Introduction to the Memory Hierachy

Azeez Bhavnagarwala

ECE 6913, Fall 2021

Computer Systems Architecture

October 14th 2021

NYU Tandon School of Engineering

Agenda

Memory Hierarchy

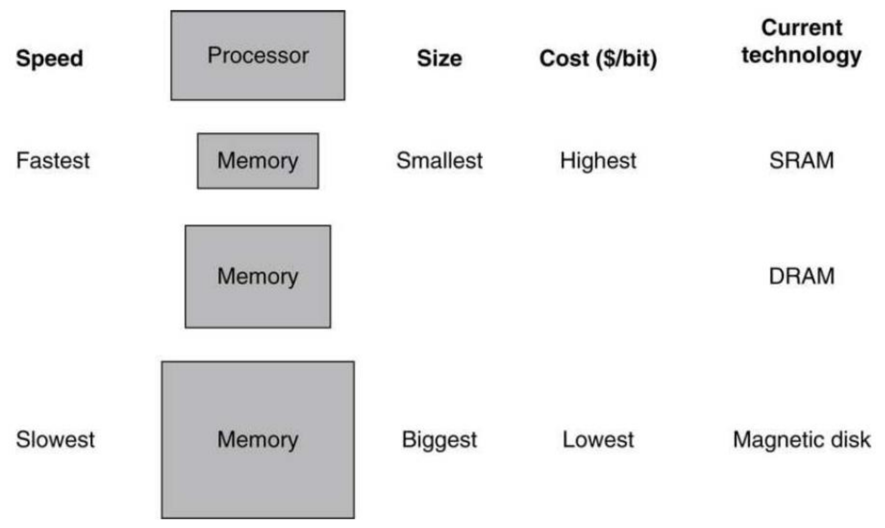
- Spatial & Temporal locality of data
- SRAM – the workhorse embedded memory technology for fast Cache memory
- Direct Mapped Cache Memory
- Write Policy
- Simple Cache Access Examples

Principle of Locality

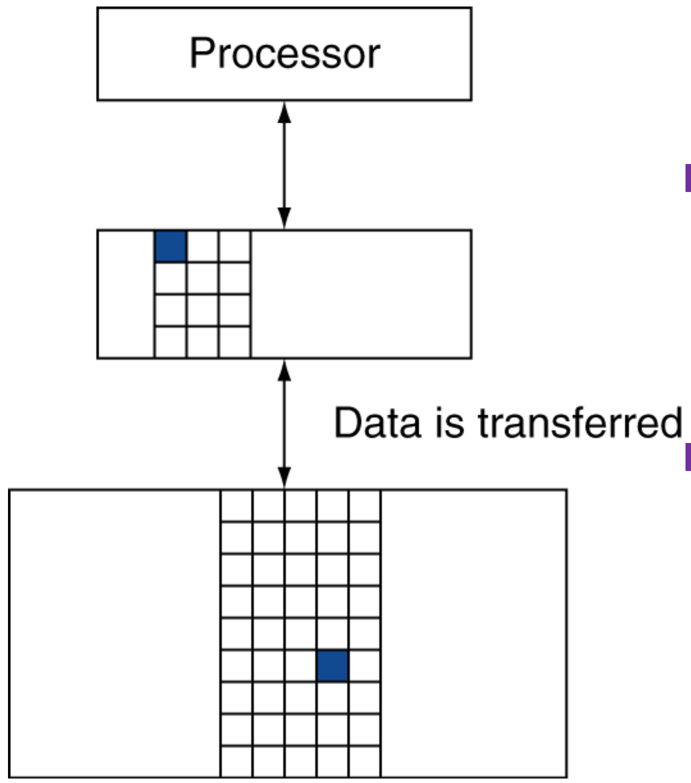
- Programs access a small proportion of their address space at any time
- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables
- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
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



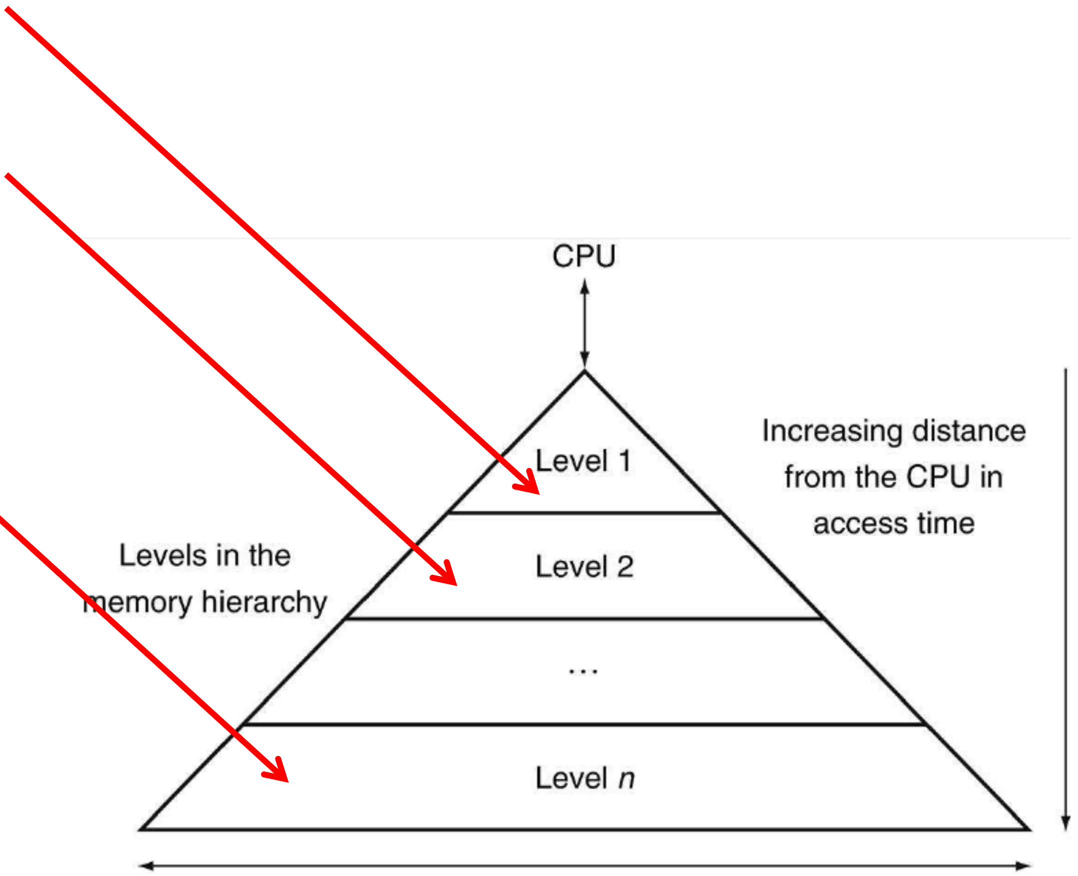
Memory Hierarchy Levels

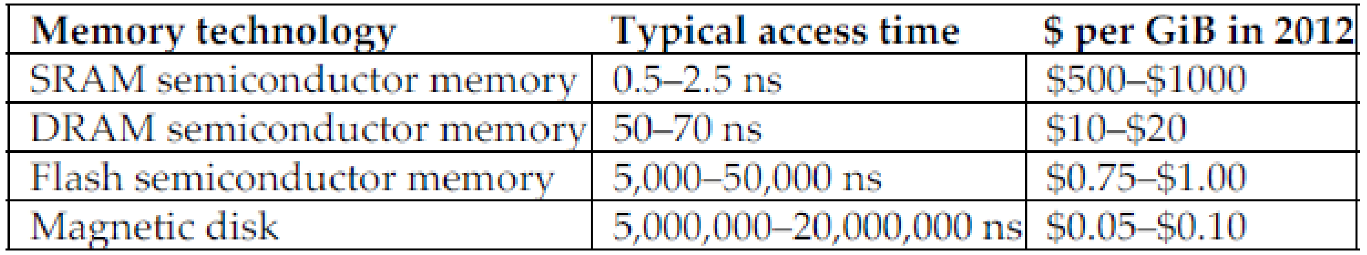


- Block (aka line): unit of copying
  - May be multiple words
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses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


CPU
Level 1
Level 2
...
Level n
Levels in the memory hierarchy
Increasing distance from the CPU in access time

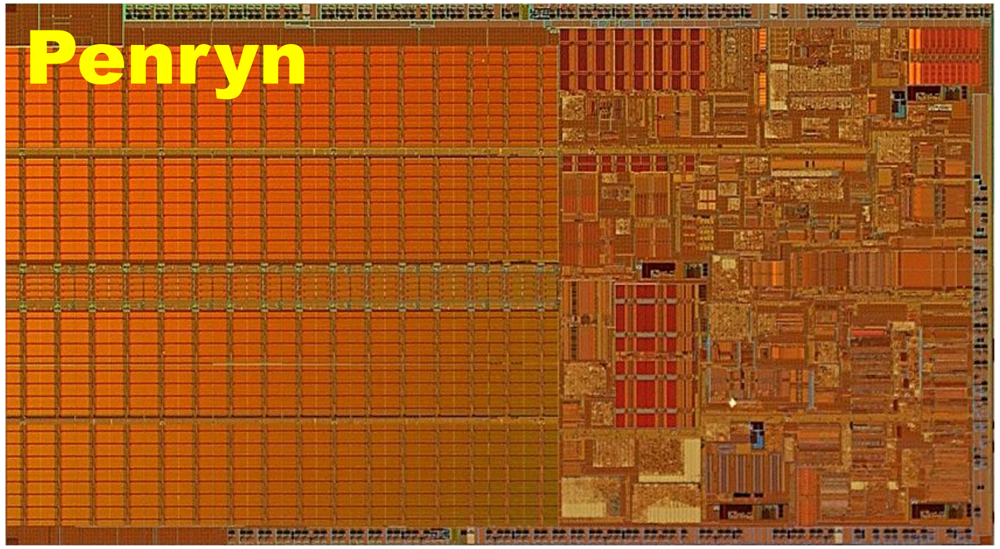


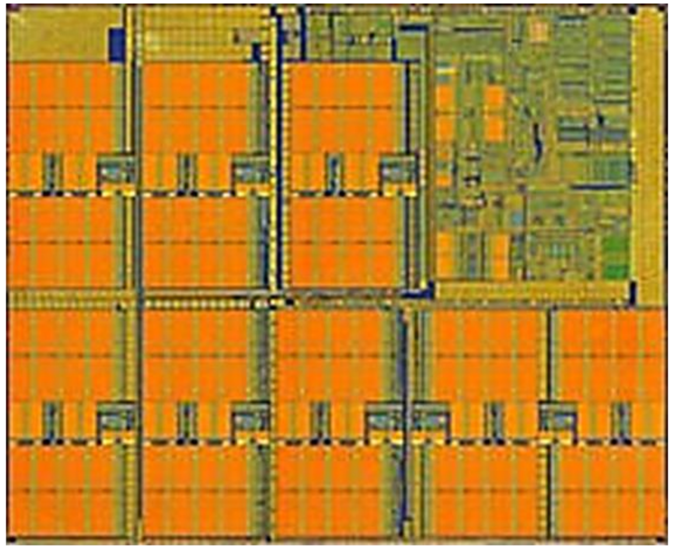
Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

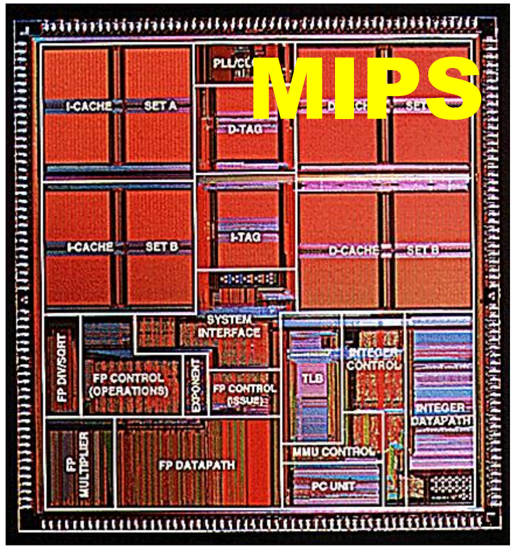
Why are SRAMs important?

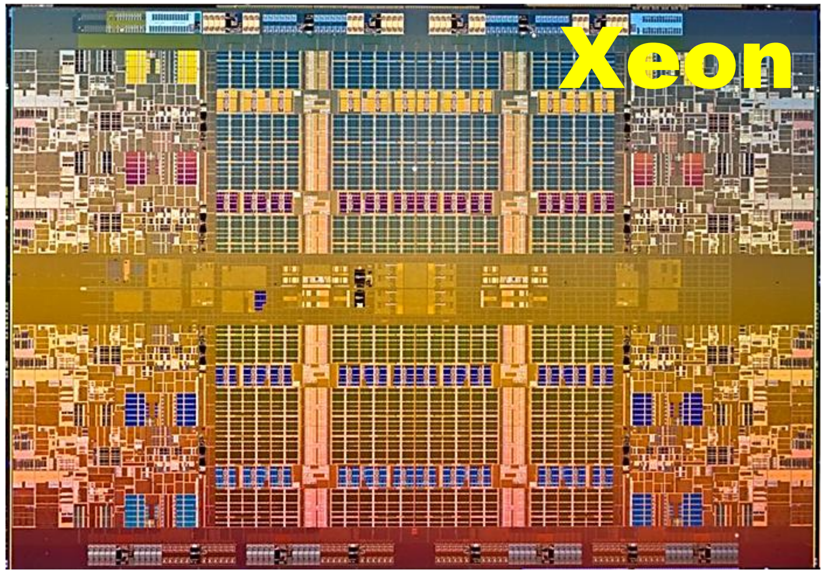
- SRAM Dominates area of Chip
  - Limits Chip Size & Cost, Chip wire-lengths
- SRAM dominates Fraction of ‘off’ transistors during active mode
  - Limits leakage power of Processor
- Clock Cycle time limited by SRAM access
  - Access Latency Limits single-thread speed of CPU, Cycle Time
- SRAM Margins 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

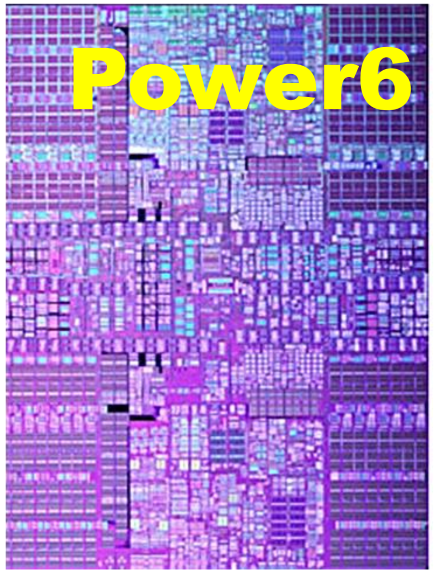
SRAM Dominates Die Size of Industry Leading CPUs

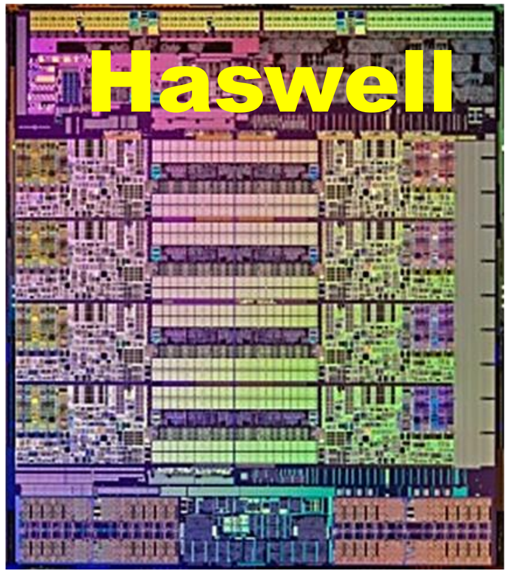


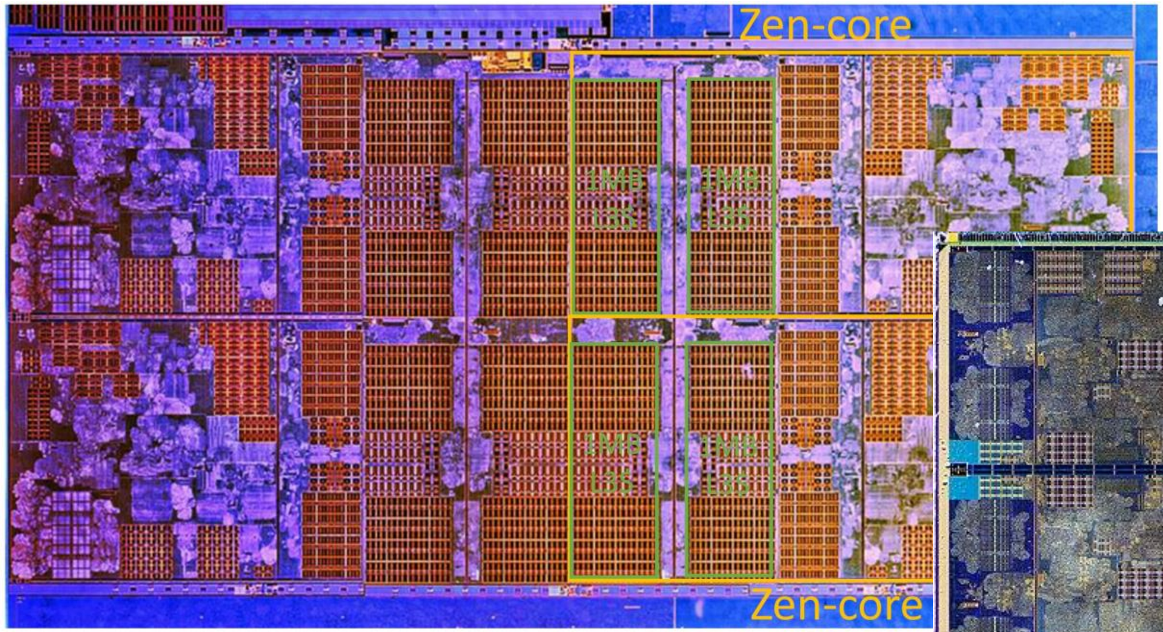


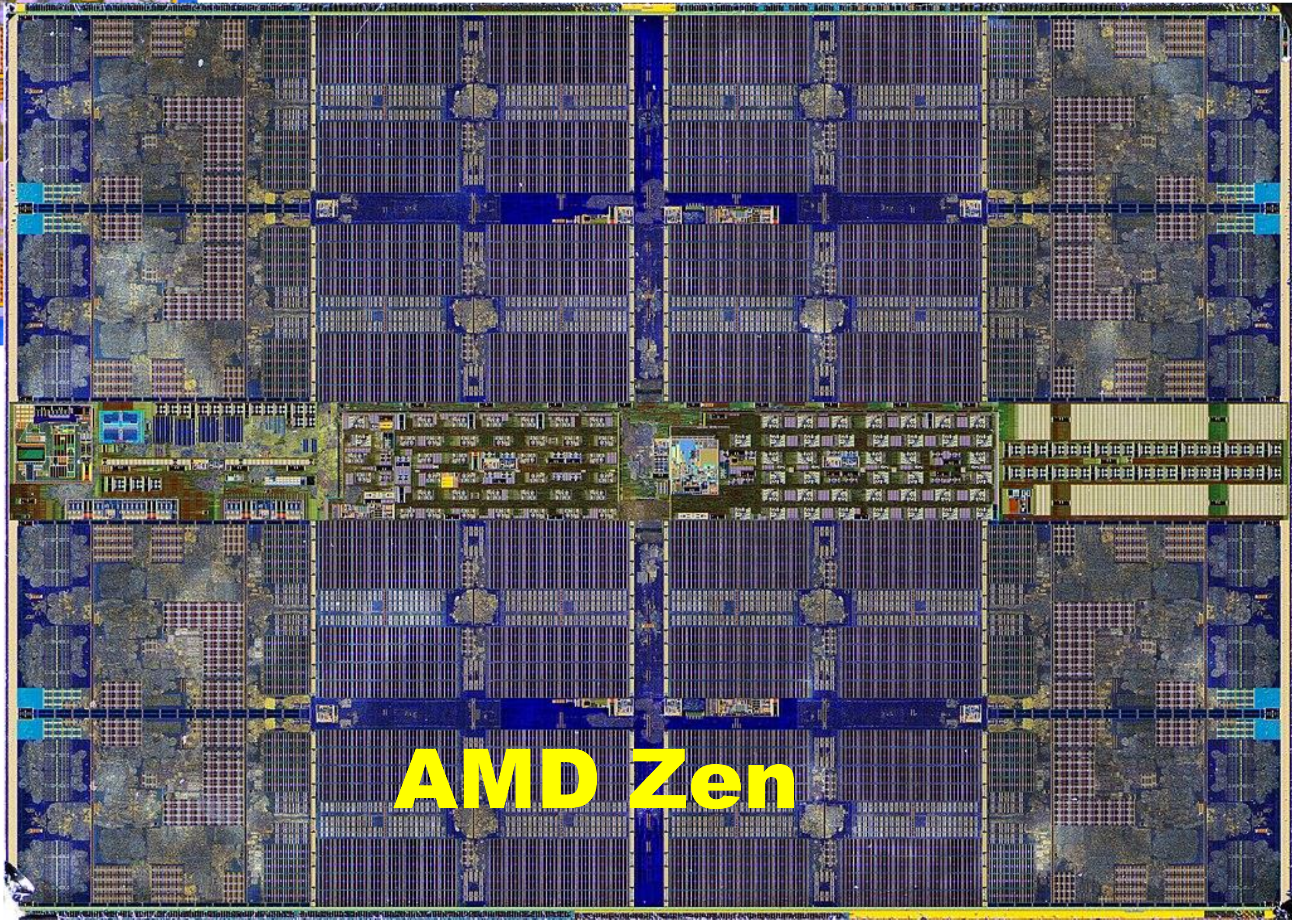






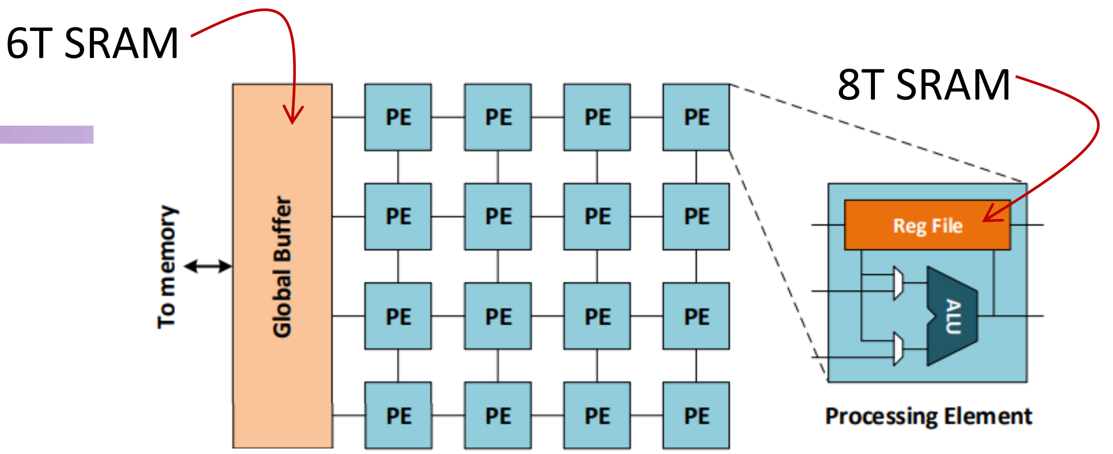


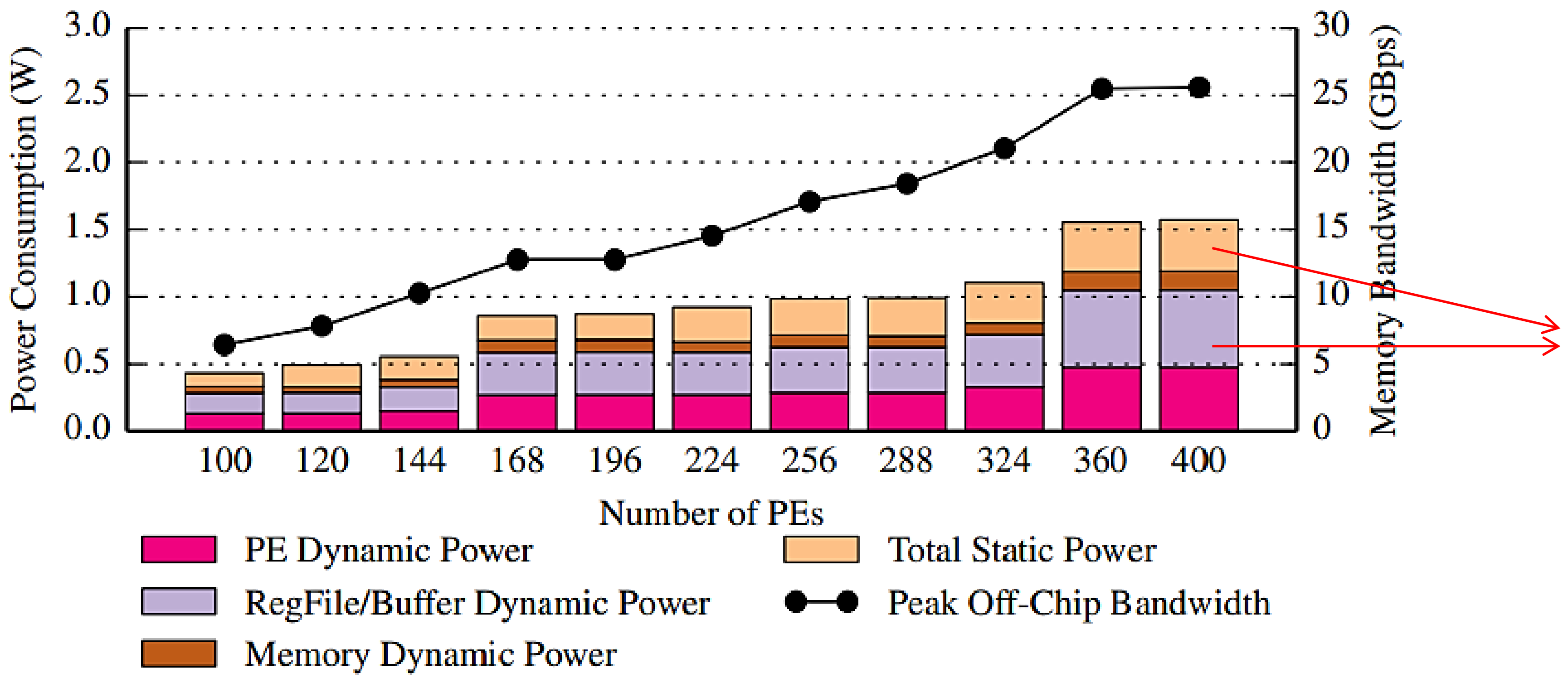

Zen-core
1MB L3$
1MB L3$
1MB L3$
1MB L3$
Zen-core


AMD Zen

SRAM in AI Hardware

- Memory bandwidth and power consumption when scaling the PE array size
- Mark Horowitz et al, TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory, ASPLOS 2017



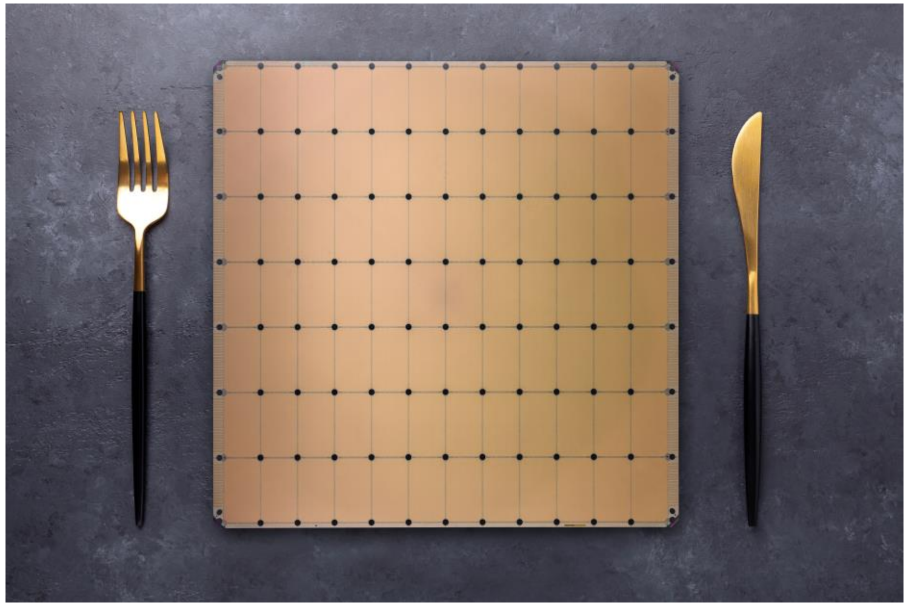


Contributions by SRAM

All SRAM Memory in Fastest AI 'chip'

- AI Hardware limited by model size, on-chip bandwidth
- Energy efficiency limited by off-chip DRAM access
- Cerebras can double the amount of computing its chips do for double the power, unlike current systems that *need more than twice as much power to double their computing capacity*

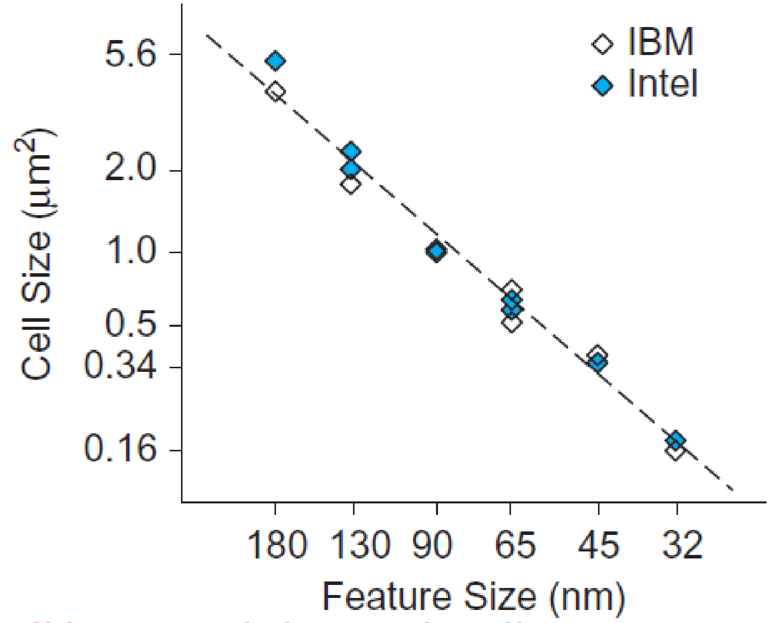
46,225	mm² silicon
2.6	Trillion transistors
850,000	AI optimized cores
40	Gigabytes on chip memory **- all SRAM**
20	Petabyte/s memory bandwidth
220	Petabit/s fabric bandwidth
1.2	Terabit/s ingest bandwidth
7nm	Process technology at TSMC



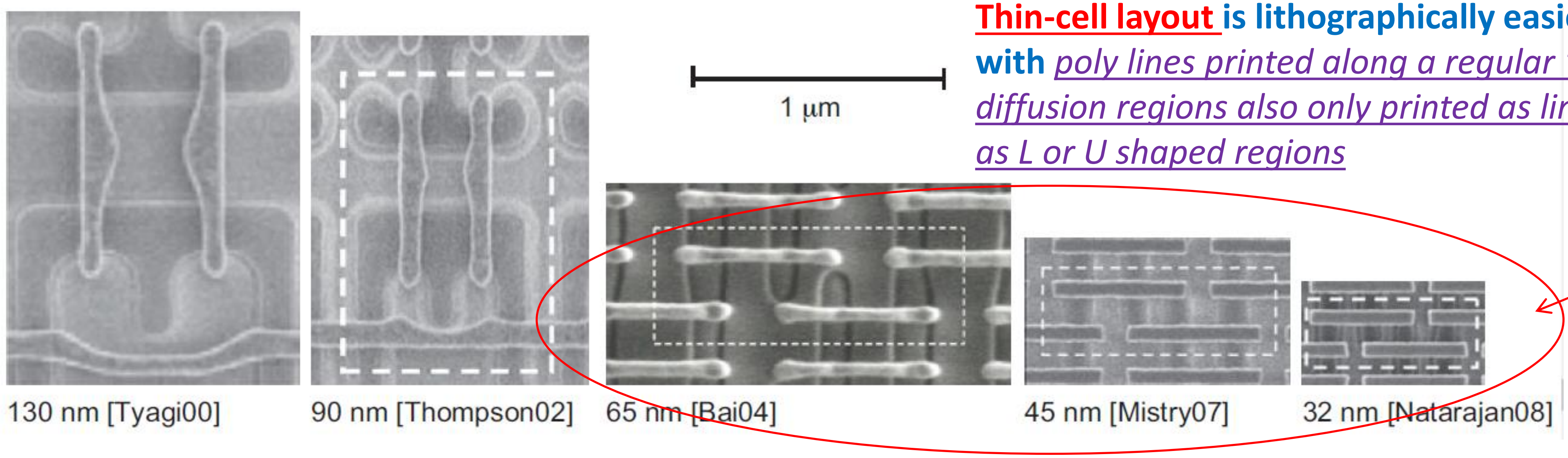
- 850,000 cores for sparse tensor operations
- Massive high bandwidth on-chip memory and interconnect orders of magnitude faster than a traditional cluster could possibly achieve

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



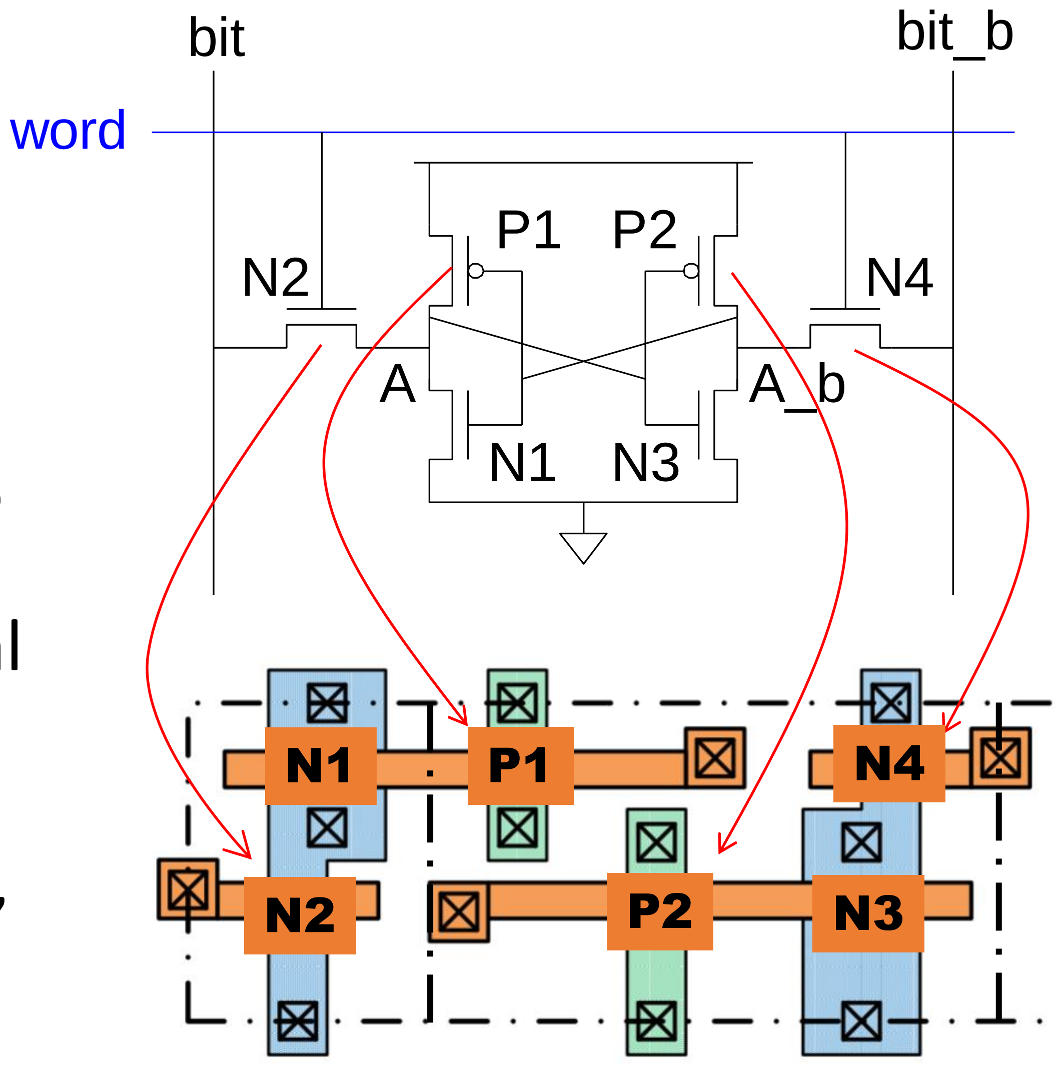
Thin-cell layout is lithographically easier to print with *poly lines printed along a regular 'pitch' and diffusion regions also only printed as lines and not as L or U shaped regions*



130 nm [Tyagi00] 90 nm [Thompson02] 65 nm [Bai04] 45 nm [Mistry07] 32 nm [Natarajan08]

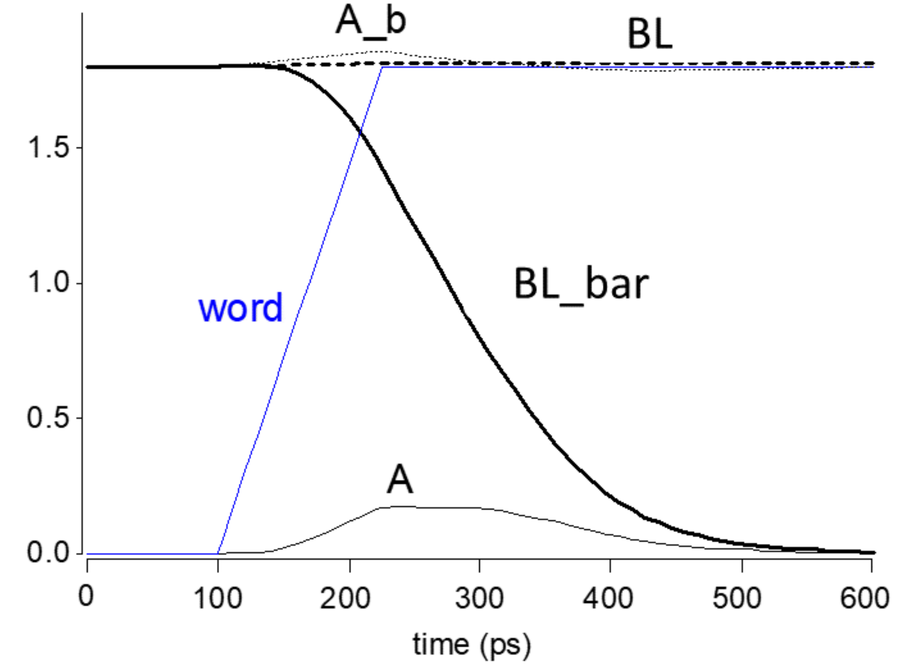
SRAM 6-Transistor ‘Thin’ cell

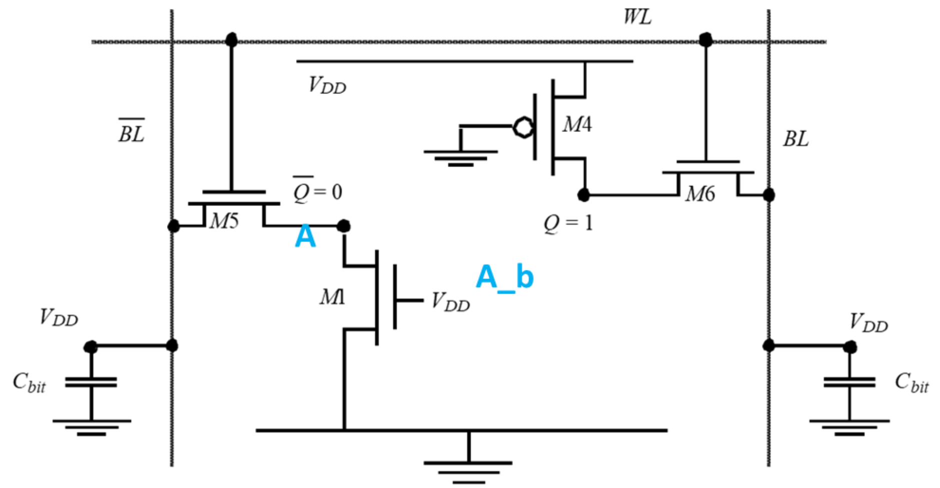
- Cell size accounts for most of array size
- Reduce cell size at expense of (process and design) complexity
- Data stored in cross-coupled inverters (N1-P1 an N3-P2)
- Access transistors N2 and N4 are selected by the WL connecting cell storage nodes to bitline pair
- Access transistors enable cell to 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

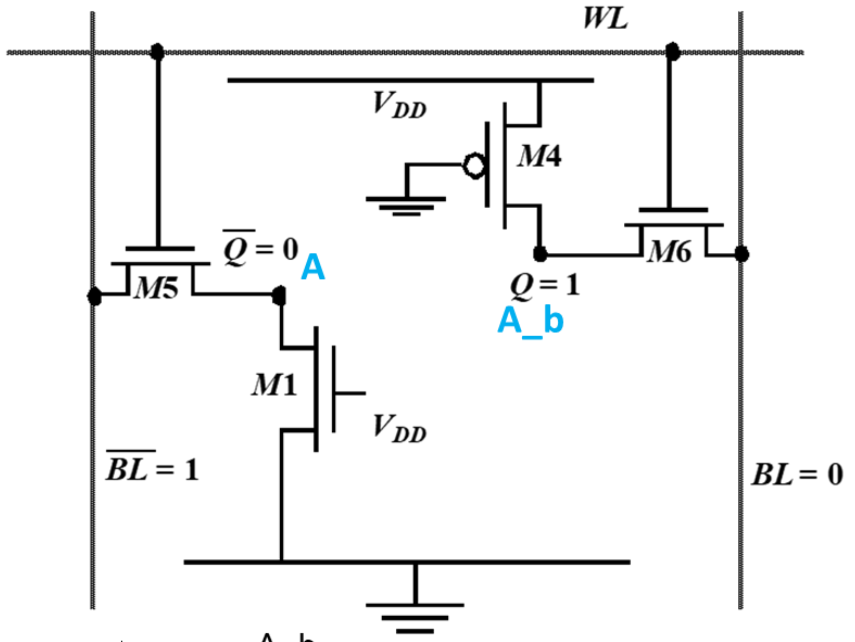
- Precharge both bitlines high
- Then turn on Word Line
- One of the two bitlines will be pulled down by the cell
  - BL_bar discharges, BL stays high
  - But A bumps up slightly
- *Read stability*
  - A, A_b must not flip

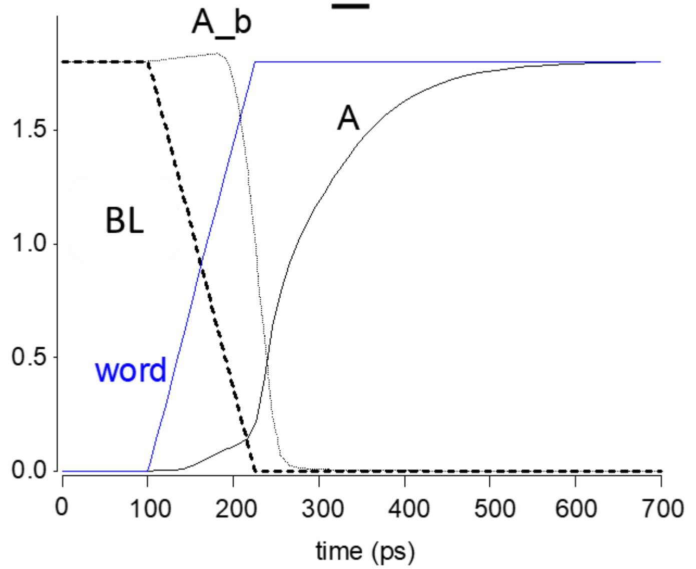




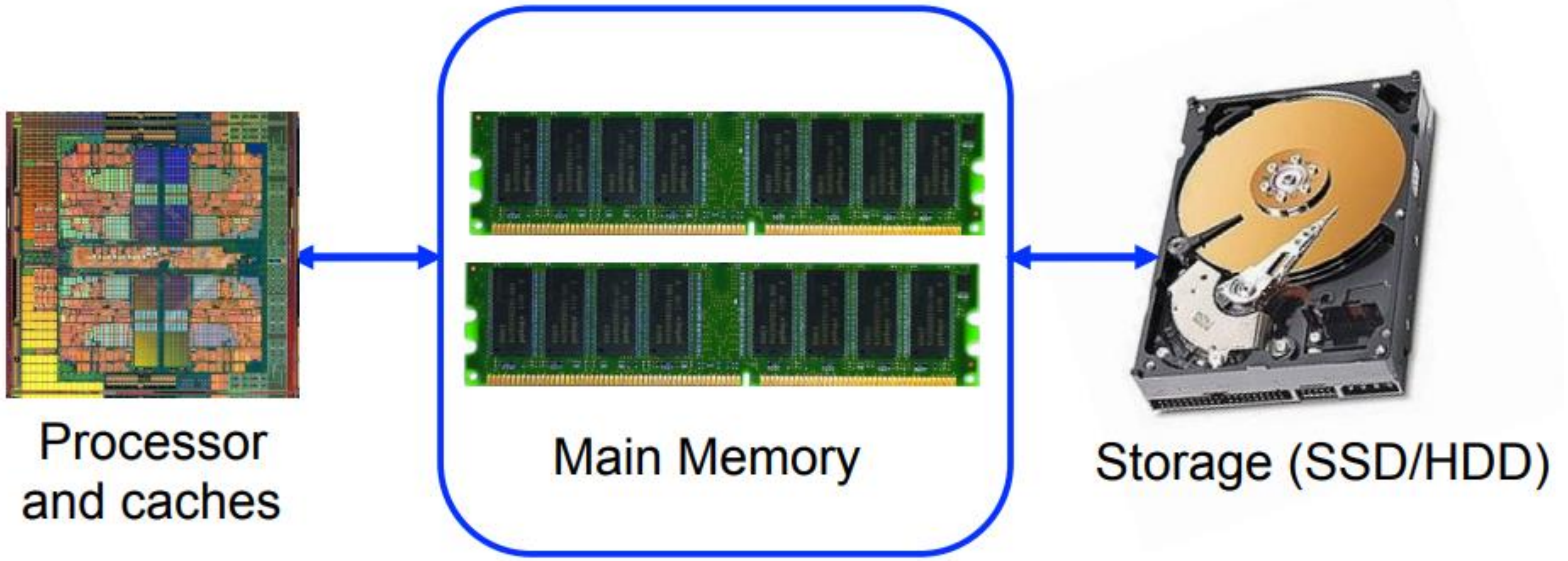
SRAM Cell Write Operation

- 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





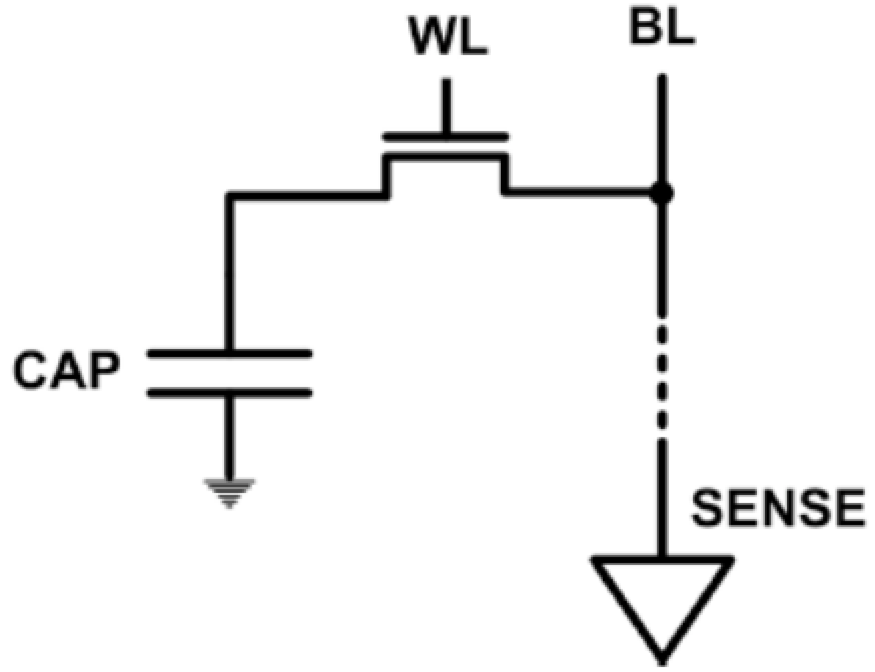
Main Memory



- 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

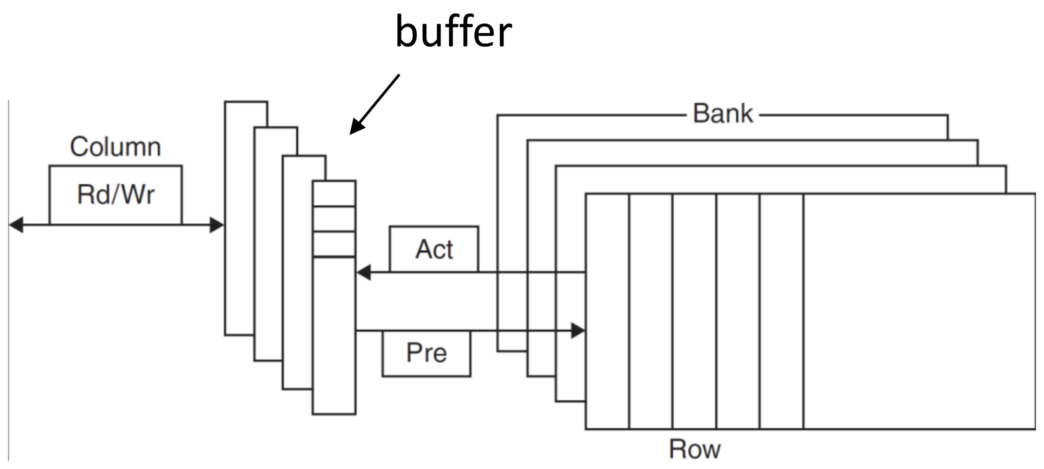
DRAM

- 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”

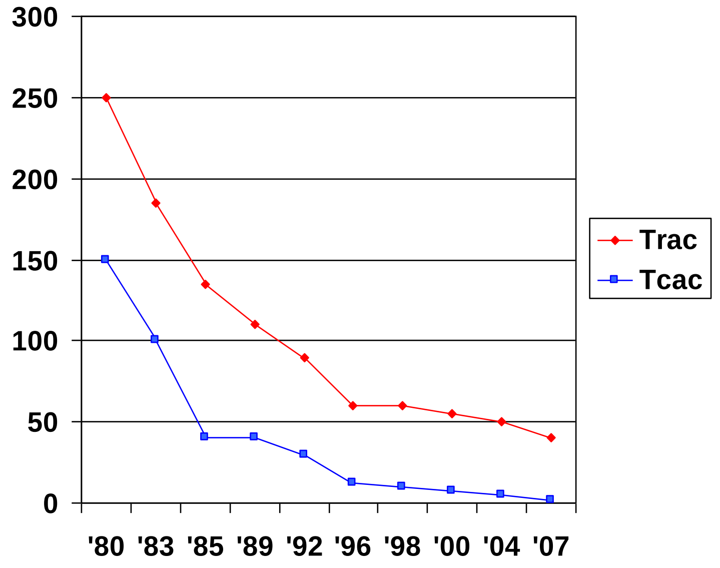


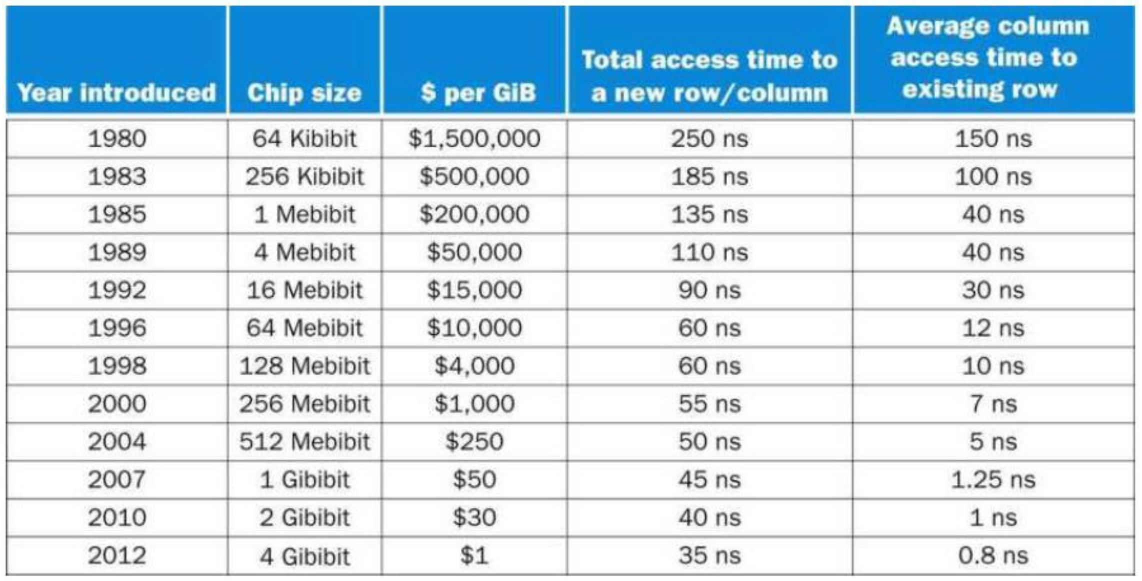
Advanced DRAM Organization

- 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



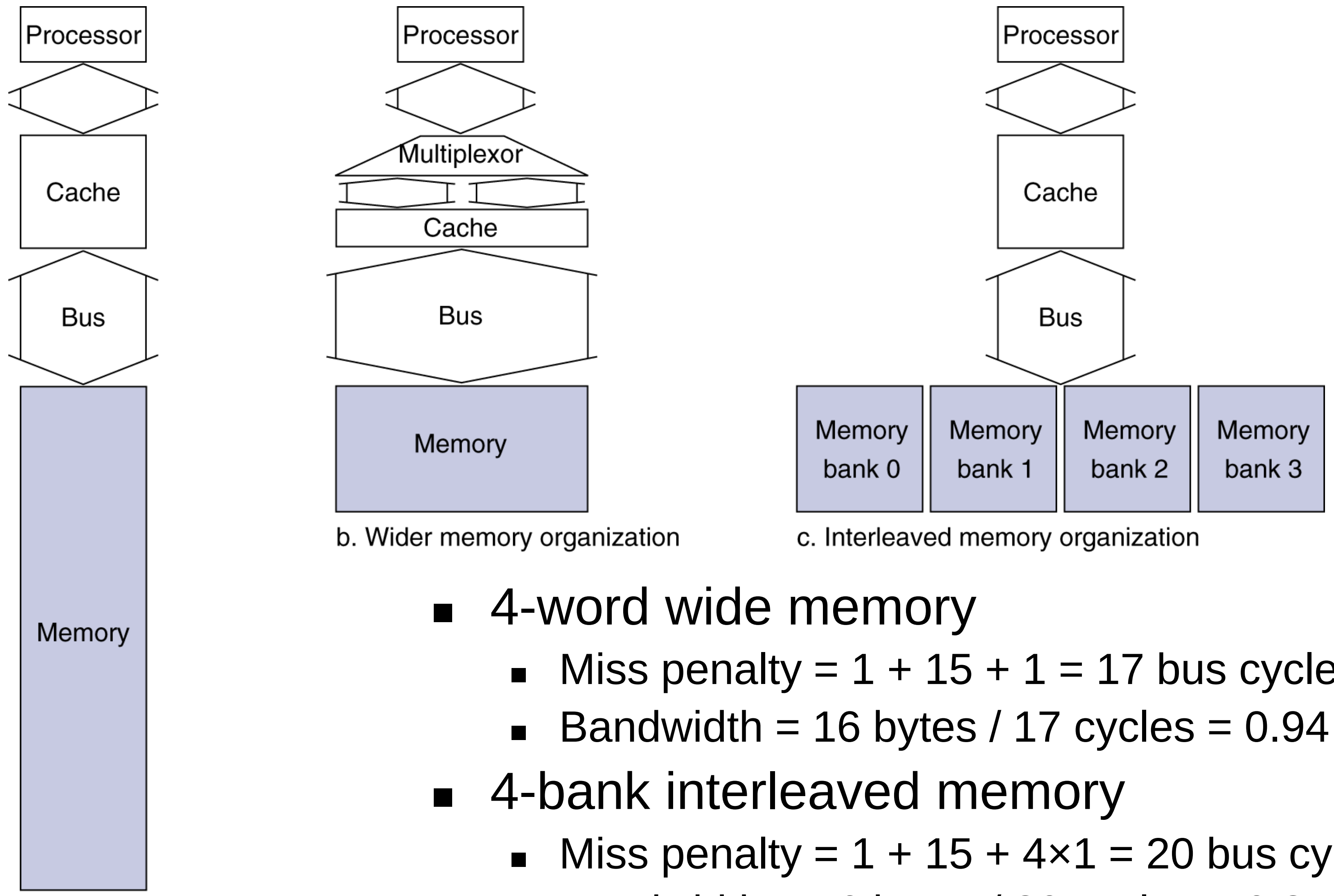


Year introduced	Chip size	$ per GiB	Total access time to a new row/column	Average column access time to existing row
1980	64 Kibibit	$1,500,000	250 ns	150 ns
1983	256 Kibibit	$500,000	185 ns	100 ns
1985	1 Mebibit	$200,000	135 ns	40 ns
1989	4 Mebibit	$50,000	110 ns	40 ns
1992	16 Mebibit	$15,000	90 ns	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

Increasing Memory Bandwidth



a. One-word-wide memory organization

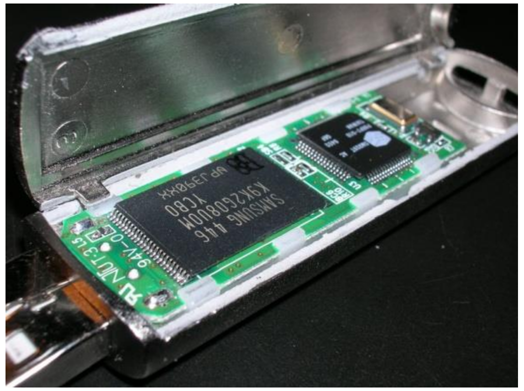
b. Wider memory organization

c. Interleaved memory organization

- 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)

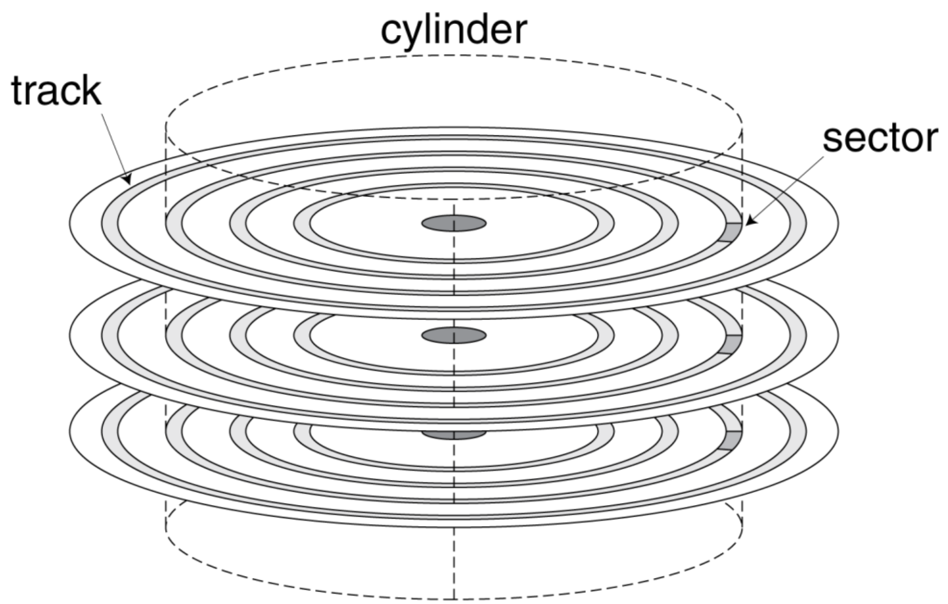


Flash Types

- 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

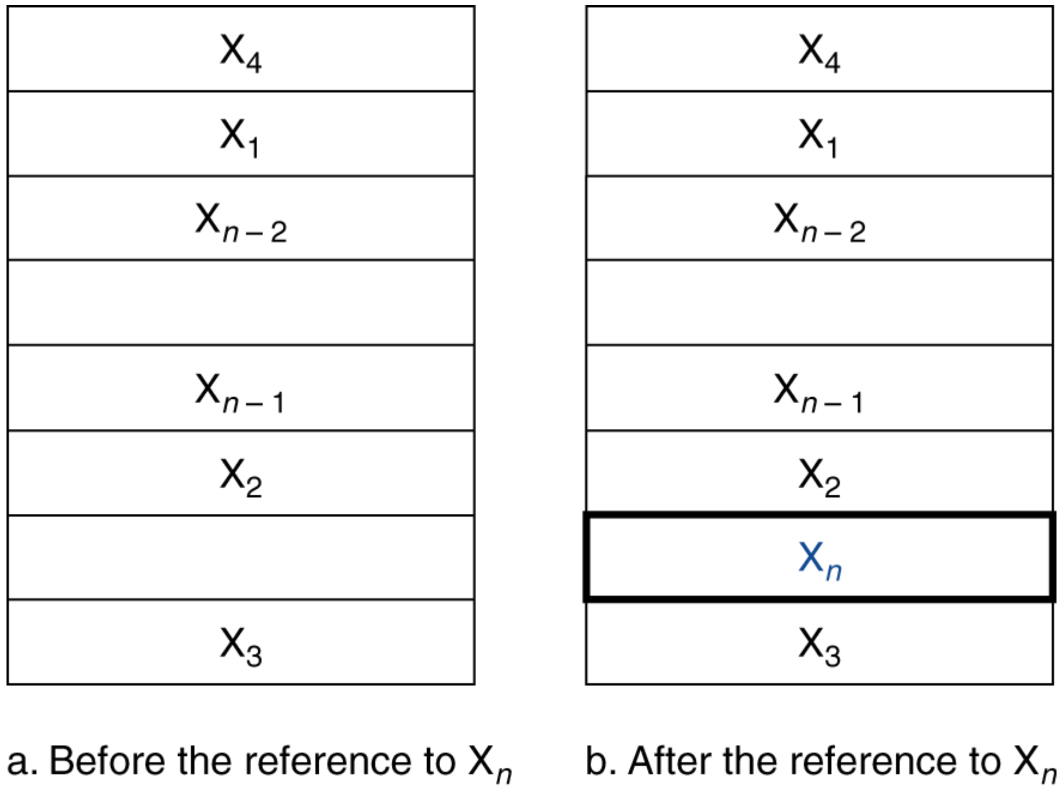
- Given
  - 512B sector, 15,000rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, idle disk
- Average read time
  - 4ms seek time
    + ½ / (15,000/60) = 2ms rotational latency
    + 512 / 100MB/s = 0.005ms transfer time
    + 0.2ms controller delay
    = 6.2ms
- If actual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

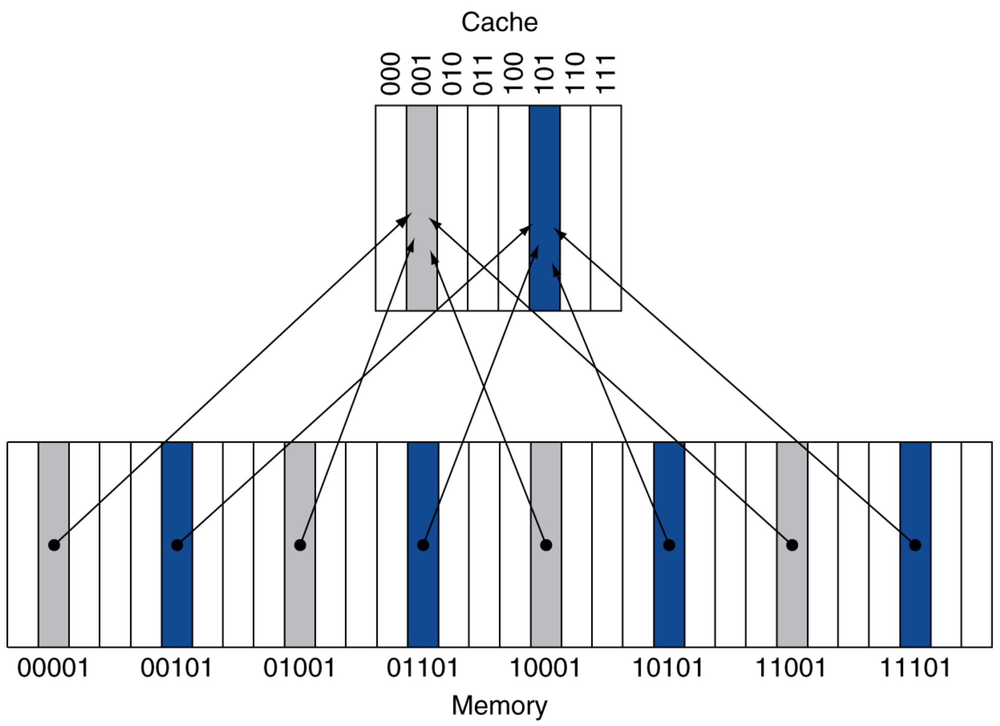
- Cache memory
  - The level of the memory hierarchy closest to the CPU
- Given accesses X_1, …, X_{n-1}, X_n



- 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

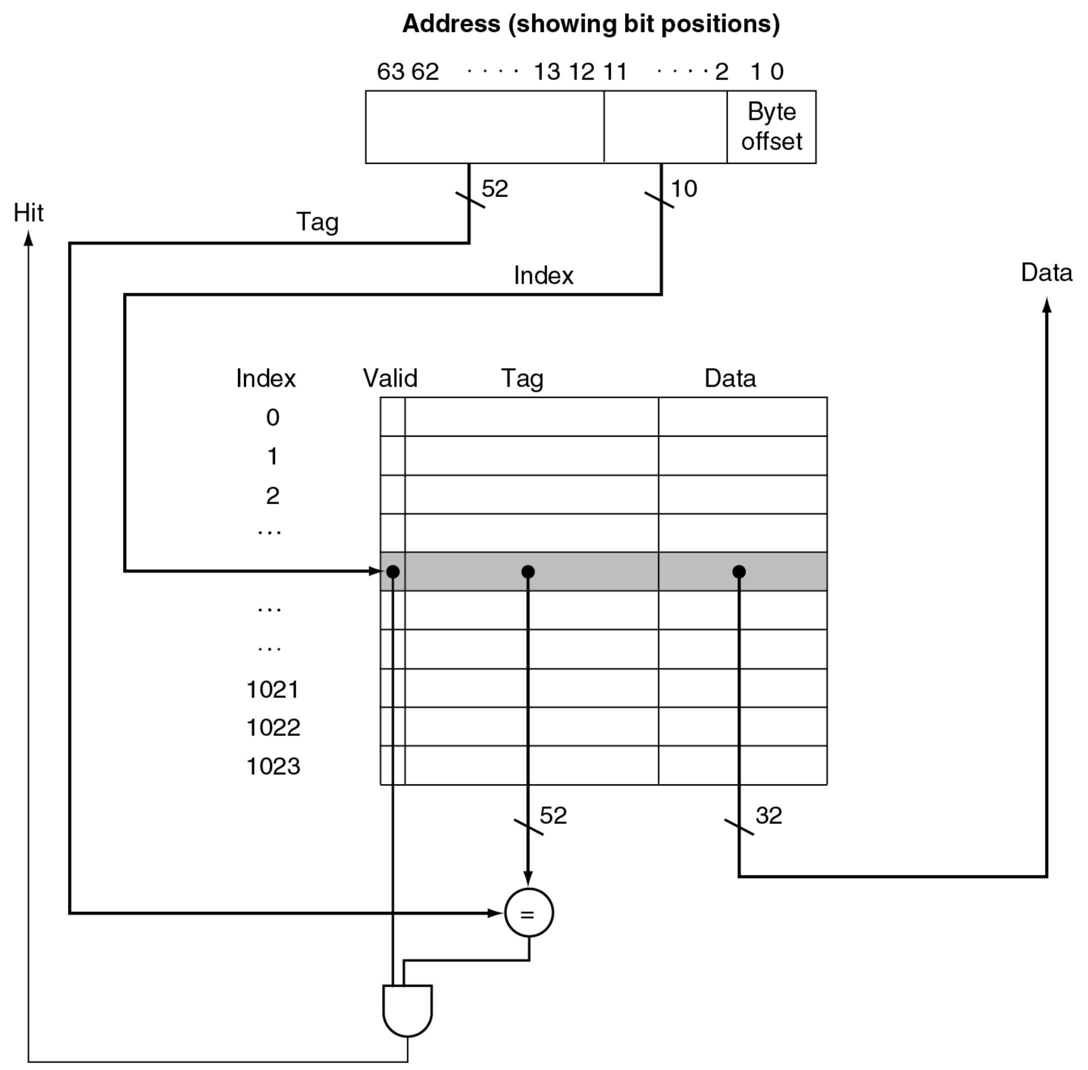
Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

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

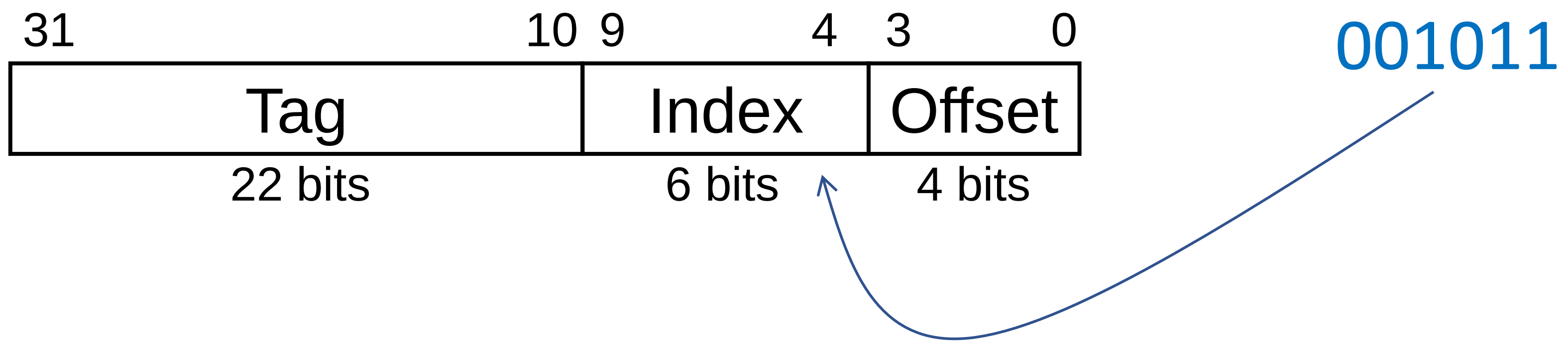
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Address Subdivision



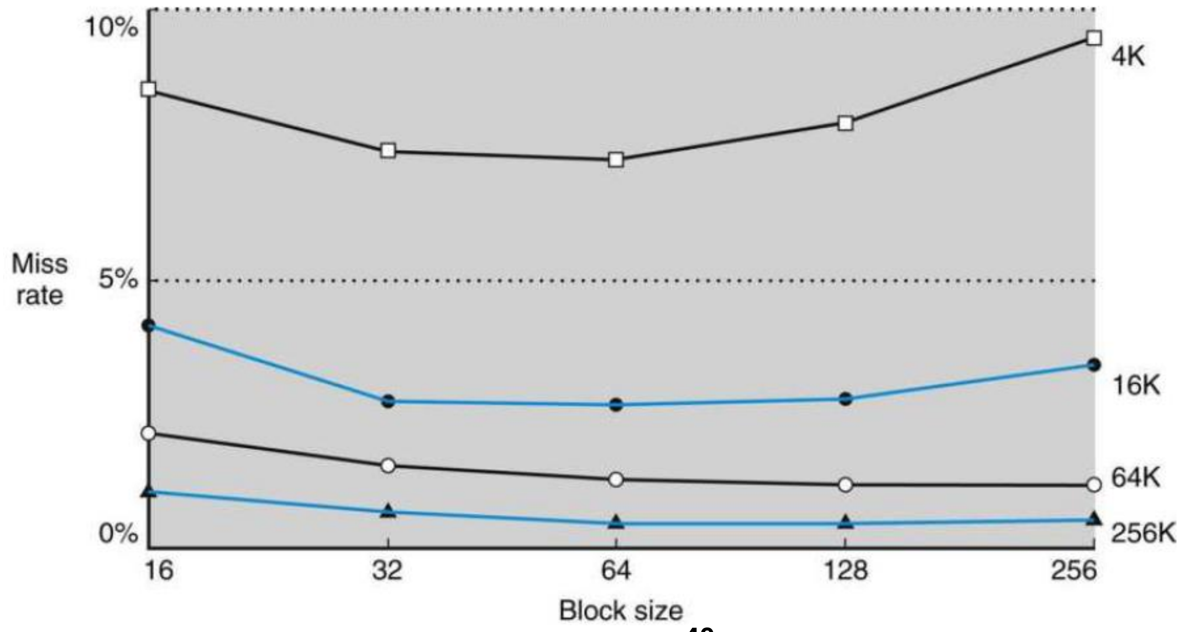
Example: Larger Block Size

- 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 $\Rightarrow$ fewer of them
- More competition between larger blocks $\Rightarrow$ 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



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*)
  - Data at desired index in cache (for read or write) is ‘*dirty*’
- 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, it must write it to store as well at some point in time
- The timing of this write is controlled by Write Policy
  - Two approaches: Write through and 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 = $1 + 0.1 \times 100 = 11$
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

Write-Back

- Alternative: On data-write hit, just update the block in cache – with it’s status ‘dirty’ (using ‘dirty bit’) *when 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*
- When a dirty block is replaced, requires 2 accesses to Memory:
    1. Write it back to memory
    2. Retrieve the needed data from memory

  Can use a write buffer 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 loaded to cache, followed by a write-hit operation. *In this approach, write misses are similar to read misses*

1.1

- 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

1.1

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

1.2

- 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

1.2

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

1.3

- 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)

1.3

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

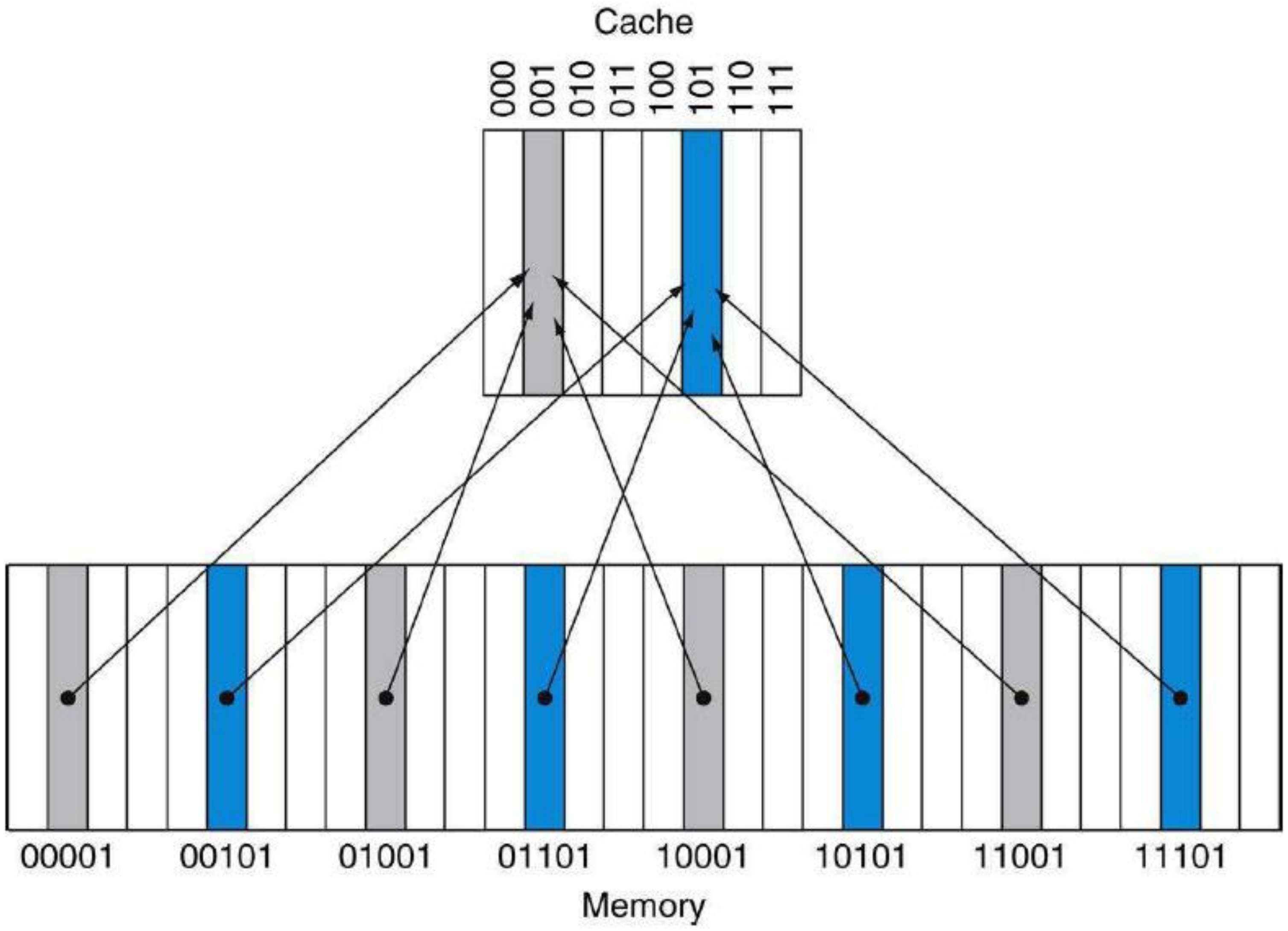
1.3

- 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 1.2, 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 1.2 are imposed by the size of the Cache itself. Fig 5.11 in text demonstrates this observation as well:

2

- 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.

2


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

2

- 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

3

- **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, 53 bits for Tag and 1 valid bit
- Total bits required = 8192 + 54x32 + 1 x 32 = *9952* bits
- **9952 / 8192 = 1.21**

3

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

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%

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 block 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 updating the block in cache following Write misses is that the data is available in the cache for a subsequent Read access*.
- The disadvantage of bringing updating the block in the 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* <u>not bringing</u> 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

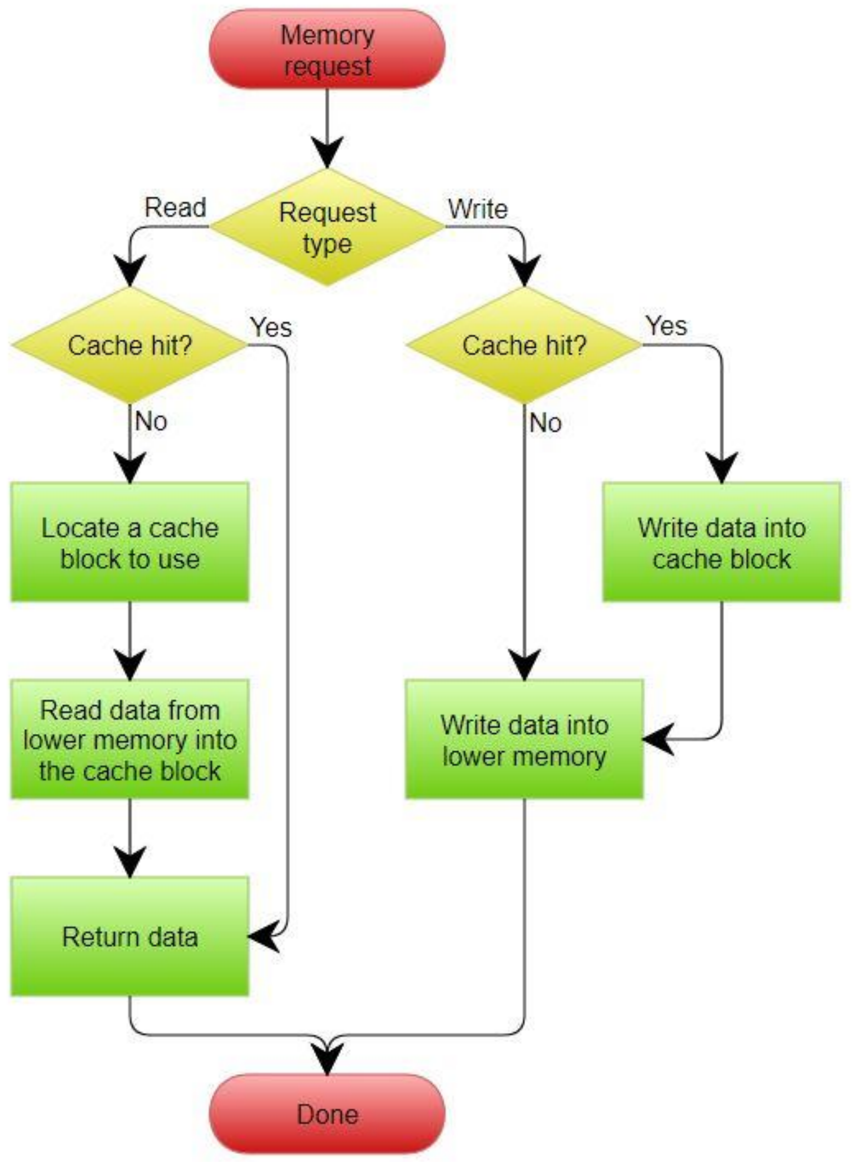
Write Back with Write Allocate:

- on **hits** it writes to cache setting dirty bit for the block, main memory is not updated;
- on **misses** it *updates the block in main memory* **and** *brings the 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:

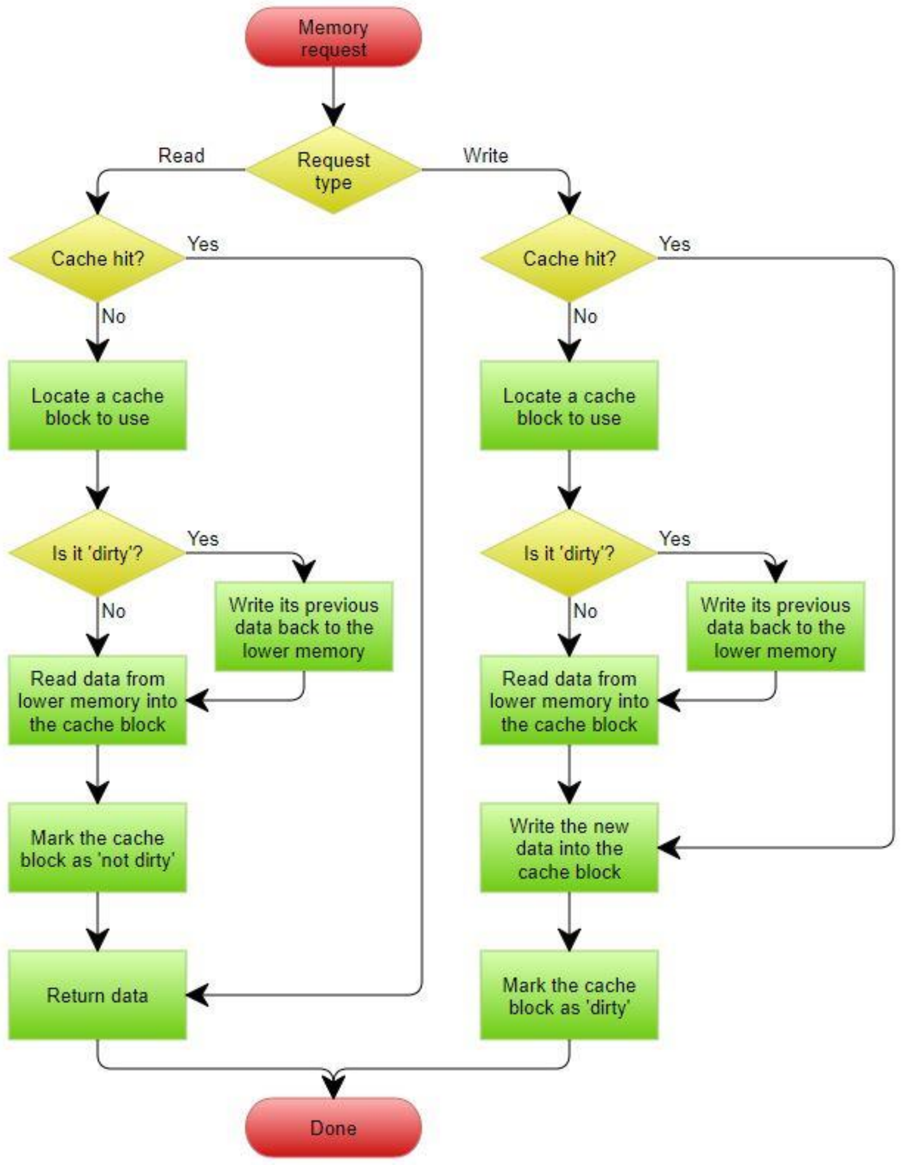
- on **hits** it writes to cache setting dirty bit for the block, main memory is not updated;
- on **misses** it *updates the block in main memory* **not** *bringing that block to the cache*;
- Subsequent writes to the same block, if the block originally caused a miss, *will generate misses* all the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

No Write Allocate



Memory request
Request type
Read
Write
Cache hit?
Yes
No
Cache hit?
Yes
No
Locate a cache block to use
Write data into cache block
Read data from lower memory into the cache block
Write data into lower memory
Return data
Done

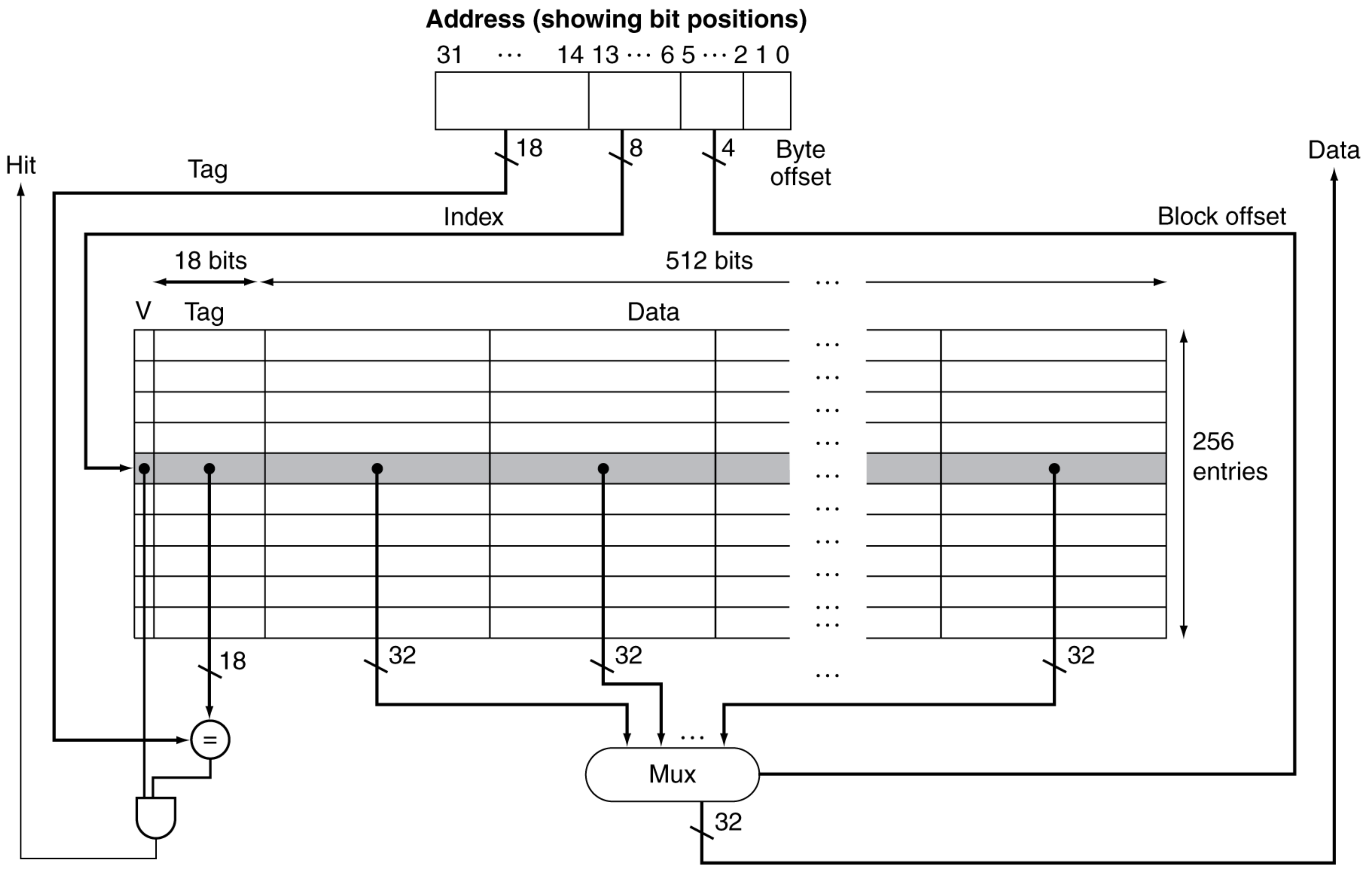
Write Allocate


Memory request
Request type
Read
Write
Cache hit?
Yes
No
Locate a cache block to use
Is it 'dirty'?
Yes
No
Write its previous data back to the lower memory
Read data from lower memory into the cache block
Mark the cache block as 'not dirty'
Return data
Cache hit?
Yes
No
Locate a cache block to use
Is it 'dirty'?
Yes
No
Write its previous data back to the lower memory
Read data from lower memory into the cache block
Write the new data into the cache block
Mark the cache block as 'dirty'
Done

Example: 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%

Example: Intrinsity FastMATH



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

Measuring Cache Performance

- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- With simplifying assumptions:

$$\text{Memory stall cycles}$$

$$= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Cache Performance Example

- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- Miss cycles per instruction = MR x MP
  - I-cache: $0.02 \times 100 = 2$
  - D-cache: $0.36 \times 0.04 \times 100 = 1.44$
- Actual CPI = 2 + 2 + 1.44 = 5.44
  - Ideal CPU is 5.44/2 =2.72 times faster

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

Performance Summary

1. When CPU performance increased
   - Miss penalty becomes more significant
2. Decreasing base CPI
   - Greater proportion of time spent on memory stalls
3. Increasing clock rate
   - Memory stalls account for more CPU cycles

- Can’t neglect cache behavior when evaluating system performance