ECE 250 / CPS 250
Computer Architecture

Caches and Memory Hierarchies

Benjamin Lee

Slides based on those from

Andrew Hilton (Duke), Alvy Lebeck (Duke)

Benjamin Lee (Duke), and Amir Roth (Penn)

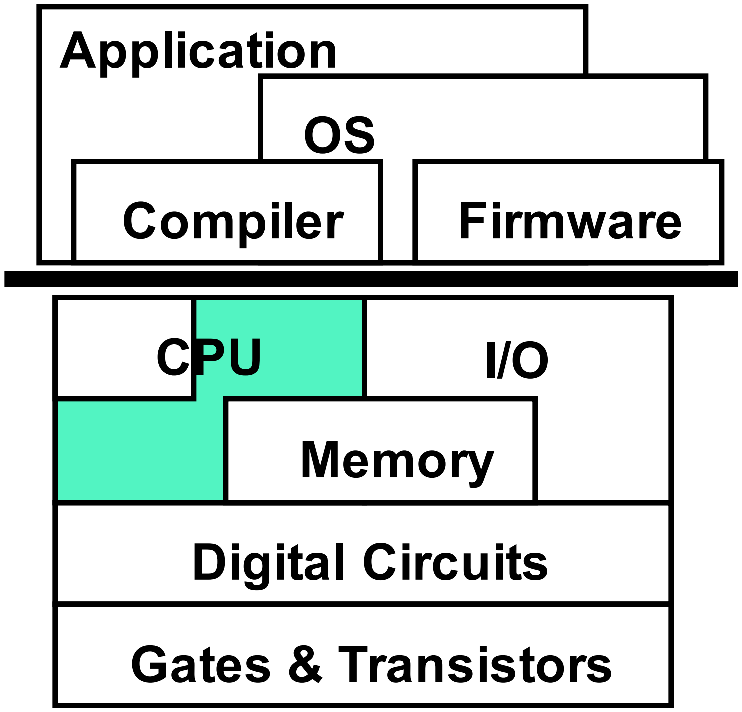
Where We Are in This Course Right Now

- So far:
  - We know how to design a processor that can fetch, decode, and execute the instructions in an ISA
  - We have assumed that memory storage (for instructions and data) is a magic black box
- Now:
  - We learn why memory storage systems are hierarchical
  - We learn about caches and SRAM technology for caches
- Next:
  - We learn how to implement main memory

Readings

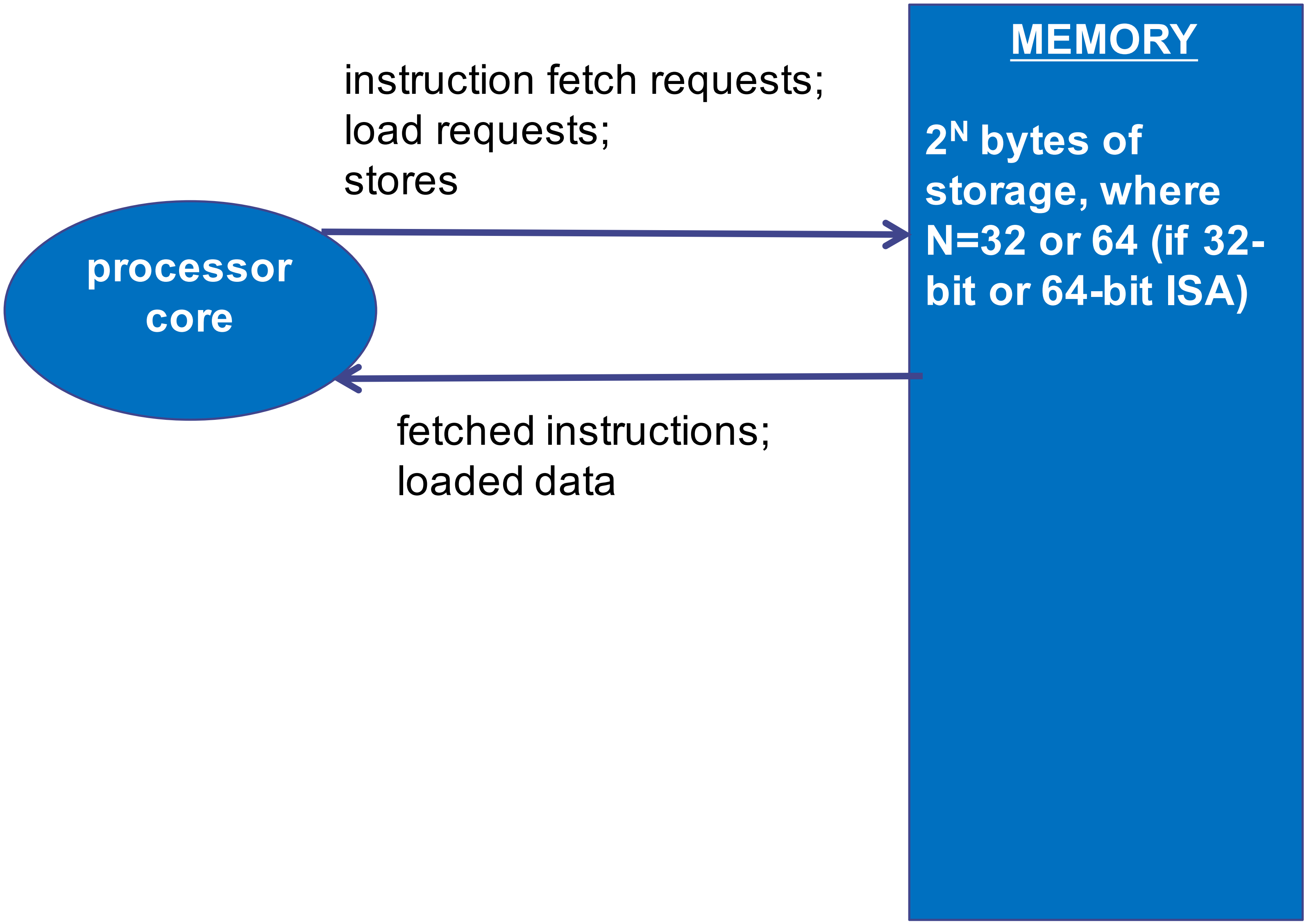
- Patterson and Hennessy
  - Chapter 5

This Unit: Caches and Memory Hierarchies

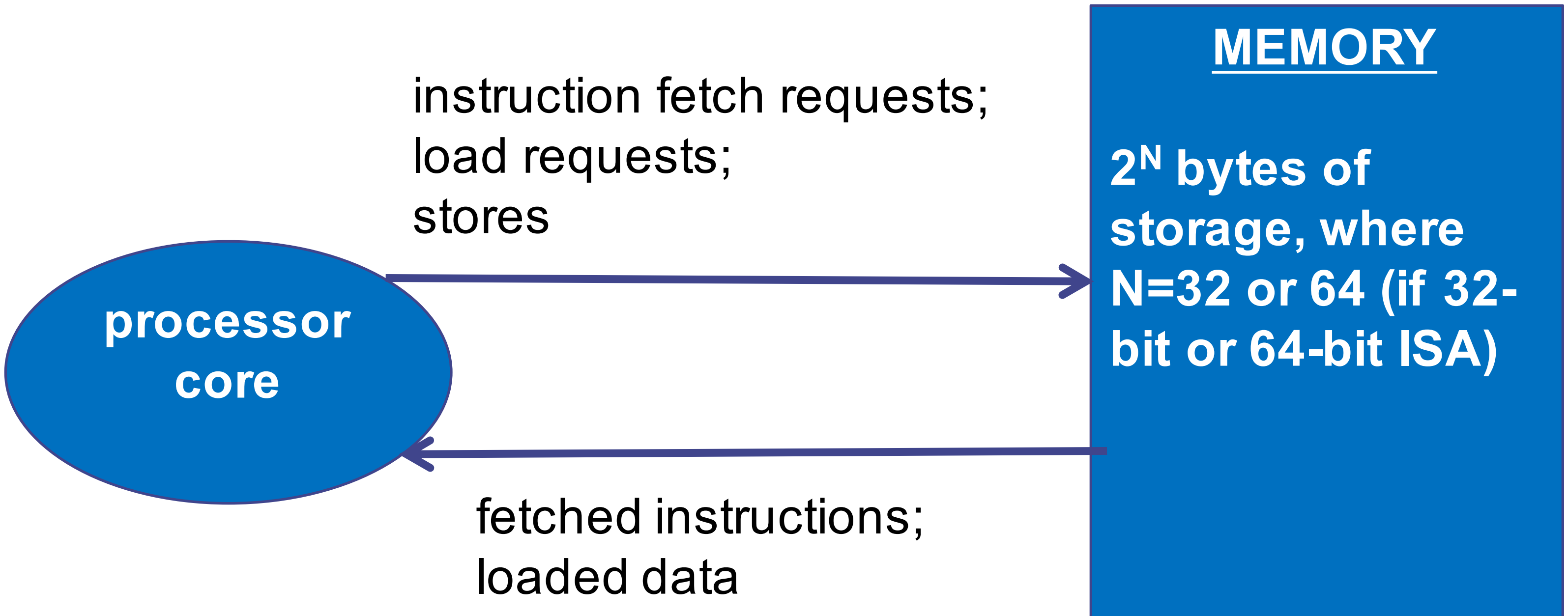


- Memory hierarchy
  - Basic concepts
- Cache organization
- Cache implementation

So far, our view of memory is simple



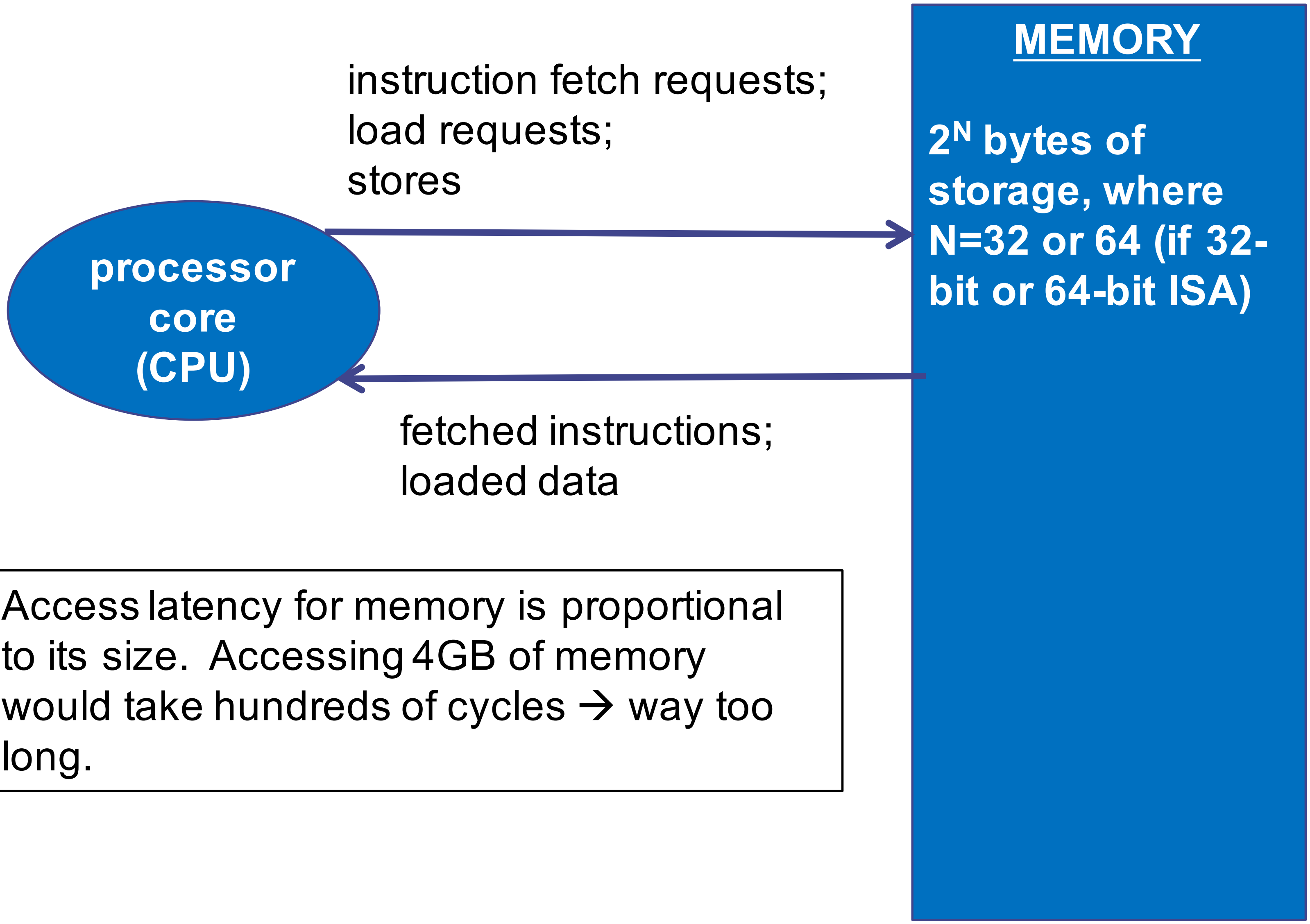
Why Isn't This Sufficient?



2^{32} bytes per running program = ridiculous
2^{64} bytes per running program = ridiculous++
→will defer this problem for a week or so

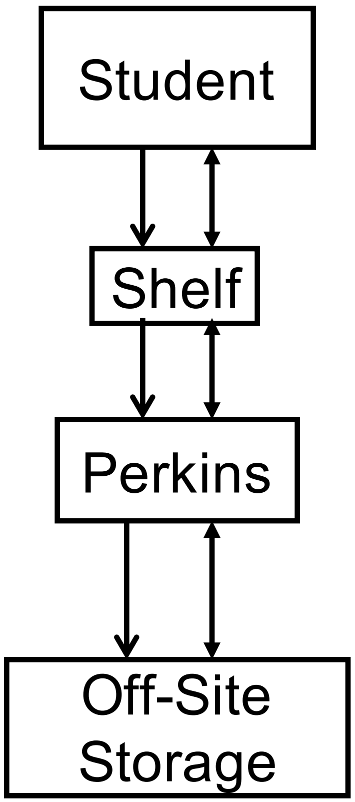
Let's assume we can buy all that memory.
So now what's the problem?

Why Isn't This Sufficient?



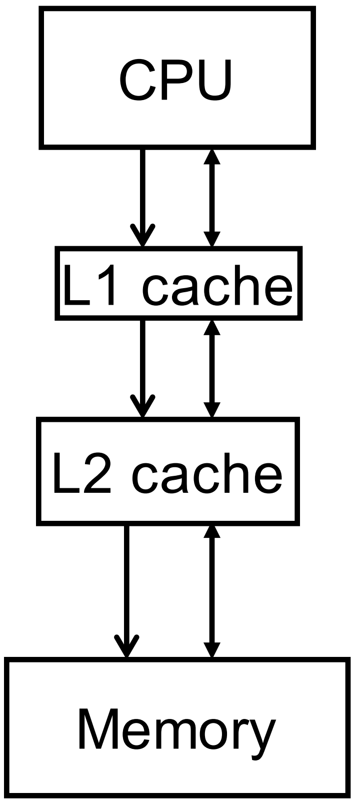
Access latency for memory is proportional to its size. Accessing 4GB of memory would take hundreds of cycles → way too long.

An Analogy: Duke’s Library System



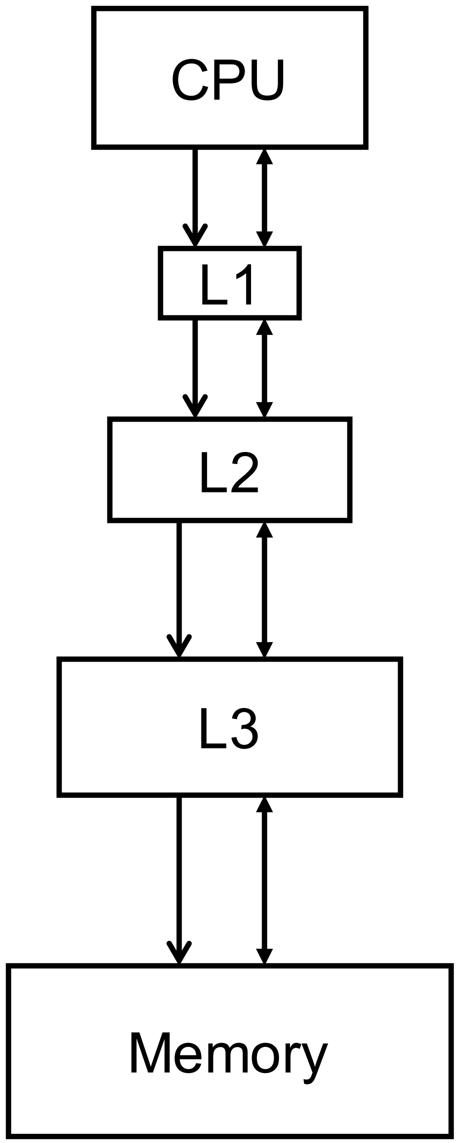
- Keep some books on shelf at home
  - Books are actively read, used
  - Small subset of all books owned by Duke
  - Fast access time
- If book not on shelf, retrieve from Perkins
  - Much larger subset of all books owned by Duke
  - Slower access time
- If book not at Perkins, retrieve from off-site
  - Guaranteed to get any book
  - Much slower access time

An Analogy: Duke's Library System



- CPU keeps small subset of memory in its level-1 (L1) cache
  - Data is actively read, used
  - Small subset of all data in memory
  - Fast access time
- If data not in cache, retrieve from level-2 (L2) cache
  - Much larger subset of all data in memory
  - Slower access time
- If data not in L2, retrieve from main memory
  - Guaranteed to get any data
  - Much slower access time

Big Concept: Memory Hierarchy



- Use hierarchy of memory components
  - Upper components (closer to CPU)
    - Fast ↔ Small ↔ Expensive
  - Lower components (further from CPU)
    - Slow ↔ Big ↔ Cheap
  - Bottom component (for now!) = what we have been calling "memory" until now
- Make average access time close to L1's
  - How?
  - Most frequently accessed data in L1
  - L1 + next most frequently accessed in L2, etc.
  - **Automatically** move data up, down hierarchy

Some Terminology

- If we access a level of memory and find what we want → called a hit
- If we access a level of memory and do NOT find what we want → called a miss

Some Goals

- Key 1: High "hit rate" → high probability of finding what we want at a given level

- Key 2: Low access latency

- Misses are expensive (take a long time)
  - Try to avoid them
  - But, if they happen, amortize their costs → bring in more than just the specific word you want → bring in a whole block of data (multiple words)

Blocks

- Block = a group of spatially contiguous and aligned bytes
  - Typical sizes are 32B, 64B, 128B
- Spatially contiguous and aligned
  - Example: 32B blocks
  - Blocks = [address 0- address 31], [32-63], [64-96], etc.
  - NOT:
    - [13-44] = unaligned
    - [0-22, 26-34] = not contiguous
    - [0-20] = wrong size (not 32B)

Why Hierarchy Works For Duke Books

- **Temporal locality**
  - Recently accessed book likely to be accessed again soon
- **Spatial locality**
  - Books near recently accessed book likely to be accessed soon (assuming spatially nearby books are on same topic)

Why Hierarchy Works for Memory

- **Temporal locality**
  - Recently executed instructions likely to be executed again soon
    - Loops
  - Recently referenced data likely to be referenced again soon
    - Data in loops, hot global data

- **Spatial locality**
  - Insns near recently executed insns likely to be executed soon
    - Sequential execution
  - Data near recently referenced data likely to be referenced soon
    - Elements in array, fields in struct, variables in stack frame

- **Locality** is one of the most important concepts in computer architecture → don't forget it!

Hierarchy Leverages Non-Uniform Patterns

- **10/90 rule (of thumb)**
  - For Instruction Memory:
    - 10% of static insns account for 90% of executed insns
    - Inner loops
  - For Data Memory:
    - 10% of variables account for 90% of accesses
    - Frequently used globals, inner loop stack variables

- What if processor accessed every block with equal likelihood? Small caches wouldn't help much.

Memory Hierarchy: All About Performance

$$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$$

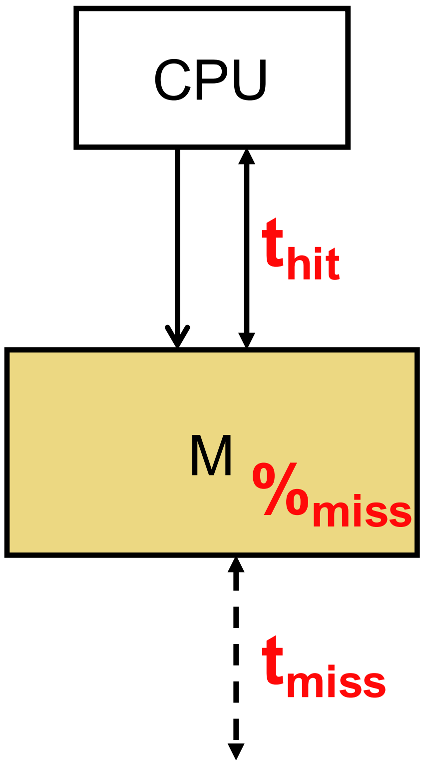
- t_{avg} = average time to satisfy request at given level of hierarchy
- t_{hit} = time to hit (or discover miss) at given level
- t_{miss} = time to satisfy miss at given level
- Problem: hard to get low t_{hit} and $\%_{miss}$ in one structure
  - Large structures have low $\%_{miss}$ but high t_{hit}
  - Small structures have low t_{hit} but high $\%_{miss}$

- Solution: use a **hierarchy** of memory structures

  "Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available … We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible."

Burks, Goldstine, and Von Neumann, **1946**

Memory Performance Equation



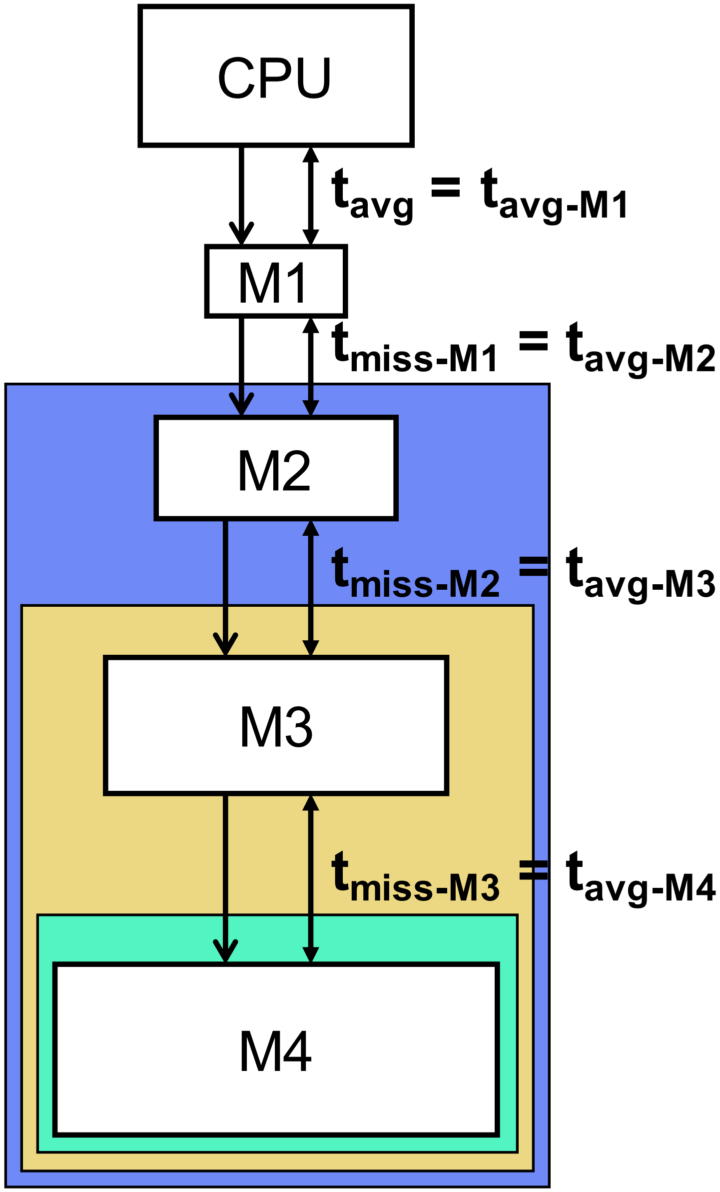
- For memory component M
  - **Access**: read or write to M
  - **Hit**: desired data found in M
  - **Miss**: desired data not found in M
    - Must get from another (slower) component
  - **Fill**: action of placing data in M

  - $\%_{miss}$ (miss-rate): #misses / #accesses
  - t_{hit}: time to read data from (write data to) M
  - t_{miss}: time to read data into M from lower level

- Performance metric
  - t_{avg}: average access time

$$t_{avg} = t_{hit} + (\%_{miss} * t_{miss})$$

Abstract Hierarchy Performance



How do we compute t_{avg} ?

$= t_{avg\text{-}M1}$

$= t_{hit\text{-}M1} + (\%_{miss\text{-}M1} * t_{miss\text{-}M1})$

$= t_{hit\text{-}M1} + (\%_{miss\text{-}M1} * t_{avg\text{-}M2})$

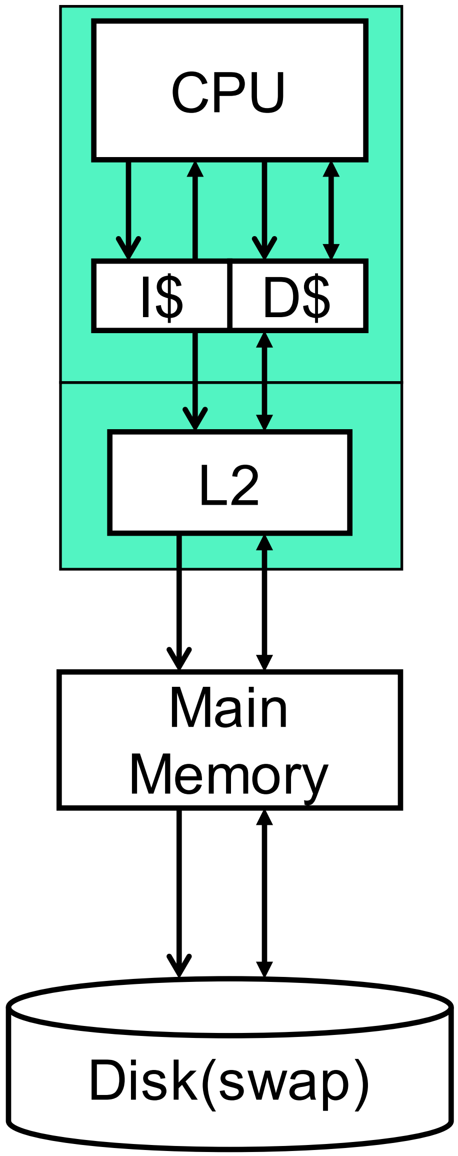
$= t_{hit\text{-}M1} + (\%_{miss\text{-}M1} * (t_{hit\text{-}M2} + (\%_{miss\text{-}M2} * t_{miss\text{-}M2})))$

$= t_{hit\text{-}M1} + (\%_{miss\text{-}M1} * (t_{hit\text{-}M2} + (\%_{miss\text{-}M2} * t_{avg\text{-}M3})))$

$= \ldots$

Note: Miss at level X = access at level X+1

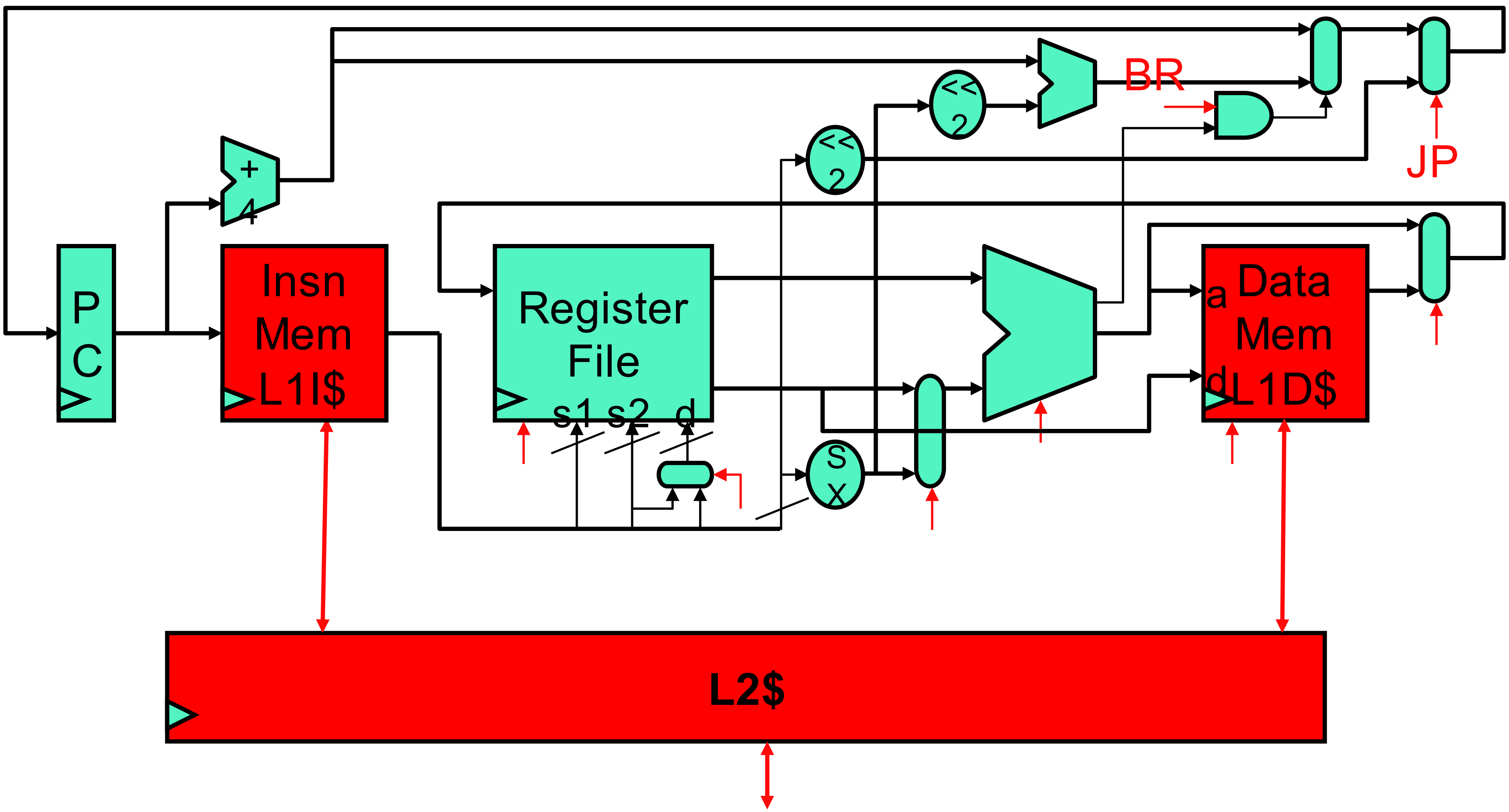
Typical Memory Hierarchy



- 1st level: **L1 I$**, **L1 D$** (L1 insn/data caches)
- 2nd level: **L2 cache (L2$)**
  - Also on same chip with CPU
  - Made of SRAM (same circuit type as CPU)
  - Managed in hardware
  - This unit of ECE/CS 250
- 3rd level: **main memory**
  - Made of DRAM
  - Managed in software
  - Next unit of ECE/CS 250
- 4th level: **disk (swap space)**
  - Made of magnetic iron oxide discs
  - Managed in software
  - Course unit after main memory
- Could be other levels (e.g., Flash, PCM, tape, etc.)

Note: many processors have L3$ between L2$ and memory

Concrete Memory Hierarchy

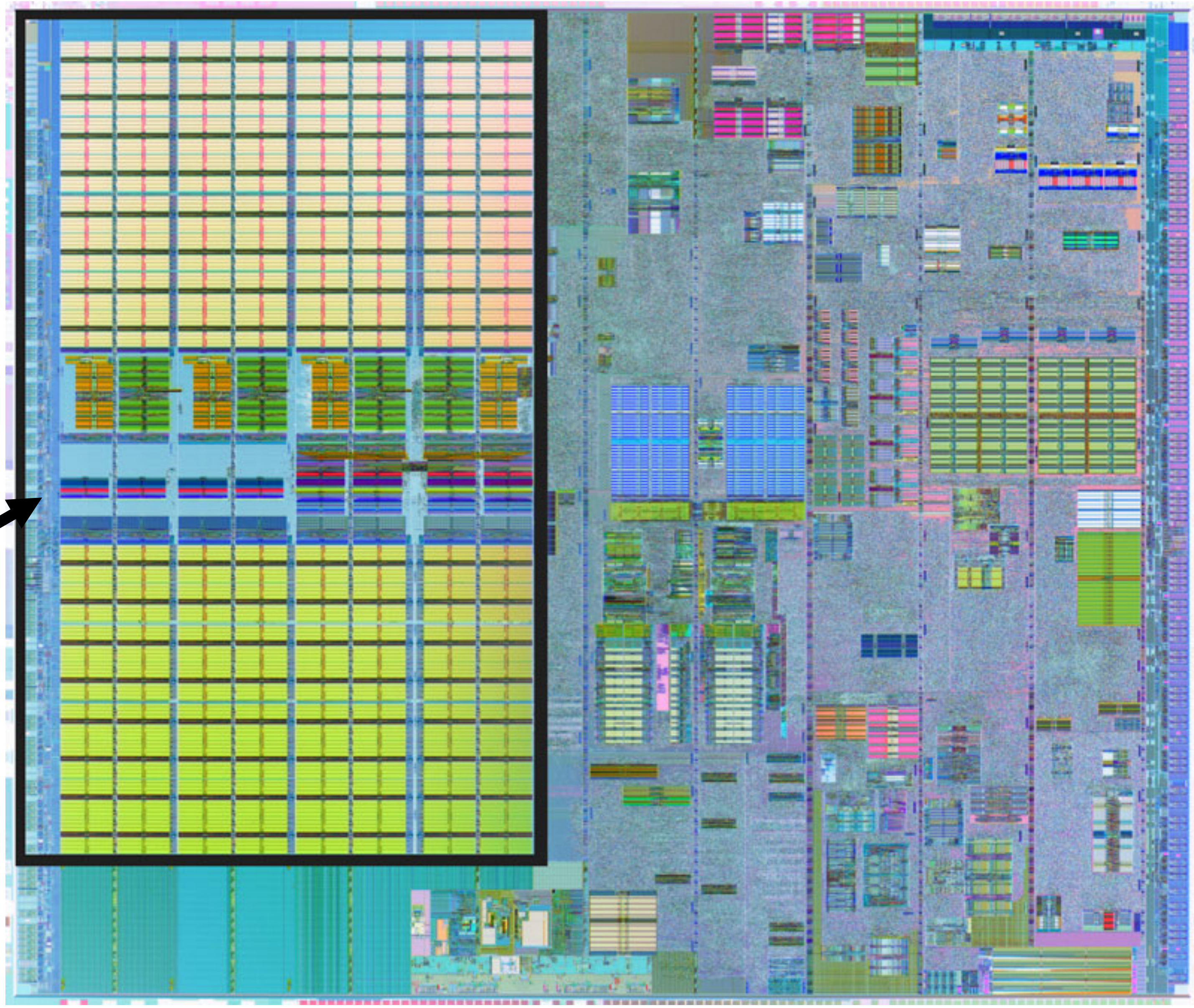


- Much of today's chips used for caches → important!

A Typical Die Photo

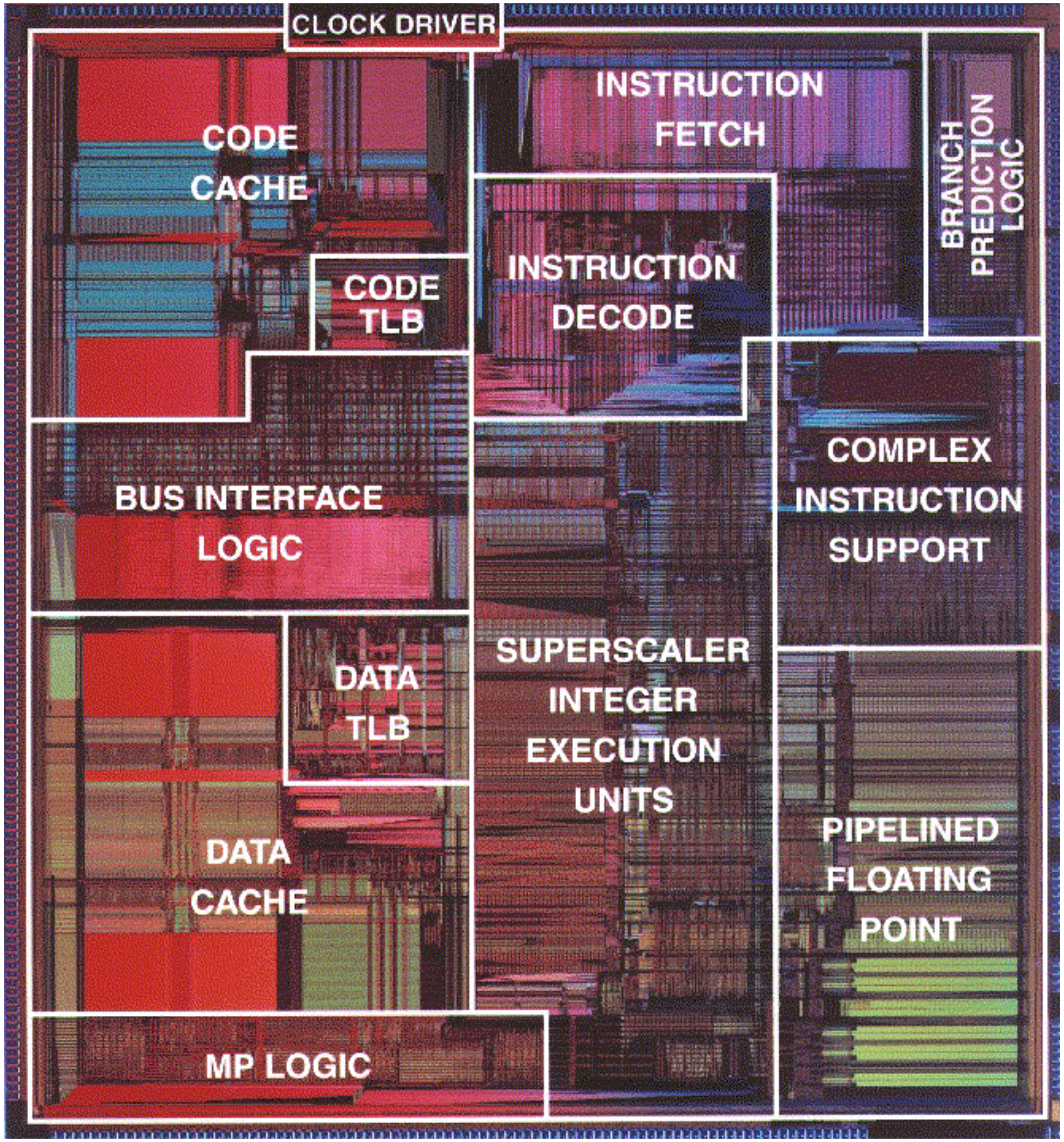
Pentium4 Prescott chip with 2MB L2$

L2 Cache



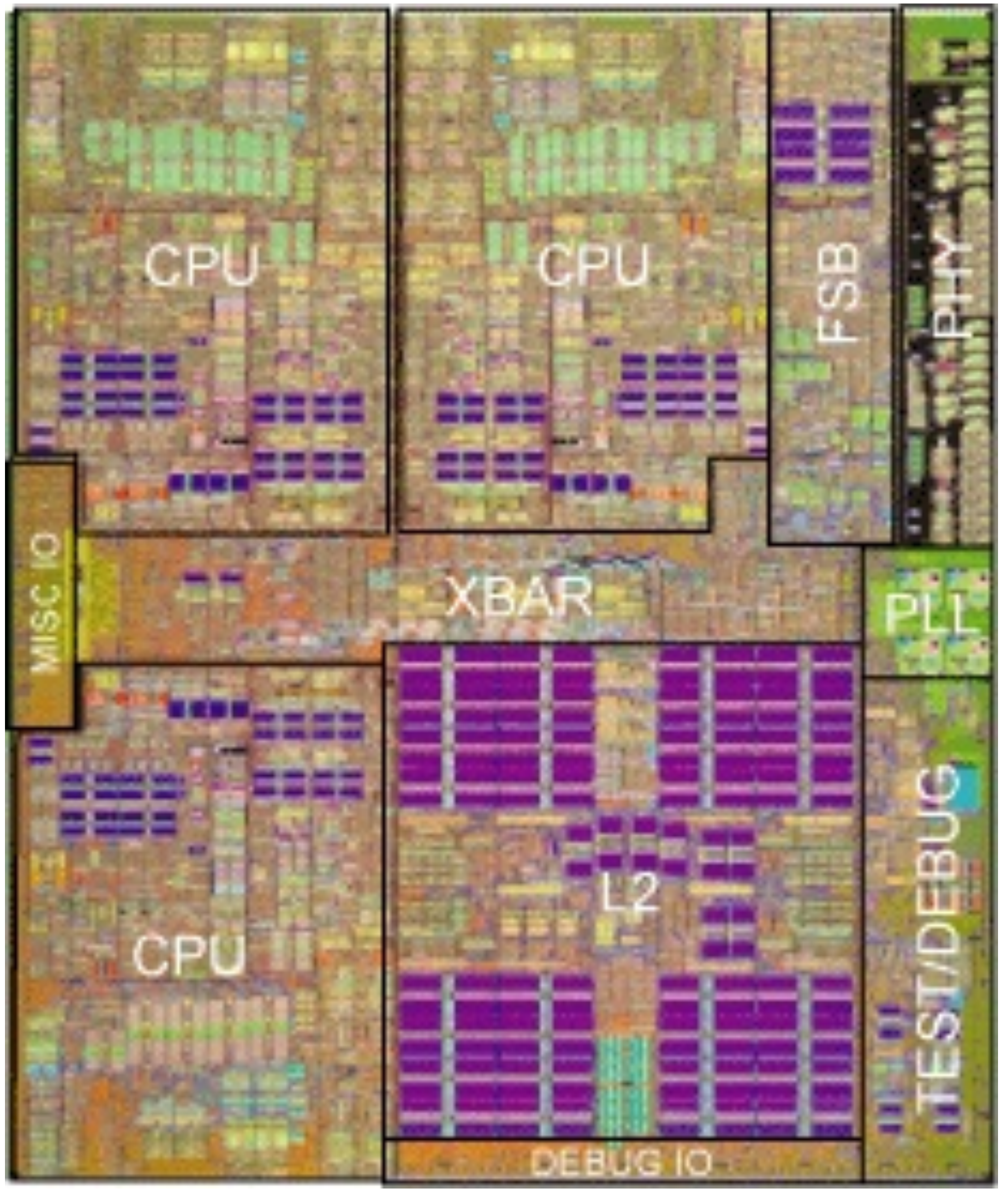
A Closer Look at that Die Photo

Pentium4 Prescott chip with 2MB L2$

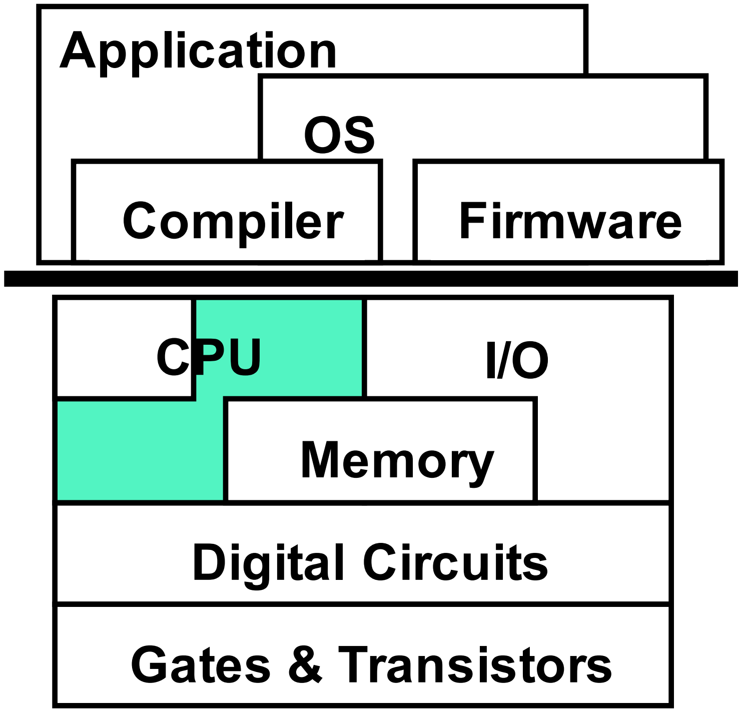


A Multicore Die Photo from IBM

IBM's Xenon chip with 3 PowerPC cores



This Unit: Caches and Memory Hierarchies



- Memory hierarchy
- Cache organization
- Cache implementation

Cache Structure

- Cache consists of frames, and each frame is the storage to hold one block of data
  - Also holds a "valid" bit and a "tag" to label the block in that frame
- Valid: if 1, frame holds valid data; if 0, data is invalid
  - Useful? Yes. Example: when you turn on computer, cache is full of invalid "data"
- Tag: specifies which block is living in this frame
  - Useful? Yes. Far fewer frames than blocks of memory!

valid	tag	block data
1	[64-95]	32 bytes of valid data
0	[0-31]	32 bytes of junk
1	[0-31]	32 bytes of valid data
1	[1024-1055]	32 bytes of valid data

Cache Example (very simplified for now)

valid	tag	block data
0	[0-31]	32 bytes of junk
0	[0-31]	32 bytes of junk
0	[0-31]	32 bytes of junk
0	[0-31]	32 bytes of junk

- When computer turned on, no valid data in cache (everything is zero, including valid bits)

Cache Example (very simplified for now)

valid	tag	block data
1	[32-63]	32 bytes of valid data
0	[0-31]	32 bytes of junk
0	[0-31]	32 bytes of junk
0	[0-31]	32 bytes of junk

- Assume CPU asks for word at byte addresses [32-35]
  - Either due to a load or an instruction fetch
- Word [32-35] is part of block [32-63]
- Miss! (no blocks in cache yet)
- Fill cache from lower level with block [32-63]
  - don't forget to set valid bit and write tag

Cache Example (very simplified for now)

valid	tag	block data
1	[32-63]	32 bytes of valid data
1	[1024-1055]	32 bytes of valid data
0	[0-31]	32 bytes of junk
0	[0-31]	32 bytes of junk

- Assume CPU asks for word [1028-1031]
  - Either due to a load or an instruction fetch
- Word [1028-1031] is part of block [1024-1055]
- Miss!
- Fill cache from lower level with block [1024-1055]

Cache Example (very simplified for now)

valid	tag	block data
1	[32-63]	32 bytes of valid data
1	[1024-1055]	32 bytes of valid data
0	[0-31]	32 bytes of junk
0	[0-31]	32 bytes of junk

- Assume CPU asks (again!) for word [1028-1031]
  - Hit! Hooray for temporal locality
- Assume CPU asks for word [1032-1035]
  - Hit! Hooray for spatial locality
- Assume CPU asks for word [0-3]
  - Miss! Don't forget those valid bits.

Where to Put Blocks in Cache

- How to decide which frame holds which block?
  - And then how to find block we're looking for?
- Some more cache structure:
  - Divide cache into sets
    - A block can only go in its set → there is a 1-to-1 mapping from block address to set
  - Each set holds some number of frames = set associativity
    - E.g., 4 frames per set = 4-way set-associative
- At extremes
  - Whole cache has just one set = fully associative
    - Most flexible (longest access latency)
  - Each set has 1 frame = 1-way set-associative = "direct mapped"
    - Least flexible (shortest access latency)

Direct-Mapped (1-way) Cache

- Assume 8B blocks
- 8 sets, 1 way/set → 8 frames
- Each block can only be put into 1 set (1 option)
  - Block [0-7] → set 0
  - Block [8-15] → set 1
  - Block [16-23] → set 2

  …
  - Block [56-63] → set 7
  - Block [64-71] → set 0
  - Block [72-79] → set 1
- Block [X-(X+7)] → set (X/8)%8
  - 1st 8=8B block, 2nd 8 = 8 sets

	way 0		
	valid	tag	data
set 0			
set 1			
set 2			
set 3			
set 4			
set 5			
set 6			
set 7			

Direct-Mapped (1-way) Cache

- Assume 8B blocks
- Consider the following stream of 1-byte requests from the CPU:
  - 2, 11, 5, 50, 67, 51, 3
- Which hit? Which miss?

	way 0		
	valid	tag	data
set 0			
set 1			
set 2			
set 3			
set 4			
set 5			
set 6			
set 7			

Problem with Direct Mapped Caches

- Assume 8B blocks
- Consider the following stream of 1-byte requests from the CPU:
  - 2, 67, 2, 67, 2, 67, 2, 67, …
- Which hit? Which miss?
- Did we make good use of all of our cache capacity?

	way 0		
	valid	tag	data
set 0			
set 1			
set 2			
set 3			
set 4			
set 5			
set 6			
set 7			

2-Way Set-Associativity

	way 0			way 1		
	valid	tag	data	valid	tag	data
set 0						
set 1						
set 2						
set 3						

- 4 sets, 2 ways/set → 8 frames (just like our 1-way cache)
  - Block [0-7] → set 0
  - Block [8-15] → set 1
  - Block [16-23] → set 2
  - Block [24-31] → set 3
  - Block [32-39] → set 0
  - Etc.

2-Way Set-Associativity

	way 0			way 1		
	valid	tag	data	valid	tag	data
set 0						
set 1						
set 2						
set 3						

- Assume the same pathological stream of CPU requests:
  - Byte addresses 2, 67, 2, 67, 2, 67, etc.
  - Which hit? Which miss?
- Now how about this: 2, 67, 131, 2, 67, 131, etc.
- How much more associativity can we have?

Full Associativity

	way 0			way 1			way 2			way 3			way 4			way 5			way 6			way 7		
	v	t	d	v	t	d	v	t	d	v	t	d	v	t	d	v	t	d	v	t	d	v	t	d
set 0																								

- 1 set, 8 ways/set → 8 frames (just like previous examples)
  - Block [0-7] → set 0
  - Block [8-15] → set 0
  - Block [16-23] → set 0
  - Etc.

Mapping Addresses to Sets

- MIPS has 32-bit addresses
  - Let's break down address into three components
- If blocks are 8B, then $\log_2 8=3$ bits required to identify a byte within a block. These bits are called block offset.
  - Given block, offset tells you which byte within block
- If there are S sets, then $\log_2 S$ bits required to indentify the set. These bits are called set index or just index.
- Rest of the bits ($32-3- \log_2 S$) specify the tag

tag	index	block offset

Mapping Addresses to Sets

- How many blocks map to the same set?
- Let's assume 8-byte blocks
  - $8=2^3$ → 3 bits to specify block offset
- Let's assume we have direct-mapped cache with 256 sets
  - 256 sets $=2^8$ sets → 8 bits to specify set index
- 2^{32} bytes of memory/(8 bytes/block) = 2^{29} blocks
- 2^{29} blocks / 256 sets = 2^{21} blocks / set
- So that means we need 2^{21} tags to distinguish between all possible blocks in the set → 21 tag bits
  - Note: 21=32-3-8 ☺

tag (21)	index (8)	block offset (3)

Mapping Addresses to Sets

tag (21)	index (8)	block offset (3)

- Assume cache from previous slide (8B blocks, 256 sets)
- Example: What do we do with the address 10?

  0000 0000 0000 0000 0000 0000 0000 1010
  - offset = 2 (2nd byte in block)
  - index=1 (set 1)
  - tag = 0
- This matches what we did before – recall:
  - Block [0-7] → set 0
  - Block [8-15] → set 1
  - Block [16-23] → set 2
  - etc.

Cache Replacement Policies

- Set-associative caches present a new design choice
  - On cache miss, which block in set to replace (kick out)?

- Some options
  - **Random**
  - **LRU (least recently used)**
    - Fits with temporal locality, LRU = least likely to be used in future
  - **NMRU (not most recently used)**
    - An easier-to-implement approximation of LRU
    - NMRU=LRU for 2-way set-associative caches
  - **FIFO (first-in first-out)**
    - When is this a good idea?

ABCs of Cache Design

- Architects control three primary aspects of cache design
  - And can choose for each cache independently
- A = Associativity
- B = Block size
- C = Capacity of cache

- Secondary aspects of cache design
  - Replacement algorithm
  - Some other more subtle issues we'll discuss later

Analyzing Cache Misses: 3C Model

- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
    - Easy to identify
  - **Capacity**: miss caused because cache is too small – would've been miss even if cache had been fully associative
    - Consecutive accesses to block separated by accesses to at least N other distinct blocks where N is number of frames in cache
  - **Conflict**: miss caused because cache associativity is too low – would've been hit if cache had been fully associative
    - All other misses

3C Example

- Assume 8B blocks
- Consider the following stream of 1-byte requests from the CPU:
  - 2, 11, 5, 50, 67, 128, 256, 512, 1024, 2
  - Is the last access a capacity miss or a conflict miss?

	way 0		
	valid	tag	data
set 0			
set 1			
set 2			
set 3			
set 4			
set 5			
set 6			
set 7			

ABCs of Cache Design and 3C Model

- **Associativity** (increase, all else equal)
  + \+ Decreases conflict misses
  + – Increases t_{hit}
- **Block size** (increase, all else equal)
  + – Increases conflict misses
  + \+ Decreases compulsory misses
  + ± Increases or decreases capacity misses
  + • Negligible effect on t_{hit}
- **Capacity** (increase, all else equal)
  + \+ Decreases capacity misses
  + – Increases t_{hit}

Inclusion/Exclusion

- If L2 holds superset of every block in L1, then L2 is inclusive with respect to L1
- If L2 holds no block that is in L1, then L2 and L1 are exclusive
- L2 could be neither inclusive nor exclusive
  - Has some blocks in L1 but not all
- This issue matters a lot for multicores, but not a major issue in this class

- Same issue for L3/L2

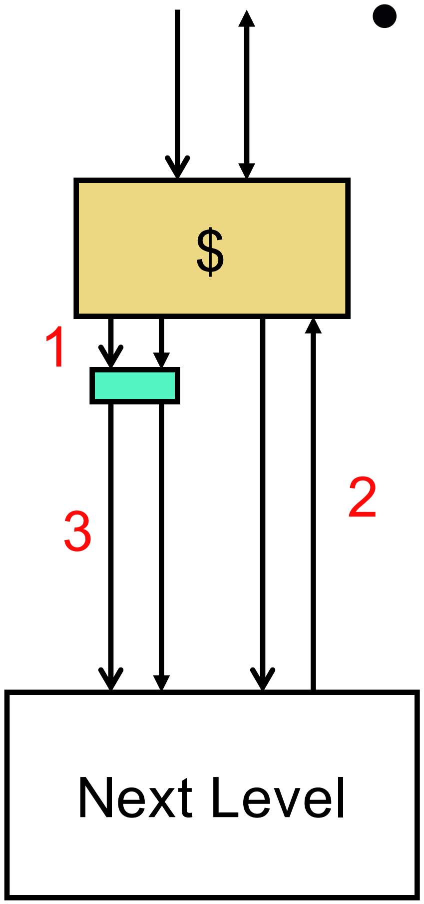
Stores: Write-Through vs. Write-Back

- When to propagate new value to (lower level) memory?
  - **Write-through**: immediately (as soon as store writes to this level)
    - \+ Conceptually simpler
    - \+ Uniform latency on misses
    - – Requires additional bandwidth to next level
  - **Write-back**: later, when block is replaced from this level
    - Requires additional "dirty" bit per block → why?
    - \+ Minimal bandwidth to next level
      - Only write back dirty blocks
    - – Non-uniform miss latency
      - Miss that evicts clean block: just a fill from lower level
      - Miss that evicts dirty block: writeback dirty block and then fill from lower level

Stores: Write-allocate vs. Write-non-allocate

- What to do on a write miss?
  - **Write-allocate**: read block from lower level, write value into it
    - + Decreases read misses
    - – Requires additional bandwidth
    - • Use with write-back
  - **Write-non-allocate**: just write to next level
    - – Potentially more read misses
    - + Uses less bandwidth
    - • Use with write-through

Optimization: Write Buffer



- **Write buffer**: between cache and memory
  - Write-through cache? Helps with store misses
    - \+ Write to buffer to avoid waiting for next level
      - Store misses become store hits
  - Write-back cache? Helps with dirty misses
    - \+ Allows you to do read (important part) first
      1. Write dirty block to buffer
      2. Read new block from next level to cache
      3. Write buffer contents to next level

Typical Processor Cache Hierarchy

- First level caches: optimized for t_{hit} and parallel access
  - Insns and data in separate caches (**I$**, **D$**) → why?
  - Capacity: 8–64KB, block size: 16–64B, associativity: 1–4
  - Other: write-through or write-back
  - t_{hit}: 1–4 cycles
- Second level cache (**L2**): optimized for $\%_{miss}$
  - Insns and data in one cache for better utilization
  - Capacity: 128KB–1MB, block size: 64–256B, associativity: 4–16
  - Other: write-back
  - t_{hit}: 10–20 cycles
- Third level caches (**L3**): also optimized for $\%_{miss}$
  - Capacity: 2–16MB
  - t_{hit}: ~30 cycles

Performance Calculation Example

- Parameters
  - Reference stream: 20% stores, 80% loads
  - L1 D$: t_{hit} = 1ns, $\%_{miss}$ = 5%, write-through + write-buffer
  - L2: t_{hit} = 10ns, $\%_{miss}$ = 20%, write-back, 50% dirty blocks
  - Main memory: t_{hit} = 50ns, $\%_{miss}$ = 0%
- What is $t_{avgL1D\$}$ without an L2?
  - Write-through + write-buffer means all stores effectively hit
  - $t_{missL1D\$} = t_{hitM}$
  - $t_{avgL1D\$} = t_{hitL1D\$} + \%_{loads} * \%_{missL1D\$} * t_{hitM}$ = 1ns+(0.8*0.05*50ns) = 3ns

Performance Calculation Example

- Parameters
  - Reference stream: 20% stores, 80% loads
  - L1 D$: t_{hit} = 1ns, $\%_{miss}$ = 5%, write-through + write-buffer
  - L2: t_{hit} = 10ns, $\%_{miss}$ = 20%, write-back, 50% dirty blocks
  - Main memory: t_{hit} = 50ns, $\%_{miss}$ = 0%
- What is $t_{avgD\$}$ with an L2?
  - $t_{missL1D\$} = t_{avgL2}$
  - Write-back (no buffer) means dirty misses cost double
  - $t_{avgL2} = t_{hitL2}+(1+\%_{dirty})*\%_{missL2}*t_{hitM}$ = 10ns+(1.5*0.2*50ns) =25ns
  - $t_{avgL1D\$} = t_{hitL1D\$} + \%_{loads}*\%_{missL1D\$}*t_{avgL2}$ = 1ns+(0.8*0.05*25ns) =2ns

Cost of Tags

- "4KB cache" means cache holds 4KB of data
  - Called **capacity**
  - Tag storage is considered overhead (not included in capacity)

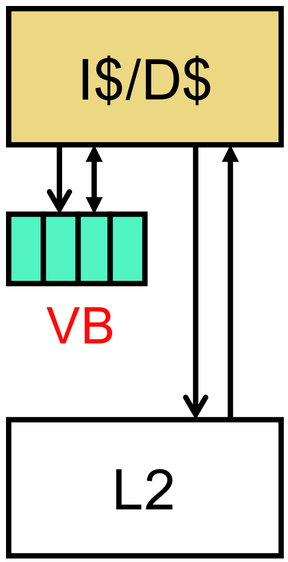
- Calculate tag overhead of 4KB cache with 1024 4B frames
  - Not including valid bits
  - 4B frames → 2-bit offset
  - 1024 frames → 10-bit index
  - 32-bit address – 2-bit offset – 10-bit index = 20-bit tag
  - 20-bit tag * 1024 frames = 20Kb tags = 2.5KB tags
  - 63% overhead → much higher than usual because blocks are so small (and cache is small)

Two (of many possible) Optimizations

- **Victim buffer**: for conflict misses
- **Prefetching**: for capacity/compulsory misses

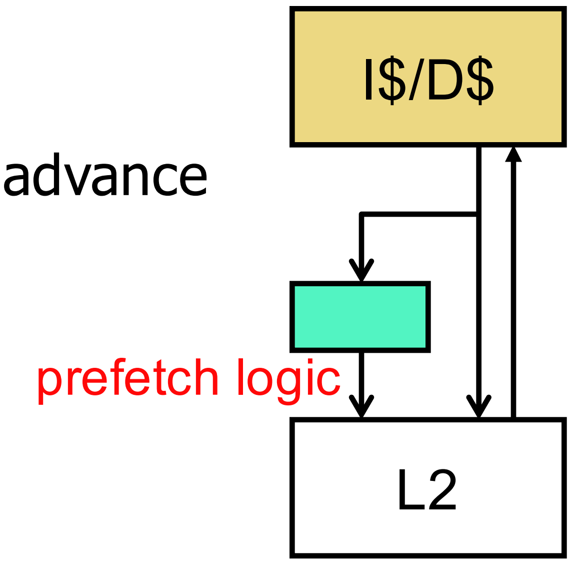
Victim Buffer

- Conflict misses: not enough associativity
  - High-associativity is expensive, but also rarely needed
    - E.g., 3 blocks map to same 2-way set, accessed (ABC)*
- **Victim buffer (VB)**: small FA cache (e.g., 4 entries)
  - Sits on I$/D$ fill path
  - VB is small → very fast
  - Blocks kicked out of I$/D$ placed in VB
  - On miss, check VB: hit ? Place block back in I$/D$
  - 4 extra ways, shared among all sets
    - + Only a few sets will need it at any given time
  - + Very effective in practice

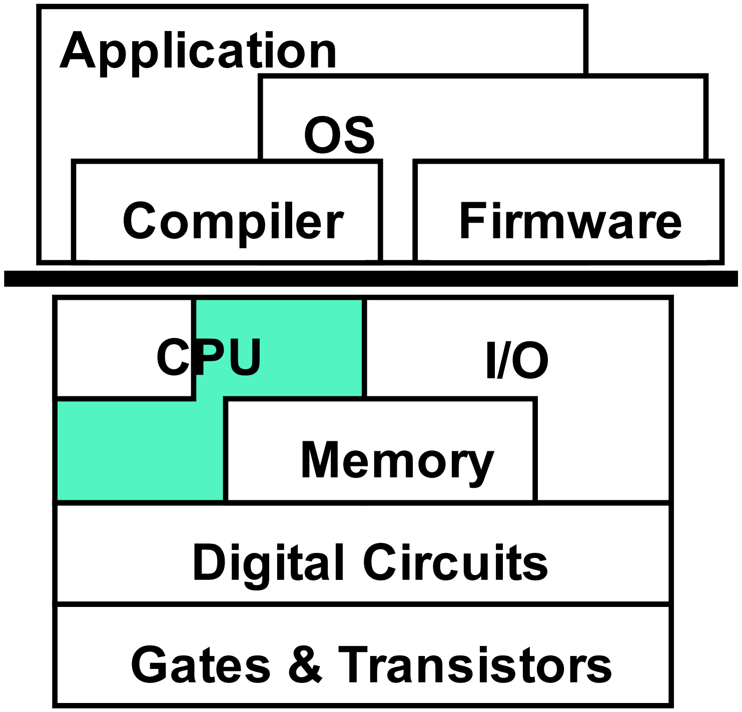


Prefetching

- **Prefetching**: put blocks in cache proactively/speculatively
  - Key: anticipate upcoming miss addresses accurately
    - Can do in software or hardware
  - Simple example: **next block prefetching**
    - Miss on address **X** → anticipate miss on **X+block-size**
    - Works for insns: sequential execution
    - Works for data: arrays
  - **Timeliness**: initiate prefetches sufficiently in advance
  - **Accuracy**: don't evict useful data



This Unit: Caches and Memory Hierarchies



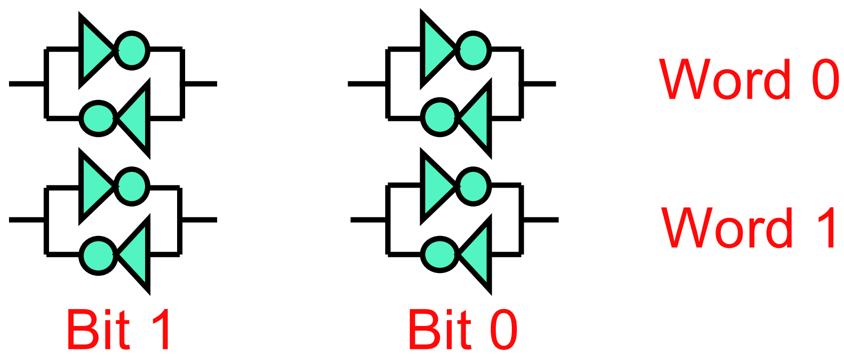
- Memory hierarchy
- Cache organization
- Cache implementation

How to Build Large Storage Components?

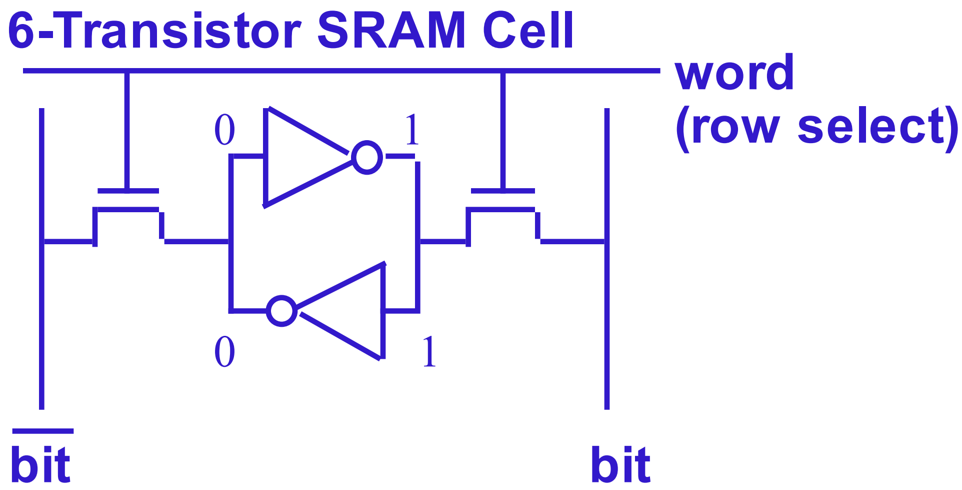
- Functionally, we could implement large storage as a vast number of D flip-flops
- But for big storage, our goal is density (bits/area)
  - And FFs are big: ~32 transistors per bit
- It turns out we can get much better density
  - And this is what we do for caches (and for register files)

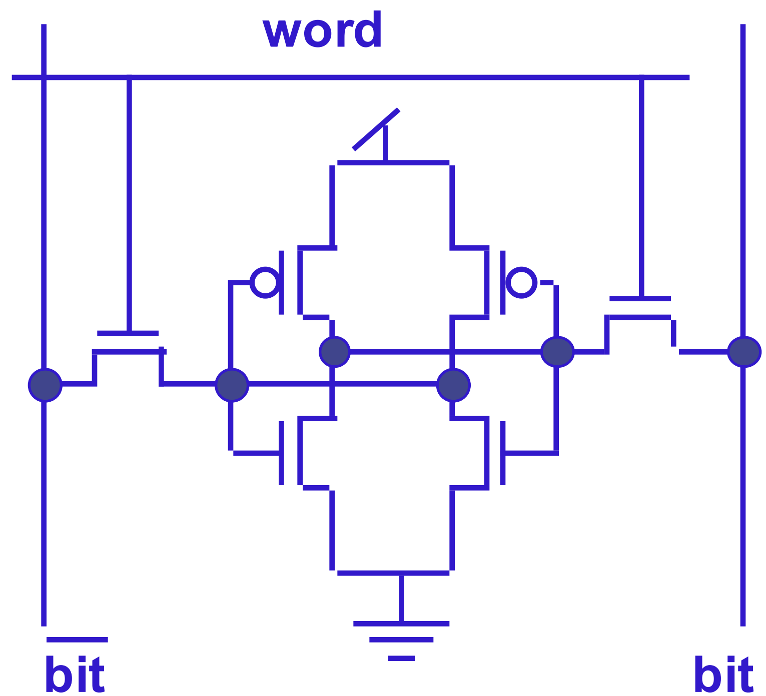
Static Random Access Memory (SRAM)

- Reality: large storage arrays implemented in “analog” way
  - Bits as cross-coupled inverters, not flip-flops
    - Inverters: 2 gates = 4 transistors per bit
    - Flip-flops: 8 gates =~32 transistors per bit
  - Ports implemented as shared buses called bitlines (next slide)
  - Called **SRAM (static random access memory)**
    - “Static” → a written bit maintains its value (doesn’t leak out)
    - But still volatile → bit loses value if chip loses power
- Example: storage array with two 2-bit words



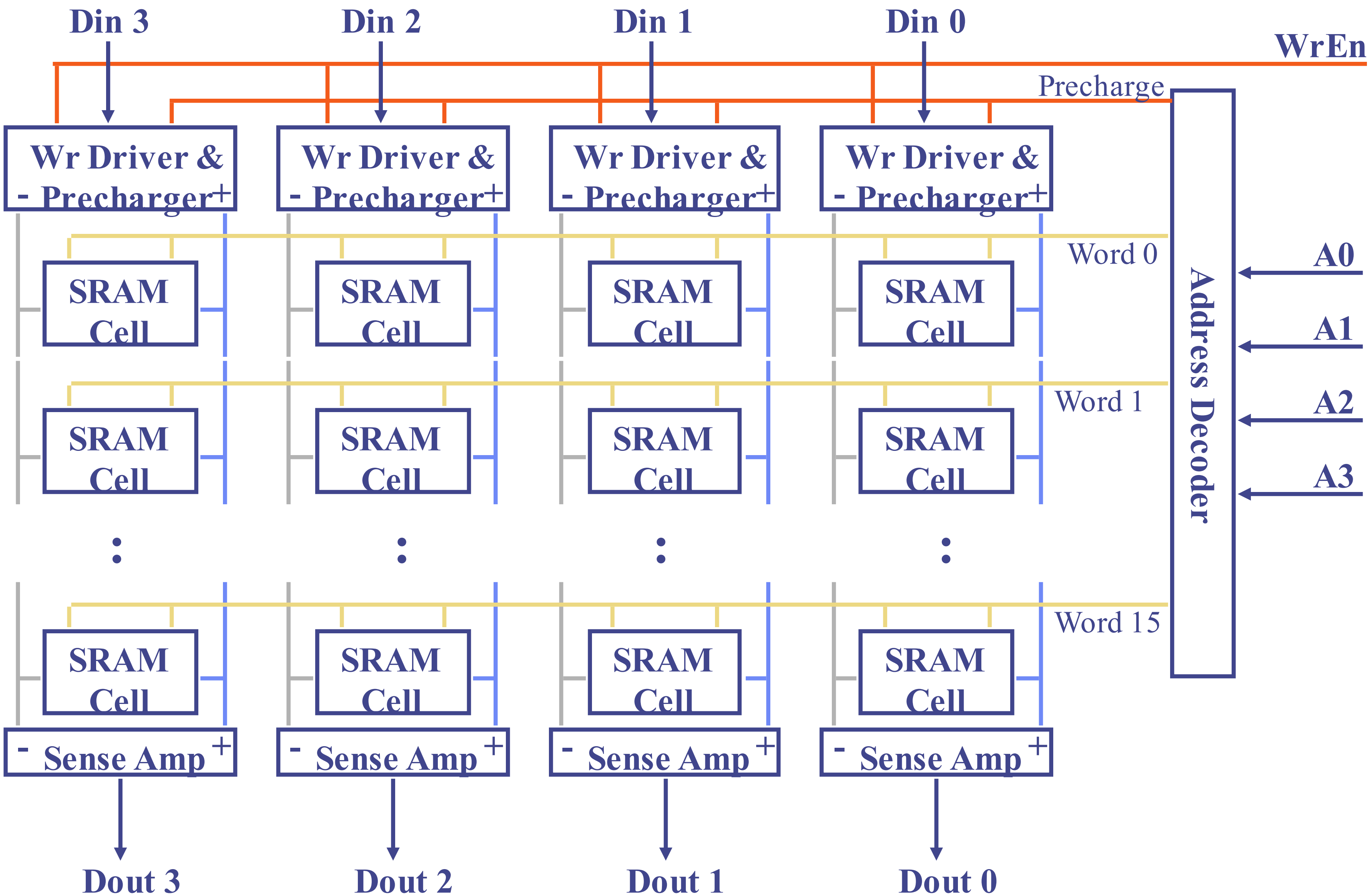
One Static RAM Cell



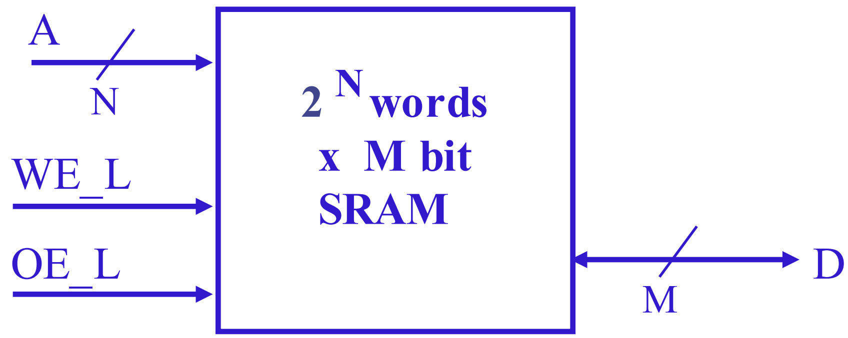


- To write (a value of 1):
  1. Drive bit lines (bit=1, $\overline{\text{bit}}$=0)
  2. Select row
- To read:
  1. Pre-charge bit and $\overline{\text{bit}}$ to Vdd (set to 1)
  2. Select row
  3. Cell pulls one line lower (pulls towards 0)
  4. Sense amp on column detects difference between bit and $\overline{\text{bit}}$

Typical SRAM Organization: 16-word x 4-bit



Logic Diagram of a Typical SRAM



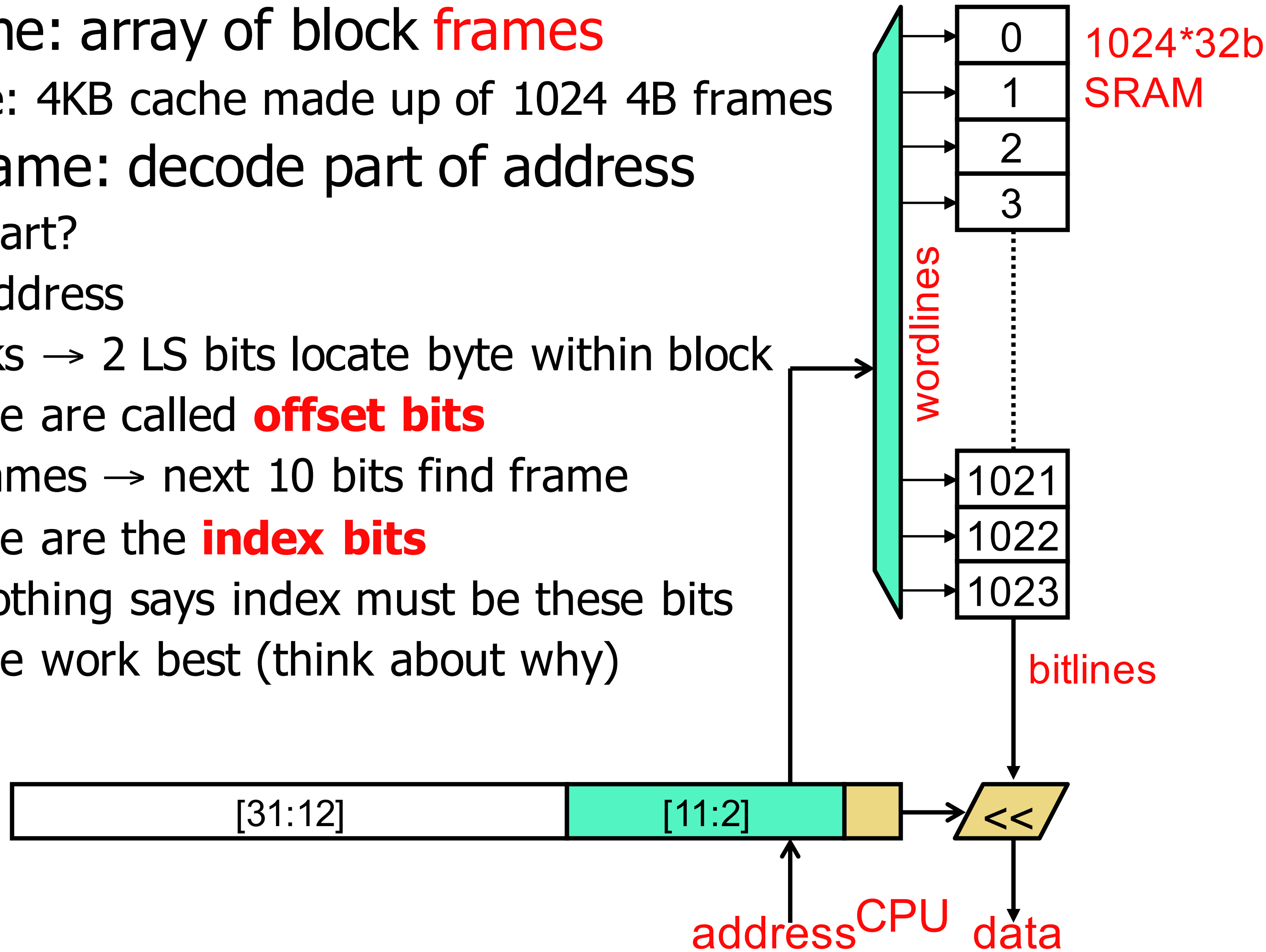
- Write Enable is usually active low (WE_L)
- Din and Dout are combined (D) to save pins:
  - A new control signal, output enable (OE_L) is needed
  - When D serves as the data input pin
    - WE_L is asserted (Low), OE_L is de-asserted (High)
  - When D serves as the data output pin
    - WE_L is de-asserted (High), OE_L is asserted (Low)
  - Both WE_L and OE_L are asserted:
    - Result is unknown. Don't do that!!!

SRAM Executive Summary

- Large storage arrays cannot be implemented "digitally"
  - Muxing and wire routing become impractical
- SRAM implementation exploits analog transistor properties
  - Inverter pair bits much smaller than flip-flop bits
  - Wordline/bitline arrangement makes for simple "grid-like" routing
  - Basic understanding of reading and writing
    - Wordlines select words
    - Overwhelm inverter-pair to write
    - Drain pre-charged line or swing voltage to read
  - Access latency proportional to √#bits * #ports
- You must understand important properties of SRAM
  - Will help when we talk about DRAM (next unit)

Basic Cache Structure

- Basic cache: array of block frames
  - Example: 4KB cache made up of 1024 4B frames
- To find frame: decode part of address
  - Which part?
  - 32-bit address
  - 4B blocks → 2 LS bits locate byte within block
    - These are called **offset bits**
  - 1024 frames → next 10 bits find frame
    - These are the **index bits**
  - Note: nothing says index must be these bits
  - But these work best (think about why)

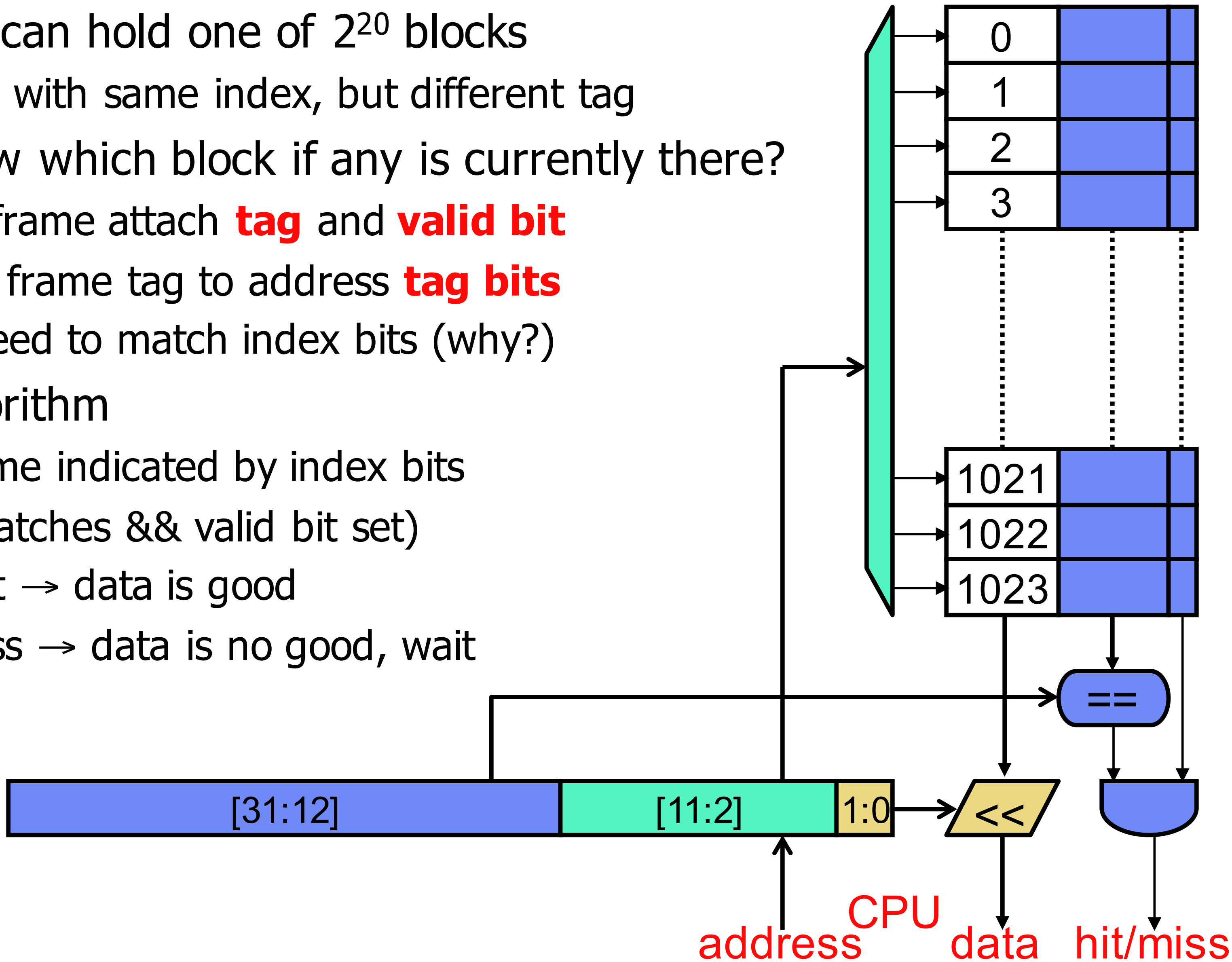


Basic Cache Structure

- Each frame can hold one of 2^{20} blocks
  - All blocks with same index, but different tag
- How to know which block if any is currently there?
  - To each frame attach **tag** and **valid bit**
  - Compare frame tag to address **tag bits**
    - No need to match index bits (why?)
- Lookup algorithm
  - Read frame indicated by index bits
  - If (tag matches && valid bit set)

    then Hit → data is good

    Else Miss → data is no good, wait

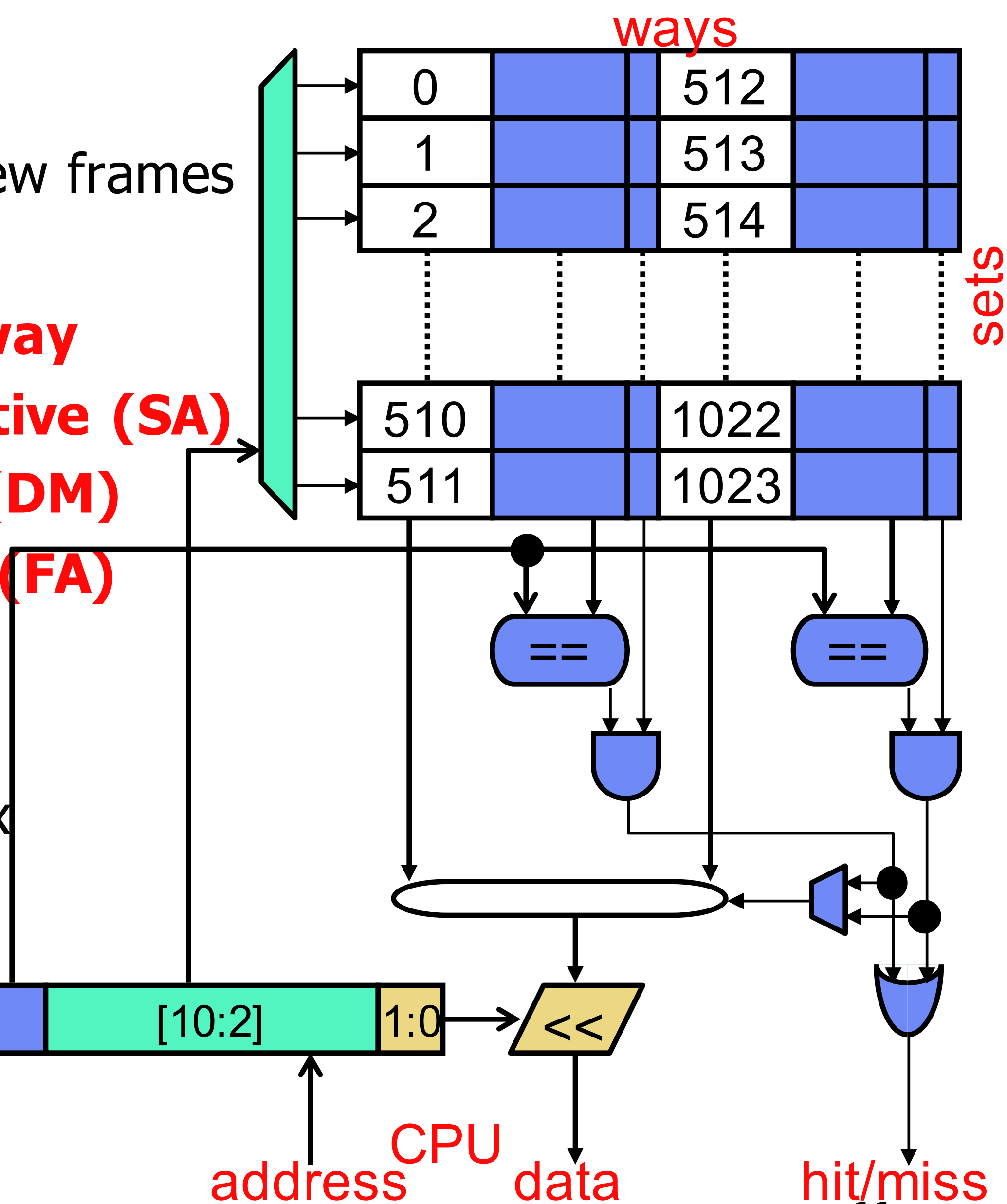


Set-Associativity

- **Set-associativity**
  - Block can reside in one of few frames
  - Frame groups called **sets**
  - Each frame in set called a **way**
  - This is **2-way set-associative (SA)**
  - 1-way → **direct-mapped (DM)**
  - 1-set → **fully-associative (FA)**

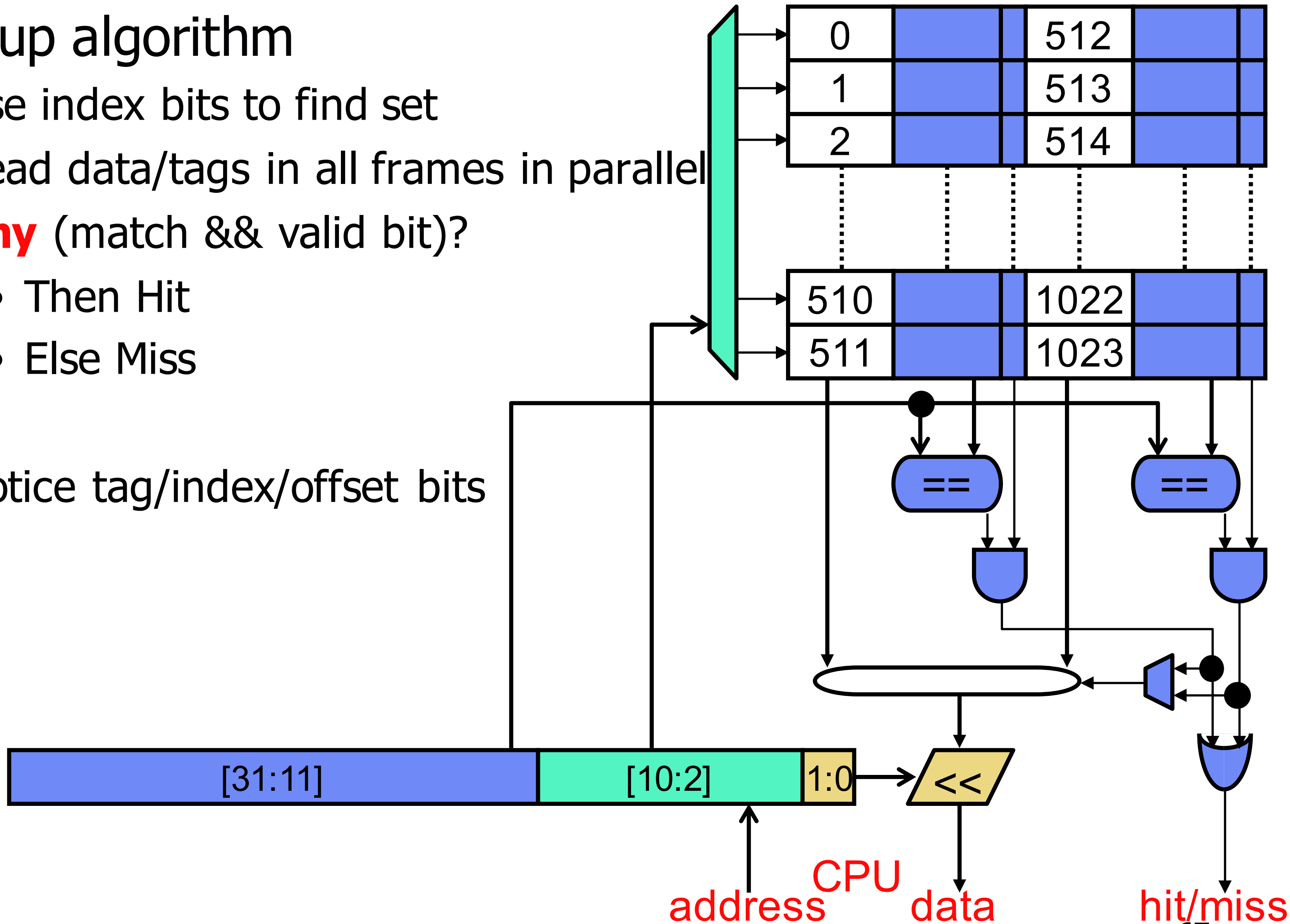
+ Reduces conflicts

– Increases $t_{hit:}$ additional mux



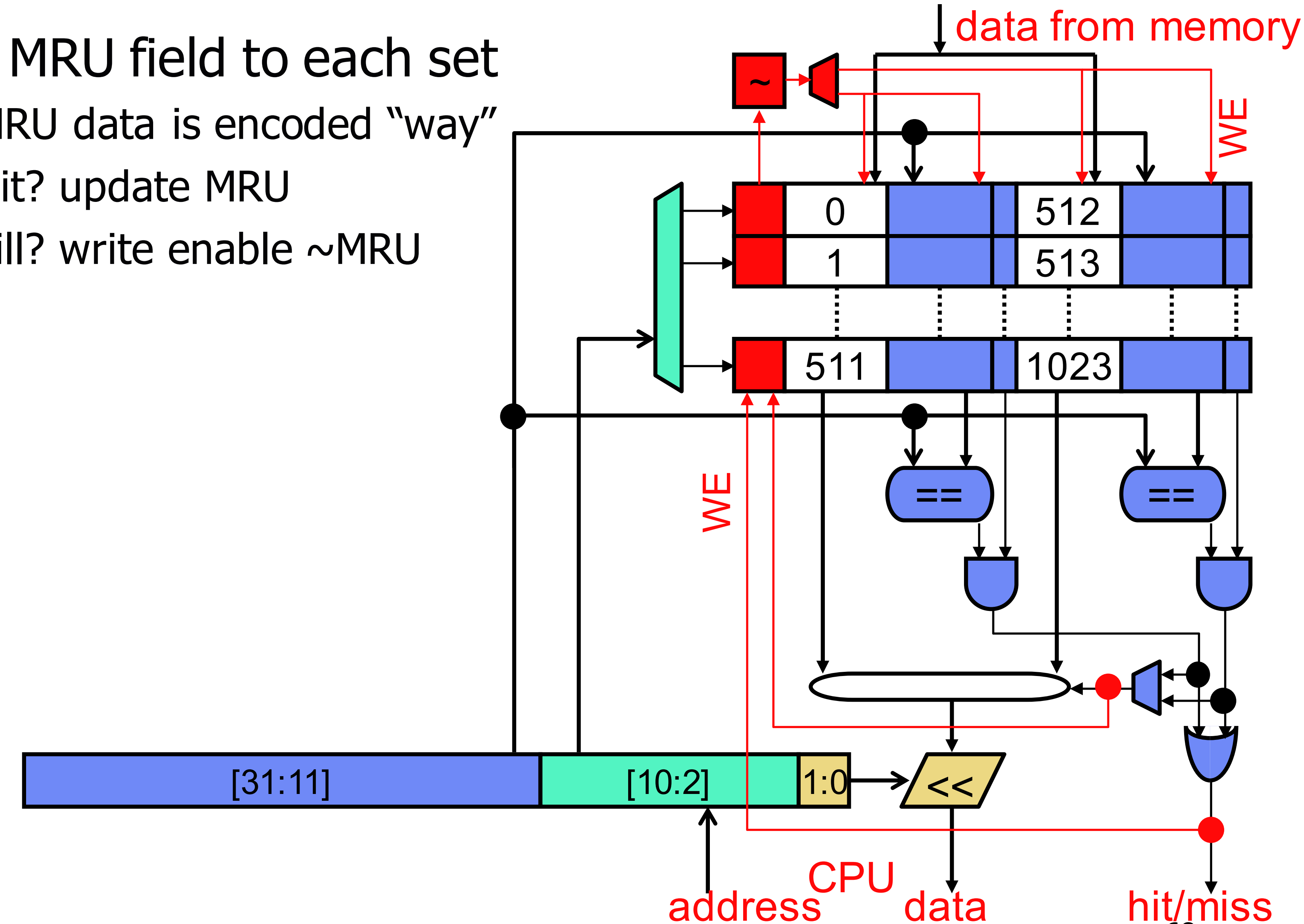
Set-Associativity

- Lookup algorithm
  - Use index bits to find set
  - Read data/tags in all frames in parallel
  - **Any** (match && valid bit)?
    - Then Hit
    - Else Miss
  - Notice tag/index/offset bits



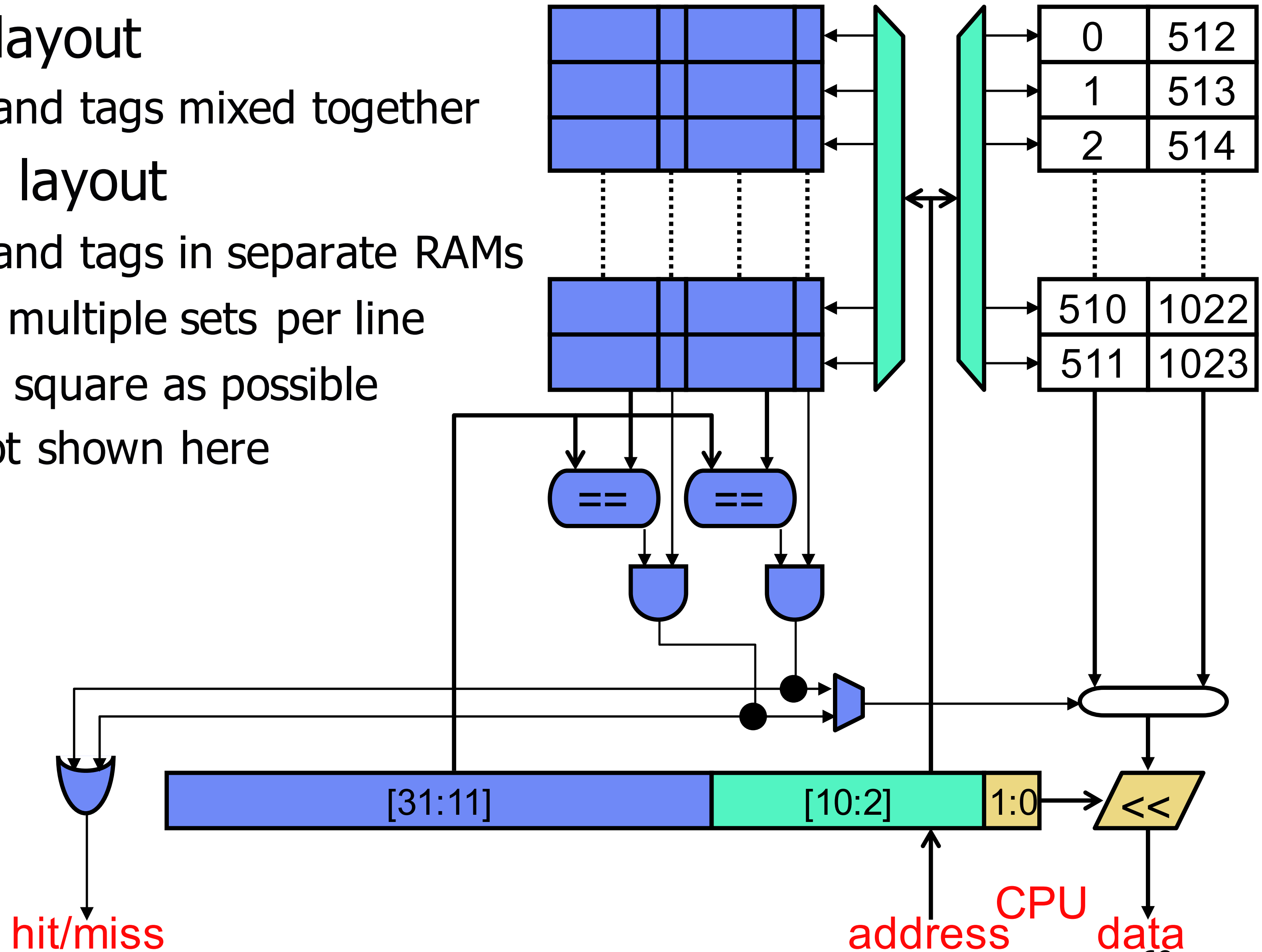
NMRU and Miss Handling

- Add MRU field to each set
  - MRU data is encoded "way"
  - Hit? update MRU
  - Fill? write enable ~MRU

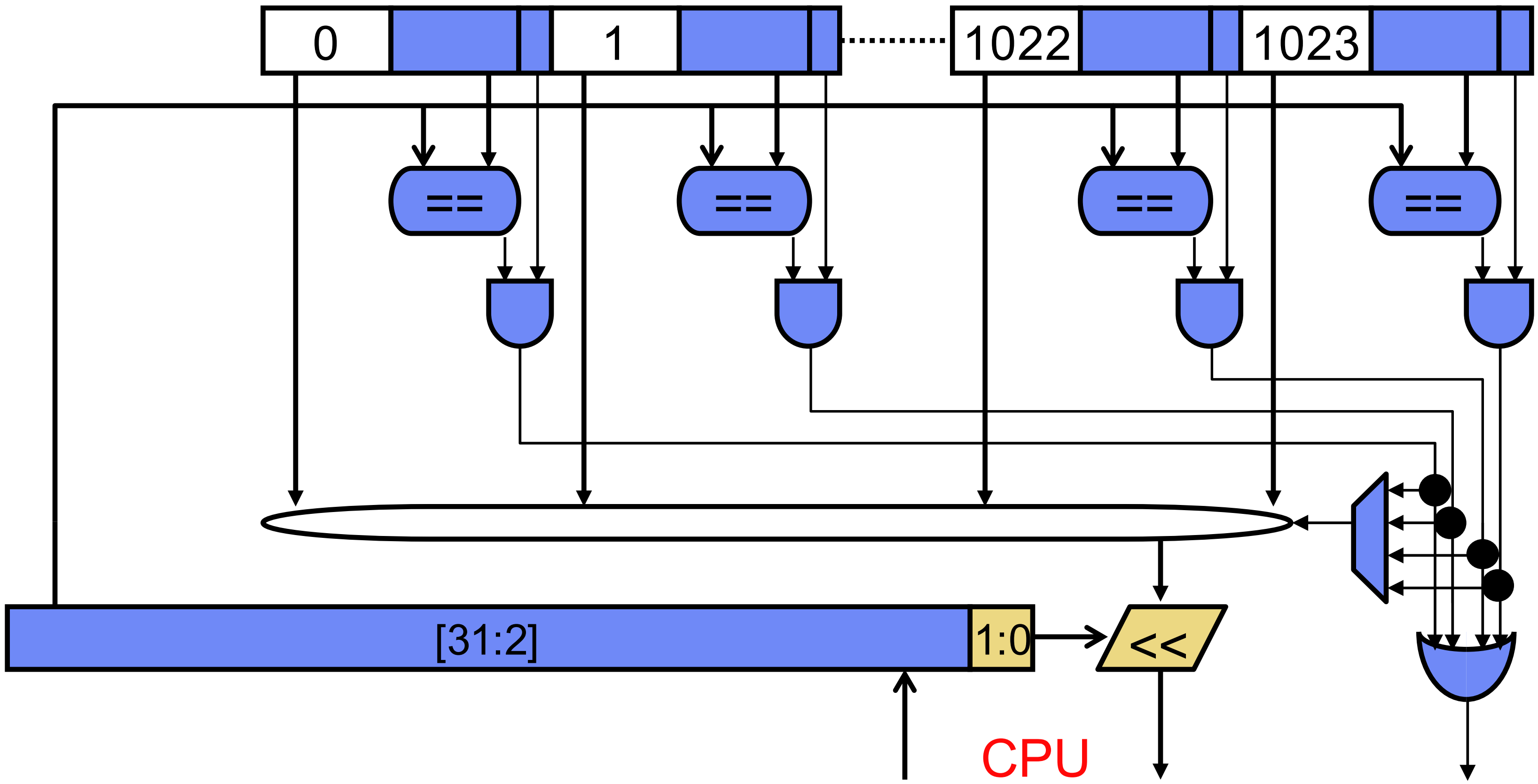


Physical Cache Layout

- Logical layout
  - Data and tags mixed together
- Physical layout
  - Data and tags in separate RAMs
  - Often multiple sets per line
    - As square as possible
    - Not shown here



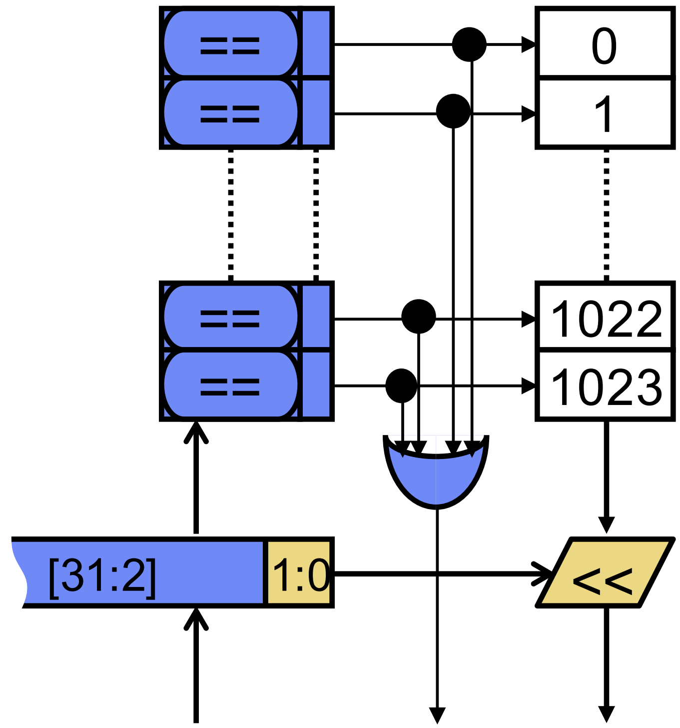
Full-Associativity



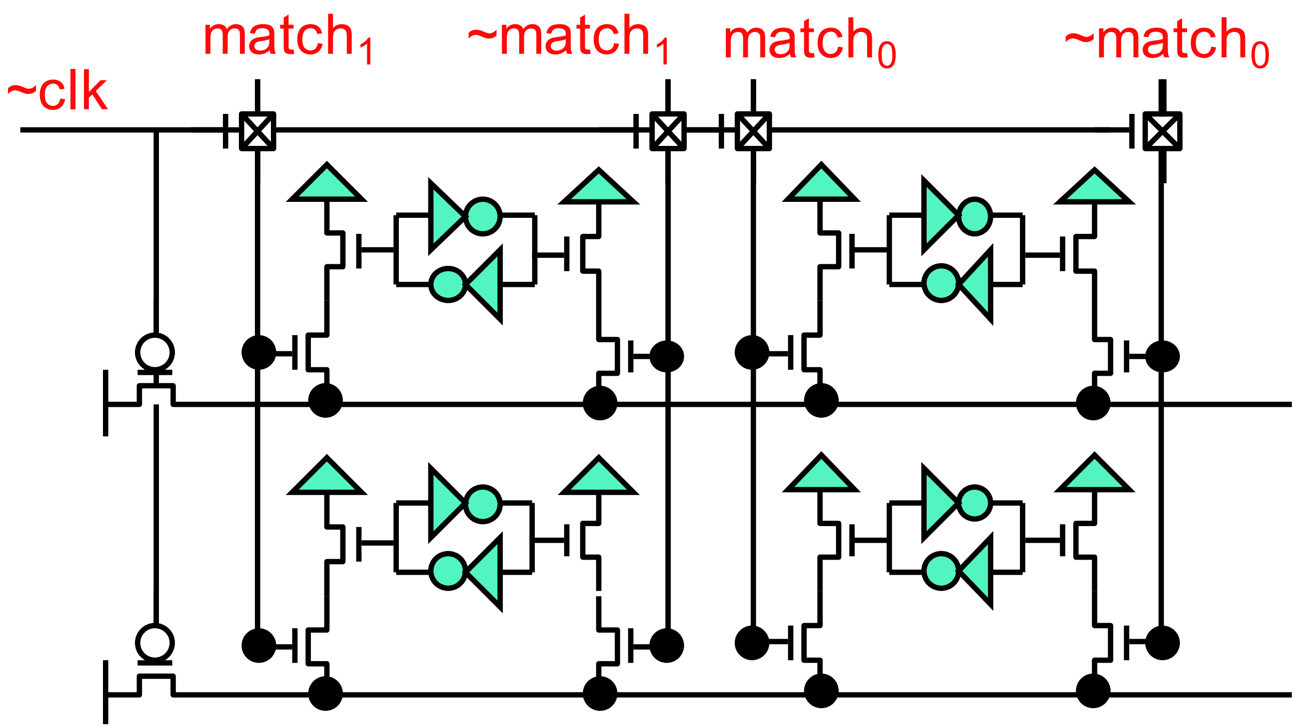
- How to implement full (or at least high) associativity?
  - Doing it this way is terribly inefficient
  - 1K matches are unavoidable, but 1K data reads + 1K-to-1 mux?

Full-Associativity with CAMs

- **CAM**: content addressable memory
  - Array of words with built-in comparators
  - Matchlines instead of bitlines
  - Output is “one-hot” encoding of match
- FA cache?
  - Tags as CAM
  - Data as RAM



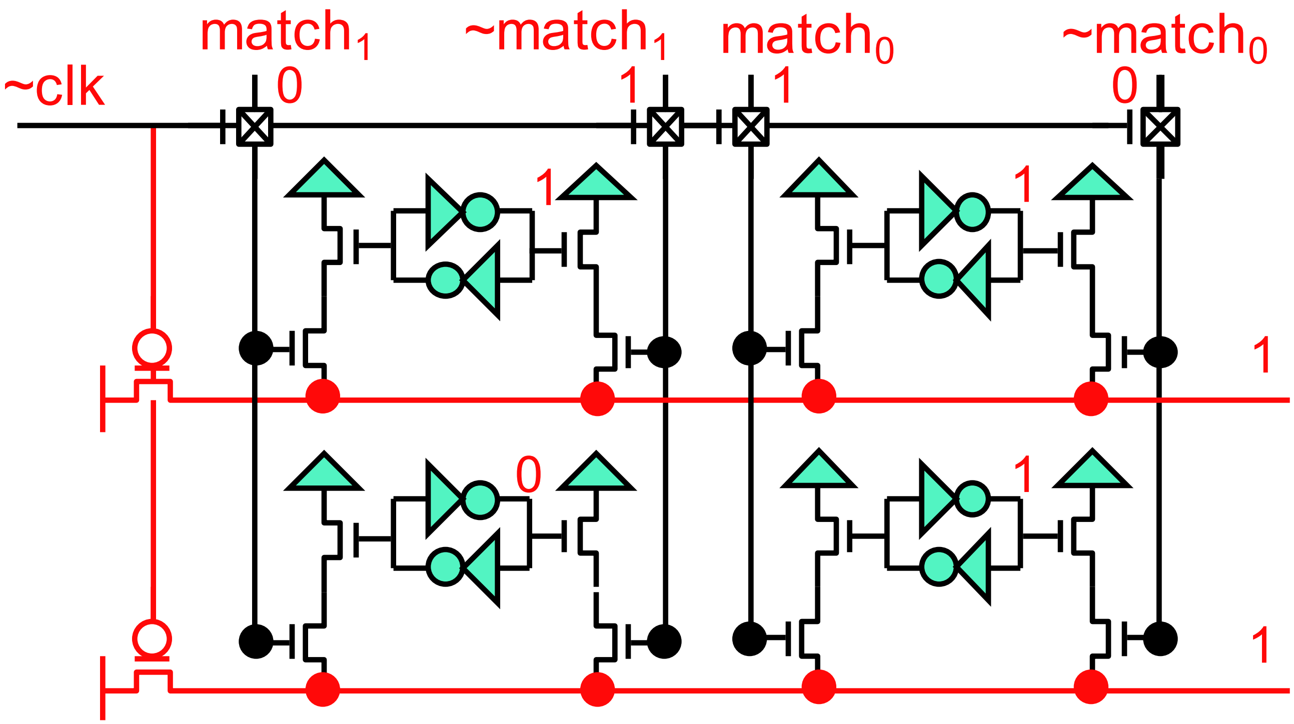
CAM Circuit



- **Matchlines** (correspond to bitlines in SRAM): inputs
- **Wordlines**: outputs

- Two phase match
  - Phase I: clk=1, pre-charge wordlines to 1
  - Phase II: clk=0, enable matchlines, non-matched bits dis-charge wordlines

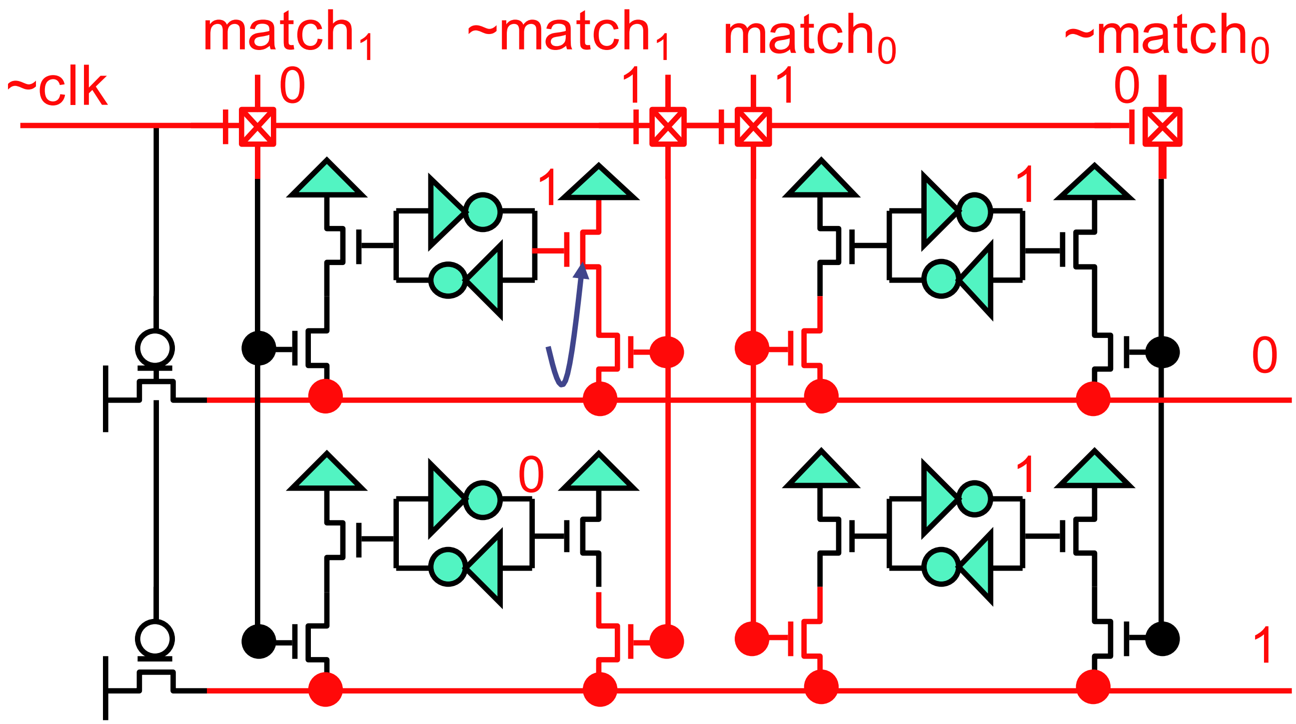
CAM Circuit In Action



- First row (1,1); Second row (0,1); Match data (0,1);

- Phase I: clk=1
  - Pre-charge wordlines to 1

CAM Circuit In Action



Looking for match with 01

- Phase I: clk=0
  - Enable matchlines (notice, match bits and value bits are flipped)
  - Any non-matching bit discharges entire wordline

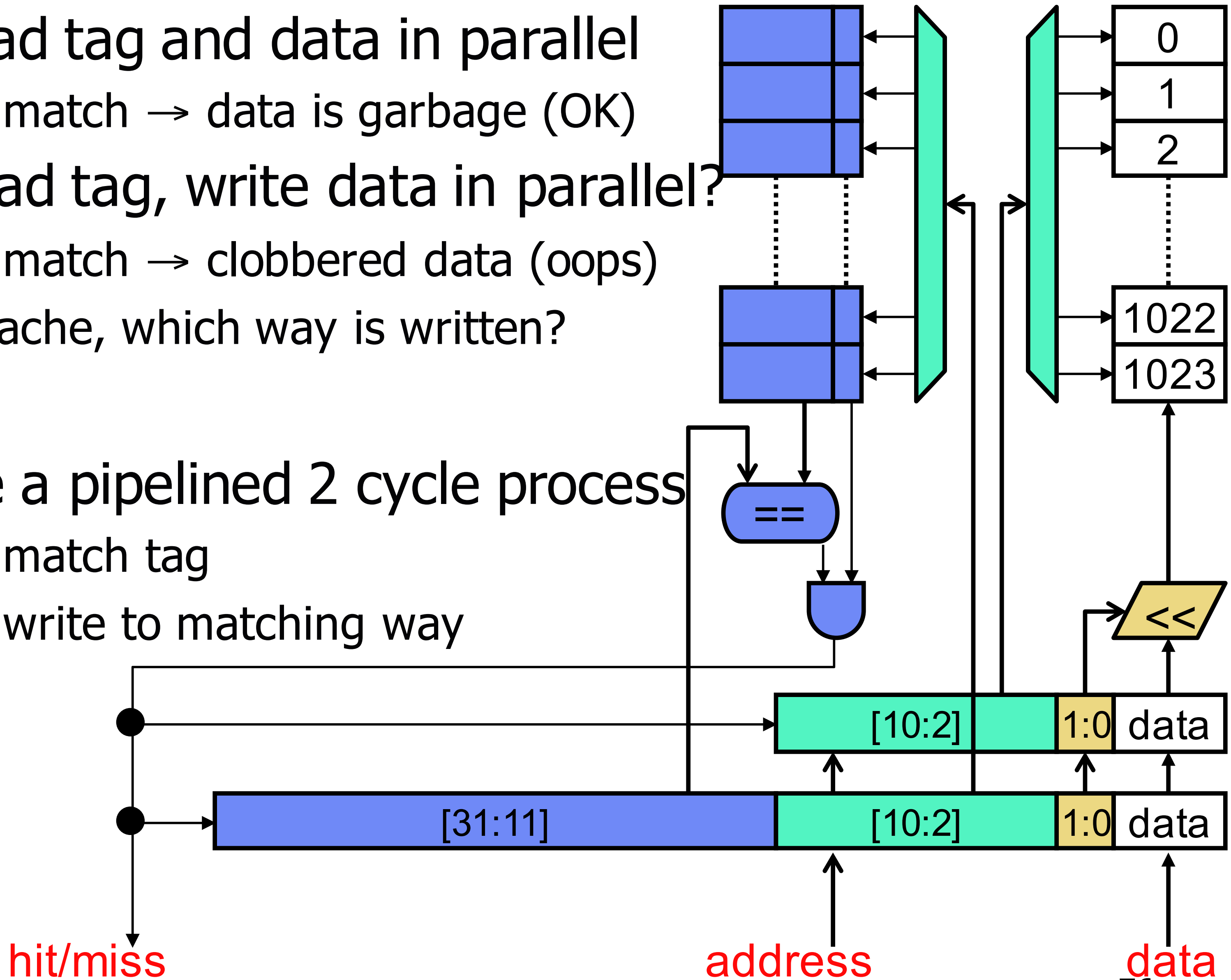
CAM Upshot

- CAMs are effective but expensive
  - – Matchlines are expensive

  - CAMs are used but only for 16 or 32 way (max) associativity
    - See an example soon
  - Not for 1024-way associativity
    - – No good way of doing something like that
    - + No real need for it either

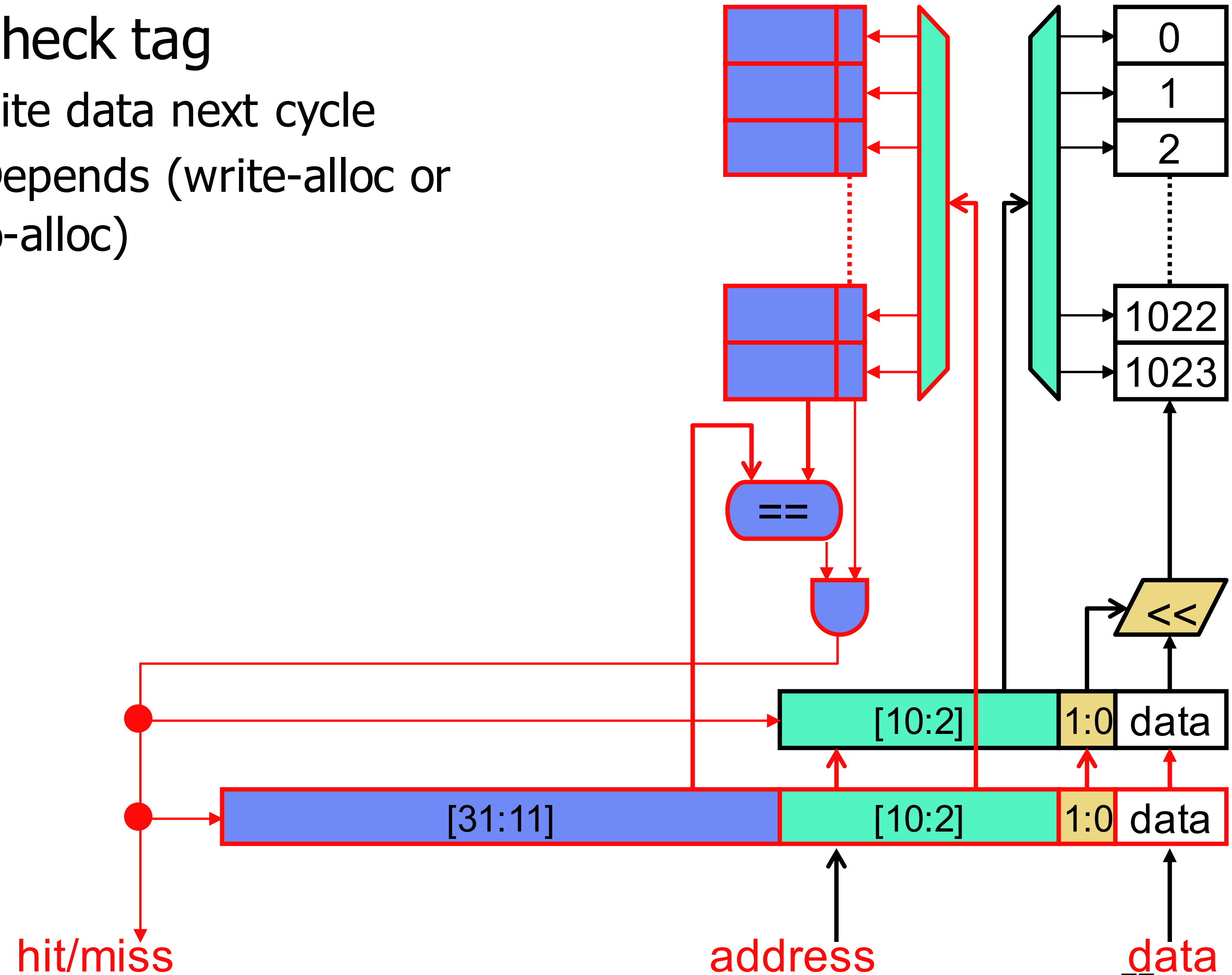
Tag + Data Access

- Reads: read tag and data in parallel
  - Tag mis-match → data is garbage (OK)
- Writes: read tag, write data in parallel?
  - Tag mis-match → clobbered data (oops)
  - For SA cache, which way is written?
- Writes are a pipelined 2 cycle process
  - Cycle 1: match tag
  - Cycle 2: write to matching way



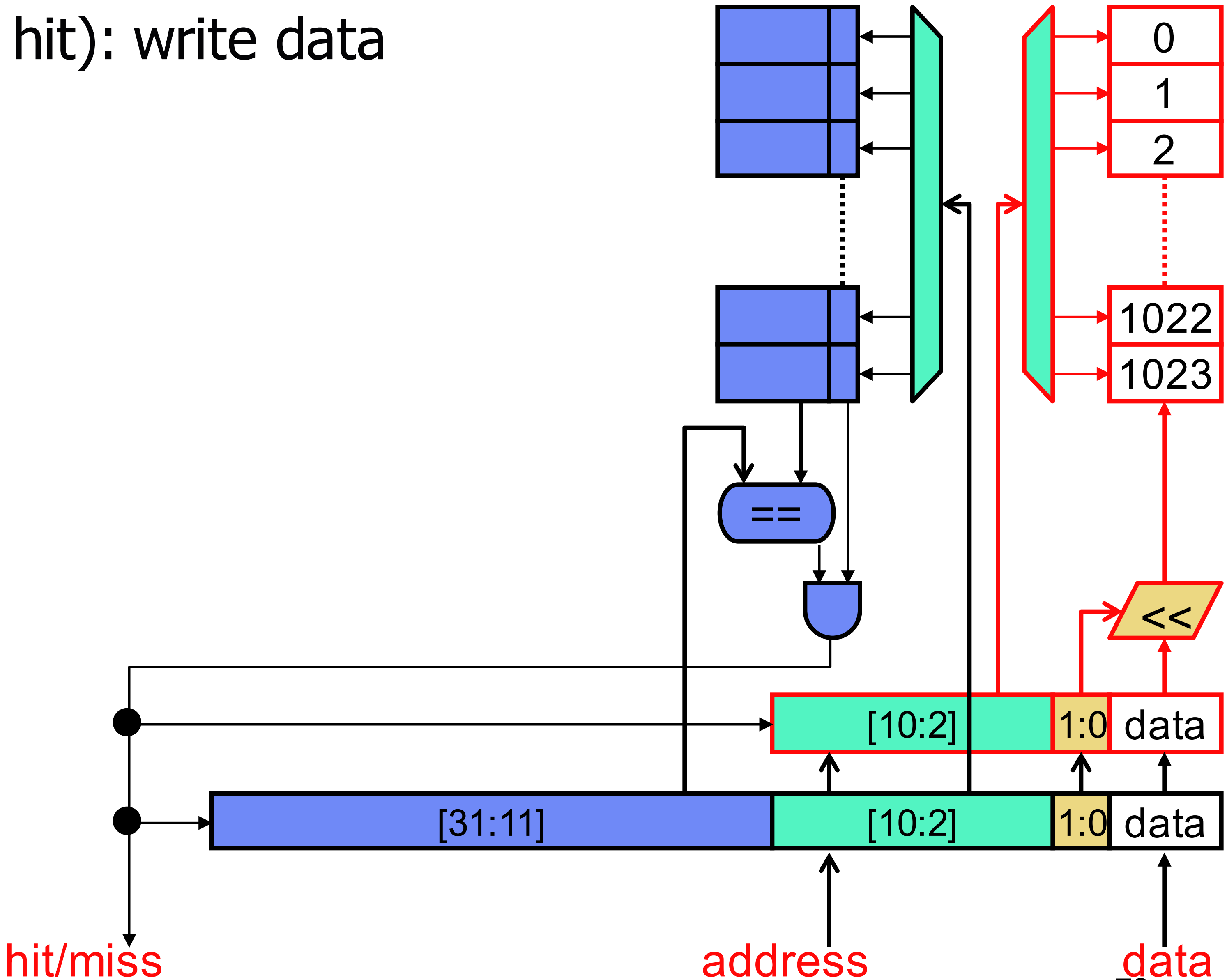
Tag + Data Access

- Cycle 1: check tag
  - Hit? Write data next cycle
  - Miss? Depends (write-alloc or write-no-alloc)

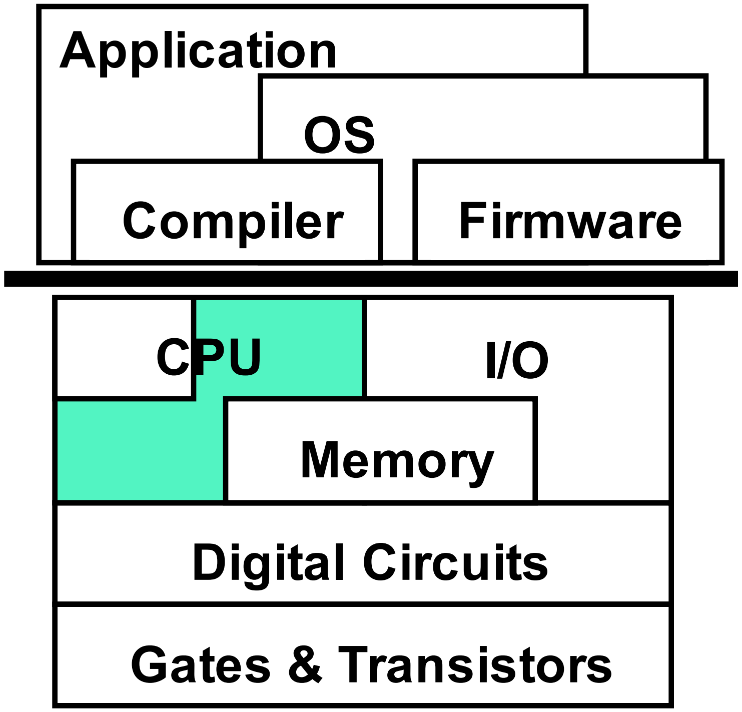


Tag + Data Access

- Cycle 2 (if hit): write data



This Unit: Caches and Memory Hierarchies



- Memory hierarchy
- Cache organization
- Cache implementation