NYU TANDON SCHOOL OF ENGINEERING

Introduction to the Memory Hierachy

Azeez Bhavnagarwala

ECE 6913, Fall 2025

Computer Systems Architecture

NYU Tandon School of Engineering

Agenda

Project

- Cycle accurate 32b RISC V Simulator
  - *Phase 1 report* due **Friday Nov 7th** 11:59PM
  - ***Phase II discussion*** in Class on ***Tue Nov 6th***
- **Quiz 2** (**Sec B: Tue Nov 11th Sec A: Thu Nov 13th**) *Review problems* in Class on *Tue/Thu Nov 4th / 6th*
  - HW 3,4,5 relevant for Mid Term 2

Memory Hierarchy

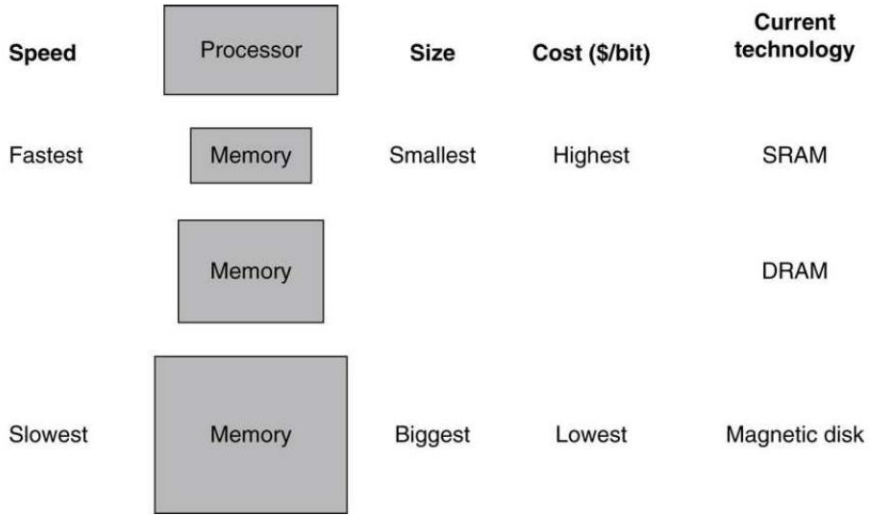
- Spatial & Temporal locality of data
- SRAM – the workhorse embedded memory technology for fast Cache memory
- Direct Mapped Cache Memory
- Write Policy
- Simple Cache Access Examples

Principle of Locality

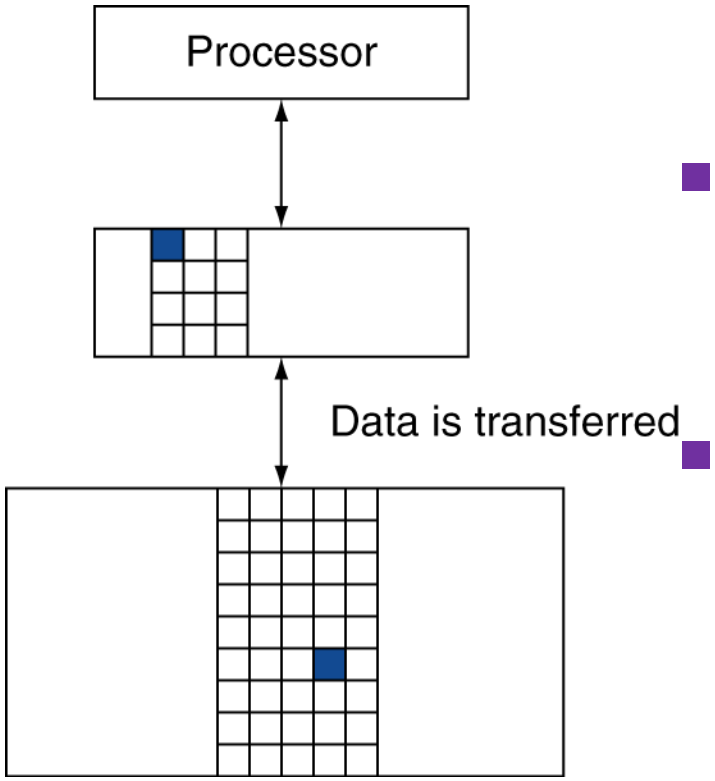
- Programs access a small proportion of their address space at any time
- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables
- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

Taking Advantage of Locality

- Memory hierarchy
- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
  - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
  - Cache memory attached to CPU



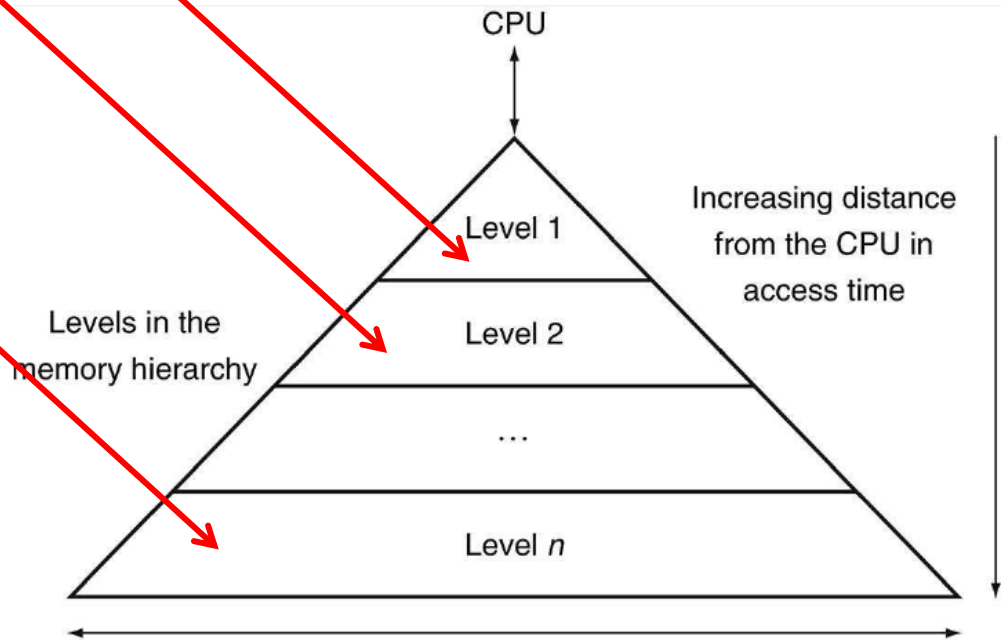
Memory Hierarchy Levels



- Block (aka line): unit of copying
  - May be multiple words
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses
      = 1 – hit ratio
  - Then accessed data supplied from upper level

Memory Technology

- Static RAM (SRAM)
  - 0.5ns – 2.5ns, $2000 – $5000 per GB
- Dynamic RAM (DRAM)
  - 50ns – 70ns, $20 – $75 per GB
- .....
- Magnetic disk
  - 5ms – 20ms, $0.20 – $2 per GB
- Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of disk

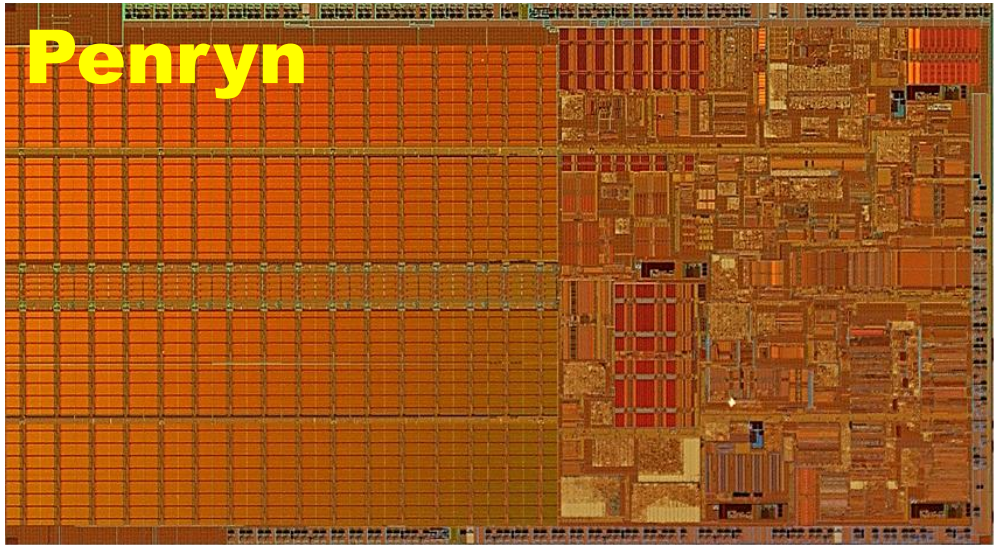


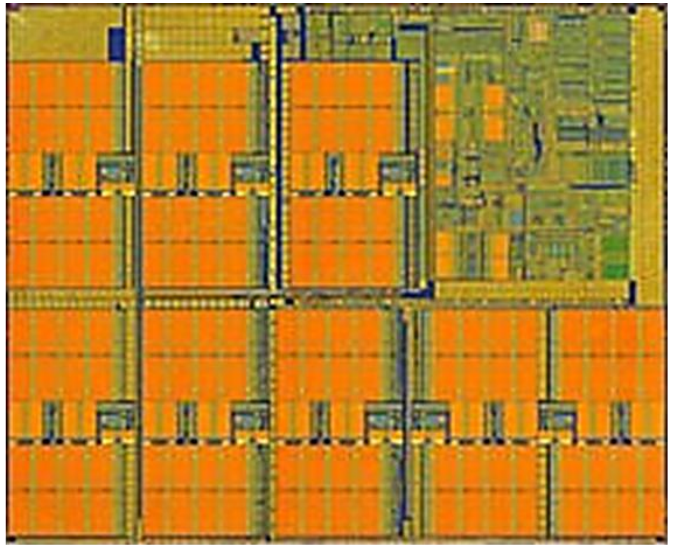
Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

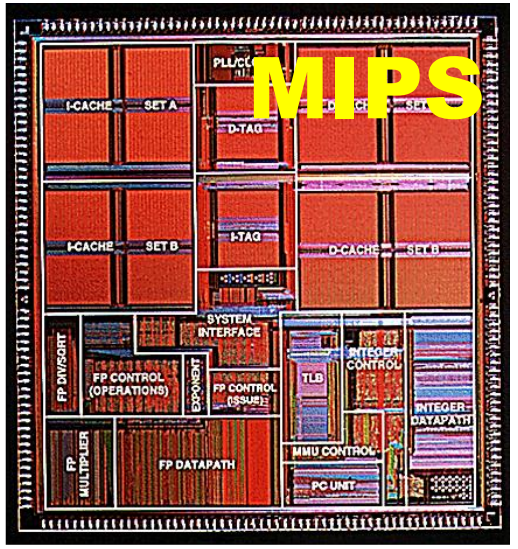
Why are SRAMs important?

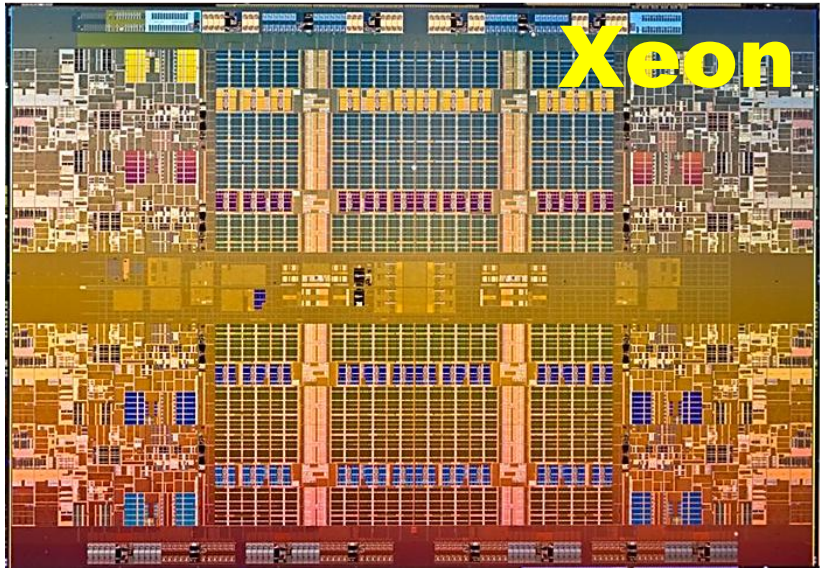
- SRAM Dominates area of Chip
  - Limits Chip Size & Cost, Chip wire-lengths
- SRAM dominates as largest Fraction of ‘off’ transistors during active mode
  - Limits leakage power of Processor
- Clock Cycle time limited by SRAM access
  - Access Latency Limits single-thread speed of CPU, Cycle Time
- SRAM Margins limit minimum Chip operating voltage
  - Minimum operating Voltage sets limits on Chip Energy Efficiency
  - Chip Yields limited by SRAM
  - New CMOS Logic Process node mostly limited by SRAM Tech Development

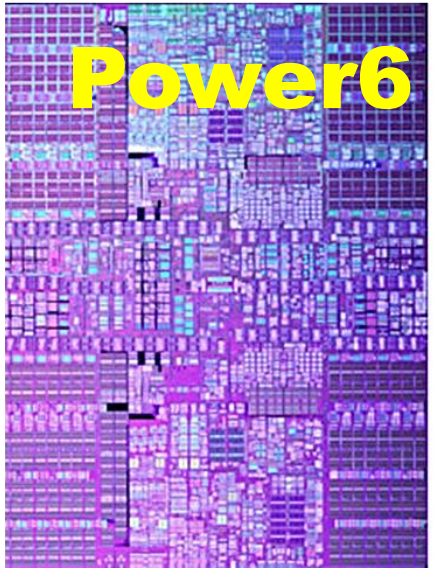
SRAM Dominates Die Size of Industry Leading CPUs

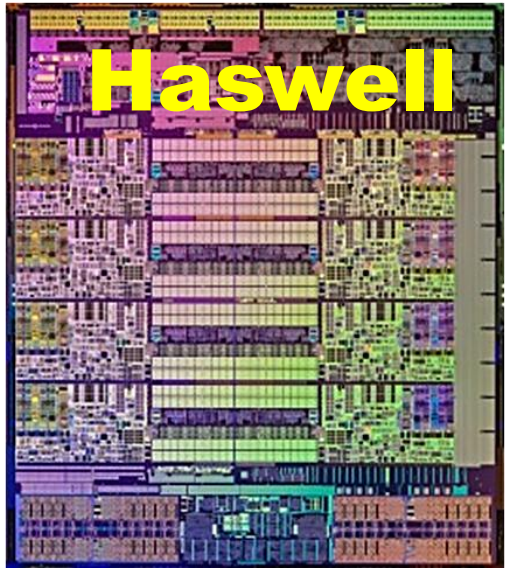


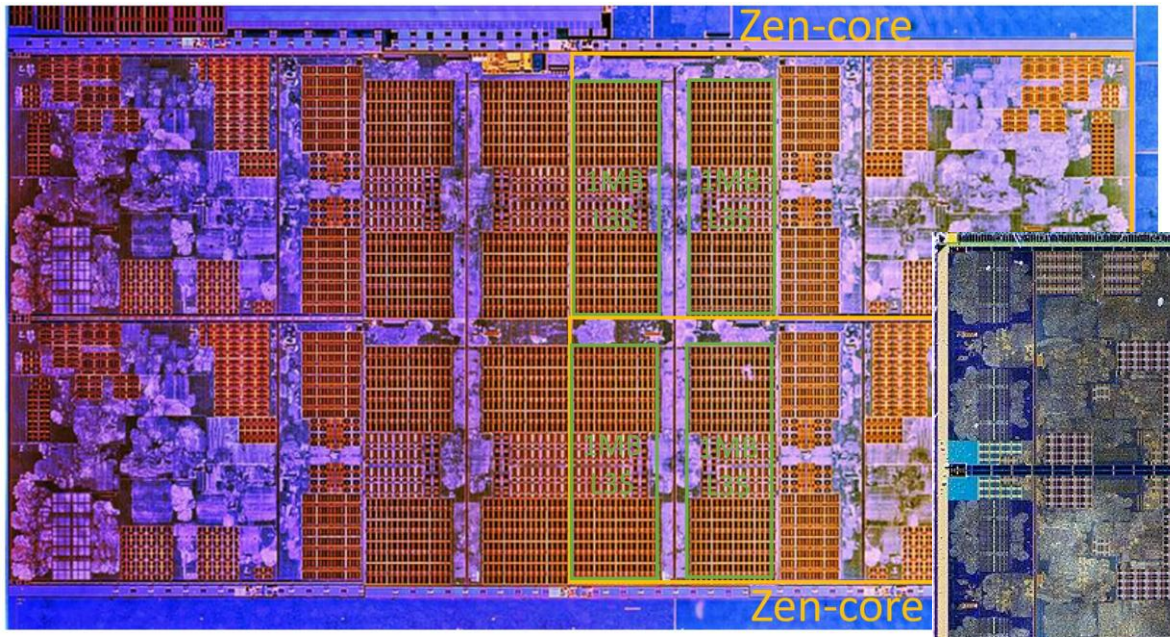


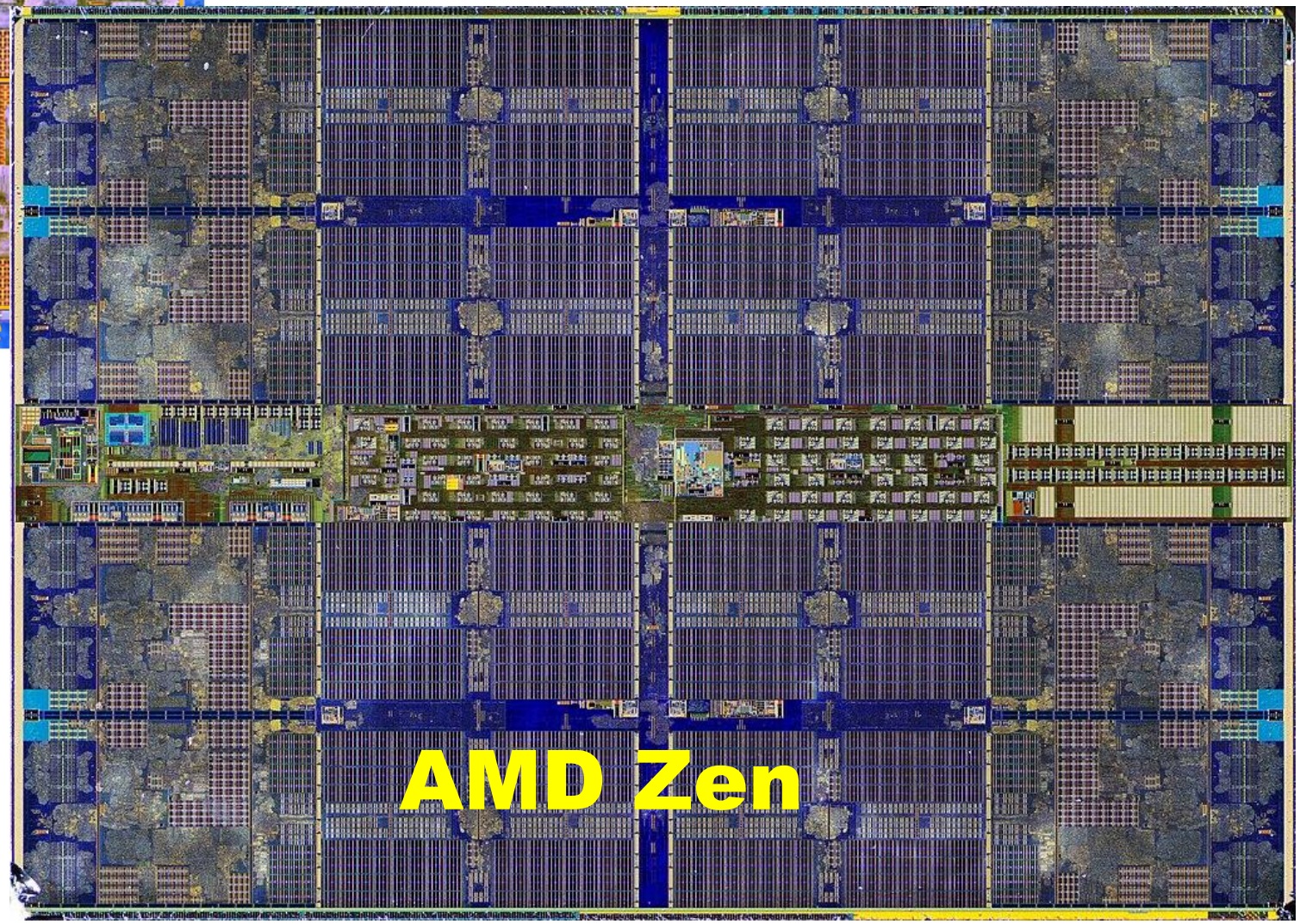








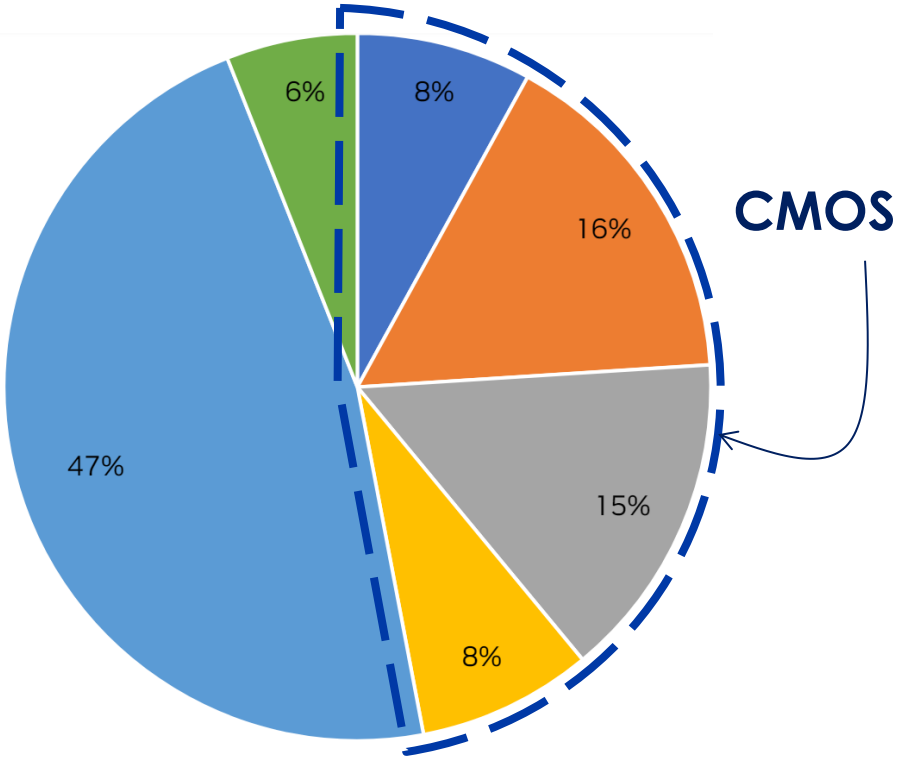

Zen-core
1MB L2
1MB L2
1MB L2
1MB L2
Zen-core


AMD Zen

Where is the Energy Going?

Bill Daly, Nvidia at Hot Chips 2023 Keynote

B. Keller et al, DL Inference accelerator prototype VLSI 22

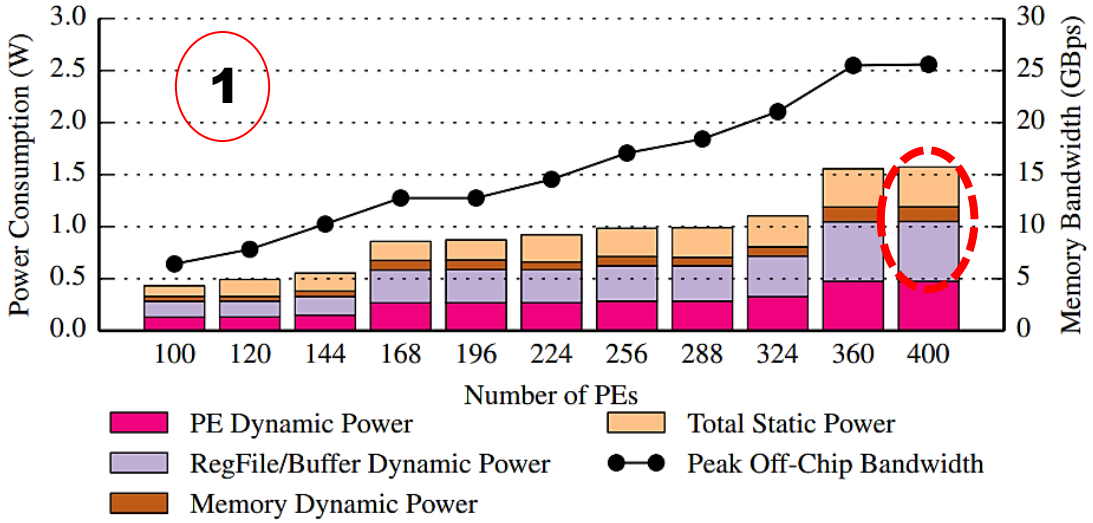


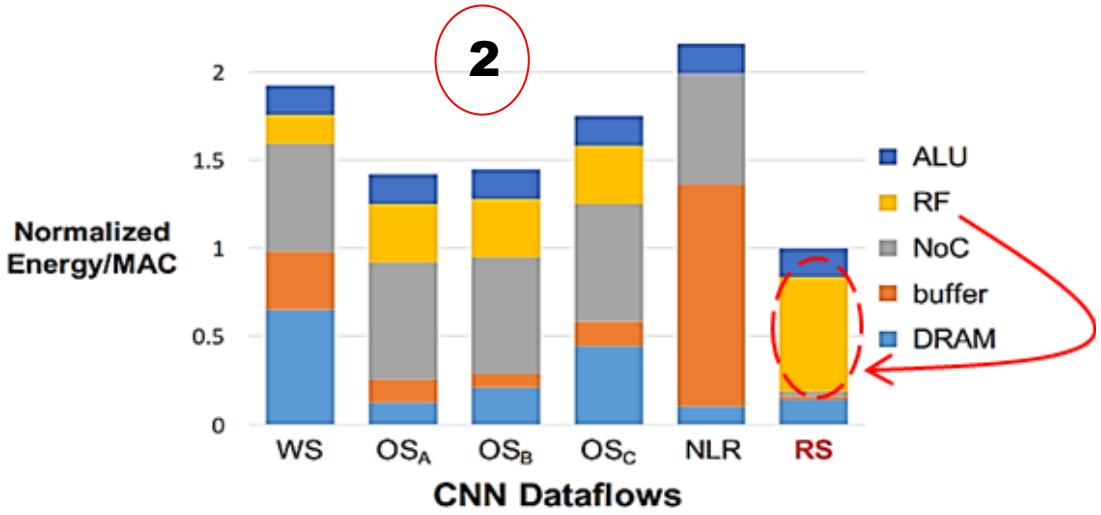
CMOS Memories: 47%

- Input Buffer
- Weight Buffer
- Accumulation Buffer
- Accumulation Collector
- Datapath + MAC
- Data Movement

CMOS Memories in Accelerators, GPUs

1. Over 2/3 of ASIC accelerator Energy (switching & leakage) consumed by SRAM buffers and Register File (RF) arrays
2. Almost 70% of MAC energy in a GPU consumed by RF SRAM arrays



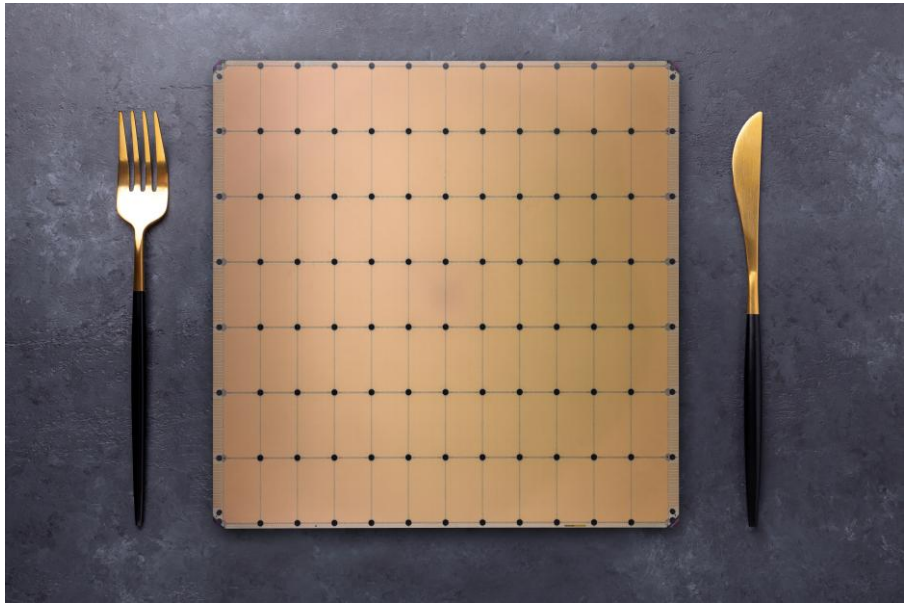


M Gao et al, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory", ASPLOS 2017, pg 751, April 2017

V Sze et al, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey" Proceedings of the IEEE, Vol 105, No. 12, Dec 2017

All SRAM Memory in Fastest AI ‘chip’

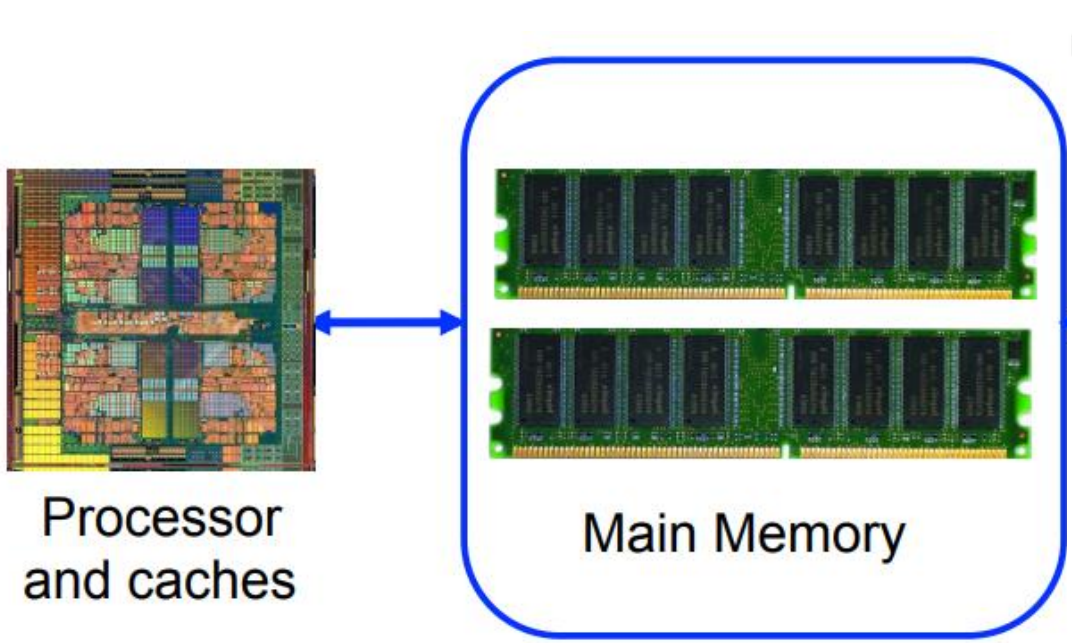
- AI Hardware limited by model size, on-chip bandwidth
- Energy efficiency limited by off-chip DRAM access
- SRAM only (44GB!) enables much lower latencies, much higher bandwidth within the mousepad sized chip (900K cores)

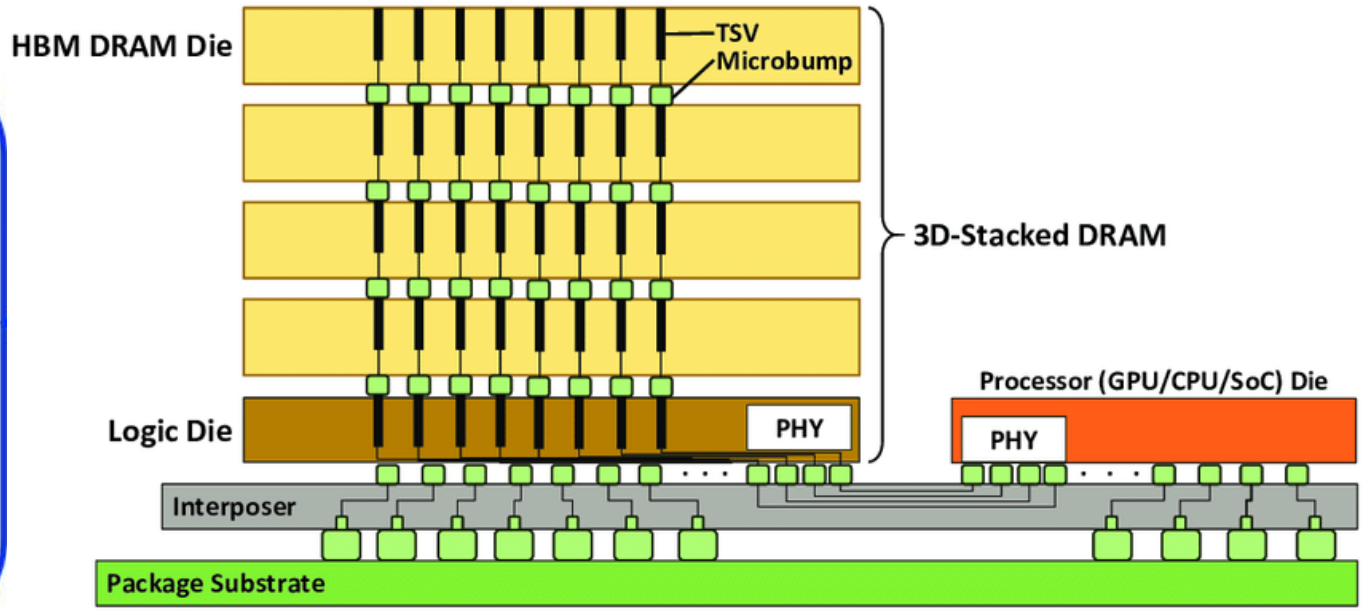


Spec	Cerebras CS-3	B200	DGX B200	GB200 NVL72	CS-3 / B200	CS-3 / DGX B200	CS-3 / NVL72
FP16 PFLOPs	125	4.4	36	360	28.4	3.5	0.3
Memory (GB)	1,200,000	192	1,536	13,500	6,250.0	781.3	88.9
NVLink | Fabric Bandwidth (TB/s)	26,750	1.8	14.4	130	14,861	1,858	206
Power (Watts)	23,000	1,000	14,300	120,000	23.0	1.6	0.2
PFLOPs / W	0.005	0.004	0.003	0.003	1.2	2.2	1.8

- 900,000 cores for sparse tensor operations
- Massive high bandwidth on-chip memory and interconnect orders of magnitude faster than a traditional cluster could possibly achieve

Main Memory

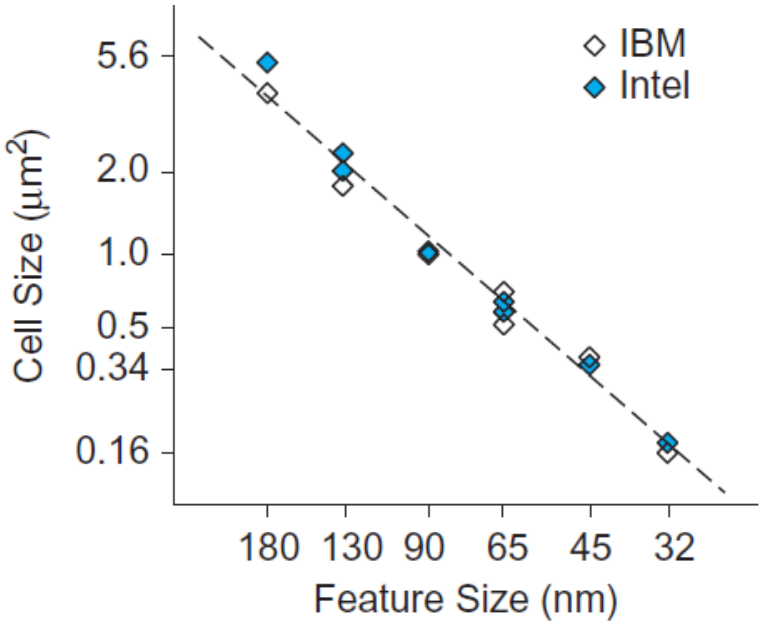




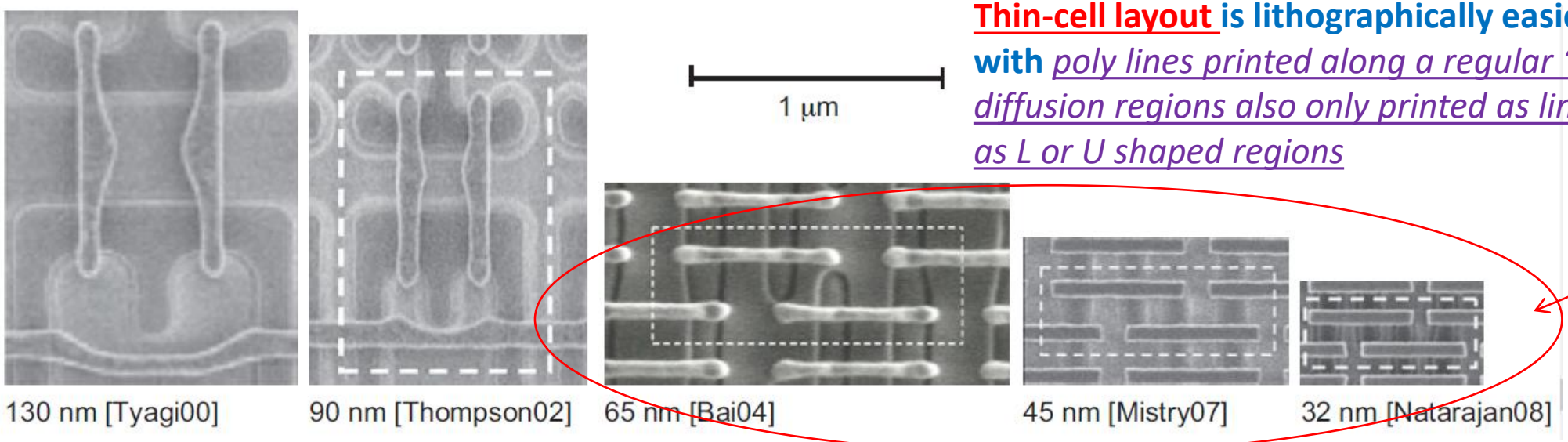
- Main memory is a critical component of all computing systems: server, mobile, embedded, desktop, sensor
- Main memory system must scale (in size, technology, efficiency, cost, and management algorithms) to maintain performance growth and technology scaling benefits

SRAM Cell Scaling

- 50% reduction in cell size each CMOS node
  - follows from 0.7x scaling of transistor dimensions
- Leakage, variability primary concerns as device geometries shrink and density increases
- 'Thin cell' layout standard today for litho-friendly cells

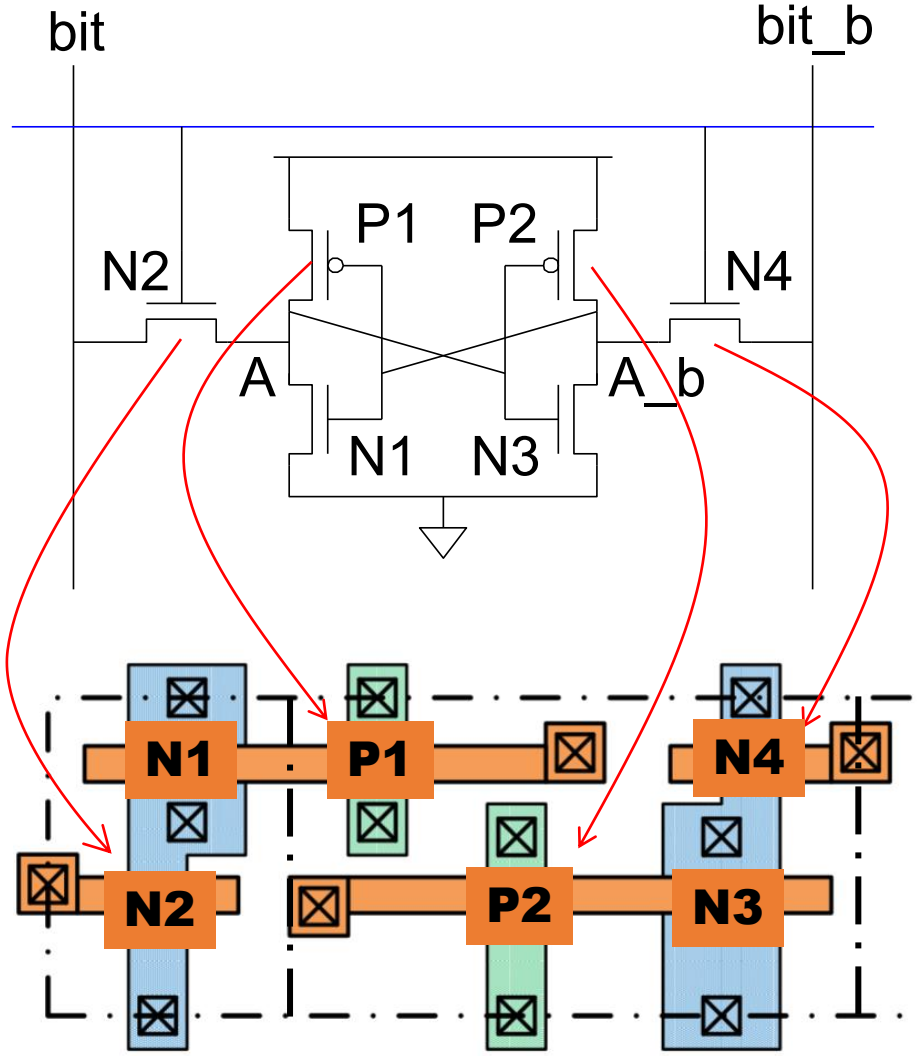


Thin-cell layout is lithographically easier to print with *poly lines printed along a regular 'pitch' and diffusion regions also only printed as lines and not as L or U shaped regions*



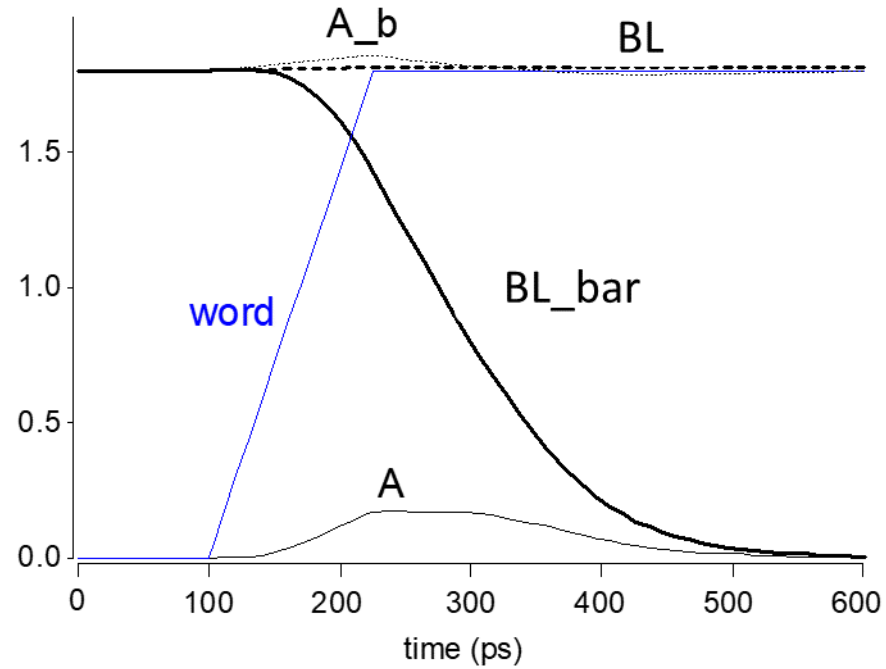
SRAM 6-Transistor 'Thin' cell

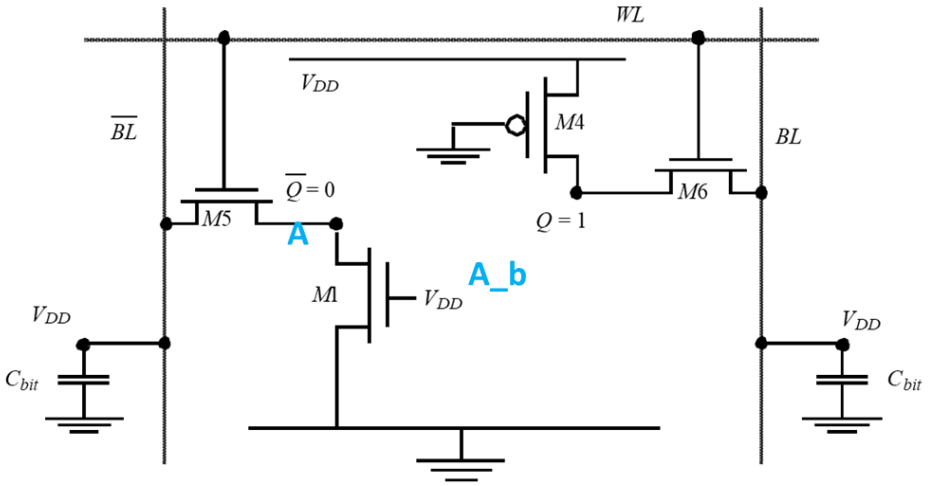
- Cell size accounts for most of array size
- Reduce cell size at expense of (process and design) complexity
- Data stored in cross-coupled inverters (N1-P1 an N3-P2)
- Access transistors N2 and N4 are selected by the WL connecting cell storage nodes to bitline pair
- Access transistors enable cell to build signal on precharged bitlines during Read
- Access transistors drive complementary data onto cell storage nodes using positive feedback of cross-coupled inverters to 'flip' the cell into intended state
- When WL=0, cells are in 'holding' data



SRAM Cell Read Operation

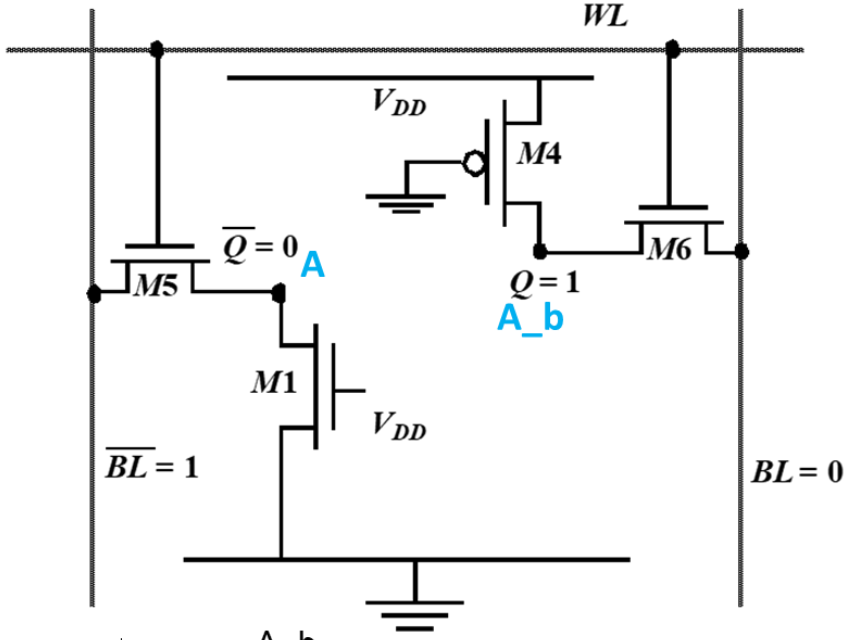
- Precharge both bitlines high
- Then turn on Word Line
- One of the two bitlines will be pulled down by the cell
  - BL_bar discharges, BL stays high
  - But A bumps up slightly
- *Read stability*
  - A, A_b must not flip

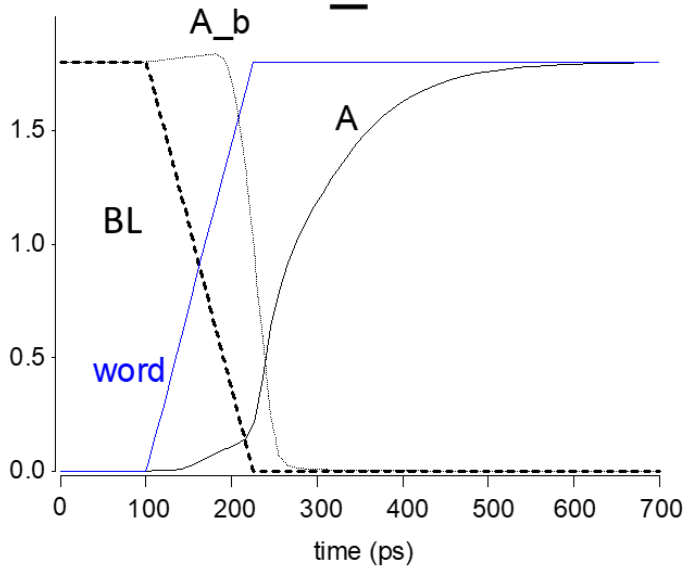




SRAM Cell Write Operation

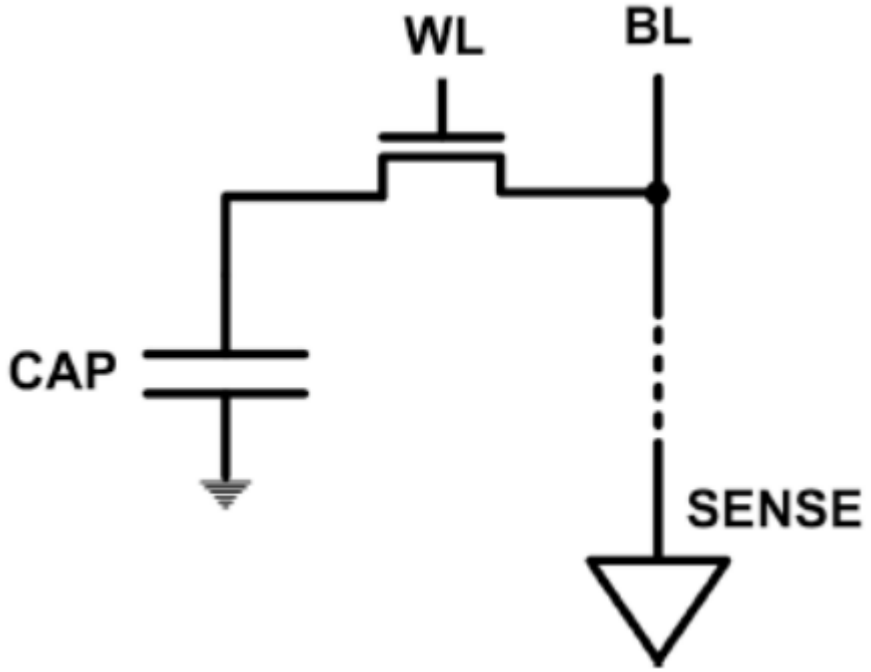
- Drive one bitline high, the other low
- Then turn on wordline
- Bitlines overpower cell with new value
- Ex: A = 0, A_b = 1, BL' = 1, BL= 0
  - Force A_b low, then A rises high
- *Writability*
  - Must overpower feedback inverter initially to enable positive feedback to complete the Write
  - M6 > M4





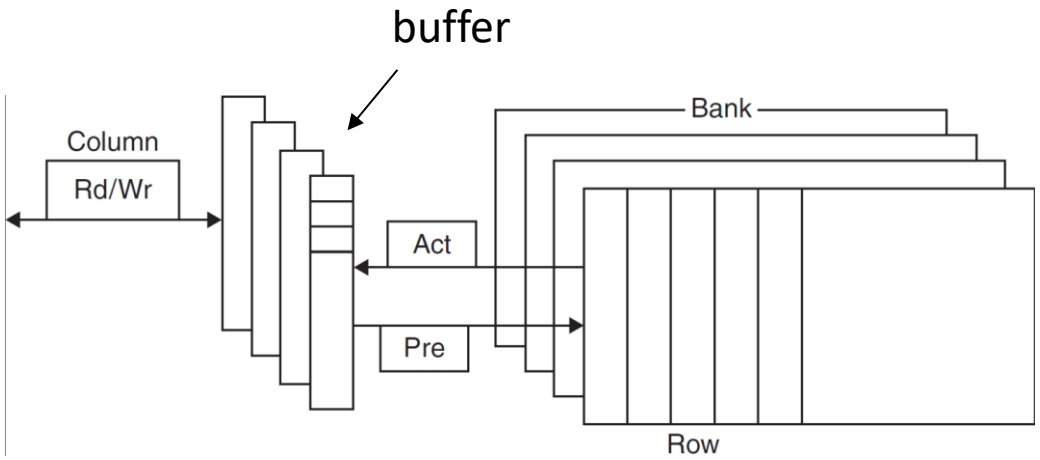
DRAM

- DRAM stores charge in a capacitor (charge-based memory)
  - Capacitor must be large enough for reliable sensing
  - Access transistor should be large enough for low leakage and high retention time



DRAM Access

- Single transistor used to access the charge
- Must periodically be refreshed
  - Read contents and write back
  - Performed on a DRAM “row”

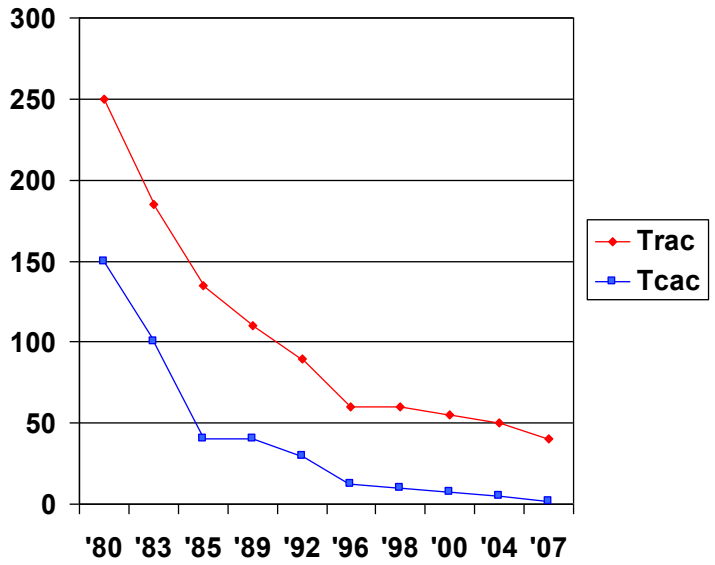


Advanced DRAM Organization

- Bits in a DRAM are organized as a rectangular array
  - DRAM accesses *an entire row*
  - *Burst mode*: supply successive words from a row with reduced latency
- Double data rate (DDR) DRAM
  - Transfer on rising and falling clock edges
- Quad data rate (QDR) DRAM
  - Separate DDR inputs and outputs

DRAM Generations

Year	Capacity	$/GB
1980	64Kbit	$1500000
1983	256Kbit	$500000
1985	1Mbit	$200000
1989	4Mbit	$50000
1992	16Mbit	$15000
1996	64Mbit	$10000
1998	128Mbit	$4000
2000	256Mbit	$1000
2004	512Mbit	$250
2007	1Gbit	$50



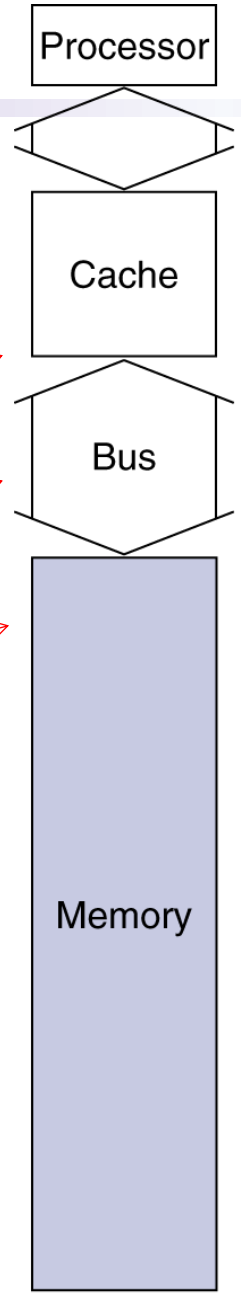
Year introduced	Chip size	$ per GiB	Total access time to a new row/column	Average column access time to existing row
1980	64 Kibibit	$1,500,000	250 ns	150 ns
1983	256 Kibibit	$500,000	185 ns	100 ns
1985	1 Mebibit	$200,000	135 ns	40 ns
1989	4 Mebibit	$50,000	110 ns	40 ns
1992	16 Mebibit	$15,000	90 ns	30 ns
1996	64 Mebibit	$10,000	60 ns	12 ns
1998	128 Mebibit	$4,000	60 ns	10 ns
2000	256 Mebibit	$1,000	55 ns	7 ns
2004	512 Mebibit	$250	50 ns	5 ns
2007	1 Gibibit	$50	45 ns	1.25 ns
2010	2 Gibibit	$30	40 ns	1 ns
2012	4 Gibibit	$1	35 ns	0.8 ns

DRAM Performance Factors

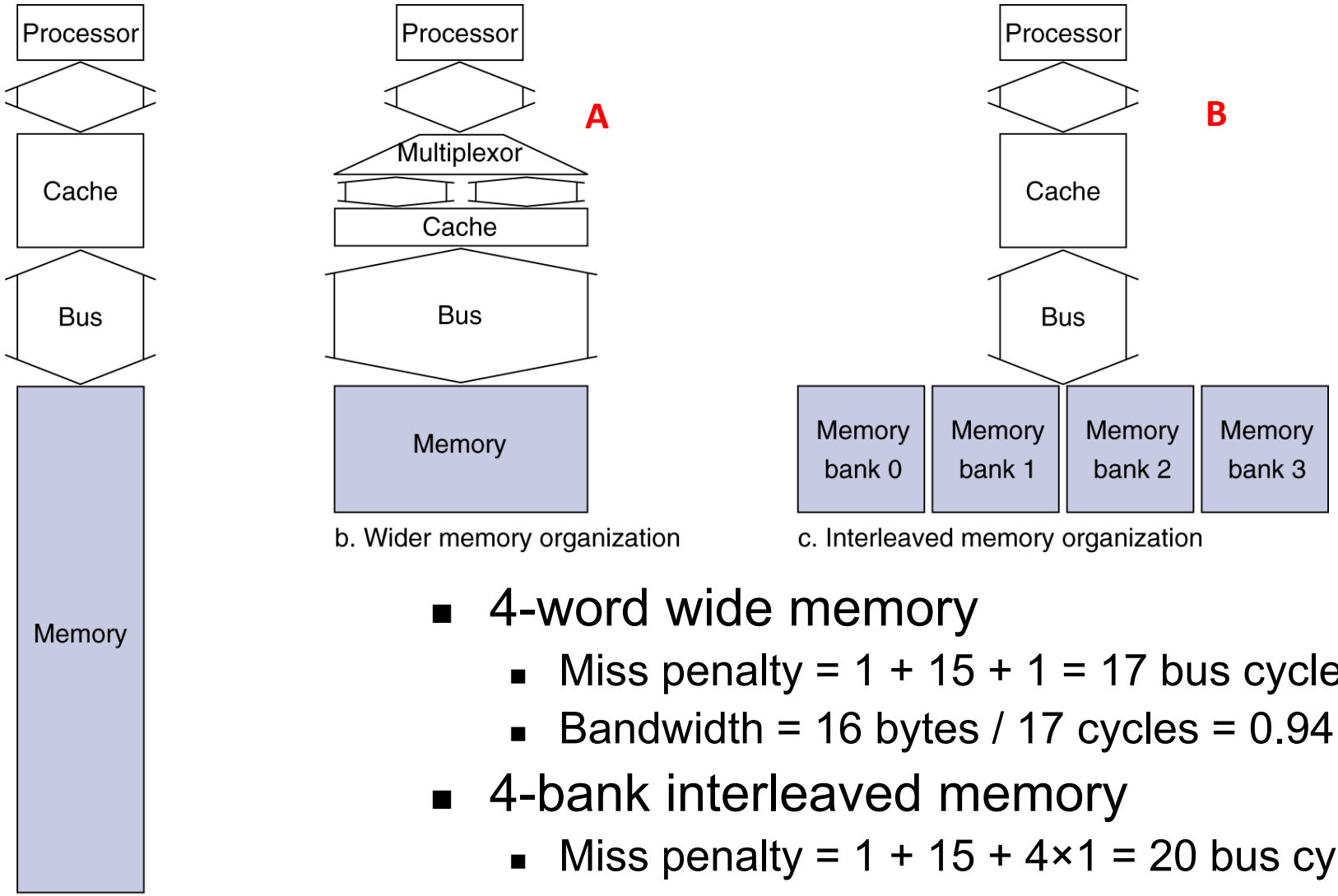
- Row buffer
  - Allows several words to be read and refreshed in parallel
- Synchronous DRAM
  - Allows for consecutive accesses in bursts without needing to send each address
  - Improves bandwidth
- DRAM banking
  - Allows simultaneous access to multiple DRAMs
  - Improves bandwidth

Main Memory Supporting Caches

- Use DRAMs for main memory
  - Fixed width (e.g., 1 word)
  - Connected by fixed-width clocked bus
    - Bus clock is typically slower than CPU clock
- Example cache block read
  - 1 bus cycle for address transfer
  - 15 bus cycles per DRAM access
  - 1 bus cycle per data transfer
- For 4-word block, 1-word-wide DRAM
  - Miss penalty = 1 + 4×15 + 4×1 = 65 bus cycles
  - Bandwidth = 16 bytes / 65 cycles = 0.25 B/cycle



Increasing Memory Bandwidth

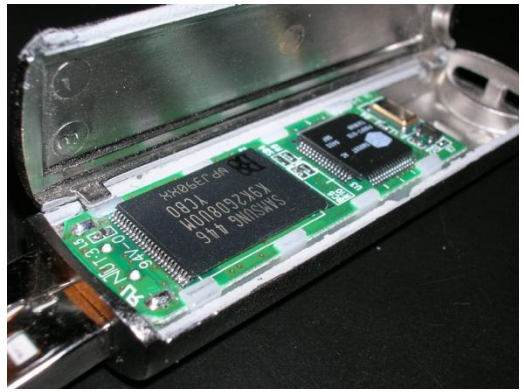


The only difference b/w A and B is the interleaved memory where Bus width is 4 words in both cases. However, only 1 word is provided to the bus per cycle in B by an interleaved Memory Bank

- 4-word wide memory
  - Miss penalty = 1 + 15 + 1 = 17 bus cycles
  - Bandwidth = 16 bytes / 17 cycles = 0.94 B/cycle
- 4-bank interleaved memory
  - Miss penalty = 1 + 15 + 4×1 = 20 bus cycles
  - Bandwidth = 16 bytes / 20 cycles = 0.8 B/cycle

Flash Storage

- Nonvolatile semiconductor storage
  - 100× – 1000× faster than disk
  - Smaller, lower power, more robust
  - But more $/GB (between disk and DRAM)

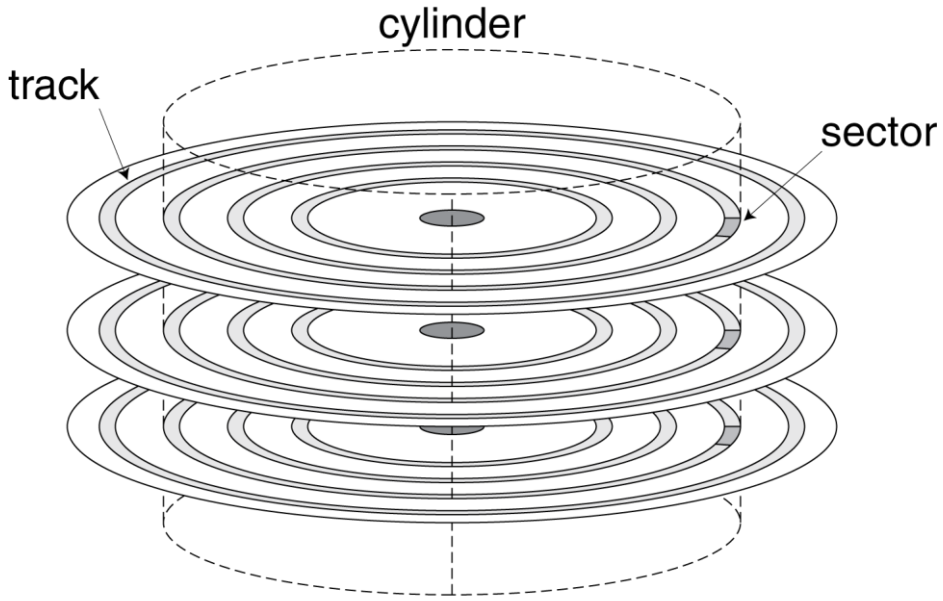


Flash Types

- NOR flash: bit cell like a NOR gate
  - Random read/write access
  - Used for instruction memory in embedded systems
- NAND flash: bit cell like a NAND gate
  - Denser (bits/area), but block-at-a-time access
  - Cheaper per GB
  - Used for USB keys, media storage, …
- Flash bits wears out after 1000's of accesses
  - Not suitable for direct RAM or disk replacement
  - Wear leveling: remap data to less used blocks

Disk Storage

- Nonvolatile, rotating magnetic storage



Disk Sectors and Access

- Each sector records
  - Sector ID
  - Data (512 bytes, 4096 bytes proposed)
  - Error correcting code (ECC)
    - Used to hide defects and recording errors
  - Synchronization fields and gaps
- Access to a sector involves
  - Queuing delay if other accesses are pending
  - Seek: move the heads
  - Rotational latency
  - Data transfer
  - Controller overhead

Disk Access Example

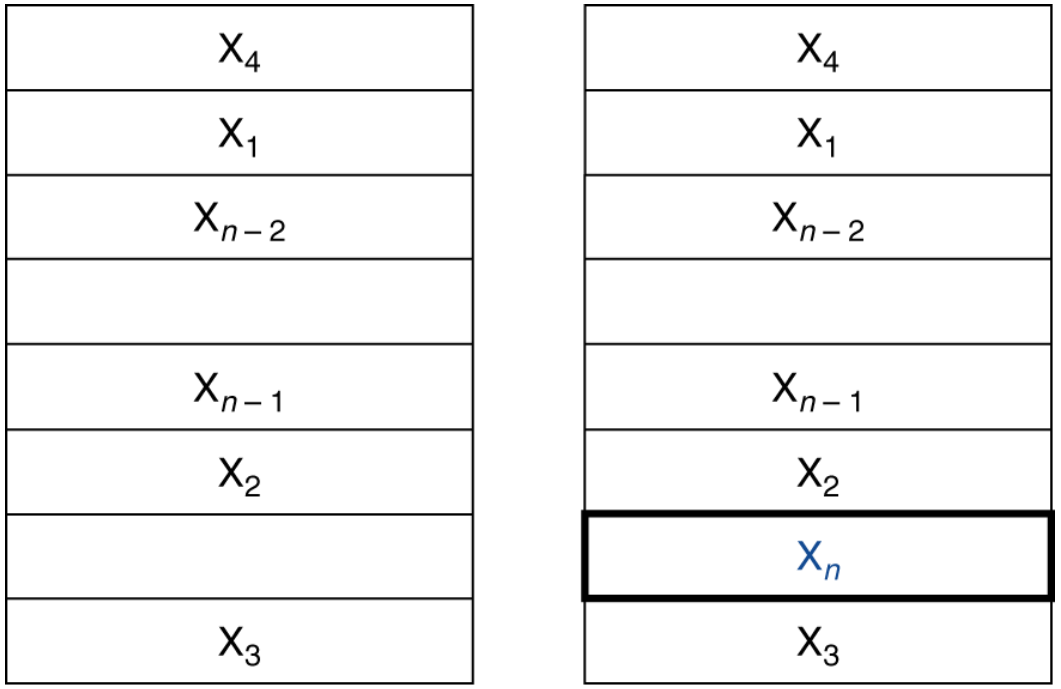
- Given
  - 512B sector, 15,000rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, idle disk
- Average read time
  - 4ms seek time
    + ½ / (15,000/60) = 2ms rotational latency
    + 512 / 100MB/s = 0.005ms transfer time
    + 0.2ms controller delay
    = 6.2ms
- If actual average seek time is 1ms
  - Average read time = 3.2ms

Disk Performance Issues

- Manufacturers quote average seek time
  - Based on all possible seeks
  - Locality and OS scheduling lead to smaller actual average seek times
- Smart disk controller allocate physical sectors on disk
  - Present logical sector interface to host
  - SCSI, ATA, SATA
- Disk drives include caches
  - Prefetch sectors in anticipation of access
  - Avoid seek and rotational delay

Cache Memory

- Cache memory
  - The level of the memory hierarchy closest to the CPU
- Given accesses X_1, …, X_{n-1}, X_n

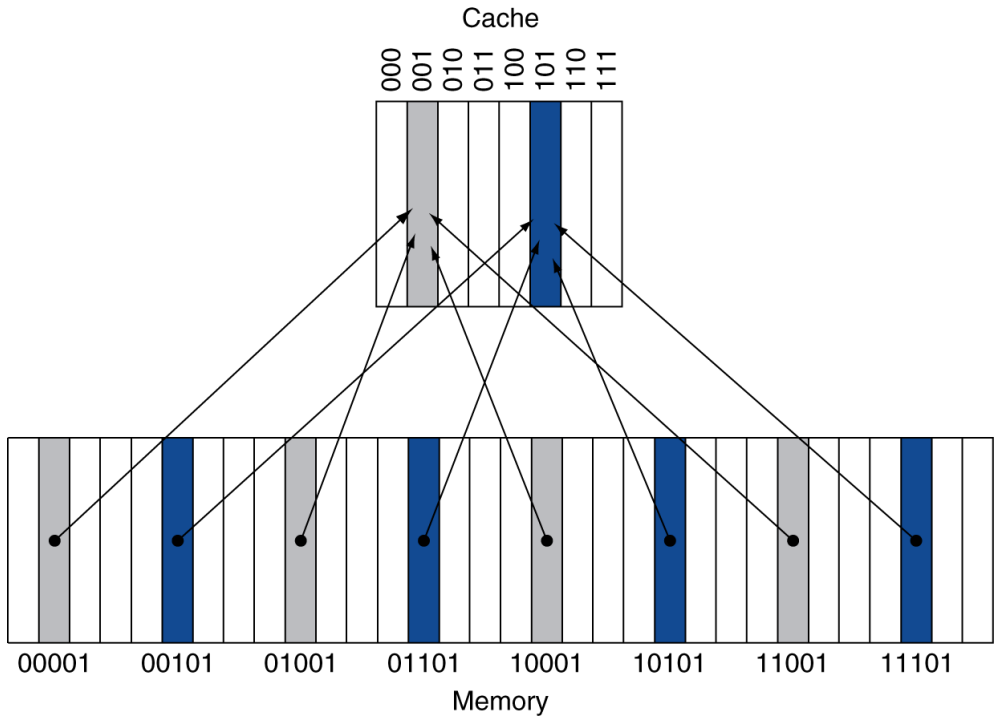


a. Before the reference to X_n b. After the reference to X_n

- How do we know if the data is present?
- Where do we look?

Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)



- #Blocks is a power of 2
- Use low-order address bits

Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data
  - Actually, only need the high-order bits
  - Called the tag
- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

Cache Example

- 8-blocks, 1 word/block, direct mapped
- Initial state

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	**Y**	**10**	**Mem[10110]**
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

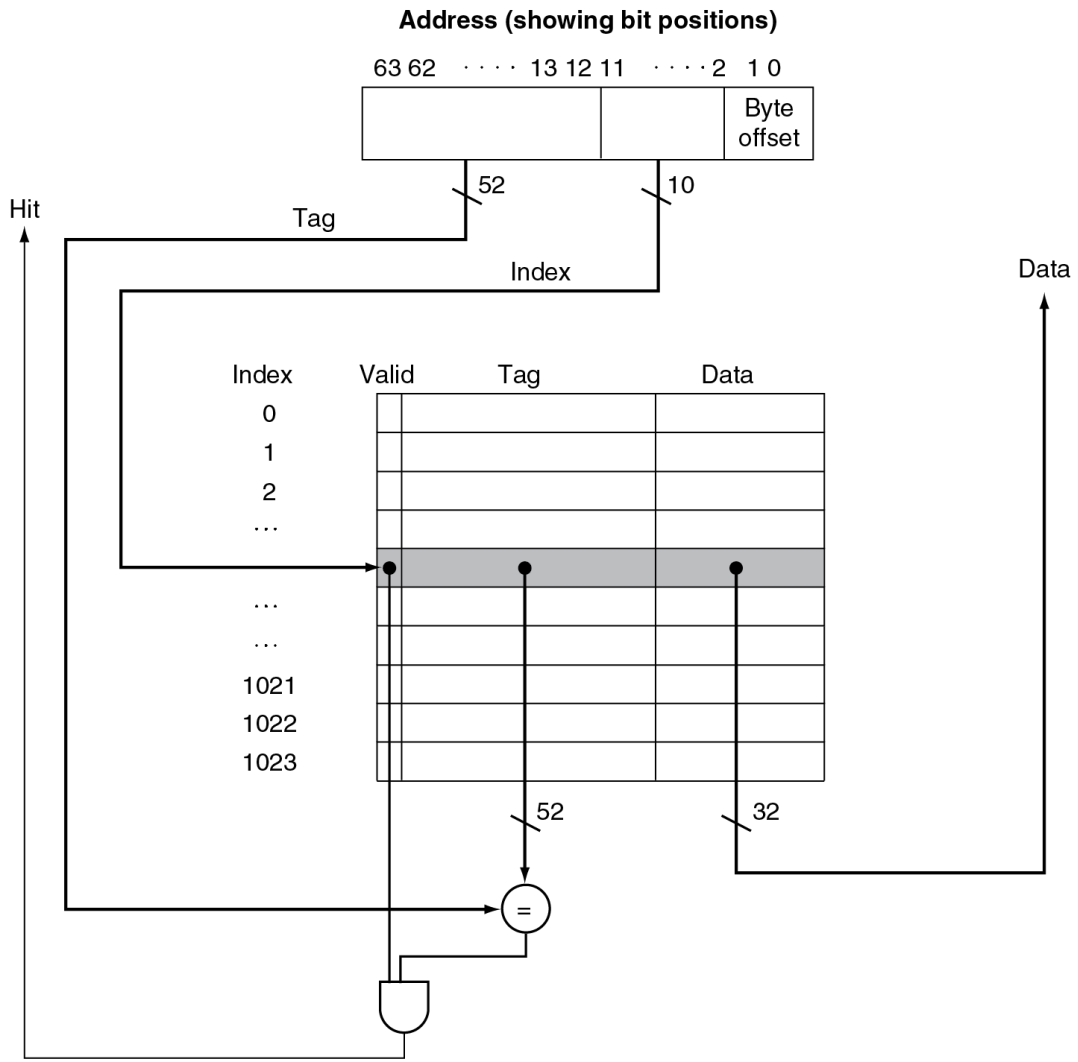
Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

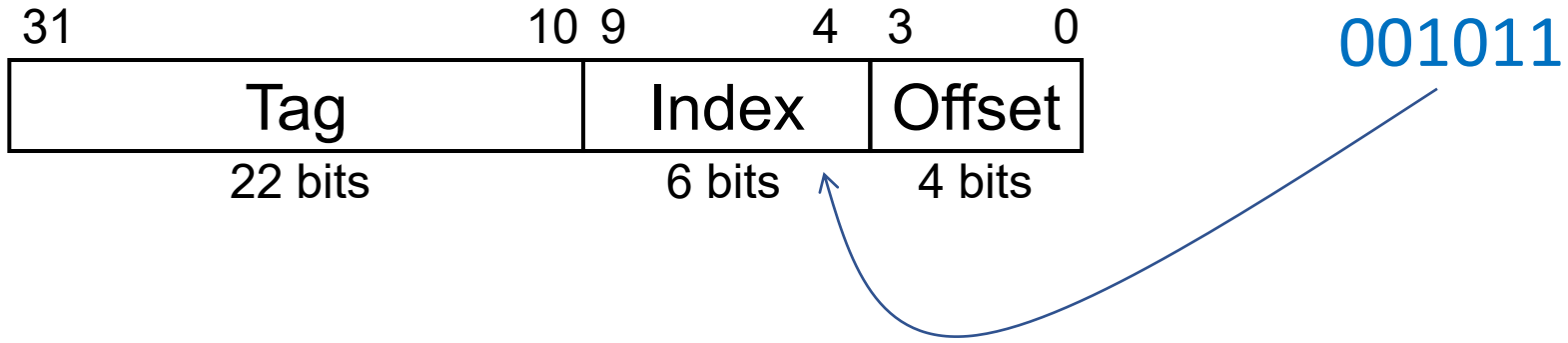
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Address Subdivision



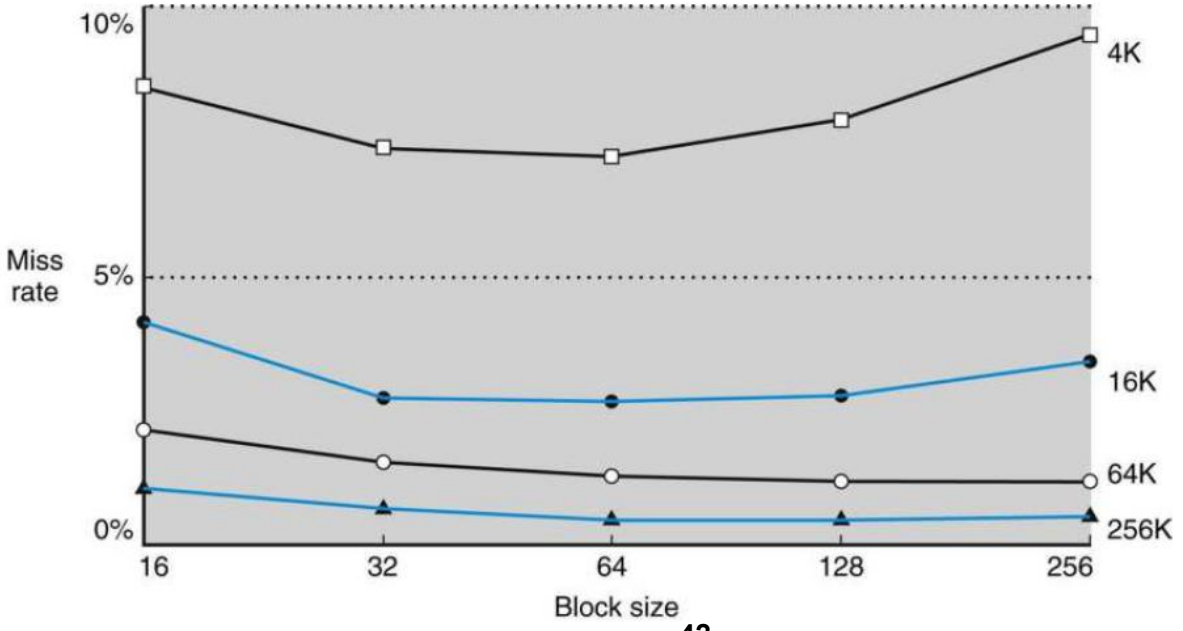
Example: Larger Block Size

- 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?
- Block address = $\lfloor 1200/16 \rfloor = 75$
- Block number = 75 modulo 64 = 11



Block Size Considerations

- Larger blocks should reduce miss rate, Due to spatial locality
- But in a fixed-sized cache, Larger blocks ⇒ fewer of them
- More competition between larger blocks ⇒ increased miss rate when entire block gets replaced with a single miss
- Larger miss penalty, due to higher latencies
  - Can override benefit of reduced miss rate - Early restart and critical-word-first can help



Example 1

- Caches are important to providing a high-performance memory hierarchy to processors.
- Below is a list of 64-bit memory address references, given as word addresses: 0x03, 0xb4, 0x2b, 0x02, 0xbf, 0x58, 0xbe, 0x0e, 0xb5, 0x2c, 0xba, 0xfd
- For each of these references, identify the binary word address, the tag, and the index given a direct-mapped cache with 16 one-word blocks. Also list whether each reference is a hit or a miss, assuming the cache is initially empty.

Part 1

- 16 words in cache, a block requested from the cache is 1 word, each memory reference is for a given word.
  - Because there are 16 words in the cache, an address X maps to the direct-mapped cache word X modulo 16.
  - That is, the low-order $\log_2 16=4$ bits : So, low order 4 bits are used as the cache index.
  - Since the cache is assumed initially empty, assume there is no valid data in it
  - Since none of the memory references repeat with identical tag and index, all of them will miss
  - Each Block now has 2 Words, with a total of 8 Blocks, larger Block, fewer Cache entries

Word Address, Tag, Index

Hex Memory Reference	Binary Reference	Tag	Index	Hit / miss
0x03	0000 0011	0	3	M
0xb4	1011 0100	b	4	M
0x2b	0010 1011	2	b	M
0x02	0000 0010	0	2	M
0xbf	1011 1111	b	f	M
0x58	0101 1000	5	8	M
0xbe	1011 1110	b	e	M
0x0e	0000 1110	0	e	M
0xb5	1011 0101	b	5	M
0x2c	0010 1100	2	c	M
0xba	1011 1010	b	a	M
0xfd	1111 1101	f	d	M

Part 2

- For each of these references, identify the binary word address, the tag, the index, and the offset given a direct-mapped cache with two-word blocks and a total size of eight blocks.
- Also list if each reference is a hit or a miss, assuming the cache is initially empty

Part 2

- The low order $\log_2 8$ = **3 *bits are used as the Cache index*** with **a one-bit field as the offset to *select a Word in the block***
- Cache again assumed to be initially empty – first access to a block is a miss with data from lower level memory written into it
- The Tag and cache index bits repeat thrice – identified in pairs with identical colors (green, brown and blue in Table below)
- A miss results in *both words* being fetched from lower level memory and written into cache so even though the same block is not requested at a later time, it registers as a hit – primary advantage of having multi-word blocks or lines
- Fewer Cache entries translates into shorter critical path to decode an entry improving Cache cycle time

Part 2

Hex Memory Reference	Binary Reference	Tag	Cache Index	Block Offset	Hit / miss
0x03	0000 0011	0	1	1	M
0xb4	1011 0100	b	2	0	M
0x2b	0010 1011	2	5	1	M
0x02	0000 0010	0	1	0	H
0xbf	1011 1111	b	7	1	M
0x58	0101 1000	5	4	0	M
0xbe	1011 1110	b	7	0	H
0x0e	0000 1110	0	7	0	M
0xb5	1011 0101	b	2	1	H
0x2c	0010 1100	2	6	0	M
0xba	1011 1010	b	5	0	M
0xfd	1111 1101	f	6	1	M

Part 3

- You are asked to optimize a cache design for the given references. There are three direct-mapped cache designs possible, all with a total of eight words of data:
  - C1 has 1-word blocks,
  - C2 has 2-word blocks, and
  - C3 has 4-word blocks.

Part 3

- Total cache size is 8 Words
  - same set of memory references
- Optimize for the number of words in a Block
- 3 possible Direct Mapped cache designs
- With 1 word, 2 words, 4 words per Block:
- Cache has 8 entries (Cache index 3 bits), 4 entries (Cache index 2 bits), 2 entries (Cache index 1 bit)

Part 3

Word Address	Binary Address	Tag (5 bits in hex)	Cache 1 block size = 1 word		Cache 2 block size = 2 word		Cache 3 block size = 4 word	
			Index (3 bits)	Hit/Miss	Index (2 bits)	Hit/Miss	Index (1 bit)	Hit/Miss
0x03	0 0000 011	0x00	3	M	1	M	0	M
0xb4	1 0110 100	0x16	4	M	2	M	1	M
0x2b	0 0101 011	0x05	3	M	1	M	0	M
0x02	0 0000 010	0x00	2	M	1	M	0	M
0xbf	1 0111 111	0x17	7	M	3	M	1	M
0x58	0 1011 000	0x0b	0	M	0	M	0	M
0xbe	1 0111 110	0x17	6	M	3	H	1	H
0x0e	0 0001 110	0x01	6	M	3	M	1	M
0xb5	1 0110 101	0x16	5	M	2	H	1	M
0x2c	0 0101 100	0x05	4	M	2	M	1	M
0xba	1 0111 010	0x17	2	M	1	M	0	M
0xfd	1 1111 101	0x1F	5	M	2	M	1	M

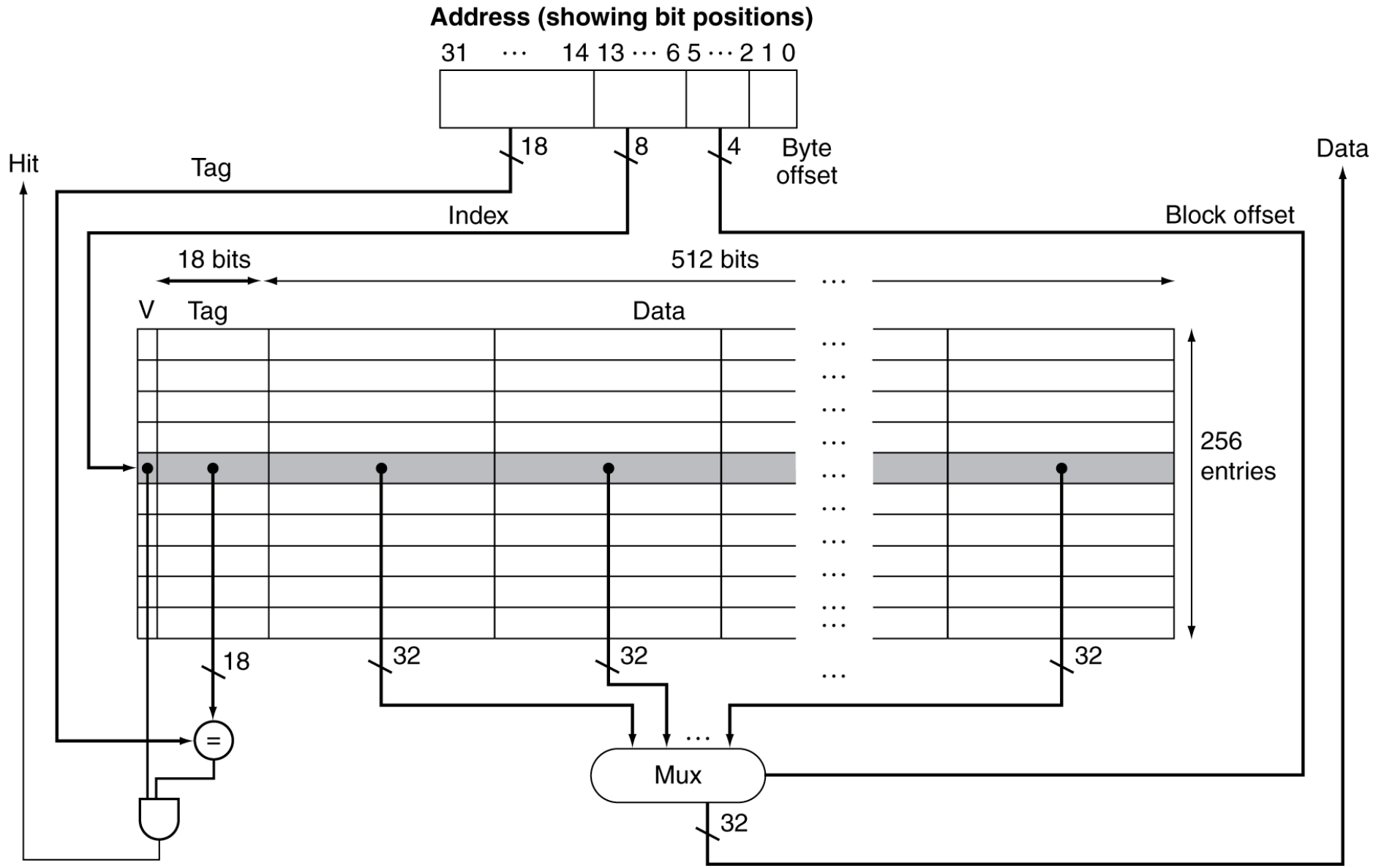
Part 3

- The 2-word and 4-word columns with entries marked in green are misses even though the tag bits and the index bits match to a previous word access (in light blue). Since an access previous to the miss (highlighted in green) corresponding to the same cache line (but different tag bits) was a miss (in yellow) that cache line was replaced and no longer holds the data required by the miss (in green)
- Note that in Part 2, we saw an increase in the Block size (in Words) lower the miss rate. However, as we increase in Word size to 4 Words, the miss rate rises. This is because as the Block size (4 Words in Cache 3) becomes comparable to the Cache size (8 Words), the competition for Blocks increases with the entire Block replaced if a single entry misses. Limits on increasing Block size to improve hit rate as seen in Part 2 are imposed by the size of the Cache itself. Fig 5.11 in text demonstrates this observation as well:

Example 2: Intrinsity FastMATH

- Embedded MIPS processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - Each 16KB: 256 blocks × 16 words/block
  - D-cache: write-through or write-back
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%

Intrinsity FastMATH



Read from Intrinsity FastMATH Cache

- 1. Send the address to the appropriate cache. The address comes either from the PC (for an instruction) or from the ALU (for data).
- 2. If the cache signals hit, the requested word is available on the data lines.
  - Since there are 16 words in the desired block, we need to select the right one.
  - A block index field is used to control the multiplexor (shown at the bottom of the figure), which selects the requested word from the 16 words in the indexed block.
- 3. If the cache signals miss, we send the address to the main memory. When the memory returns with the data, we write it into the cache and then read it to fulfill the request

Miss Rates of the Intrinsity Cache

- Miss rates for the instruction and data caches - The combined miss rate is the effective miss rate per reference for each program after accounting for the differing frequency of instruction and data accesses

Instruction miss rate	Data miss rate	Effective combined miss rate
0.4%	11.4%	3.2%

FIGURE 5.13 Approximate instruction and data miss rates for the Intrinsity FastMATH processor for SPEC CPU2000 benchmarks.

The combined miss rate is the effective miss rate seen for the combination of the 16 KiB instruction cache and 16 KiB data cache. It is obtained by weighting the instruction and data individual miss rates by the frequency of instruction and data references.

Processor Performance and Cache Reads

- CPU Time = clock cycles CPU spends executing Program + clock cycles CPU spends waiting for Memory System

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory-stall clock cycles}) \times \text{Clock cycle time}$$

- Memory-stall clock cycles come primarily from cache misses
- Memory-stall clock cycles can be defined as the sum of the stall cycles coming from reads plus those coming from writes
- The read-stall cycles can be defined in terms of the (i) number of read accesses per program (ii) the miss penalty in clock cycles for a read and (iii) the read miss rate:

$$\text{Read-stall cycles} = \frac{\text{Reads}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty}$$

Processor Performance and Cache Writes

- For a write-through scheme, we have two sources of stalls: (i) write misses, which usually require that we fetch the block before continuing the write and (ii) write buffer stalls, which occur when the write buffer is full when a write happens
- The cycles stalled for writes equal the sum of these two:

$$\text{Write-stall cycles} = \left(\frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty} \right) + \text{Write buffer stalls}$$

- In most write-through cache organizations, the read and write miss penalties are the same (the time to fetch the block from memory)
- *If* we assume that the *write buffer stalls are negligible*, we can combine the reads and writes by using a single miss rate and the miss penalty:

$$\text{Memory-stall clock cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

Data Memory

$$\text{Memory-stall clock cycles} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Instruction Memory

Example

- Assume the miss rate of an instruction cache is 2% and the miss rate of the data cache is 4%. If a processor has a CPI of 2 without any memory stalls, and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect cache that never missed. Assume the frequency of all loads and stores is 36%

$$\text{Instruction miss cycles} = I \times 2\% \times 100 = 2.00 \times I$$

$$\text{Data miss cycles} = I \times 36\% \times 4\% \times 100 = 1.44 \times I$$

- The total number of memory-stall cycles is 2.00 I +1.44 I =3.44 I - This is more than three cycles of memory stall per instruction!
- The total CPI including memory stalls is 2 +3.44 =5.44
- Since there is no change in instruction count or clock rate, the ratio of the CPU execution times is

$$\frac{\text{CPU time with stalls}}{\text{CPU time with perfect cache}} = \frac{I \times \text{CPI}_{\text{stall}} \times \text{Clock cycle}}{I \times \text{CPI}_{\text{perfect}} \times \text{Clock cycle}} = \frac{\text{CPI}_{\text{stall}}}{\text{CPI}_{\text{perfect}}} = \frac{5.44}{2}$$

Memory Performance must scale w Processor

- What happens if the processor is made faster, but the memory system is not? `IC X CPI x Tcycle`
- The amount of time spent on memory stalls (miss penalty) will take up *an increasing fraction of the execution time*
- Speed-up the computer in this example by reducing its CPI from 2 to 1 without changing the clock rate, which might be done with an improved pipeline
- CPI of 1 +3.44 =4.44
- Only a ~ 20% improvement in CPI!

Average Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss Rate × Miss Penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

Example 4

- Cache block size (B) can affect both miss rate and miss latency.
- Assuming a machine with a base CPI of 1, and an average of 1.35 references (both instruction and data) per instruction, *find the block size that minimizes the total miss latency* given the following miss rates for various block sizes.

8: 4%	16: 3%	32: 2%	64: 1.5%	128: 1%

Miss rate is reduced by increasing Block size (in Bytes) by taking advantage of spatial locality of data

Block size impact on Cache Performance

- Larger Block size improves miss rate until Block size becomes comparable to cache size which is when miss rate begins to increase
- Larger Block size degrades miss penalty since it *takes longer to transfer a larger Block* from RAM to Cache
- AMAT= Miss Rate x Miss Latency, when **AMAT is a linear function of Miss Latency**

Optimal Block size for ML = **20 x B cycles**?

- AMAT for B = 8: 0.040 × (20 × 8) = **6.40**
- AMAT for B = 16: 0.030 × (20 × 16) = 9.60
- AMAT for B = 32: 0.020 × (20 × 32) = 12.80
- AMAT for B = 64: 0.015 × (20 × 64) = 19.20
- AMAT for B = 128: 0.010 × (20 × 128) = 25.60

Miss Rate dependence on Block Size (Bytes)

8: 4%	16: 3%	32: 2%	64: 1.5%	128: 1%

B = 8 is optimal. The *smallest* Block size yields the lowest AMAT since the miss latency is the lowest with the fewest Bytes that need to be transferred on a Miss

Part 2

Optimal block size for a miss latency of **24+B cycles**?

- For cases where the *Miss Latency begins to increase noticeably only for Block sizes larger than some threshold* size:
- AMAT = Miss Rate x (24 + B)

AMAT for B = 8: 0.040 × (24 + 8) = 1.28

AMAT for B = 16: 0.030 × (24 + 16) = 1.20

AMAT for B = 32: 0.020 × (24 + 32) = **1.12**

AMAT for B = 64: 0.015 × (24 + 64) = 1.32

AMAT for B = 128: 0.010 × (24 + 128) = 1.52

B = 32 is optimal

Part 3

Optimal block size for *constant miss latency*?

B = 128 is optimal

Minimizing the miss rate minimizes the total miss latency.

Memory Performance must scale w Processor

- When CPU performance increased: `IC X CPI x Tcycle`
  - Miss penalty becomes more significant

1. Decreasing base CPI
   - Greater proportion of time spent on memory stalls
2. Increasing clock rate
   - Memory stalls account for more CPU cycles

- Can't neglect cache behavior when evaluating system performance

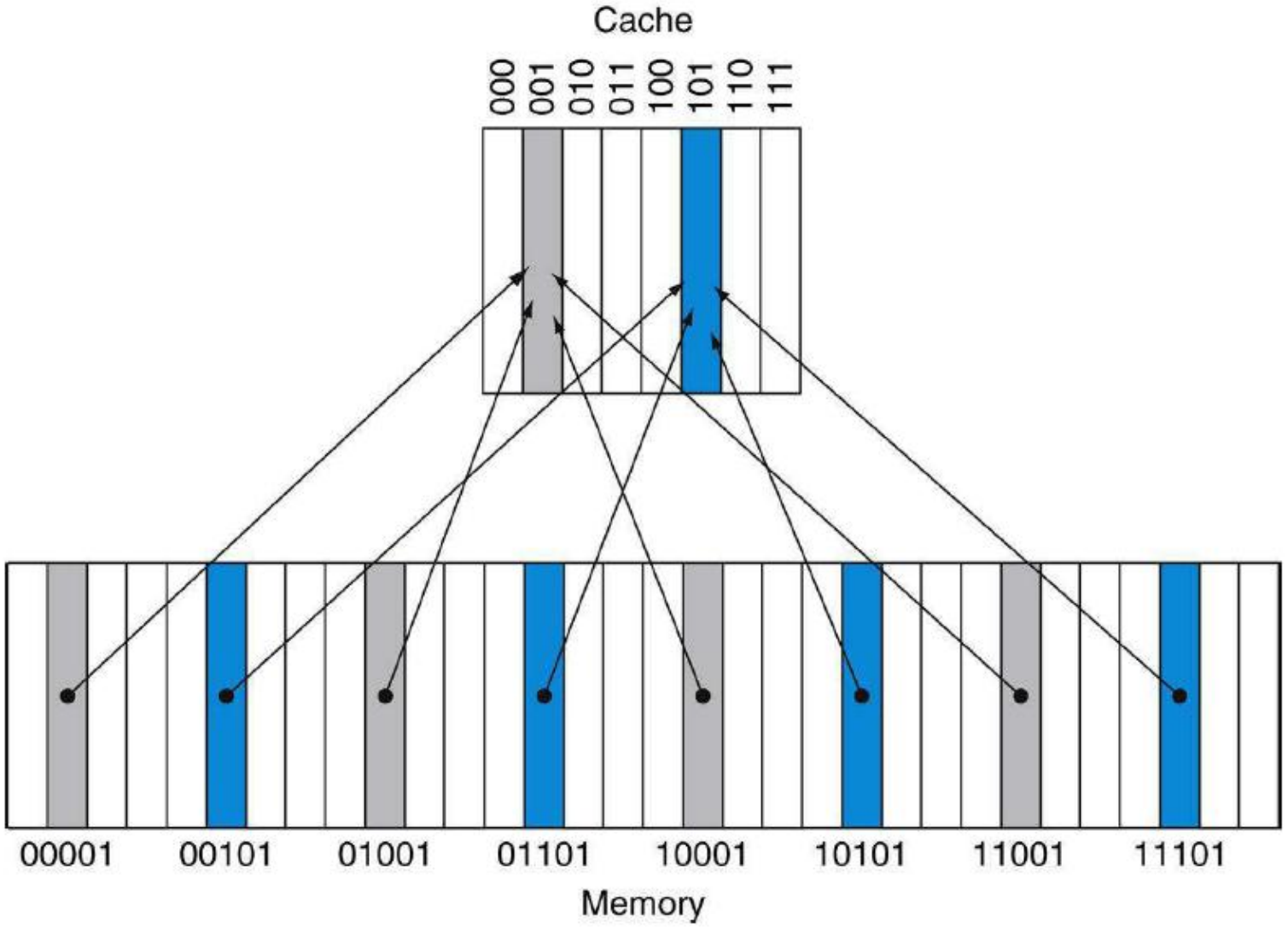
Unique Cache Index

- In any direct mapped cache – with a 10-bit index in this given problem,
- The 10-bit cache index must be unique to any given block address. In other words, a given block address cannot map to more than one cache index – as shown in the color-coded figure (5.8 from text) below. If this were not the case, then a given block address could map to multiple cache locations.
- So, at a minimum, any function that produces a unique 10-bit output corresponding to the 10-bit cache index and which can cover all possible cache blocks, is sufficient.

Example 5

- Section 5.3 shows the typical method to index a direct-mapped cache, specifically (Block address) modulo (Number of blocks in the cache).
- Assuming a 64-bit address and 1024 blocks in the cache, consider a different indexing function, specifically (Block address[63:54] XOR Block address[53:44]).
- Is it possible to use this to index a direct-mapped cache?
- If so, explain why and discuss any changes that might need to be made to the cache. If it is not possible, explain why

Example 5


Cache
000
001
010
011
100
101
110
111
00001
00101
01001
01101
10001
10101
11001
11101
Memory

Example 5

- Clearly, [block address modulo number of blocks in cache] satisfies this minimum requirement
- Assuming a 10-bit cache index that identifies one of 1024 cache entries in a direct mapped cache are given by bits [53:44] in a 64 bit memory address – lets assume for simplicity that bits [43:0] correspond to block and byte offsets and that the 20 most significant bits of the 64 bit address [63:44] correspond to the Memory address space of 1M Blocks
- Let's use, for any given 64 bit Memory address M:
- Proposed Indexing Function: M[63:54] XOR M[53:44]
- Observation A: For each unique set of bits in M[63:54], there are exactly 1024 possible combinations of M[53:44] in the 64 bit address provided corresponding to the opportunity to map the unique M[63:54] bits to any of exactly 1024 cache entries addressed by M[53:44]
- Observation B: For each unique set of bits in [63:54] there is exactly only ONE result of the XOR function for each of the 1024 combinations in [53:44] satisfying the unique cache index for any given memory address vector of 64 bits
- From the above 2 observations, the proposed XOR function to index the cache in a direct mapped cache is sufficient

Example 7

- For a direct-mapped cache design with a 64-bit address, the following bits of the address are used to access the cache.

Tag	Index	Offset
63–10	9–5	4–0

- What is the cache block size (in words)?
- How many blocks does the cache have?
- What is the ratio between total bits required for such a cache implementation over the data storage bits?
- For each Memory reference shown in Table below, list (1) its tag, index, and offset, (2) whether it is a hit or a miss, and (3) which bytes were replaced (if any).
- What is the hit ratio?

Address												
Hex	00	04	10	84	E8	A0	400	1E	8C	C1C	B4	884
Dec	0	4	16	132	232	160	1024	30	140	3100	180	2180

Example 7

- **Each Block has 32 Bytes** (offset is 5 bits wide) or 4 64-bit words or 4 8-Byte words (total of 32 Bytes).
- 2 bits determine one of 4 (64-bit) Words in the Block, 3 least significant bits determine the byte in each 64-bit word
- 5 bits in the index field indicate **32 Blocks or 32 lines in the cache**
- The cache stores 32 Blocks x 4 Words/Block x 8 Bytes/word = 1024 Bytes = *8192* bits
- In addition to data, 54 bits for Tag and 1 valid bit
- Total bits required = 8192 + 54x32 + 1 x 32 = *9952* bits
- **9952 / 8192 = 1.21**

Example 7

Byte Address	Binary Address	Tag	Index	Offset	Line replaced	Hit/Miss
0x00	00 0 0000 0 0000	0x0	0x00	0x00	No	M
0x04	00 0 0000 0 0100	0x0	0x00	0x04	No	H
0x10	00 0 0000 1 0000	0x0	0x00	0x10	No	H
0x84	00 0 0100 0 0100	0x0	0x04	0x04	No	M
0xe8	00 0 0111 0 1000	0x0	0x07	0x08	No	M
0xa0	00 0 0101 0 0000	0x0	0x05	0x00	No	M
0x400	01 0 0000 0 0000	0x1	0x00	0x00	**Yes**	M
0x1e	00 0 0000 1 1110	0x0	0x00	0x1e	**Yes**	M
0c8c	00 0 0100 0 1100	0x0	0x04	0x0c	No	H
0xc1c	11 0 0000 1 1100	0x3	0x00	0x1c	**Yes**	M
0xb4	00 0 0101 1 0100	0x0	0x05	0x14	No	H
0x884	10 0 0100 0 0100	0x2	0x04	0x04	**Yes**	M

Part 3

- The Cache line is replaced only when the tag bits of the memory reference change. All 32 bytes in that cache line are replaced.
- Hit Ratio = 4/12 = 33.33%

Cache Misses

- On cache hit, CPU proceeds normally
- When does a cache 'miss'
  - *Tag bits* of cache do not match leading bits of requested address (for read or write)
  - Data at desired index in cache (for read) is invalid (*valid bit*)
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
- Instruction cache miss
    - Restart instruction fetch
- Data cache miss
    - Complete data access

Write Policy

- When a system writes to cache (assuming it is a hit), it must write it to store (next level in the Memory hierarchy) as well at some point in time
- The timing of this write is controlled by Write Hit Policy
  - Two approaches: Write through or Write Back

Write-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: *also update memory*
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: Write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full – simple design solution: make buffer deeper to hold more words (2-4 for example)
    - No amount of buffering will help if the rate at which Write requests are generated (Write Bandwidth demanded) exceed the rate at which Write-Through updates memory (Write Bandwidth available)
    - Even if the Bandwidth demanded is less than the Bandwidth available at which Write-Through updates memory, it can be insufficient when Write requests are issued in bursts
    - Write Hit policy is determined by the architect according to the Write bandwidth available

Write-Back

- One Solution to maintain performance of the memory system is to lower the bandwidth demanded for Writes using a 'Write-Back' Write Hit policy
- On data-write hit, just update the block in cache – with it's status bit updated to 'dirty' - *representing that data is inconsistent with lower level memory*
- The new value is written only to the block in the cache.
- The modified block is written to the lower level of the hierarchy *only when it is replaced in the cache (on a Read or Write Miss)*
- When a dirty block is replaced, requires 2 accesses to Memory:
  1. Write 'dirty' block back to memory so memory has the most recent updated value
  2. Fetch the needed data from memory

  Can use a write buffer for above 'dirty' block move, to allow replacing block to be read first from lower level memory

Write Allocation

- What should happen on a write miss?
- No data is returned to the requester on write operations
  - A decision needs to be made on write misses, whether or not data would be loaded into the cache
- ***No Write Allocate:***
  - Data at the missed-write location is not loaded to cache, and is written directly to the backing store. *In this approach, data is loaded into the cache on read misses only*
- ***Write Allocate:***
  - Data at the missed-write location is (1) loaded to cache, followed by (2) a write-hit operation. *In this approach, write misses are similar to read misses*

Write Miss

- What happens when a Write is a ‘miss’? (When Tag bits don’t match)
  - Assuming a Write Through Cache

1. Fetch Block corresponding to Address of intended Write, from Memory
2. After Block is fetched, overwrite the word in that Block
3. Write that overwritten Word (**only**) into Memory [use write buffer]

Write Allocate

- This is simple, but would not provide good performance
- Alternatively,
  - Simply Write that Word directly to Memory [use write buffer]

No Write Allocate

Write Miss

- What happens when a Write is a ‘miss’? (When Tag bits don’t match)
  - Assuming a Write Back Cache

1. Fetch Block corresponding to Address of intended Write, from Memory
2. If ‘dirty’, the Block to be replaced is written to Memory [use Write Buffer]

- With Write Back, 2 cycles are necessary: First we must determine if the Write is a hit or a miss because we cannot overwrite the cache if it is a miss and if the cache entry ‘dirty’ bit is set [cache entry inconsistent with memory]
  - Processor places the new data to be written - into a store buffer and does a cache lookup in the first cycle.
  - If the cache is a hit, the new data is moved from the store buffer into the cache on the next cache access cycle
- If it is a miss
  - The (‘dirty’) data in the cache to be replaced is moved from the write buffer to Memory before the fetched data from Memory overwrites it

Write Policy

Write hit policy	Write miss policy
Write Through	Write Allocate
Write Through	No Write Allocate
Write Back	Write Allocate
Write Back	No Write Allocate

Write Through with Write Allocate:

- on Write **hits** it writes to cache and main memory
- on Write **misses (tag mismatch)** it *updates the word in main memory and brings the block to the cache* (so that the next Write or Read to the same cache line will not miss)
- *The benefit of bringing the updated block from main Memory into cache following Write misses is that the data is available in the cache for a subsequent Read access*.
- The disadvantage of bringing updated block from main Memory into cache following a write miss is that *in a subsequent Write hit, this data in the cache that costed bandwidth and performance with a Write allocate miss policy is overwritten anyways* and the memory is still updated on this subsequent Write. So, *Bringing the block to cache from updated memory on a write miss would not make a lot of sense*.

Write Through with No Write Allocate:

- On **hits** it writes to cache and main memory;
- On **misses** it *updates the block in main memory* not bringing that block to the cache;
- Subsequent writes to the block will *update main memory because Write Through policy is employed*. So, some *time is saved not bringing the block in the cache on a miss* because it appears useless anyway.
- However, *subsequent Reads to this address will report a miss because the cache line was not updated with a no write allocate policy.* These Read misses likely evict the cache line (since the cache line is inconsistent with memory) requiring the memory to be updated

Write Back with Write Allocate:

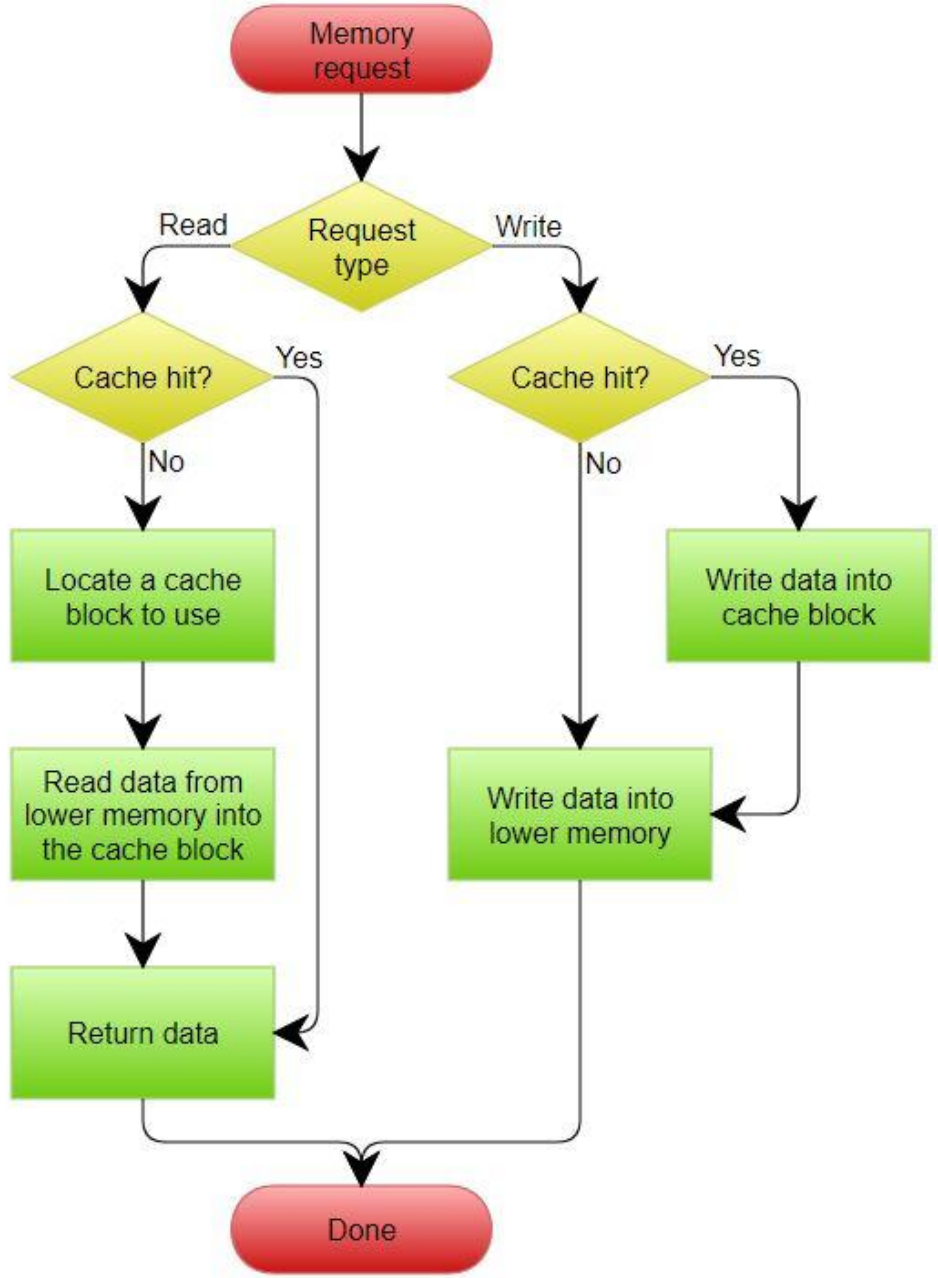
- on **hits** it writes to cache *setting dirty bit* for the block, *main memory is not updated*;
- on **misses** it *updates the 'dirty' block in main memory* **and** *brings the newly requested block to the cache*;
- *Subsequent writes* to the same block, if the block originally caused a miss, *will hit in the cache next time*, setting dirty bit for the block. That *will eliminate extra memory accesses and result in very efficient execution* compared with Write Through with Write Allocate combination.

Write Back with No Write Allocate:

- on **hits** it *writes to cache setting dirty bit* for the block, *main memory is not updated*;
- on **misses** it *updates the block in main memory* **not** *bringing that block to the cache*;
- Subsequent writes to the same block, if the block originally caused a miss, *will generate misses* all the way and *result in very inefficient execution*.

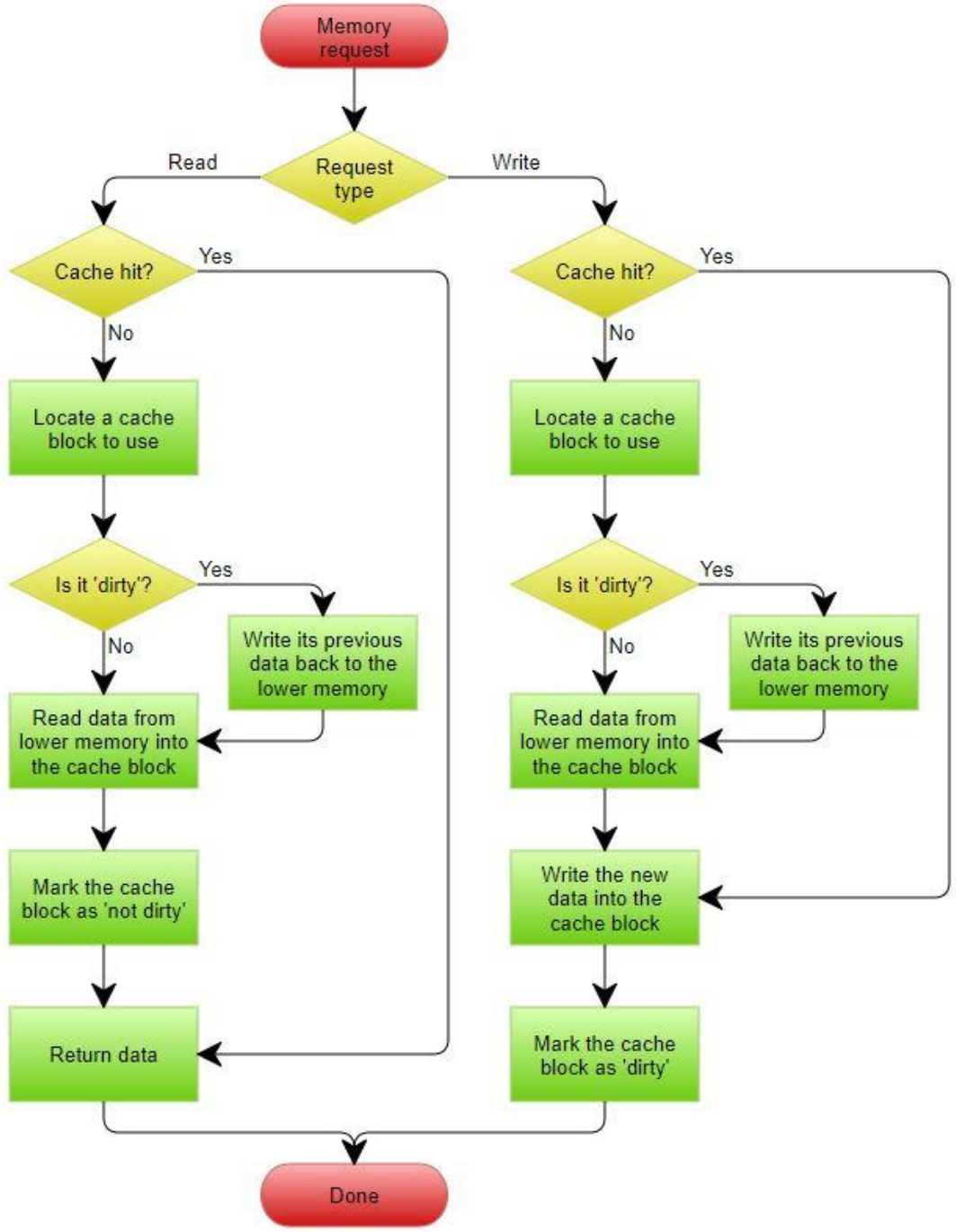
No Write Allocate

Write Through hit policy assumed



Write Allocate

Write Back hit policy assumed



Average Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss Rate × Miss Penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

Improving Cache Performance

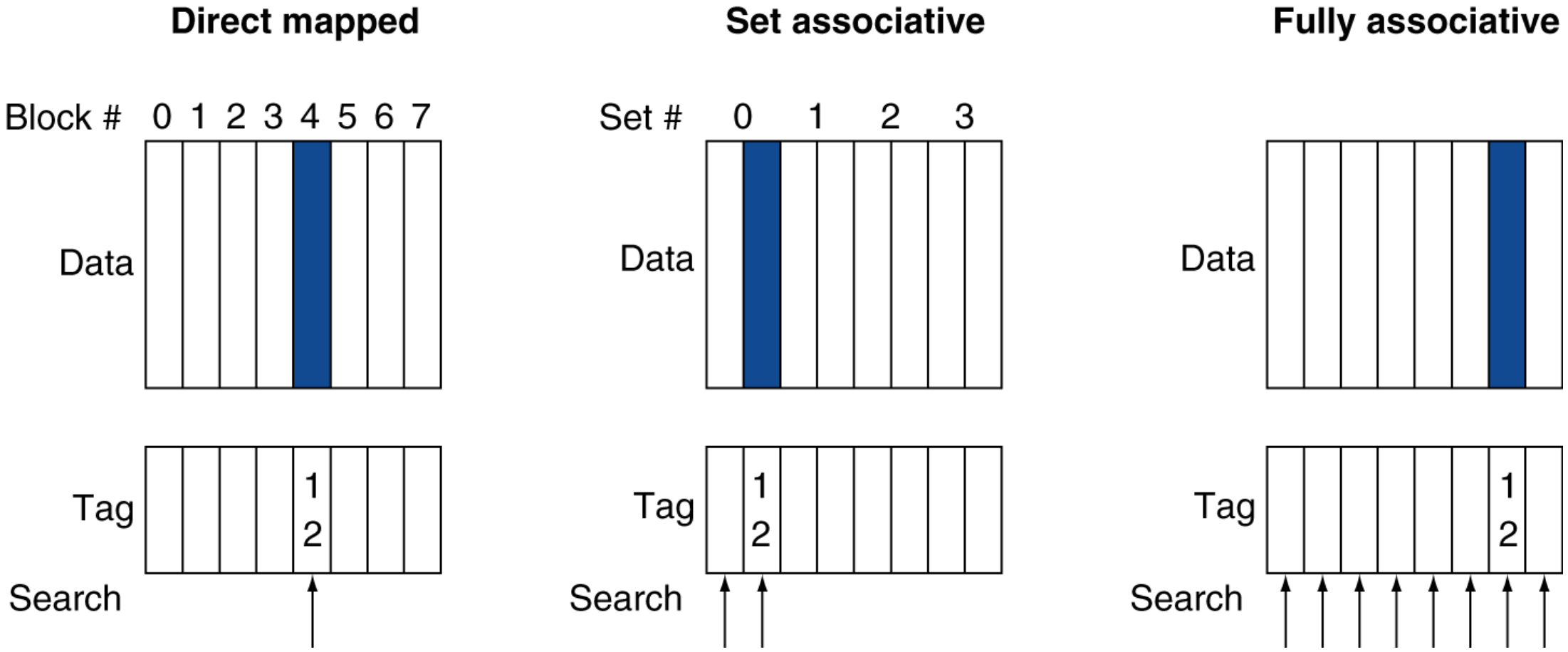
- Two Techniques to improve Cache Performance:

1. Reduce Miss Rate
   - Increase Cache size (within limits of cycle time)
   - Increase Block size (to within at least 4x smaller than cache size)
   - Associative Caches – Memory address no longer mapped to only one unique location in the cache – improves miss rate substantially
2. Reduce Miss Penalty
   - Improve bandwidth b/w cache and next level
   - Add level in memory hierarchy within Processor chip so likelihood of processor finding the requested data within chip is much higher (compared to off-chip DRAM)
     - Cost of processor goes up as chip gets bigger with bigger L2, L3 caches – so does the miss penalty in number of clock cycles
     - Trade-off decreasing miss rate with increasing miss penalty and chip cost to minimize average memory access time (AMAT) within cost constraint

Associative Caches

- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- *n*-way set associative
  - Each set contains *n* entries
  - Block number determines which set
    - (Block number) modulo (#Sets in cache)
  - Search all entries in a given set at once
  - *n* comparators (less expensive)

Associative Cache Example



Spectrum of Associativity

- For a cache with 8 Blocks

One-way set associative

(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8

1. Direct mapped

All memory references miss because 4 of the 5 references can be mapped only to Block 0 in a direct mapped cache

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Replacement Policy

- Direct mapped: no choice
- Set associative
  - Prefer to replace a non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
  - Gives approximately the same performance as LRU for high associativity

Associativity Example

2. 2-way set associative

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

Mem[6] replaces Mem[8] and not Mem[0] because Mem[8] was not used as recently as Mem[0]

3. Fully associative

Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

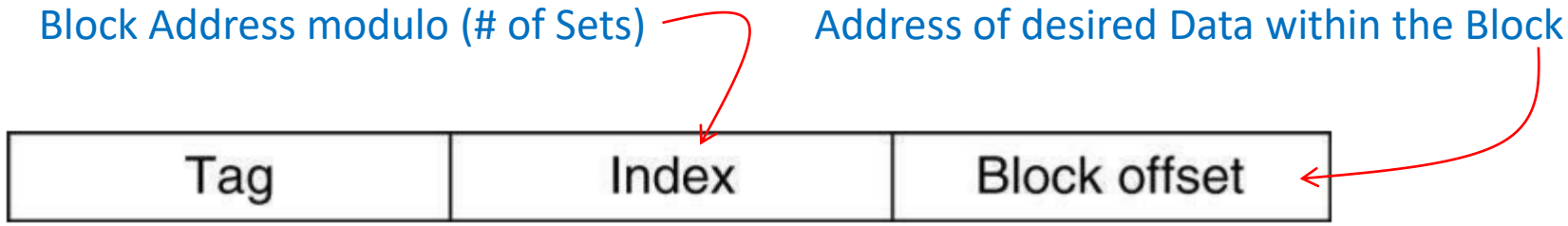
As the cache gets more associative, each set can hold more Blocks substantially raising the hit rate

How Much Associativity

- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%
- While increasing the Associativity lowers the miss rate, it increases the delay overhead in identifying a hit because the tag bits of all entries in the set must be compared to the tag bits of the requested address before a ‘hit’ or a ‘miss’ can be determined

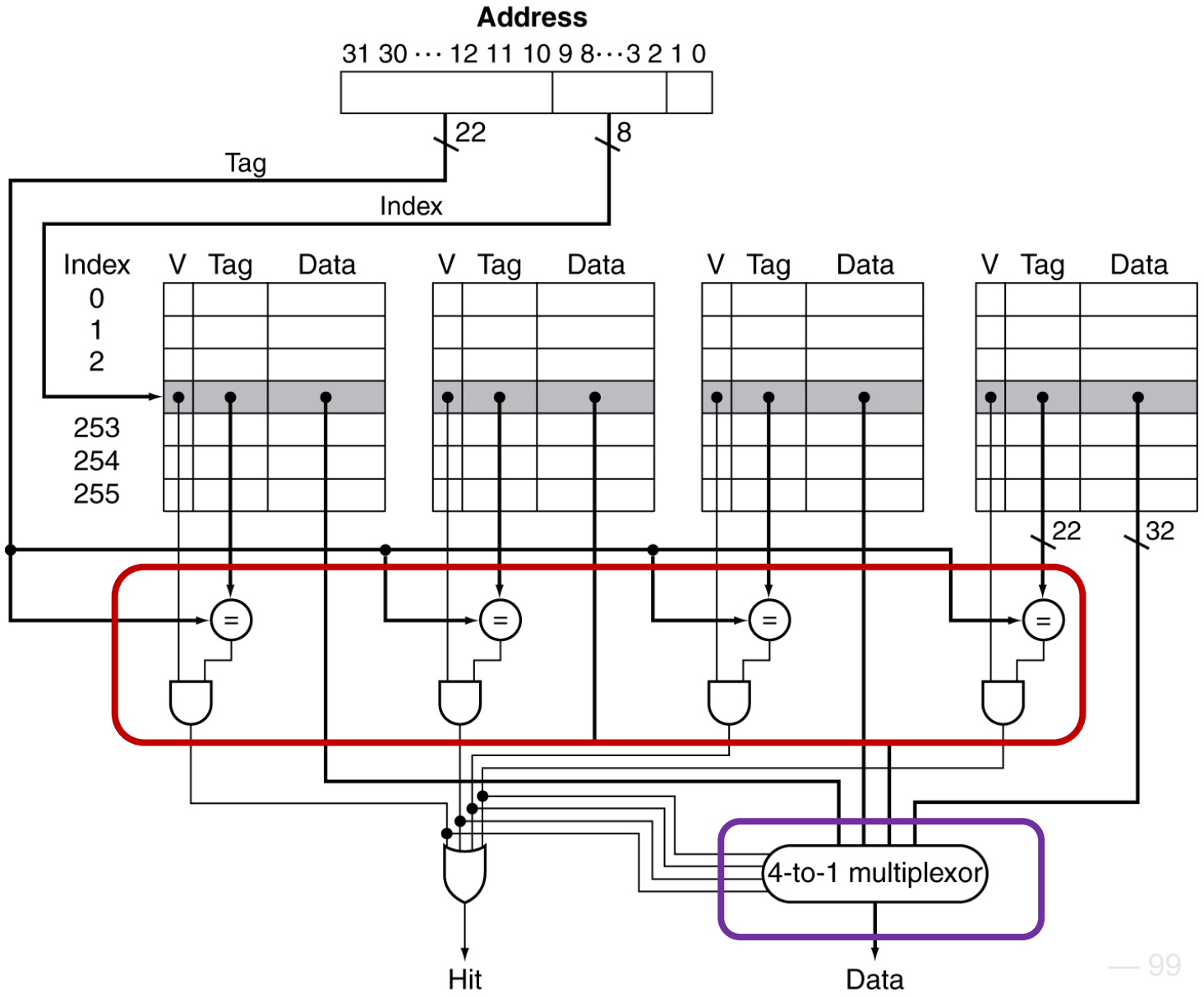
Observations on Associative Caches

- For a given cache size, increasing associativity increases the number of Blocks per set (and lowers the number of sets by the same factor)
- The number of Blocks per set is the number of simultaneous compares the access must pursue (of Tags of each Block with the Block address of the access)
- Each factor of 2 increase in the associativity decreases the index field width by 1 bit and increases the Tag field by 1 bit
- In a fully associative cache there is only 1 set – all Block Tags in the cache must be compared to the Block address of the access in parallel:
  - There are no bits in the index field
  - The entire address excluding the Block offset is compared against the Tag of every Block in the cache



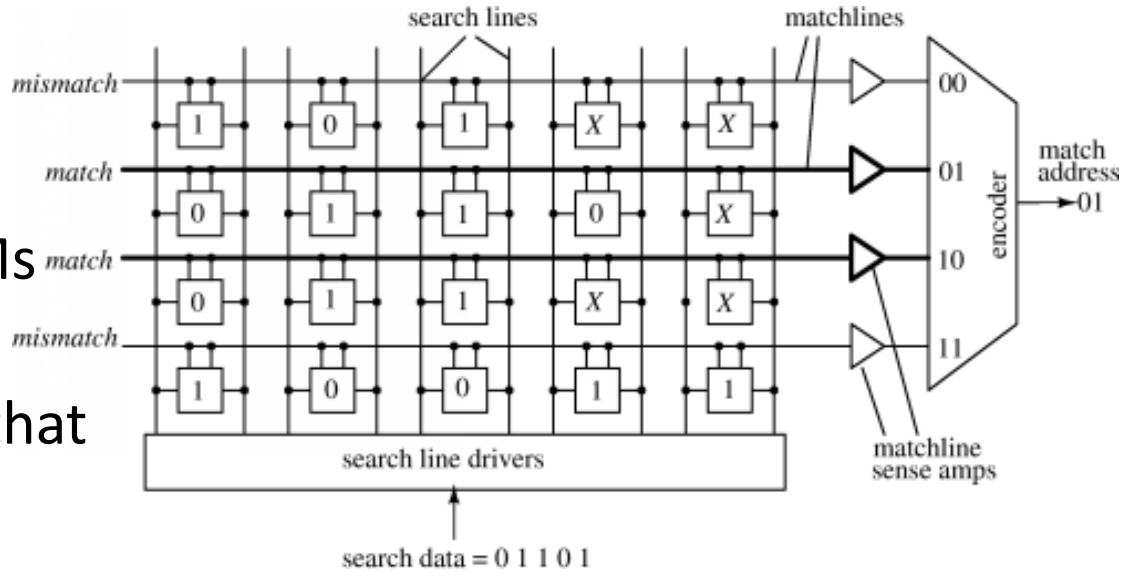
Set Associative Cache Organization

- In a 4-way set associative cache, 4 **comparators** are needed, together with a 4- to-1 **multiplexor** to choose among the four potential members of the selected set
- The overheads of an associative cache are the **comparators** and additional latency from the task to compare and select from among the elements of the set
- **Comparators** determine which element of the selected set matches the tag.
  - The output of the comparators is used to select the data from one of the 4 blocks of the indexed set using a multiplexor with a decoded select signal



Fully Associative Cache using CAM

- Instead of supplying an address and reading a word like a RAM, you send the data and the CAM looks to see if it has a copy and returns the index of the matching row
- CAMs mean that cache designers can afford to implement much higher set associativity than if they needed to build the hardware out of SRAMs and comparators
- Increasing set associativity requires overheads that are increasingly inefficient to implement using SRAM and Comparators (previous slide)
- Fully Associative Caches where the entire Tag array must be searched for the given Block address (excluding Block offset) – are best implemented with CAMs



Example

- Increasing associativity requires more comparators and more tag bits per cache block.
- Assuming a cache of 4096 blocks, a four-word block size, and a 64-bit address, find the total number of sets and the total number of tag bits for caches that are direct-mapped, two-way and four-way set associative, and fully associative.

Example

- Given: 4-Word Block size. i.e., 4 Words x 4 Bytes/Word = 16 Bytes = 2^4 Bytes
- A 64 bit address thus requires 64 – 4 = 60 bits for Tag and Index fields
- The Direct Mapped Cache has as many Sets as Blocks – thus 4096 Blocks indexed with **12** bits in the index field (2^{12} = 4096)
  - Total number of Tag bits = [60 -12] x 4096 entries = 48 x 4096 = 197Kb
- As Associativity increases by a factor of 2, the number of bits in the index field decreases by 1 and the Tag field increases by 1 bit
  - 2-way associative cache will have 11 bits in the index field and 49 bits in the Tag field

  number of Tag bits = [60 -11] x 2 x 2048 entries = 49 x 2048 = 401 Kb
  - 4-way associative cache will have 10 bits in the index field and 50 bits in the Tag field

  number of Tag bits = [60 -10] x 4 x 1024 entries = 50 x 1024 = 205 Kb
  - Fully Associative Cache has only 1 set with 4096 Blocks, so Tag has 60 bits (no index field anymore)

  number of Tag bits = [60] x 4096 x 1 entries = 60 x 4096 x 1 = 246 Kb

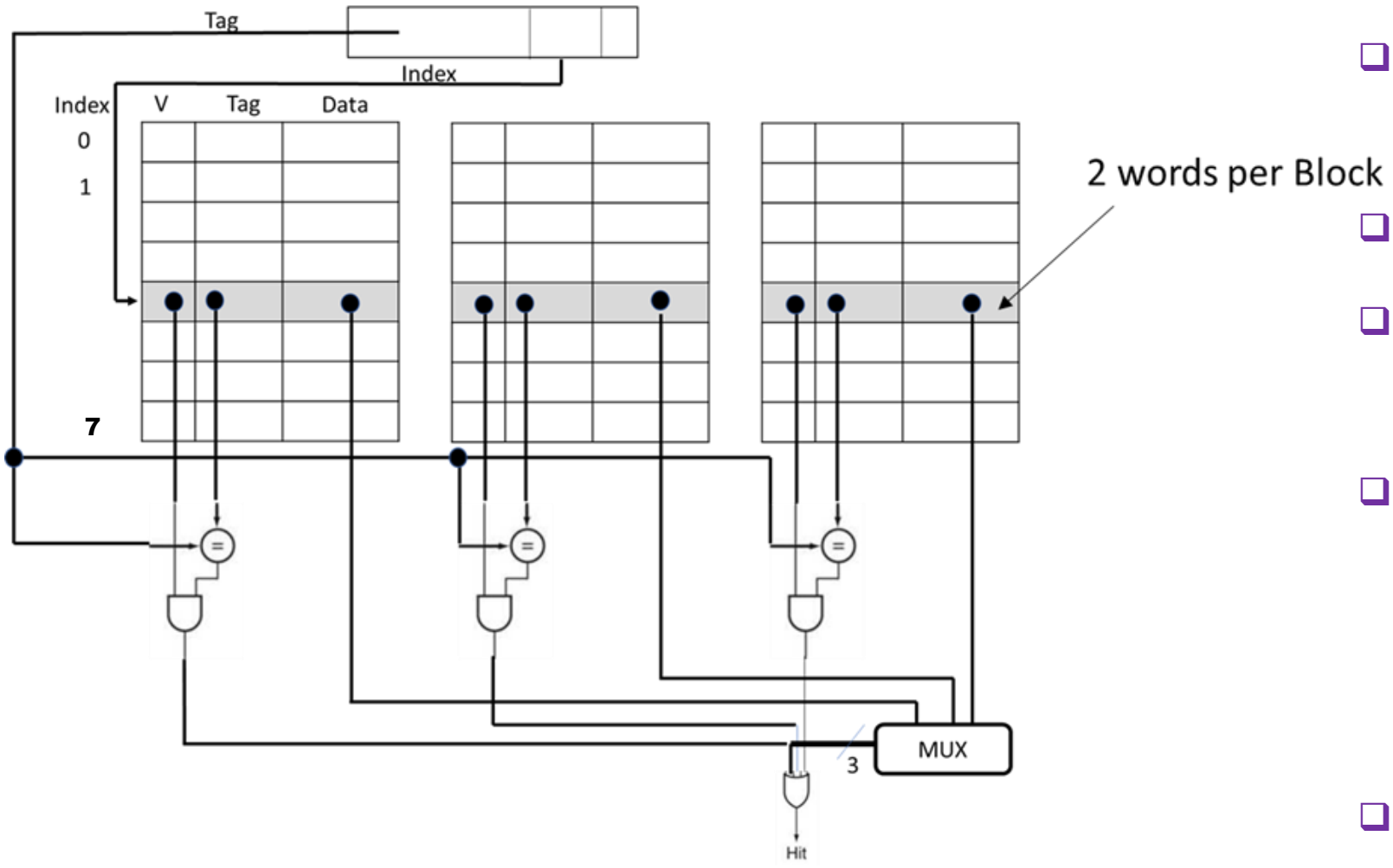
Associative Caches

- This exercise examines the effect of different cache designs, specifically comparing associative caches to the direct-mapped caches from *Section 5.4*.
- Given a sequence of word address shown:

```
0x03, 0xb4, 0x2b, 0x02, 0xbe, 0x58, 0xbf, 0x0e, 0x1f, 0xb5,
                    0xbf, 0xba, 0x2e, 0xce
```

- Sketch the organization of a three-way set associative cache with two-word blocks and a total size of 48 words
- Show the width of the tag and data fields

3-Way Set Associative Cache



- A DM cache would have 24 lines 2 Blocks each
- 3-way set associative
- 8 lines of 2 Words in each of 3 sets
- 1 bit (pick Word in Block) + 3 index bits (1 of 8 lines) and the remainder (4 bits) for Tag
- Assume LRU policy

Trace the behavior of the cache

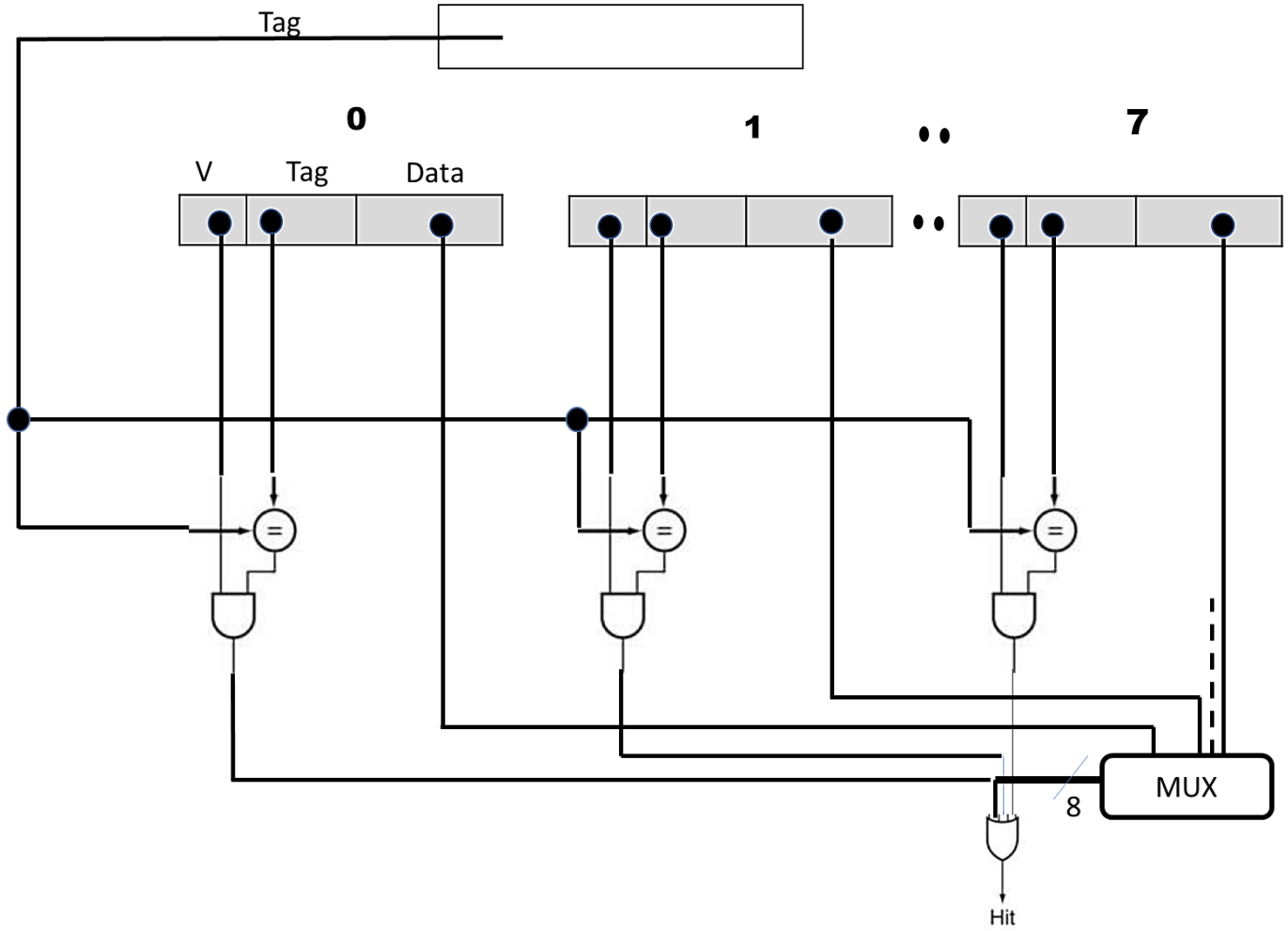
- Assume a true LRU replacement policy.
- For each memory reference, identify
  - *the binary word address,*
  - *the tag,*
  - *the index,*
  - *the offset*
  - *whether the reference is a hit or a miss, and*
  - *which tags are in each way of the cache after the reference has been handled.*

Cache Behavior in response to given Memory Refs

Hex	Binary	Tag	Index	Offset	Hit / Miss
0x03	0000 001 1	0x0	1	1	M
0xb4	1011 010 0	0xb	2	0	M
0x2b	0010 101 1	0x2	5	1	M
0x02	0000 001 0	0x0	1	0	H
0xbe	1011 111 0	0xb	7	0	M
0x58	0101 100 0	0x5	4	0	M
0xbf	1011 111 1	0xb	7	1	H
0x0e	0000 111 0	0x0	7	0	M
0x1f	0001 111 1	0x1	7	1	M
0xb5	1011 010 1	0xb	2	1	H
0xbf	1011 111 1	0xb	7	1	H
0xba	1011 101 0	0xb	5	0	M
0x2e	0010 111 0	0x2	7	0	M
0xce	1100 111 0	0xc	7	0	M

Fully Associative 8-W Cache with 1-Word Blocks

- Sketch the Organization – show width of tag and data fields
- A fully associative cache does not have an index or offset fields in the address
- All of the bits in the cache are a single bit for the Valid bit (V), Tag bits (64) and Data bits (64)



Trace the behavior of the cache

- Assume a true LRU replacement policy.
- For each memory reference, identify
  - *the binary word address,*
  - *the tag,*
  - *whether the reference is a hit or a miss, and*
  - *contents of the cache after each reference has been handled*

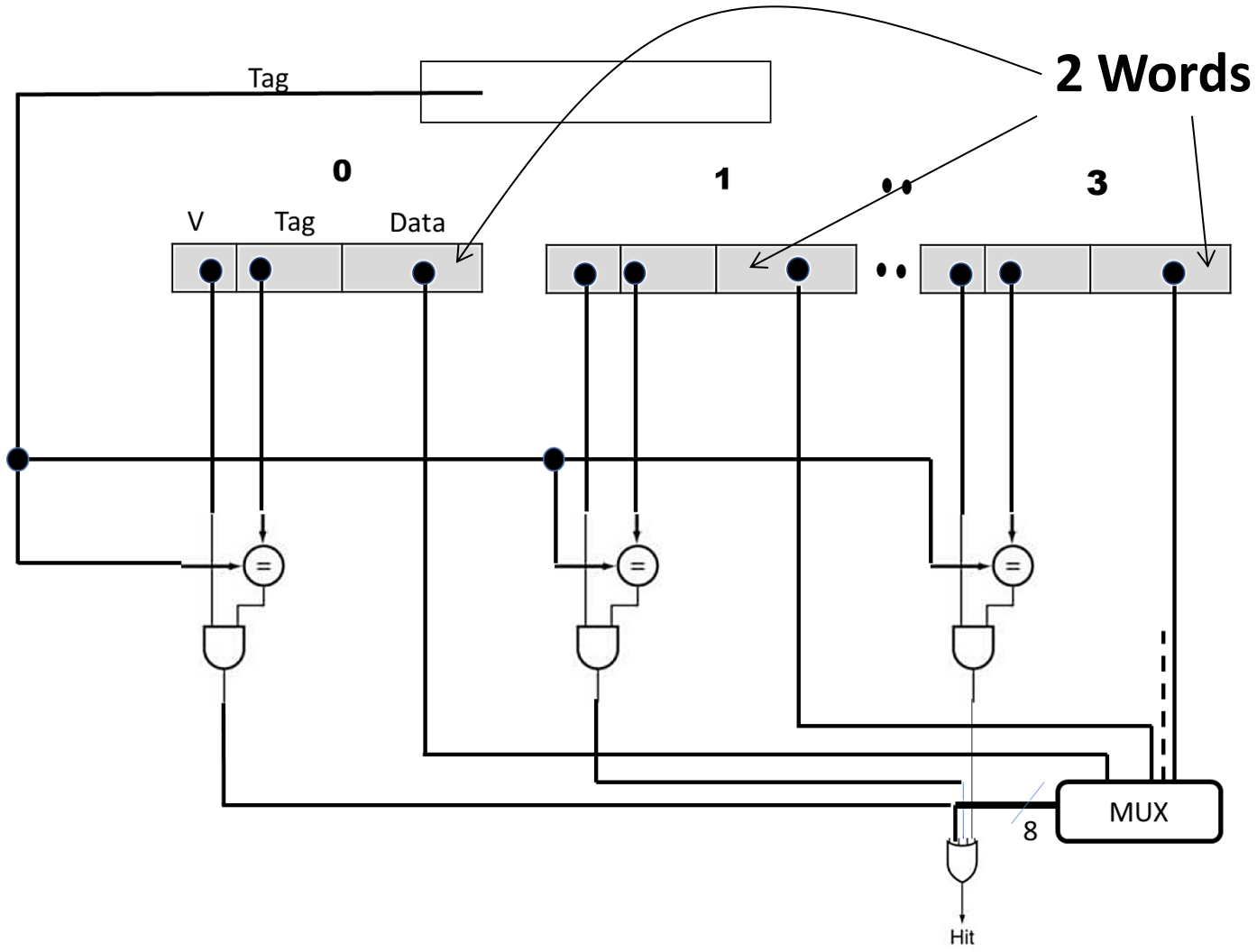
Trace the behavior of the cache

- Cache has a total size of **eight words.**
- Because this cache is **fully associative** and has **one-word blocks,** *there is no index and no offset*. Consequently, the *word address is equivalent to the tag*

Hex	Binary	Tag	Hit/Miss	Content
0x03	0000 0011	0x03	M	3
0xb4	1011 0100	0xb4	M	3, b4
0x2b	0010 1011	0x2b	M	3, b4, 2b
0x02	0000 0010	0x02	M	3, b4, 2b, 2
0xbe	1011 1110	0xbe	M	3, b4, 2b, 2, be
0x58	0101 1000	0x58	M	3, b4, 2b, 2, be, 58
0xbf	1011 1111	0x**bf**	M	3, b4, 2b, 2, be, 58, **bf**
0x0e	0000 1110	0x0e	M	3, b4, 2b, 2, be, 58, **bf**, e
0x1f	0001 1111	0x1f	M	b4, 2b, 2, be, 58, **bf**, e, 1f
0xb5	1011 0101	0xb5	M	2b, 2, be, 58, **bf**, e, 1f, b5
0xbf	1011 1111	0x**bf**	H	2b, 2, be, 58, e, 1f, b5, **bf**
0xba	1011 1010	0xba	M	2, be, 58, e, 1f, b5, bf, ba
0x2e	0010 1110	0x2e	M	be, 58, e, 1f, b5, bf, ba, 2e
0xce	1100 1110	0xce	M	58, e, 1f, b5, bf, ba, 2e, ce

Fully Associative 8-W Cache with 2-Word Blocks

- Sketch the Organization – show width of tag and data fields
- A fully associative cache does not have an index or offset fields in the address
- All of the bits in the cache are a single bit for the Valid bit (V), Tag bits (63) and Data bits (128)



Trace the behavior of the cache

- For each Memory reference, identify
- *the binary word address,*
- *the tag,*
- *the index,*
- *the offset,*
- *whether the reference is a hit or a miss,*
- *the contents of the cache after each reference has been handled*
- *Assume LRU replacement policy*

Trace the Behavior of the Cache

Hex	Binary	Tag	Offset	Hit/Miss	Content
0x03	**000 0001** 1	**0x01**	1	M	**[2,3]**
0xb4	101 1010 0	0x5a	0	M	**[2,3],** [b4,b5]
0x2b	001 0101 1	0x15	1	M	**[2,3],** [b4,b5], [2a,2b]
0x02	**000 0001** 0	**0x01**	0	**H**	[b4,b5], [2a,2b], **[2,3]**
0xbe	101 1111 0	0x5f	0	M	[b4,b5], [2a,2b] [2,3], **[be,bf]**
0x58	010 1100 0	0x2c	0	M	[2a,2b] [2,3], **[be,bf],** [58,59]
0xbf	101 1111 1	0x5f	1	**H**	[2a,2b] [2,3], [58,59], **[be,bf]**
0x0e	000 0111 0	0x07	0	M	[2,3], [58,59], **[be,bf],** [e,f]
0x1f	000 1111 1	0x0f	1	M	[58,59], **[be,bf],** [e,f], [1e,1f]
0xb5	101 1010 1	0xb5	1	M	**[be,bf],** [e,f], [1e,1f], [b4,b5]
0xbf	101 1111 1	0xbf	1	**H**	[e,f], [1e,1f], [b4,b5], **[be,bf]**
0xba	101 1101 0	0xba	0	M	[1e,1f], [b4,b5], [be,bf], [ba,bb]
0x2e	001 0111 0	0x2e	0	M	[b4,b5], [be,bf], [ba,bb], [2e,2f]
0xce	110 0111 0	0xce	0	M	[be,bf], [ba,bb], [2e,2f], [ce,cf]

Repeat for MRU (Most Recently Used) Replacement

- For the first memory reference in Hex: `0x03`, if we invert the last bit in this address string (to get the address of the second word in the Block) we would have the memory reference in Hex: `0x02`. Thus these 2 memory references are identified in the Content column – for this pair and for following pairs.
- In the Content column, the pair of memory references in italics correspond to the candidate words that are replaced in the next cycle with an MRU replacement policy. Content column **entries in bold** correspond to entries that are responsive to a hit – that were loaded into the cache from previous memory requests and are requested again

Repeat for MRU (Most Recently Used) Replacement

Hex	Binary	Tag	Offset	Hit/Miss	Content
0x03	**000 0001** 1	**0x01**	1	M	[2,3]
0xb4	**101 1010** 0	**0x5a**	0	M	[2,3], [b4,b5]
0x2b	001 0101 1	0x15	1	M	**[2,3],** [b4,b5], [2a,2b]
0x02	**000 0001** 0	**0x01**	0	**H**	[b4,b5], [2a,2b], **[2,3]**
0xbe	101 1111 0	0x5f	0	M	[b4,b5], [2a,2b] [2,3], *[be,bf]*
0x58	010 1100 0	0x2c	0	M	[b4,b5], [2a,2b] [2,3], *[58,59]*
0xbf	101 1111 1	0x5f	1	M	[b4,b5], [2a,2b] [2,3], *[be,bf]*
0x0e	000 0111 0	0x07	0	M	[b4,b5], [2a,2b] [2,3], *[e,f]*
0x1f	000 1111 1	0x0f	1	M	**[b4,b5],** [2a,2b] [2,3], [1e,1f]
0xb5	**101 1010** 1	**0x5a**	1	**H**	[2a,2b] [2,3], [1e,1f], ***[b4,b5]***
0xbf	101 1111 1	0x5f	1	M	[2a,2b] [2,3], [1e,1f], *[be,bf]*
0xba	101 1101 0	0x5d	0	M	[2a,2b] [2,3], [1e,1f], *[ba,bb]*
0x2e	001 0111 0	0x17	0	M	[2a,2b] [2,3], [1e,1f], *[2e,2f]*
0xce	110 0111 0	0x67	0	M	[2a,2b] [2,3], [1e,1f], *[ce,cf]*

Repeat using Optimal Replacement Policy

- **Optimal Replacement Policy: One that yields lowest miss-rate**

- For the 6th memory reference, 0x58, the **LRU** policy used
- For the 8th memory reference, 0x0e, the **MRU** policy used
- For the 9th memory reference, 0x1f, the **MRU** policy used
- For the 12-14th memory reference, 0xbf-0xce, the **LRU** policy used

Repeat using Optimal Replacement Policy

	Hex	Binary	Tag	Offset	Hit/Miss	Content
1	0x03	**000 0001** 1	**0x01**	1	M	[2,3]
2	0xb4	**101 1010** 0	**0x5a**	0	M	[2,3], [b4,b5]
3	0x2b	001 0101 1	0x15	1	M	[2,3], [b4,b5], [2a,2b]
4	0x02	**000 0001** 0	**0x01**	0	**H**	[2,3], [b4,b5], [2a,2b]
5	0xbe	101 1111 0	0x5f	0	M	*[2,3],* [b4,b5], [2a,2b] [be,bf]
6	0x58	010 1100 0	0x2c	0	M	[58,59], [b4,b5], [2a,2b], [be,bf]
7	0xbf	101 1111 1	0x5f	1	**H**	*[58,59],* [b4,b5], [2a,2b], [be,bf]
8	0x0e	000 0111 0	0x07	0	M	*[e,f],* [b4,b5], [2a,2b], [be,bf]
9	0x1f	000 1111 1	0x0f	1	M	[1e,1f], [b4,b5], [2a,2b], [be,bf]
10	0xb5	**101 1010 1**	**0x5a**	1	**H**	[1e,1f], **[b4,b5],** [2a,2b], [be,bf]
11	0xbf	101 1111 1	0x5f	1	**H**	[1e,1f], *[b4,b5],* [2a,2b], **[be,bf]**
12	0xba	101 1101 0	0x5d	0	M	[1e,1f], [ba,bb], *[2a,2b],* [be,bf]
13	0x2e	001 0111 0	0x17	0	M	[1e,1f], [ba,bb], [2e,2f], *[be,bf]*
14	0xce	110 0111 0	0x67	0	M	[1e,1f], [ba,bb], [2e,2f], [ce,cf]

Multilevel Caches

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

Multilevel Cache Example

- Suppose we have a processor with a base CPI of 1.0, assuming all references hit in the primary cache, and a clock rate of 4 GHz.
- Assume a main memory access time of 100 ns, including all the miss handling.
- Suppose the miss rate per instruction at the primary cache is 2%. How much faster will the processor be if we add a secondary cache that has a 5- ns access time for either a hit or a miss and is large enough to reduce the miss rate to main memory to 0.5%?

Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just primary cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9

Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
- CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
- Performance ratio = 9/3.4 = **2.6**

Cache Size Impact on Performance

- Access time proportional to cache size
  - more bits, longer wires, more decode gate delays etc.
- Miss rate lower for larger cache size
  - more likely to find data in the cache if it is bigger, even though it is slower
- So **what optimal value** of each enables best Cache performance?

	L1 Size	L1 Miss Rate	L1 Hit Time
P1	2 KiB	8.0%	0.66 ns
P2	4 KiB	6.0%	0.90 ns

- Assume: RAM takes 70ns to access, 36% of all instructions access data cache. Table above is data for 2 processors P1, P2

Clock rates for P1, P2

- Assuming L1 access determines cycle time for P1, P2 (that is assuming no need to go to RAM) what are the clock rates for P1, P2?
- Clock rate *is generally limited by the L1 cache* – since the access to the L1 must complete within 1 clock cycle.
- Smaller L1s permit faster clocks, so:
- Cycle times for P1 and P2 : **P1: 1.515 GHz; P2: 1.11 GHz**

AMAT for P1, P2 (without L2)

- What is AMAT for P1, P2 assuming the RAM Latency given and hit rates in table?
- Average Memory Access Time (in cycles)
- = # of cycles for a hit + Miss Rate x Miss Penalty Miss penalty
  - for P1 = 70ns/0.66ns = 107 cycles Miss penalty
  - for P2 = 70ns/0.90ns = 78 cycles
  - P1: AMAT = 1 + 0.08 x 107 cycles = 9.56 cycles or 6.31 ns
  - P2: AMAT = 1 + 0.06 x 78 cycles = 5.68 cycles or 5.11 ns

CPI for P1 and P2?

- base CPI = 1.0 without any memory stalls
- Total CPI for P1 and P2 are determined by calculating:
- For **P1:**
  - Each instruction requires 1 cycle for a hit 1 cycle
  - If the instruction fetch from ***L1 Instruction memory*** misses at Miss Rate of 8%, the instruction incurs an additional delay of 0.08 x 107 cycles = 8.56 cycles
  - 36% of the instructions also require access to ***L1 Data Memory*** at a Miss rate of 8% which incur an additional delay of + 0.36 x 0.08 x 107 cycles = 3.08 cycles
  - So, 1 + 0.08 x 107 + 0.36 x 0.08 x 107 = ***12.64 cycles @ 0.66ns/cycle = 8.34ns***
- For **P2:**
  - Each instruction requires 1 cycle for a hit 1 cycle
  - If the instruction fetch from ***L1 Instruction memory*** misses at Miss Rate of 6%, the instruction incurs an additional delay of 0.06 x 78 cycles = 4.68 cycles
  - 36% of the instructions also require access to Data Memory at a Miss rate of 6% which incur an additional delay of + 0.36 x 0.06 x 78 cycles = 1.68 cycles
  - So, 1 + 0.06 x 78 + 0.36 x 0.06 x 78 = ***7.36 cycles @ 0.66ns/cycle = 6.63ns***

AMAT for P1 with the addition of an L2 cache

L2 Size	L2 Miss Rate	L2 Hit Time
1 MiB	95%	5.62 ns

- An L2 access requires 9 cycles (5.62ns/0.66ns).
- All memory accesses require at least one cycle.
- 8% of memory accesses miss in the L1 cache and make an L2 access, which takes 9 cycles.
- 95% of all L2 access are misses and require a 107 cycle RAM lookup.
- AMAT = 1 + .08[9 + 0.95*107] = 9.85 cycles – worse than without the L2 (9.56 cycles)

AMAT for P2 with the addition of an L2 cache

L2 Size	L2 Miss Rate	L2 Hit Time
1 MiB	95%	5.62 ns

- An L2 access requires 7 cycles (5.62ns/0.9ns).
- All memory accesses require at least one cycle.
- 6% of memory accesses miss in the L1 cache and make an L2 access, which takes 7 cycles.
- 95% of all L2 access are misses and require a 78 cycle RAM lookup.
- AMAT = 1 + .06[7 + 0.95*78] = 5.87 cycles – worse than without the L2 (5.68 cycles)

Multilevel Cache Considerations

- Primary cache
  - Focus on minimal hit time
- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size

Example: 2-level Cache

- **Write Through, Write Back**
- **Write Allocate, No Write Allocate**

L1	L2
Write through, non-write allocate	Write back, write allocate

- Buffers b/w L1 & L2, b/w L2 & Memory?
  - The L1 cache has a low write miss penalty while the L2 cache has a high write miss penalty since the latency between RAM and L2 is much higher than the latency between L1 and L2.
  - A *write buffer between the L1 and L2 cache* would effectively pipeline the write to the L2 cache enabling it *to require only one cycle for the Write*.
  - Since the buffer would hold data to be written into L2 from L1 and can be designed to be deep enough, it can prevent stalls from subsequent write misses in L1.
  - The L2 cache would benefit from *write buffers between L1 and L2* when replacing a dirty block in L2, *since the new block would be read into and held by the buffer between L1 and L2 before* the dirty block is physically written to memory, only after which it could be overwritten in the L2 by the new block

Example 2-level Cache

- **Write Through, Write Back**
- **Write Allocate, No Write Allocate**

L1	L2
Write through, non-write allocate	Write back, write allocate

- Procedure of handling L1 write-miss, considering the components involved and the possibility of replacing a dirty block?
- On an L1 write miss, the word is written directly to L2 *without bringing its block into the L1 cache [write through with no write allocate policy]*. If this results in an L2 miss, its block must be brought into the L2 cache from Memory, possibly replacing a dirty block, which must first be written to memory

Memory Bandwidth for Read, Write in WT/WA cache

- Given:
  - Write-through, Write Allocate as the Write Hit & Write Miss policies
  - CPI =2 (or 0.5 Instructions/cycle), Block size = 64 Bytes
  - Reads:
    - 100% of instructions generate a Read Instruction Cache request with a 0.3% miss rate
    - 25% of instructions generate a Read Data Cache request with 2% miss rate
  - Writes:

Data Reads per 1000 Instructions	Data Writes per 1000 Instructions	Instruction Cache Miss Rate	Data Cache Miss Rate	Block Size (bytes)
250	100	0.30%	2%	64

Memory Bandwidth for Read

- What is the bandwidth b/w RAM and Instruction Cache? (note: only Read ops performed on Instruction memory)
  - 0.3% of instructions read from Instruction cache register a Read miss
  - So, 0.003 x 0.5 instructions /cycle = missed instructions per cycle
  - Block of 64 Bytes corresponding to the miss must be fetched from memory
  - So, [0.003 misses/I] x [0.5 I/cycle] x [64 bytes/miss] = 0.096 Bytes/cycle
- What is the Read bandwidth b/w RAM and Data Cache
  - 25% of instructions generate a read request to Data cache
  - [0.25 Reads/I] x [0.5 I/cycle] = 0.125 Reads/cycle
  - 2% of these miss: So, [0.02 misses/Read]x[0.125 Reads/cycle] = 0.0025 Read misses/cycle
  - Each miss requests 1 block (64 Bytes), So: [0.0025 Read miss/cycle]x64 Bytes = 0.16 Bytes/cycle

Memory Bandwidth for Write

- What is the Write bandwidth b/w RAM and Data Cache
  - 10% of instructions generate a write request to Data cache
  - [0.10 Writes/I] x [0.5 I/cycle] = 0.05 Writes/cycle
- Since this is a Write-through hit policy, all words written to data cache must be written to RAM
  - So, [0.05 Writes/cycle]x[8 Bytes/word] x [1 word/write-through] = 0.4Bytes/cycle write bandwidth consumed
- Since this is a Write Allocate Miss policy, A write miss (rate = 2%) must also Read from RAM the Block corresponding to the missing entry (to be written into) into the Data Cache
  - [0.05 Writes/cycle]x[0.02 misses/write]x[64Bytes/miss] = 0.064 Bytes/cycle
- Assuming each write miss attempts to write only 1 word to the Block brought in from RAM, into the RAM to update it:
  - [0.05 Writes/cycle]x[0.02 misses/write]x[8Bytes/word]x[1 word/miss] = 0.008 Bytes/cycle (included in above calculation of BW for all words written to RAM w Write-through

Total Bandwidth

- Read: 0.096 (Instruction memory) + 0.16 (data memory) + 0.064 (Write-miss in Write-through cache with Write Allocate) Bytes/cycle = 0.32 Bytes/cycle
- Write: 0.4Bytes/cycle (all Word Writes to Data Cache written to RAM)

Assume a Write Back Cache Hit Policy

- For a write-back, write-allocate cache, assuming 30% of replaced data cache blocks are dirty, what are the **read** and **write** bandwidths needed for a CPI of 2?
- With a write-back, write allocate cache*, data are only written to memory on a cache miss*.
- But, it is written to memory *on every cache miss* (both read and write), because any line could have dirty data when evicted, even if the eviction is caused by a read request
- assuming *30% of replaced data cache blocks are dirty* what are the read and write bandwidths needed assuming the same CPI?

Write Data Bandwidth

- # of Read & Write Accesses to Cache:
- [0.5 inst/cycle] x [0.10 Write Data Accesses/instruction + 0.25 Read Data Accesses/instruction] = 0.175 Accesses/cycle
- [0.175 Accesses/cycle] x [0.02 misses /Access] = 0.0035 misses/cycle

Of these misses, 30% are 'dirty' – that is they need to be updated in RAM before being evicted from Cache

- [0.0035 misses/cycle] x [0.3 blocks/miss] = 0.00105 blocks/cycle
- [0.00105 blocks/cycle] x [64 bytes/block] = =**0.0672 bytes/cycle**