques exploit instruction-level parallelism (ILP)

Review: What is ILP?

- Independence among instructions

- Example with ILP
-
```
add $t0, $t1, $t2
or  $t3, $t1, $t2
sub $t4, $t1, $t2
and $t5, $t1, $t2
```

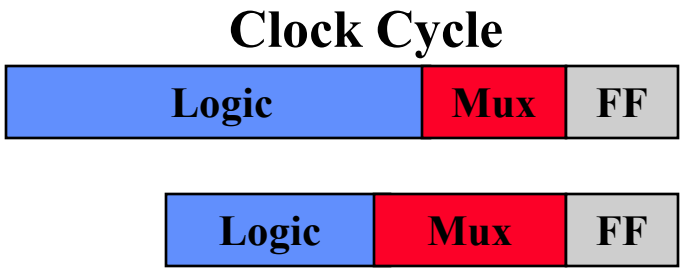
- Example with no ILP
-
```
add $t0, $t1, $t2
or  $t3, $t0, $t2
sub $t4, $t3, $t2
and $t5, $t4, $t2
```



ILP in real programs is limited

Limits of Advanced Pipelining

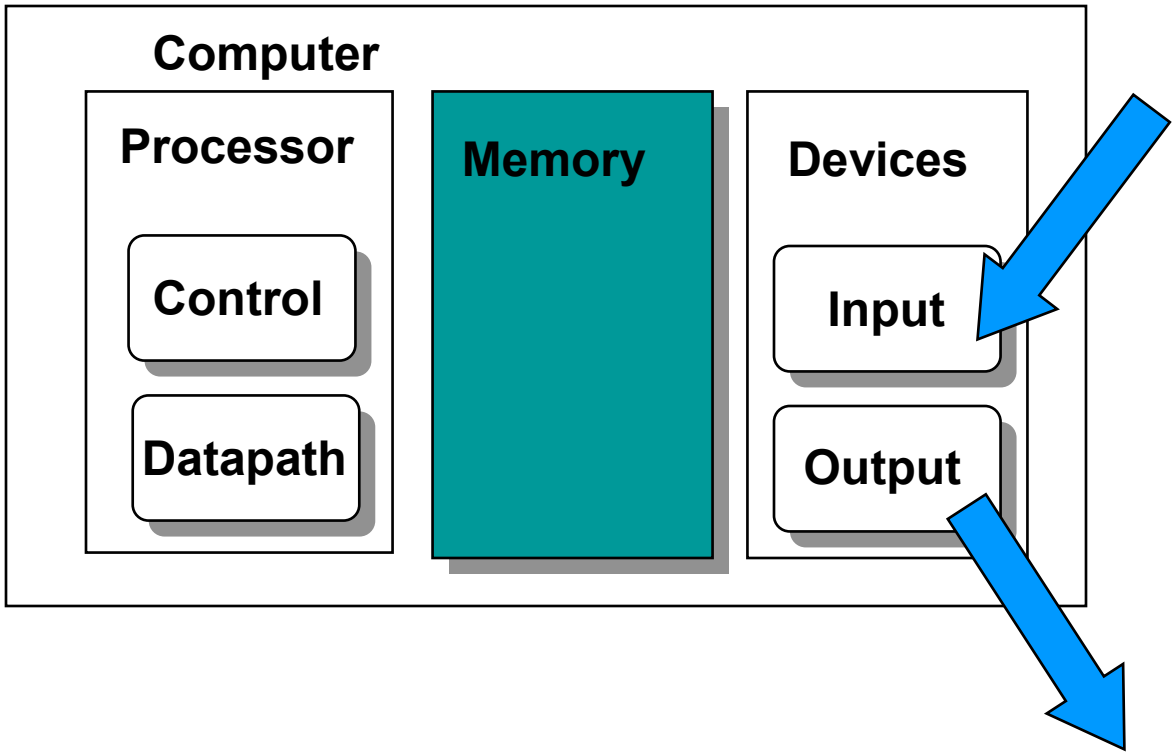
- Limited ILP in real programs
- Pipeline overhead
  - Branch and load delays exacerbated
  - Clock cycle timing limits



- Limited branch prediction accuracy (85%-98%)
  - Even a few percent really hurts with long/wide pipes!
- Memory inefficiency
  - Load delays + # of loads/cycle
- Complexity of implementation
  - Too expensive to design
  - Requires too much power management

- We have reached the limits today!

Five Components



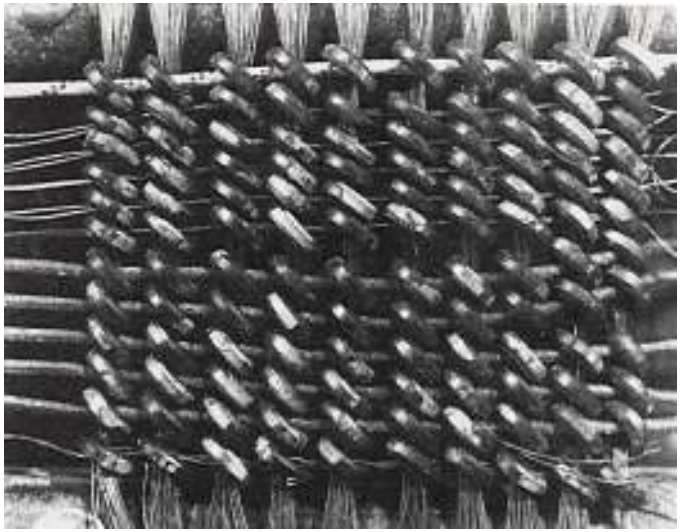
- Datapath
- Control
- Memory
- Input
- Output

Memory Systems Outline

- Memory System Basics
  - A little history
  - Memory technologies
    - SRAM, DRAM, Disk
- Principle of locality
  - Program characteristics
- Memory Hierarchies
  - Exploiting locality

In The Beginning

- In the earliest electronic computing memory was very hard
  - Memory devices used dots on a CRT screen, and stuff that was even worse!
  - In late 40's core memory was invented
    - This was the dominant memory until the mid 70's
    - Threaded by hand!



http://www.columbia.edu/acis/history/core.html

- Early semiconductor memory
  - Static memory
    - 256 bits
  - Dynamic memory
    - MOS,
    - 1K bits in 1970, i1103

DRAMs

- In the late 70's a type of memory became cheapest/bit
  - DRAM – dynamic random access memory
- It was built in MOS technology
  - So it was reasonably fast
- Processors could just access that memory when they needed it
  - Processors were slower than memory

- But as technology scaled:
  - Memories got denser and a little faster
  - Processors got a lot faster

- But before we talk about this problem,
  - Let's review the kinds of memory

Random Access Memory (RAM)

- Dynamic Random Access Memory (DRAM)
  - High density, low power, cheap, but slow
  - Dynamic since data must be “refreshed” regularly (“leaky buckets”)
  - Contents are lost when power is lost
- Static Random Access Memory (SRAM)
  - Lower density, (about 1/10 density of DRAM)
    - This means it is going to cost more
  - Static since data is held “forever” and do not require a refresh
    - But only when the power is on
  - Fast access time, often 2 to 10 times faster than DRAM
- Flash memory
  - Holds contents without power
  - Data written in blocks
  - Generally slow

Actually Situation Is More Complex

- There really is even cheaper storage
  - That does not need power to remember data
  - This is a disk drive
  - Today this costs about $1/GB

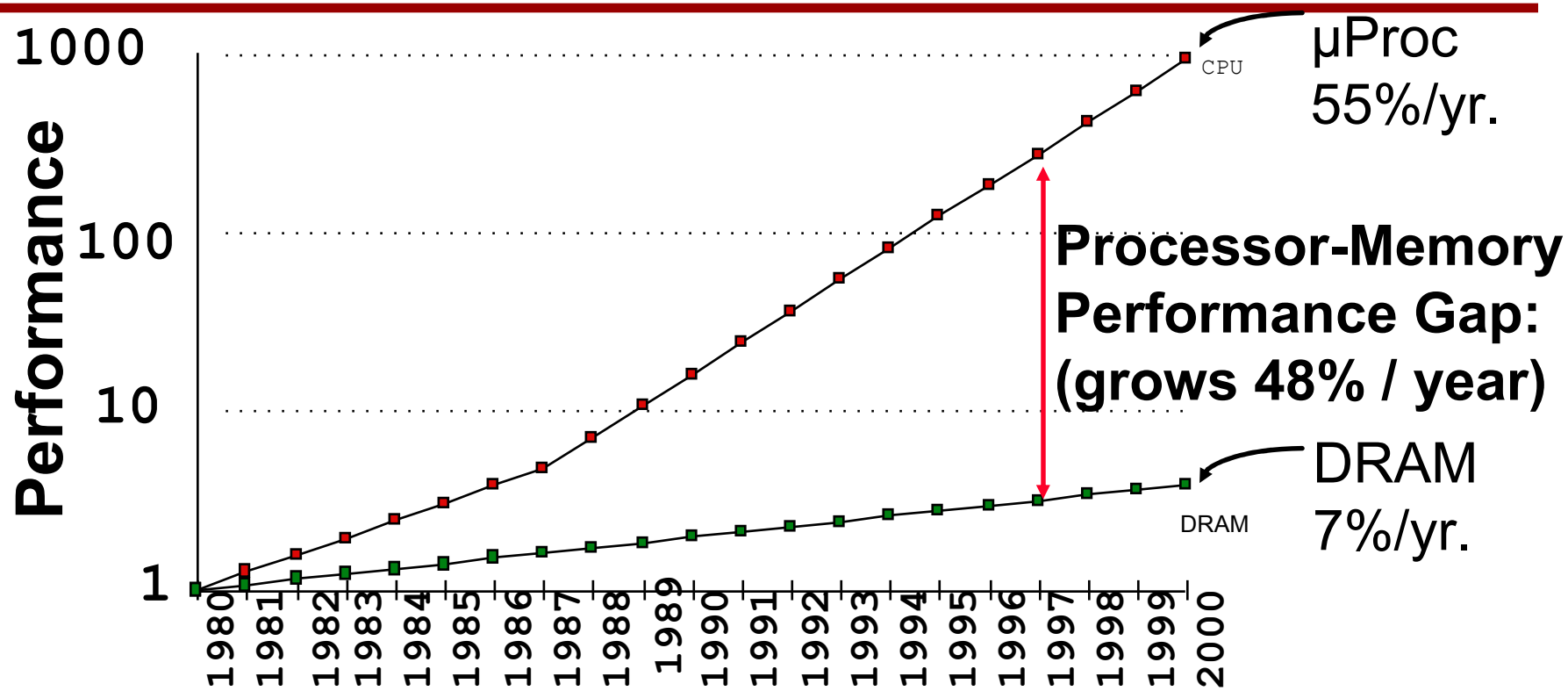
- The access to these type of devices is different
  - Block access
    - Mechanically move a arm to the right track
    - Wait for the data you want to rotate under the head
  - Very slow access times (ms)
    - Reasonable transfer rates

Memory Speeds and Costs

- The cheaper the memory,
  - The slower the access time

Technology	Access Time	¢/MB in 2002
SRAM	.5-25 ns	¢30 - ¢50
DRAM	60-120 ns	¢10 - ¢20
Disk	5-10 ms	¢0.1 - ¢0.3

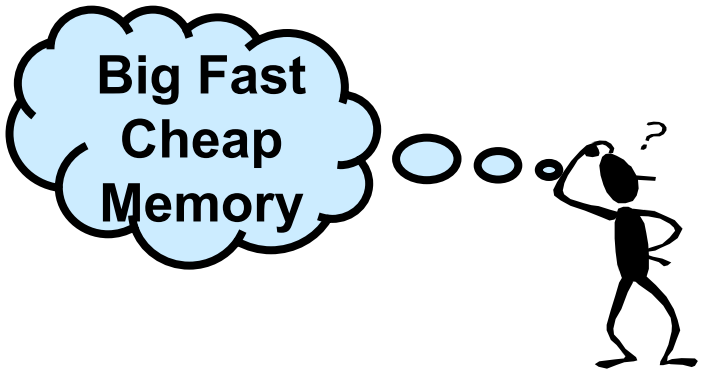
The Processor-Memory Performance Gap



- A single-cycle access to DRAM memory in 1980 takes 100s of cycles in 2002
- Can we afford to stall the pipeline for that long?

The Memory Problem

- Build a big, fast, cheap memory
- Big memories are slow
  - Even when built from fast components
- Fast memories are expensive



Programs Have Locality

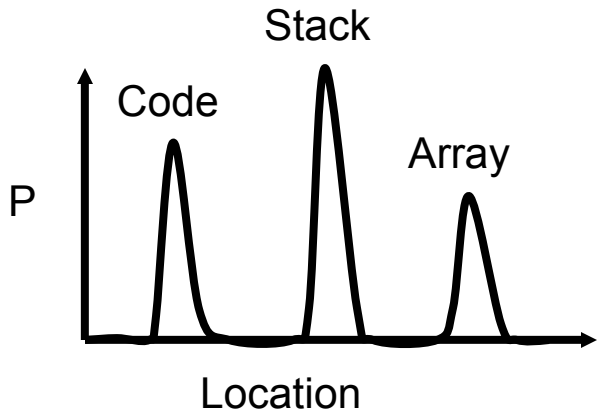
- Principle of Locality
  - Programs access a relatively small portion of the address space at any given time
  - Can tell what memory locations a program will reference in the future by looking at what it referenced recently in the past

- Two Types of Locality
  - **Temporal Locality** - If an item has been referenced recently, it will tend to be referenced again soon
  - **Spatial Locality** - If an item has been referenced recently, nearby items will tend to be referenced soon
    - Nearby refers to memory addresses
  - Examples

Locality Examples

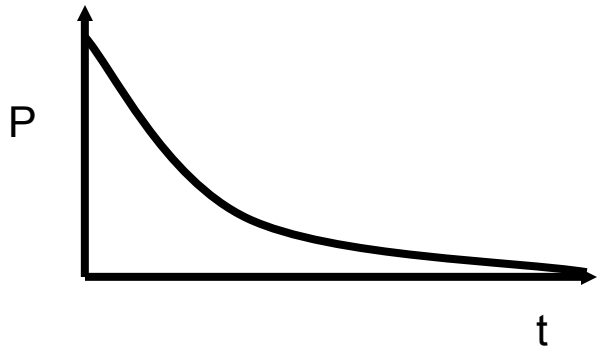
- Spatial Locality
  - Likely to reference data near recent references
  - Example:

```
for (i=0; i<N; i++)
        a[i]= …
```



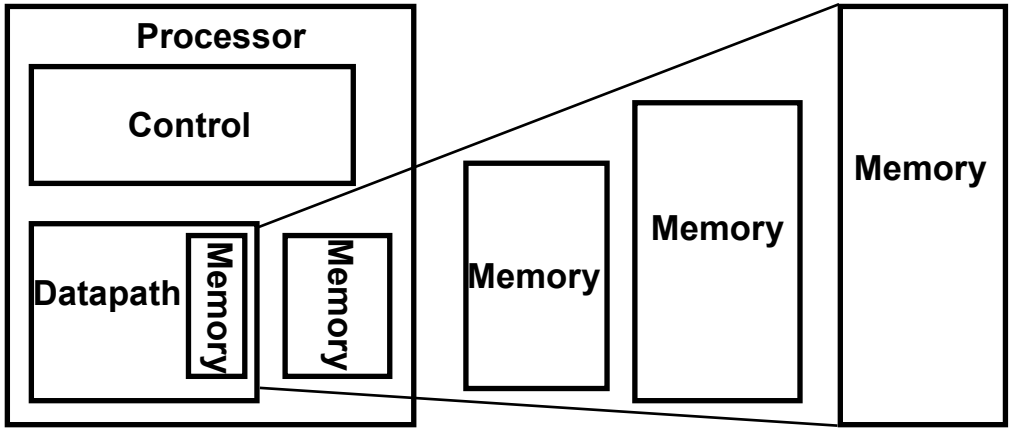
- Temporal Locality
  - Likely to reference the same data that was referenced recently
  - Example:

```
for (i=1; i<N; i++)
    a[i]= f(a[i-1]);
```



The Solution

- Memory can be arranged as hierarchies
- The goal is to provide the illusion of lots of fast memory
  - But how do you manage this, and make it work?

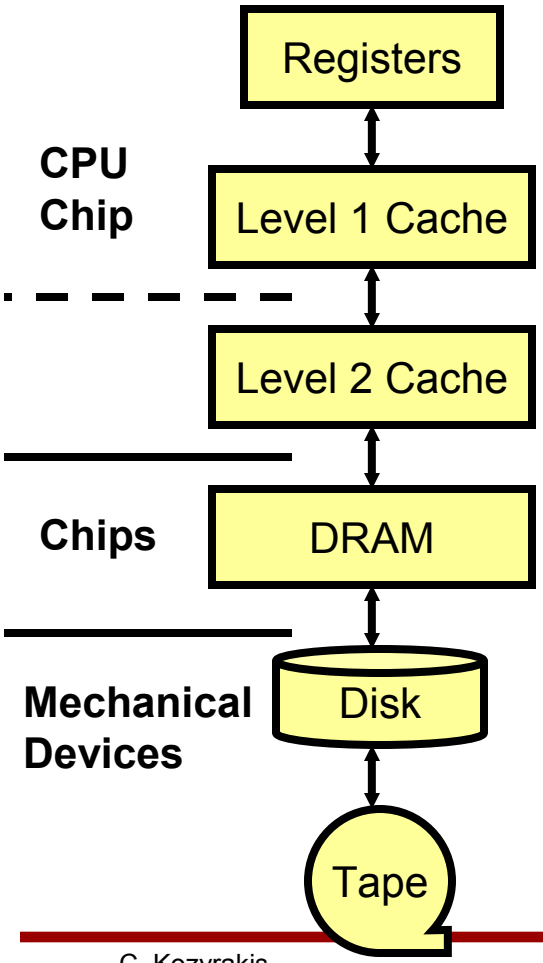


Speed:	**Fastest**	**Slowest**
Size:	**Smallest**	**Biggest**
Cost:	**Highest**	**Lowest**

Caches

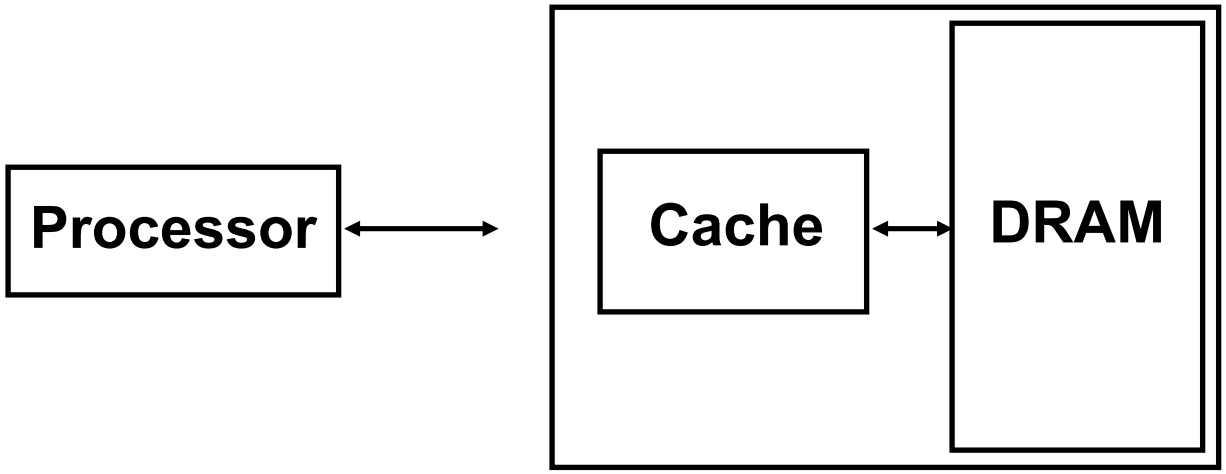
- A cache is an interim storage component
  - Functions as a buffer for larger, slower storage components
- Exploits principle of locality
  - Provide as much inexpensive storage space as possible
  - Offer access speed equivalent to the fastest memory
    - For data in the cache
    - Key is to have the right data cached
- Computer systems often use multiple caches
- Cache ideas are not limited to hardware designers
  - Example: Web caches widely used on the Internet

Typical Memory Hierarchy: Everything is a Cache for Something Else



	Access time	Capacity	Managed by
Registers	1cycle	~500B	software/compiler
Level 1 Cache	1-3cycles	~64KB	hardware
Level 2 Cache	5-10cycles	1-10MB	hardware
DRAM	~100cycles	~10GB	Software/OS
Disk	10^6-10^7cycles	~100GB	software/OS

Cache Goal: Transparent Access



- Processor will send references to the “cache”
  - If it has the data (a “hit”), it returns it quickly
  - If it does not have the data,
    - The reference “misses” in the cache
    - It gets the data from the DRAM
- Processor must deal with variable access time for memory

Want low miss rates

Average Memory Access Times

- Need to define an average access time
  - Since some will be fast and some slow

    Access time = hit time + miss rate $\times$ miss penalty

  - The hope is that the hit time will be low and the miss rate low since the miss penalty is so much larger than the hit time

- Average Memory Access Time (AMAT)
  - Formula can be applied to any level of the hierarchy
    - Access time for that level
  - Can be generalized for the entire hierarchy
    - Average access time that the processor sees for a reference

How Processor Handles a Miss

- Assume that cache access occurs in 1 cycle
  - Hit is great, and basic pipeline is fine

    CPI penalty = ~~hit time +~~ miss rate × miss penalty

- A miss stalls the pipeline
  - For an instruction or data miss:
    - Stall the pipeline (you don't have the data it needs)
    - Send the address that missed to the memory
    - Instruct main memory to perform a read and wait
    - When access complete, return the data to the processor
    - Restart the instruction

- But how do we get the data we need in the cache?

Exploiting Locality

- Need to update the contents of the cache to useful stuff
  - Leverage locality

- Spatial locality
  - Rather than fetching just the word that missed
  - Fetch a block of data around the word that missed
    - If you need these words (and you often do) they will now hit
    - This is also good since you can build memory systems that deliver large blocks of data once they access it (disk/DRAM)
- Temporal locality
  - Keep more recently accessed data items closer to the processor, so when we need space in the cache, evict the old ones

How To Build A Cache?

- Big question
  - I need to map a large address space into a small memory
    - Want associative lookup
- How do I do that?
  - Can build full associative lookup in hardware, but complex
  - Need to find a simple but effective solution
  - Two common techniques:
    - Direct Mapped
    - Set Associative

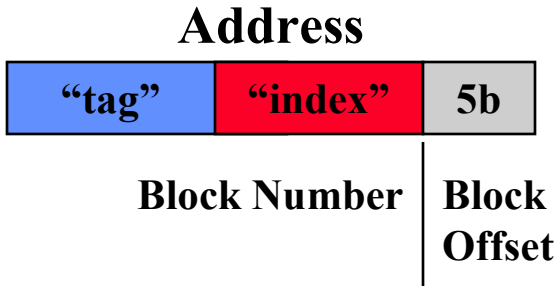
- Details
  - Block size
  - Write policy
  - Replacement policy

Terminology

- *Block* – Minimum unit of information transfer between levels of the hierarchy
  - Block addressing varies by technology at each level
  - Blocks are moved one level at a time
- *Hit* – Data appears in a block in upper level
- *Hit rate* – Percent of accesses found
- *Hit time* – Time to access at upper level
  - Hit time = Cache access time + Time to determine hit/miss
- *Miss* – Data was not in upper level and had to be fetched from a lower level
- *Miss rate* – Percent of misses (1 - Hit rate)
- *Miss penalty* – Overhead in getting data from a lower level
  - Miss penalty = Lower level access time + Time to deliver to upper level + Cache replacement / forward to processor time
  - Miss penalty is usually much larger than the hit time

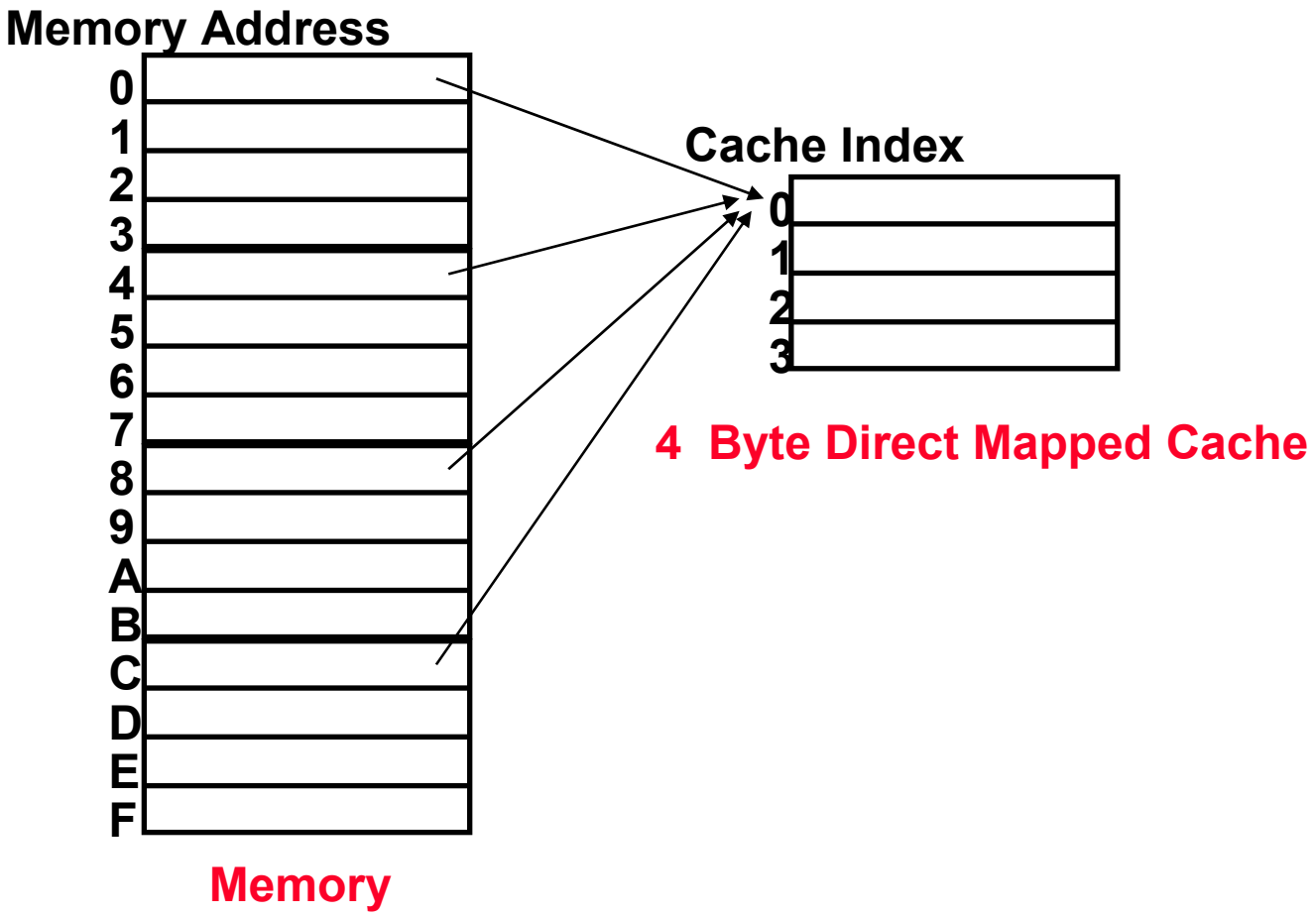
Direct Mapped Cache

- Cache uses a hash of address to index into the memory
  - Hash is generally very simple – lower order address bits
    - If you are fetching 32 byte blocks
    - Need one index per block
    - So "hash" starts after the 5 LSB address bits
    - It is the block number modulo cache size



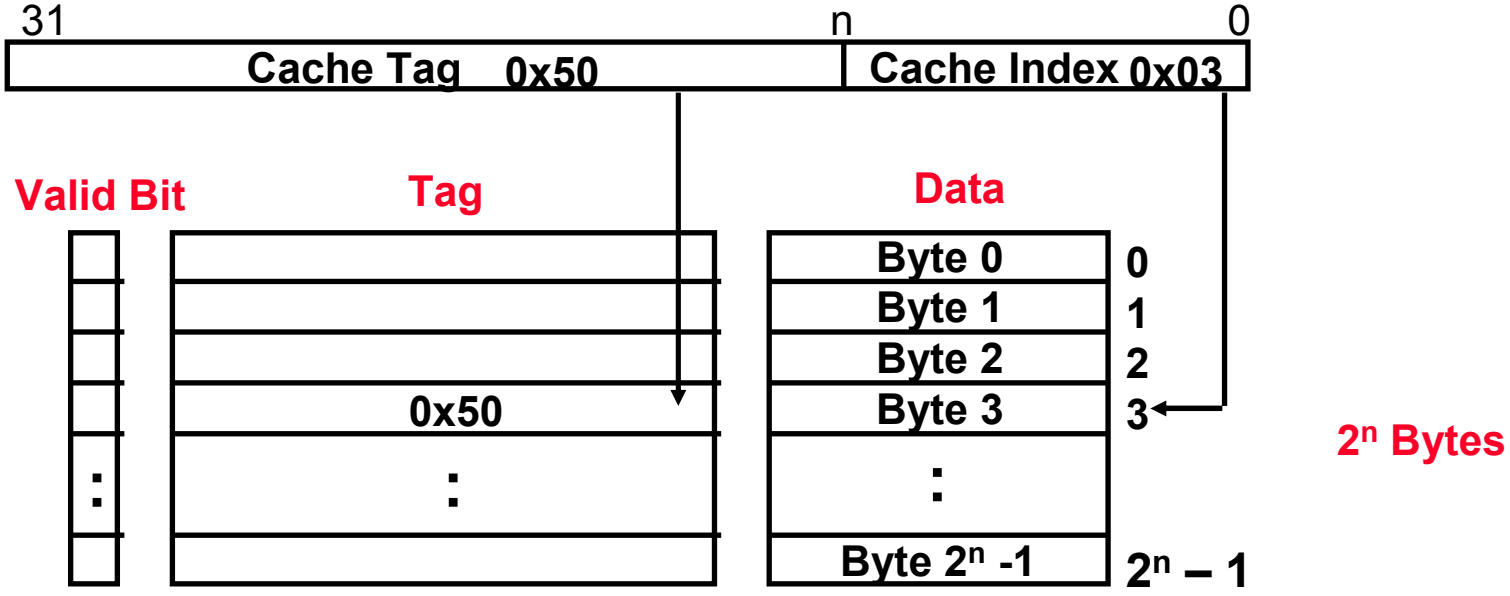
- Lookup is pretty simple
  - "Index" into the cache with block number modulo cache size
  - Read out both data, and "tag"
    - Tag is the rest of the address bits not used to index cache
  - Compare Tag with MSBs of address you want
  - If they match, you have a hit, otherwise it is a miss
  - Need tag to identify which data is being stored
  - Need valid bit for empty cache lines

Direct Mapped Cache



Cache Tag & Index

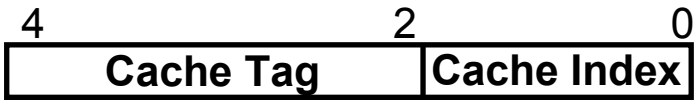
- Assume a 32 bit memory address
- Assume we also have a 2^n byte direct mapped cache with 1 byte blocks
  - Cache index - The lower n bits of the memory address
  - Cache tag - The upper (32-n) bits of the memory address

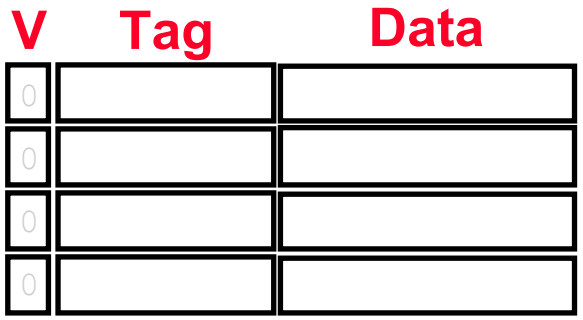


Cache Access Example (0)

- Assume a 4 byte direct-mapped cache with 1 byte blocks
- Consider the following 5 byte memory accesses

```
000 01
001 10
000 01
001 10
000 10
```

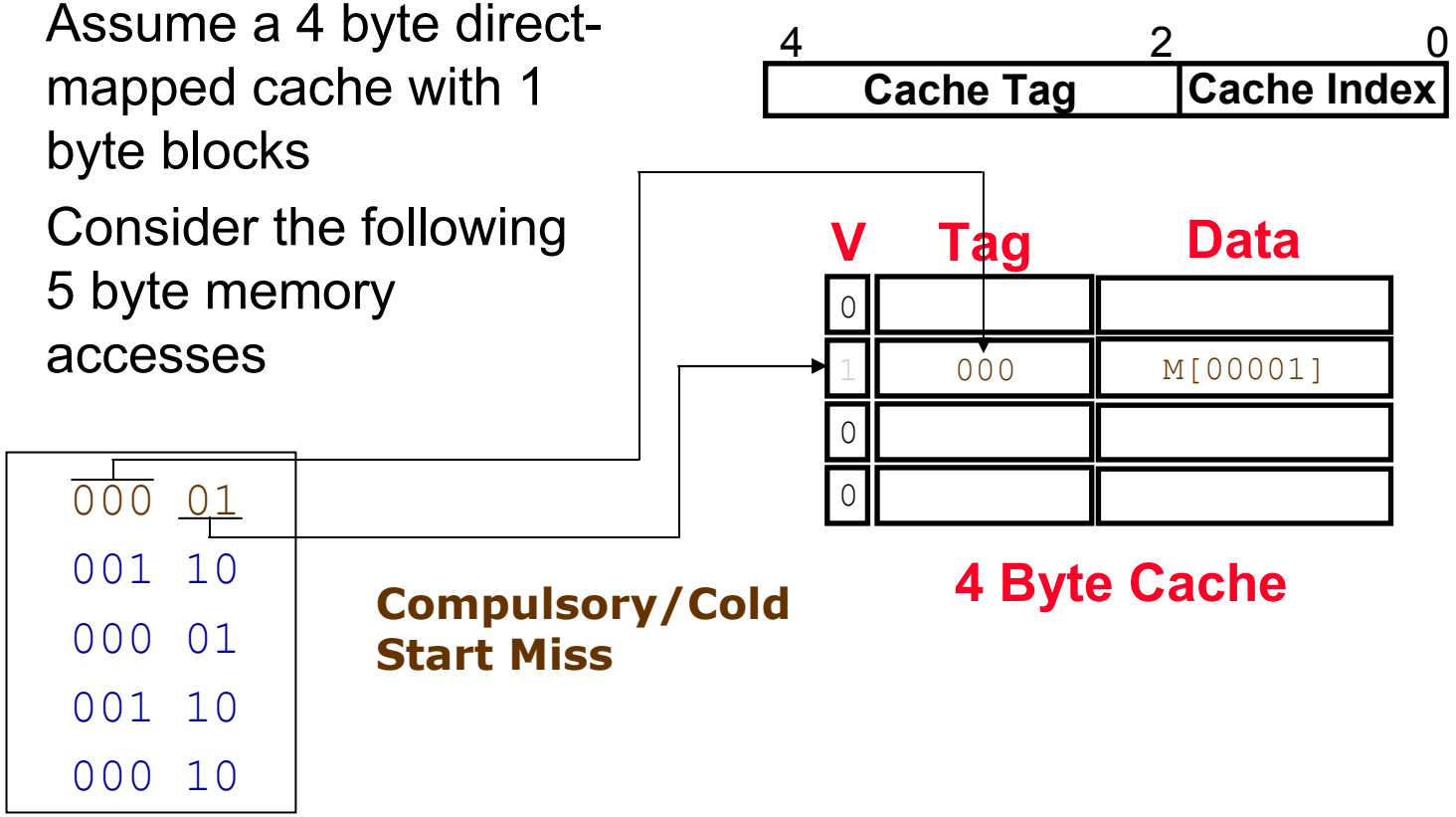




4 Byte Cache

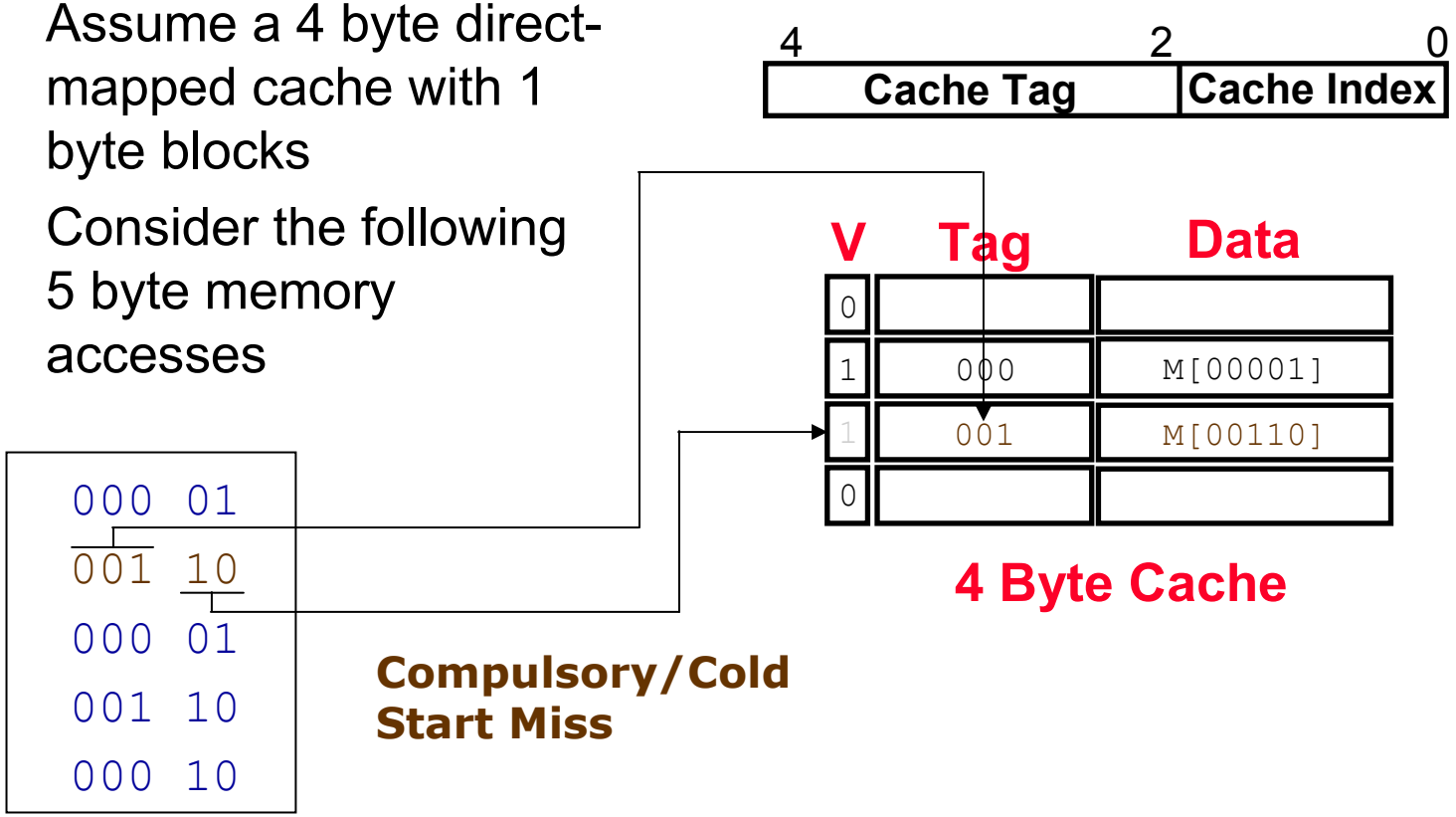
Cache Access Example (1)

- Assume a 4 byte direct-mapped cache with 1 byte blocks
- Consider the following 5 byte memory accesses



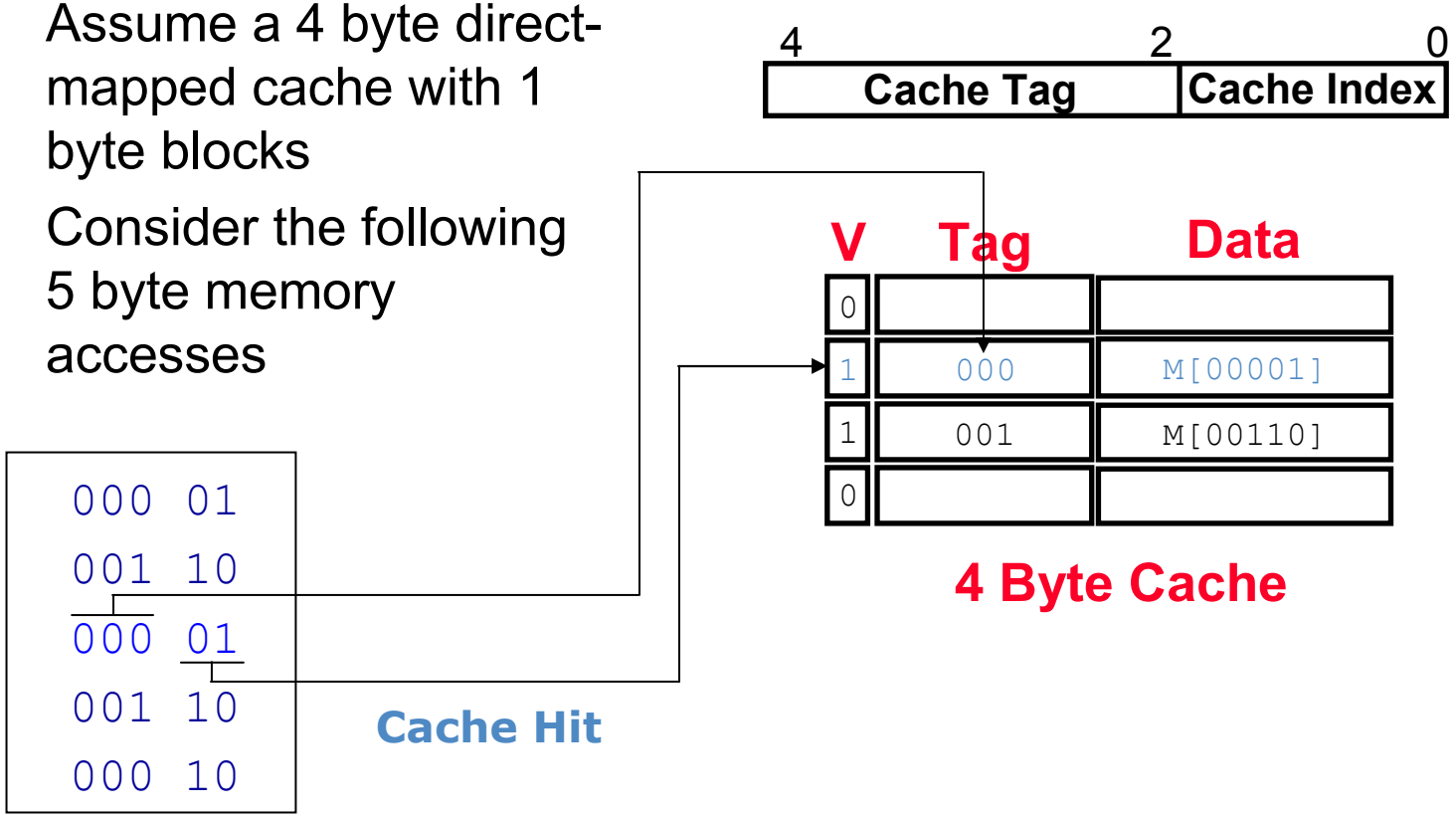
Cache Access Example (2)

- Assume a 4 byte direct-mapped cache with 1 byte blocks
- Consider the following 5 byte memory accesses



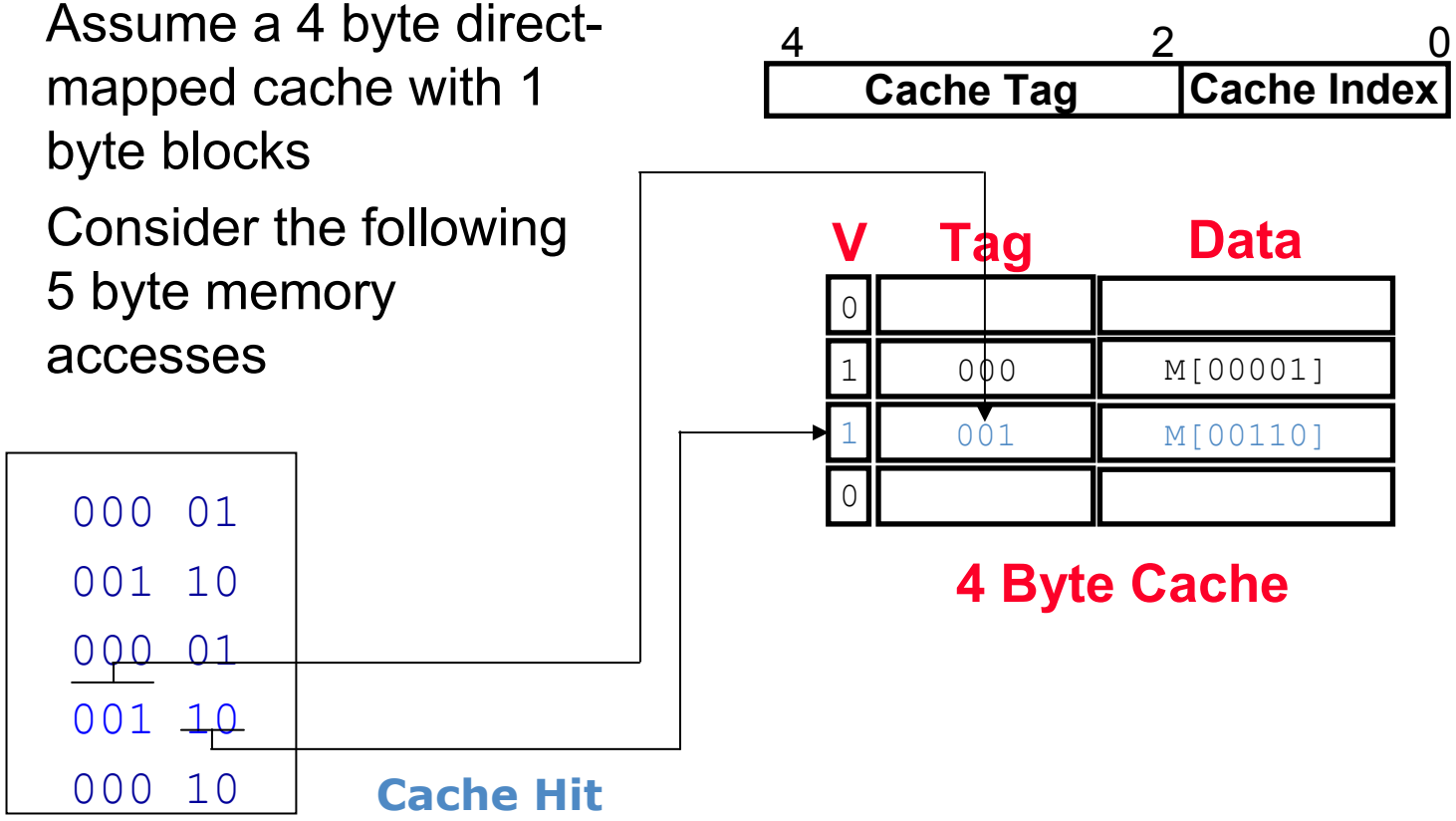
Cache Access Example (3)

- Assume a 4 byte direct-mapped cache with 1 byte blocks
- Consider the following 5 byte memory accesses



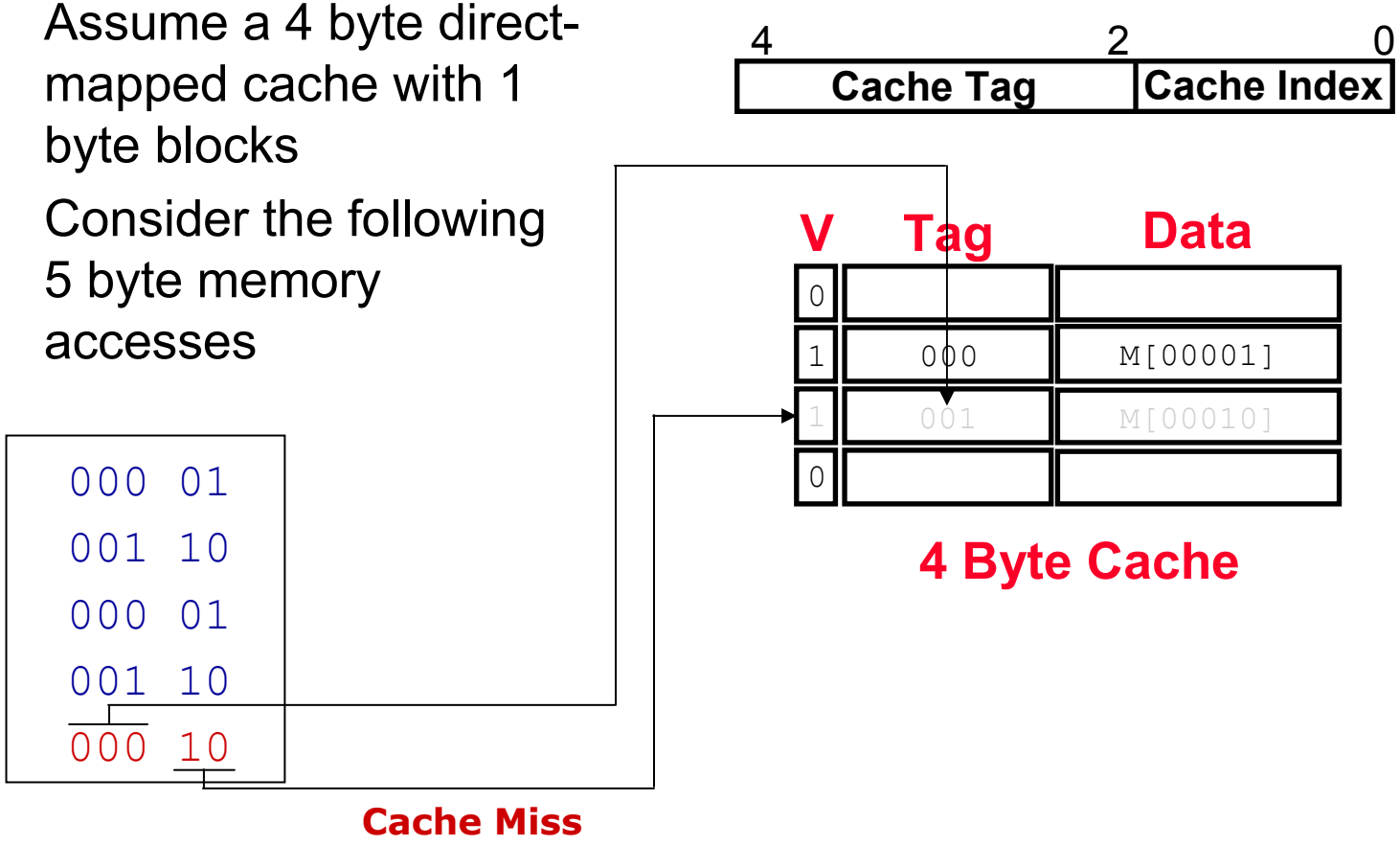
Cache Access Example (4)

- Assume a 4 byte direct-mapped cache with 1 byte blocks
- Consider the following 5 byte memory accesses



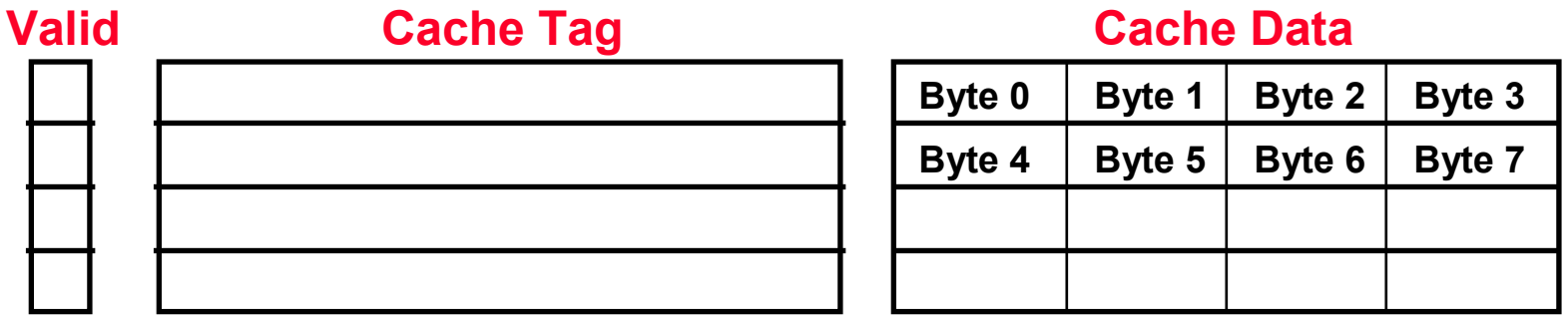
Cache Access Example (5)

- Assume a 4 byte direct-mapped cache with 1 byte blocks
- Consider the following 5 byte memory accesses



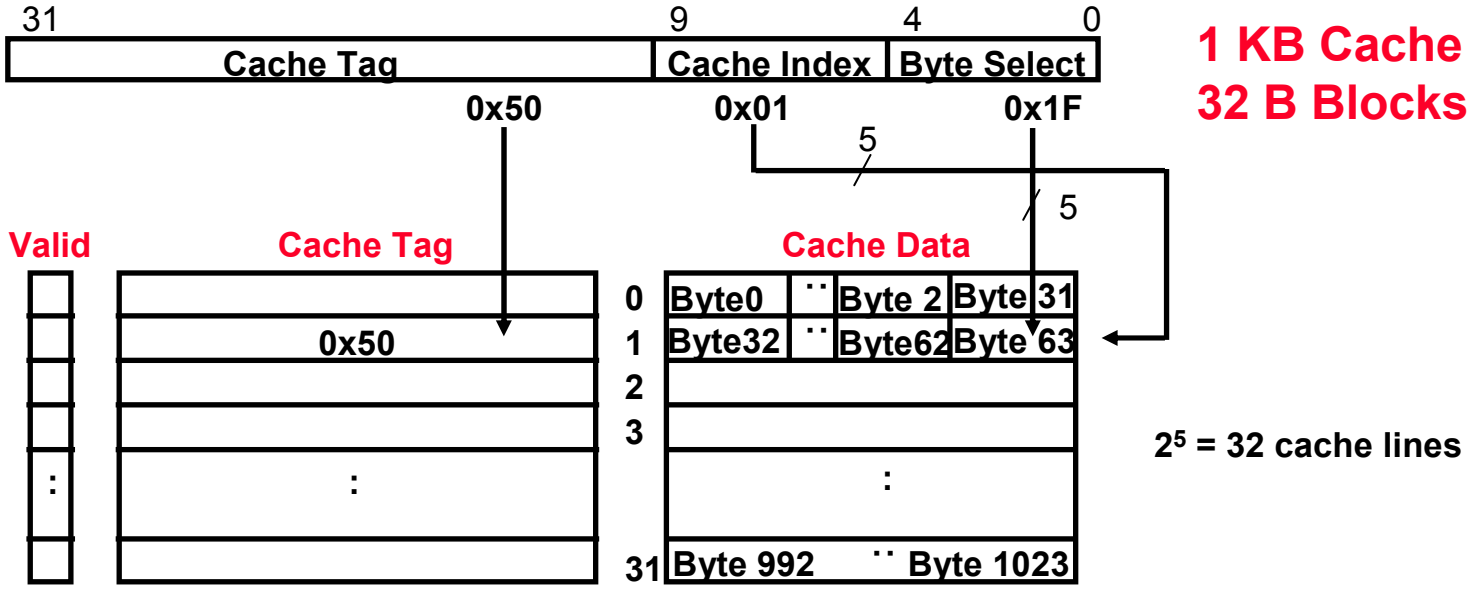
Cache Blocks

- Previous example was 4 Byte Direct Mapped Cache
  - Each block was 1 byte wide
  - Strategy took advantage of temporal locality since if a byte is referenced, it will tend to be referenced soon
  - Did not take advantage of spatial locality
- To take advantage of spatial locality, increase block size
  - Bonus: Reduces size of tag memory, too!



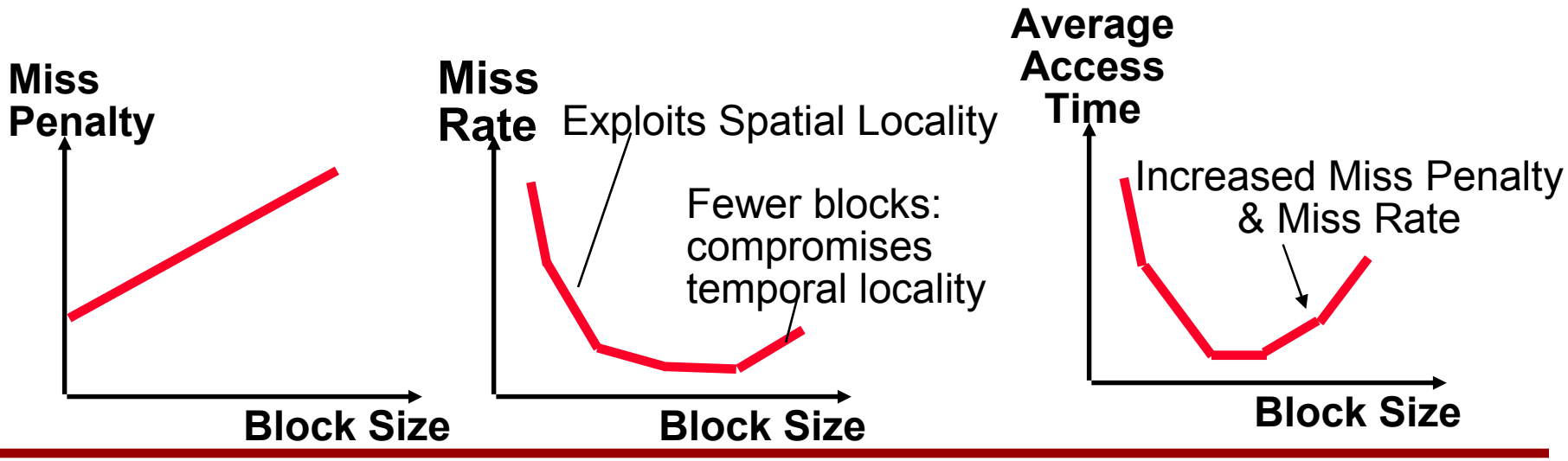
Cache Block Example

- Assume a 2^n byte direct mapped cache with 2^m byte blocks
  - Byte select – The lower m bits
  - Cache index - The lower (n-m) bits of the memory address
  - Cache tag - The upper (32-n) bits of the memory address



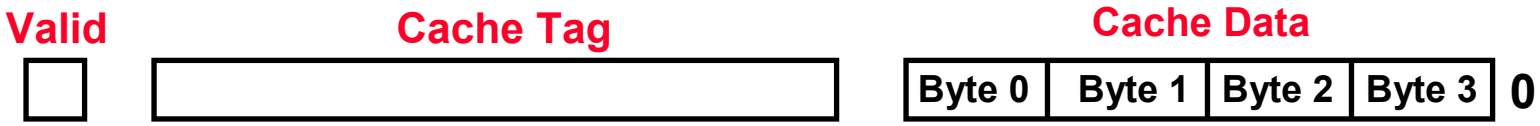
Block Sizes

- Larger block sizes take advantage of spatial locality
  - Also incurs larger miss penalty since it takes longer to transfer the block into the cache
  - Large block can also increase the average time or the miss rate
- Tradeoff in selecting block size
- Average Access Time = Hit Time • (1-MR) + Miss Penalty • MR



Direct Mapped Problems

- Conflict misses could in turn cause other conflict misses
- Consider an extreme example where both the cache size and block size are 4 bytes



  - Only one entry in the cache
  - Conflict misses will be continuous when accessing >1 line
- Such a situation is referred to as *thrashing* where cache continually loading data into the cache but evicting it before it can be used again
- How can the severity of the conflicts be reduced?

Fully Associative Cache

- Opposite extreme in that it has no cache index to hash
  - Use any available entry to store memory elements
  - No conflict misses, only capacity misses
  - Must compare cache tags of *all* entries in parallel to find the desired one