CS/ECE 250: Computer Architecture

Caches and Memory Hierarchies

Benjamin C. Lee
Duke University

Slides from Daniel Sorin (Duke)
and are derived from work by
Amir Roth (Penn) and Alvy Lebeck (Duke)

Administrivia

- Homework #4
  - Due 11:55pm on Tuesday, Apr 1

- Midterm II – Tuesday, Mar 25
  - Covers logic design, datapath and control
  - Does not include memory hierarchy

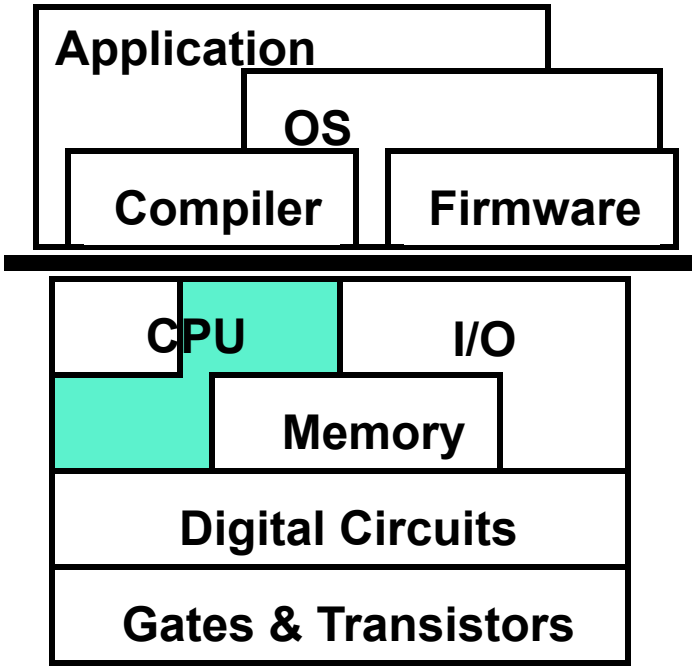
- Two more homeworks
  - HW #5 Friday, Apr 11 – Memory hierarchy
  - HW #6 Wednesday, Apr 23 -- Exceptions, I/O, pipelining

- Reading
  - Chapter 5, Patterson and Hennessy

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

This Unit: Caches and Memory Hierarchies



- Memory hierarchy
  - Basic concepts
- SRAM technology
  - Transistors and circuits
- Cache organization
  - ABC
  - CAM (content associative memory)
  - Classifying misses
  - Two optimizations
  - Writing into a cache
- Some example calculations

Readings

- Patterson and Hennessy
  - Chapter 5

Storage

- We have already seen some storage implementations
  - **Individual registers**
    - For singleton values: e.g., PC, PSR
    - For μarch/transient values: e.g., in pipelined design
  - **Register File**
    - For architectural values: e.g., ISA registers

- What else is there?

Storage Hierarchy

- Registers
  - Few locations: e.g., 32 4-byte words
  - Accessible directly via user-level ISA: multiple specifiers per insn
  - **Volatile** (values disappear when power goes off)
- **Memory**
  - Many (but finite) locations: e.g., 2^{32} bytes
  - Accessible indirectly via user-level ISA: one specifier per insn
  - Also volatile
- Disk
  - "Infinitely" many locations
  - Not accessible to user-level ISA (only via OS SYSCALL)
  - Non-volatile

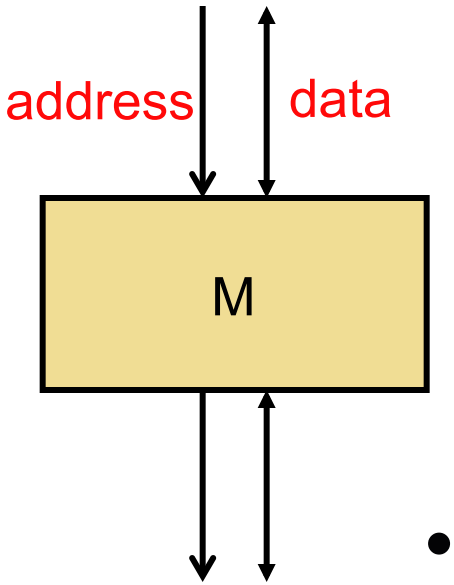
Storage Hierarchy

- Registers vs. memory
  - Direct specification (fast) vs. address calculation (slow)
  - Few addresses (small & fast) vs. many (big & slow)
  - Not everything can be put into registers (e.g., arrays, structs)

- Memory vs. disk
  - Electrical (fast) vs. electro-mechanical (extremely slow)
  - Disk is so slow (relatively), it is considered I/O

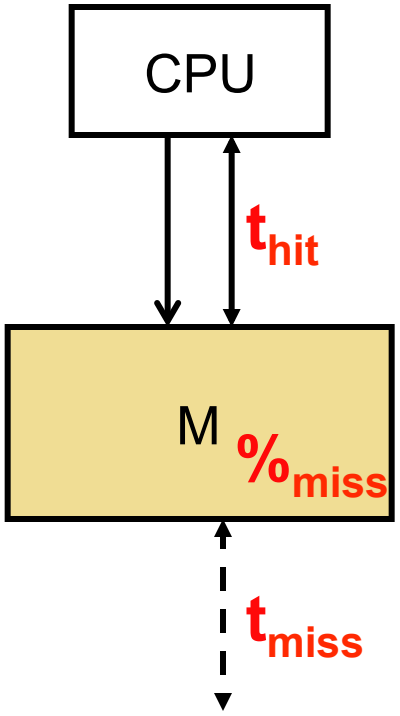
- We will talk just about memory for **instructions** and **data**

(CMOS) Memory Components



- Interface
  - N-bit **address** bus (on N-bit machine)
  - **Data** bus
    - Typically read/write on same data bus
  - Can have multiple **ports**: address/data bus pairs
  - Can be **synchronous**: read/write on clock edges
  - Can be **asynchronous**: untimed "handshake"
- Performance
  - Access time proportional to (#ports) * √(#bits)
  - √(#bits)? Proportional to max wire length
    - More about this a little later …

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

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

Memory Hierarchy

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

- 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
  - A very old (by computer standards) idea:

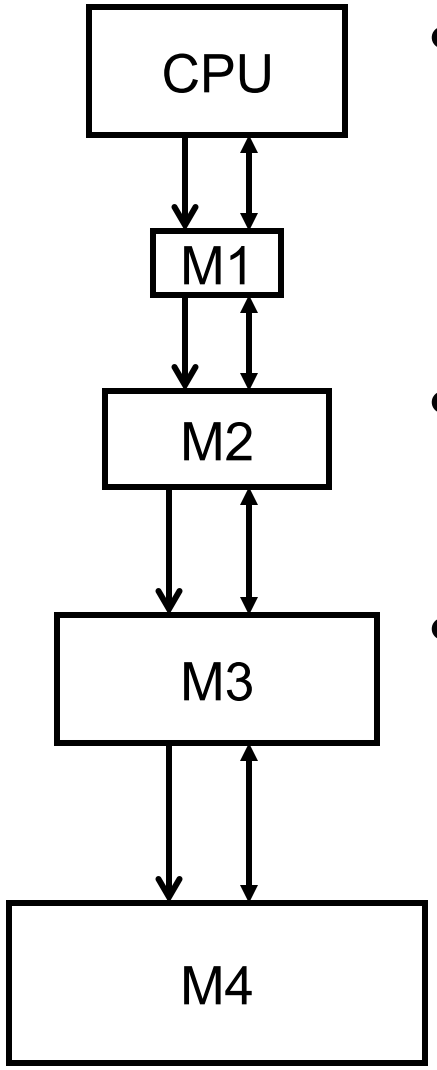
> "Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available … We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible."

Burks, Goldstine, and Von Neumann

"Preliminary discussion of the logical design of an electronic computing instrument"

IAS memo **1946**

Abstract Memory Hierarchy



- Hierarchy of memory components
  - Upper components (closer to CPU)
    - Fast ↔ Small ↔ Expensive
  - Lower components (further from CPU)
    - Slow ↔ Big ↔ Cheap
- Connected by buses
  - Which we will ignore for now
- Make average access time close to M1's
  - How?
  - Most frequently accessed data in M1
  - M1 + next most frequently accessed in M2, etc.
  - **Automatically** move data up & down hierarchy

Why Hierarchy Works I

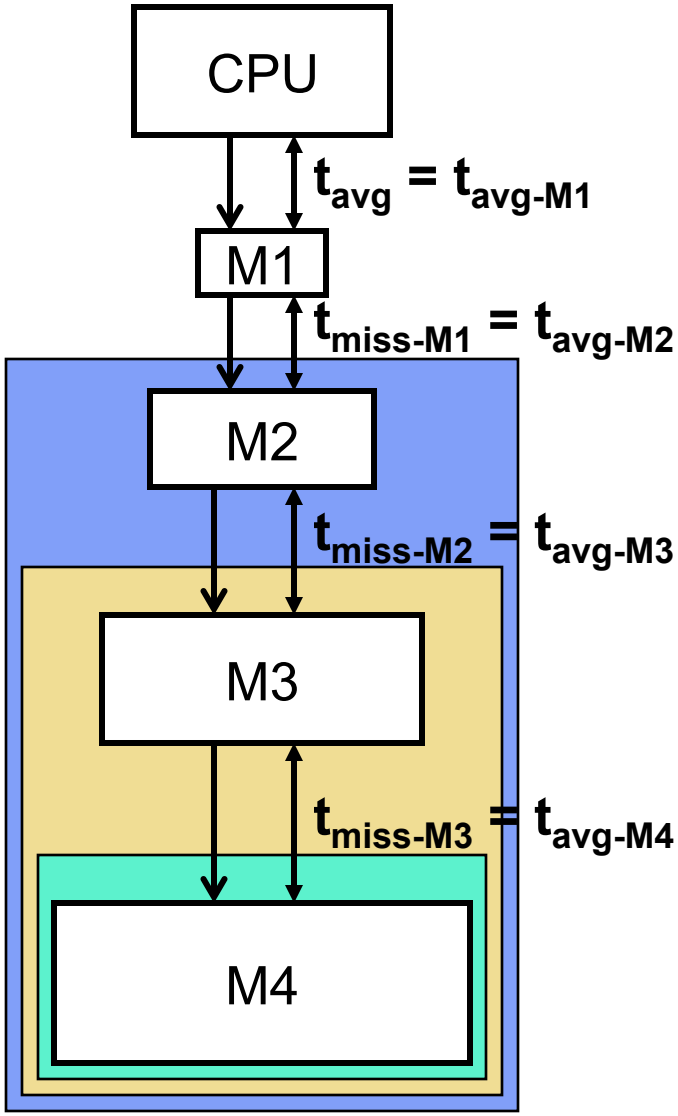
- **10/90 rule (of thumb)**
  - For Instruction Memory:
    - 10% of static insns account for 90% of executed insns
    - Inner loops
  - For Data Memory:
    - 10% of variables account for 90% of accesses
    - Frequently used globals, inner loop stack variables

Why Hierarchy Works II

- **Temporal locality**
  - Recently executed insns likely to be executed again soon
    - Inner loops (next iteration)
  - Recently referenced data likely to be referenced again soon
    - Data in inner loops, hot global data
  - Hierarchy can be **"reactive"**: move things up when accessed

- **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 an array, fields in a struct, variables in frame
  - Hierarchy can be **"proactive"**: move things up speculatively

Abstract Hierarchy Performance



How do we compute t_{avg} ?

$= t_{avg-M1}$

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

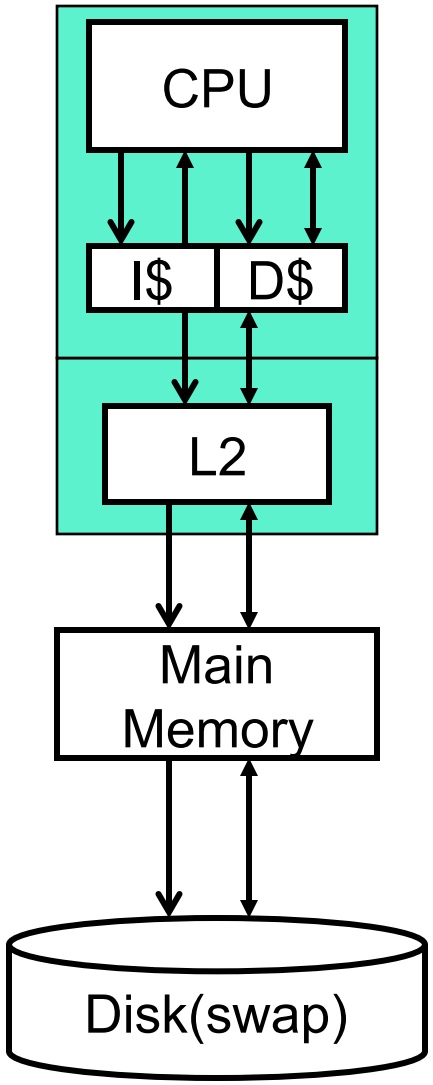
$= t_{hit-M1} + (\%_{miss-M1} * t_{avg-M2})$

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

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

$= \ldots$

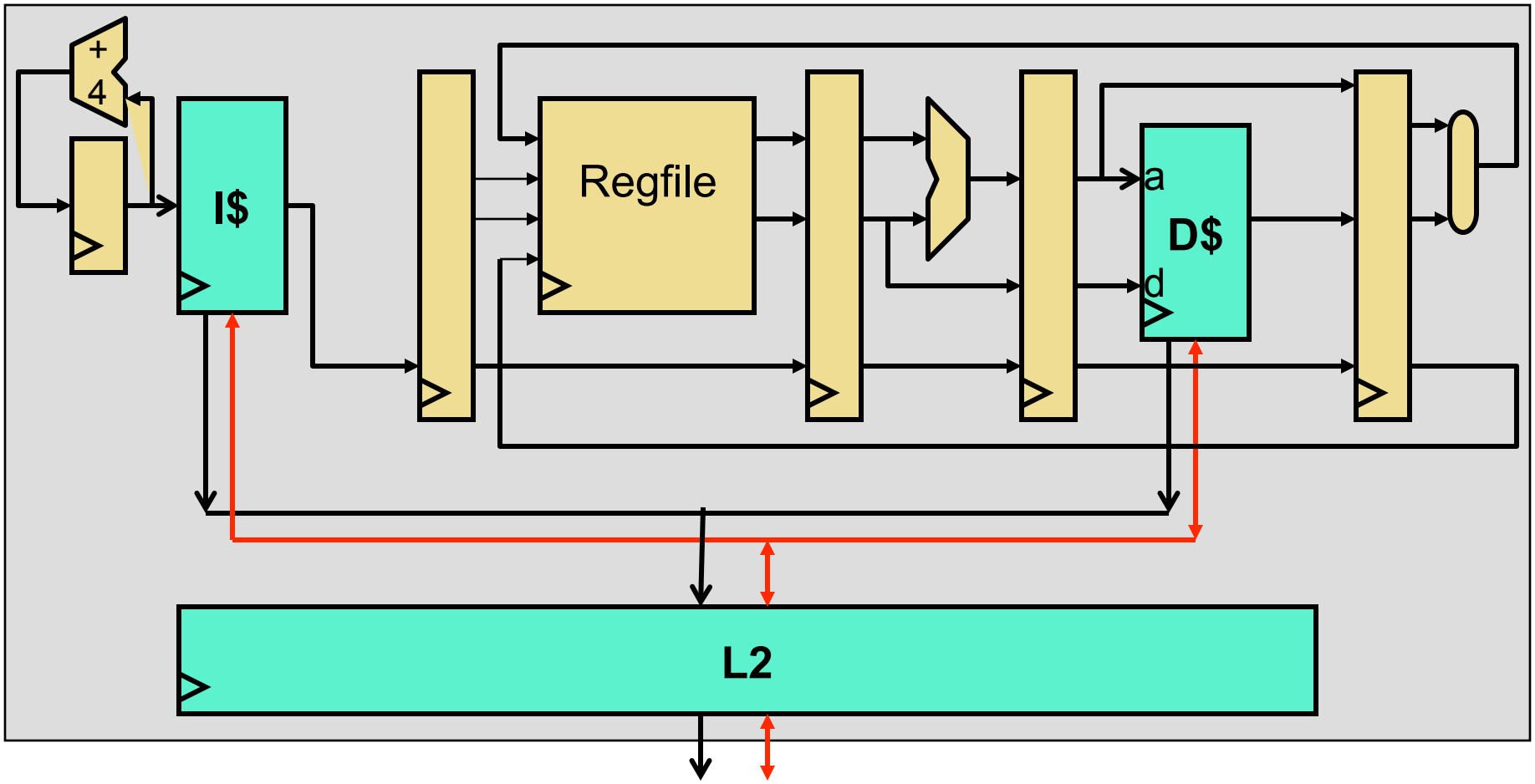
Concrete Memory Hierarchy



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

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

Concrete Memory Hierarchy

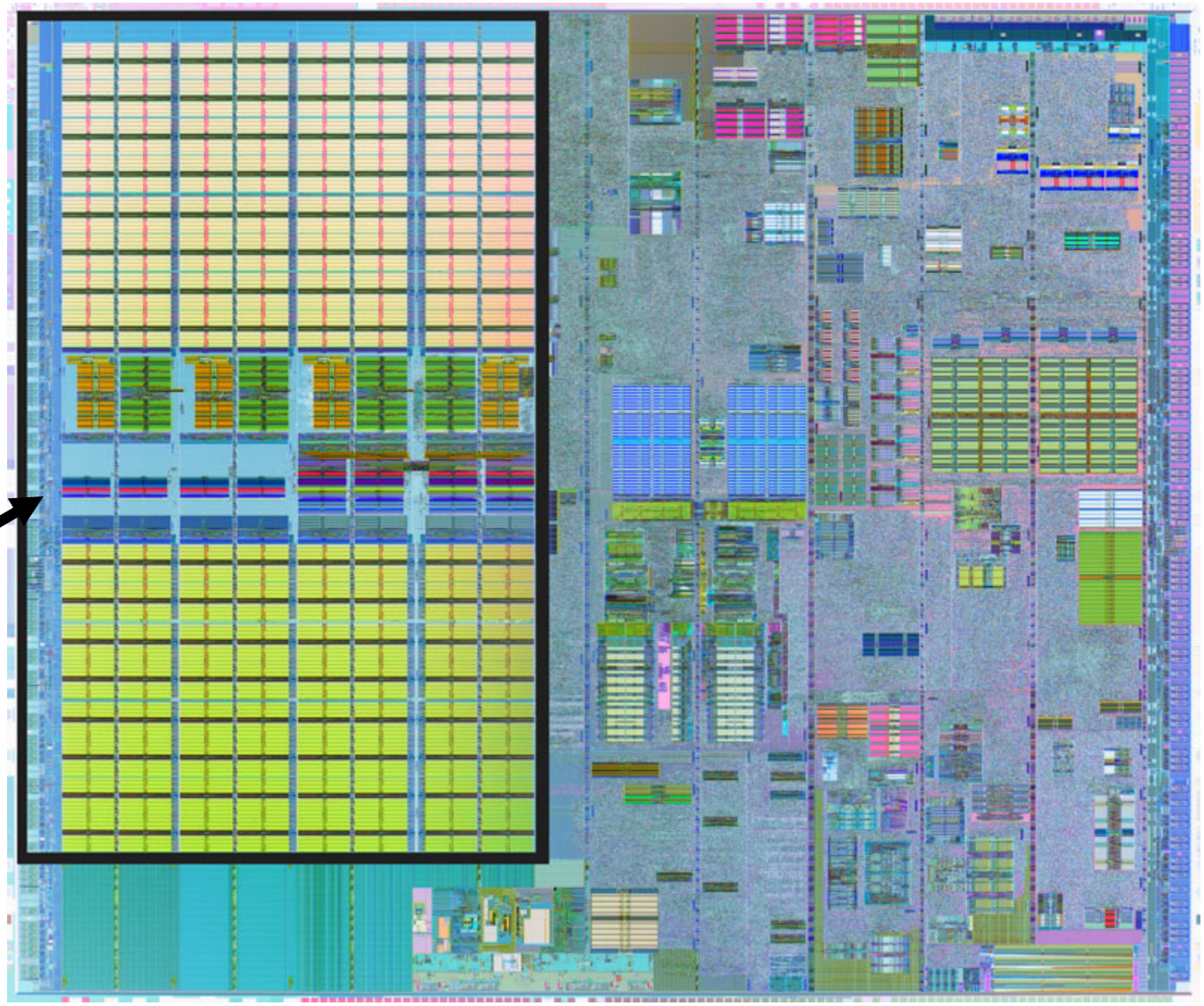


- Much of today's chips used for caches → important!

A Typical Die Photo

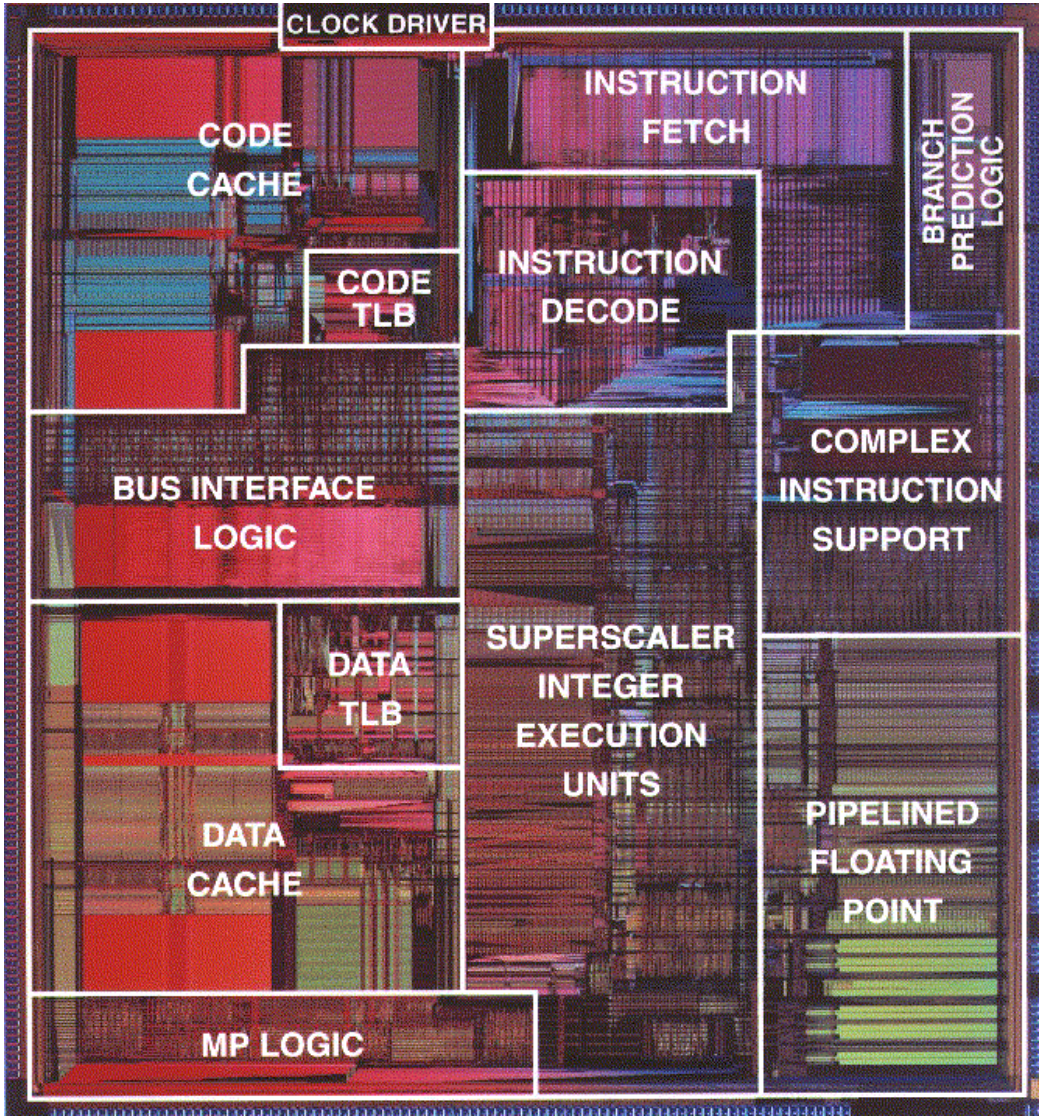
Pentium4 Prescott chip with 2MB L2$

L2 Cache



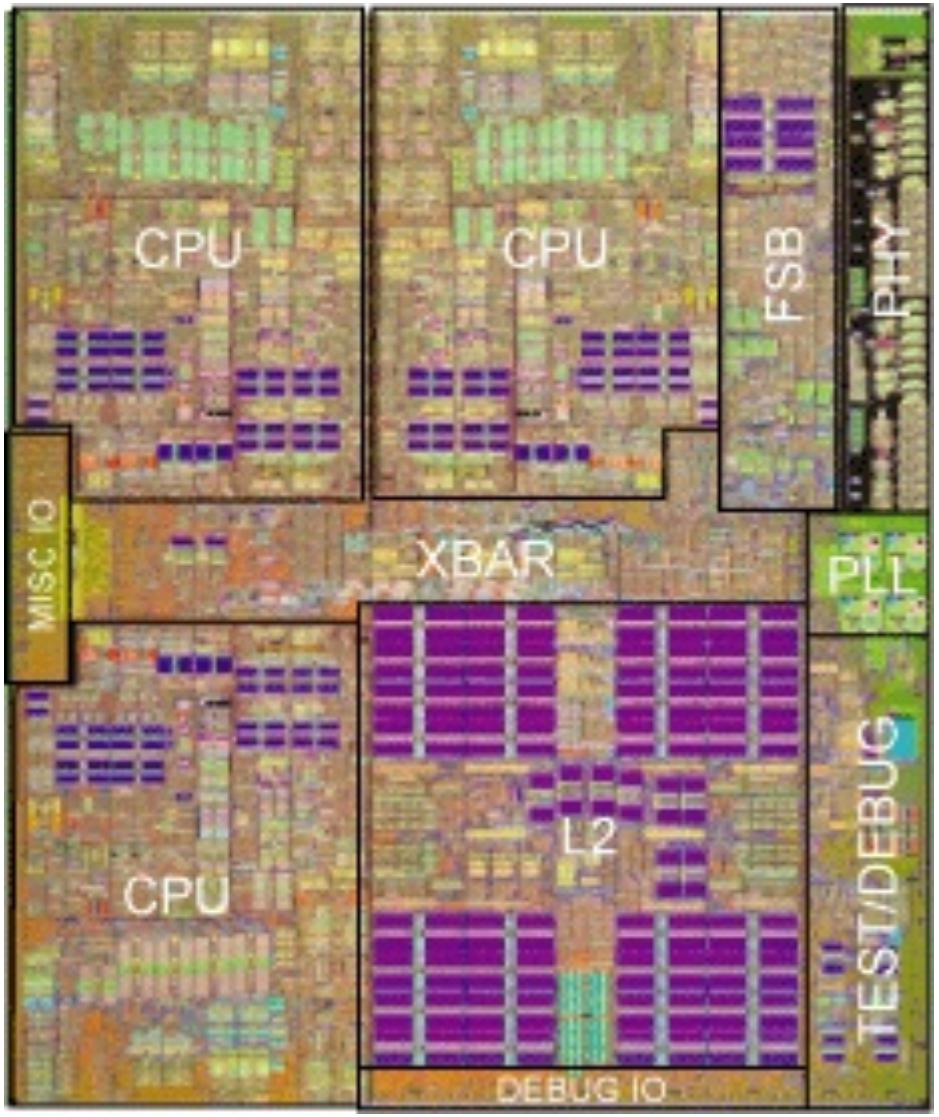
A Closer Look at that Die Photo

Pentium4 Prescott chip with 2MB L2$

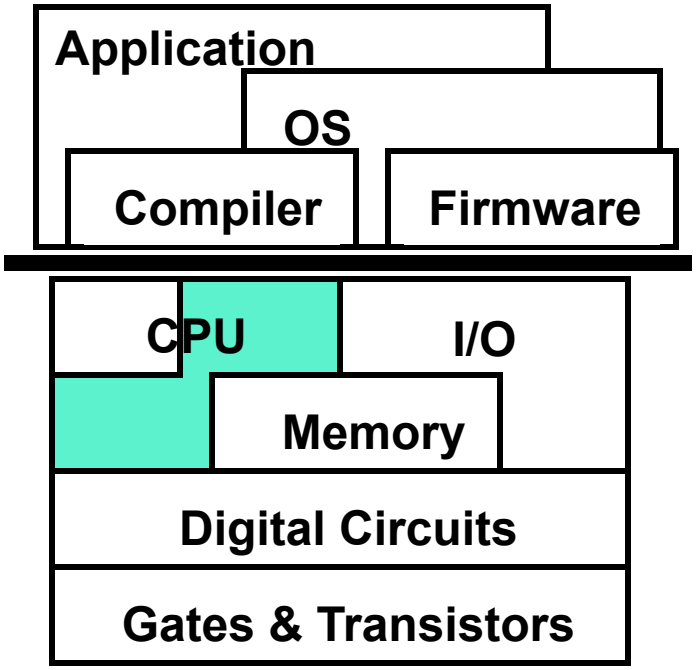


A Multicore Die Photo from IBM

IBM's Xenon chip with 3 PowerPC cores

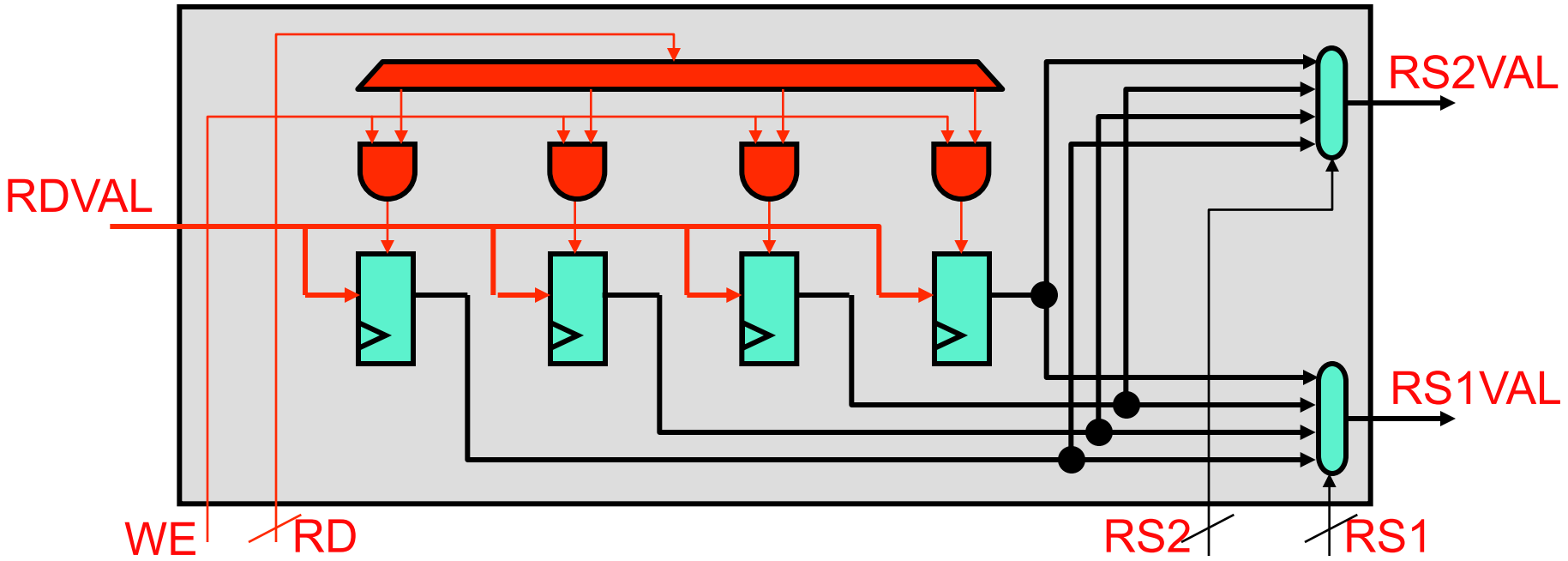


This Unit: Caches and Memory Hierarchies



- Memory hierarchy
  - Basic concepts
- SRAM technology
  - Transistors and circuits
- Cache organization
  - ABC
  - CAM (content associative memory)
  - Classifying misses
  - Two optimizations
  - Writing into a cache
- Some example calculations

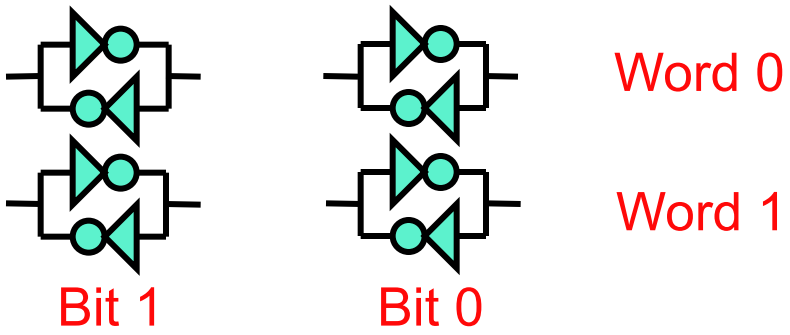
Implementing Big Storage Arrays



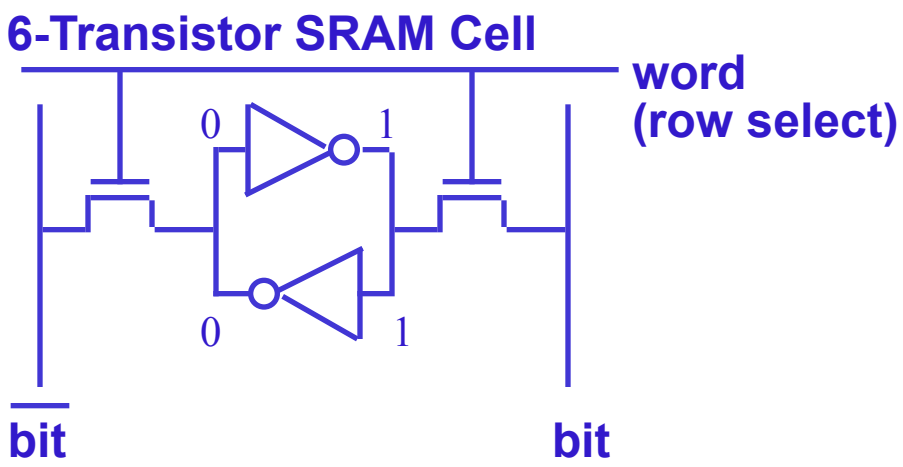
- Register file: bits as flip-flops, read ports as muxes
  - Not realistic, even if we replace muxes with tri-state buffers
  - MIPS register file: each read port is a 32-bit 32-to-1 mux?
    - Just routing the wires would be a nightmare
  - What about a **cache**? each read port is a 1024-to-1 mux? Yuck!

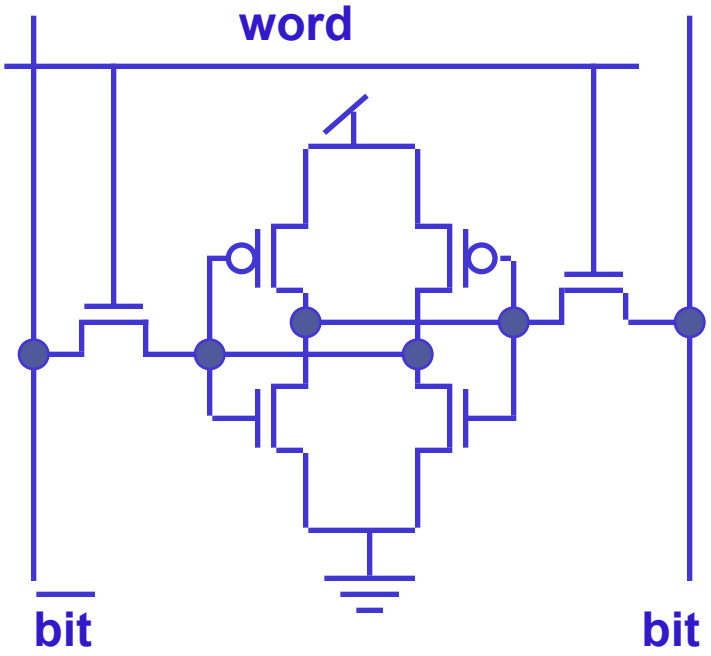
SRAM

- 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 (but still volatile)
- Example: storage array with two 2-bit words



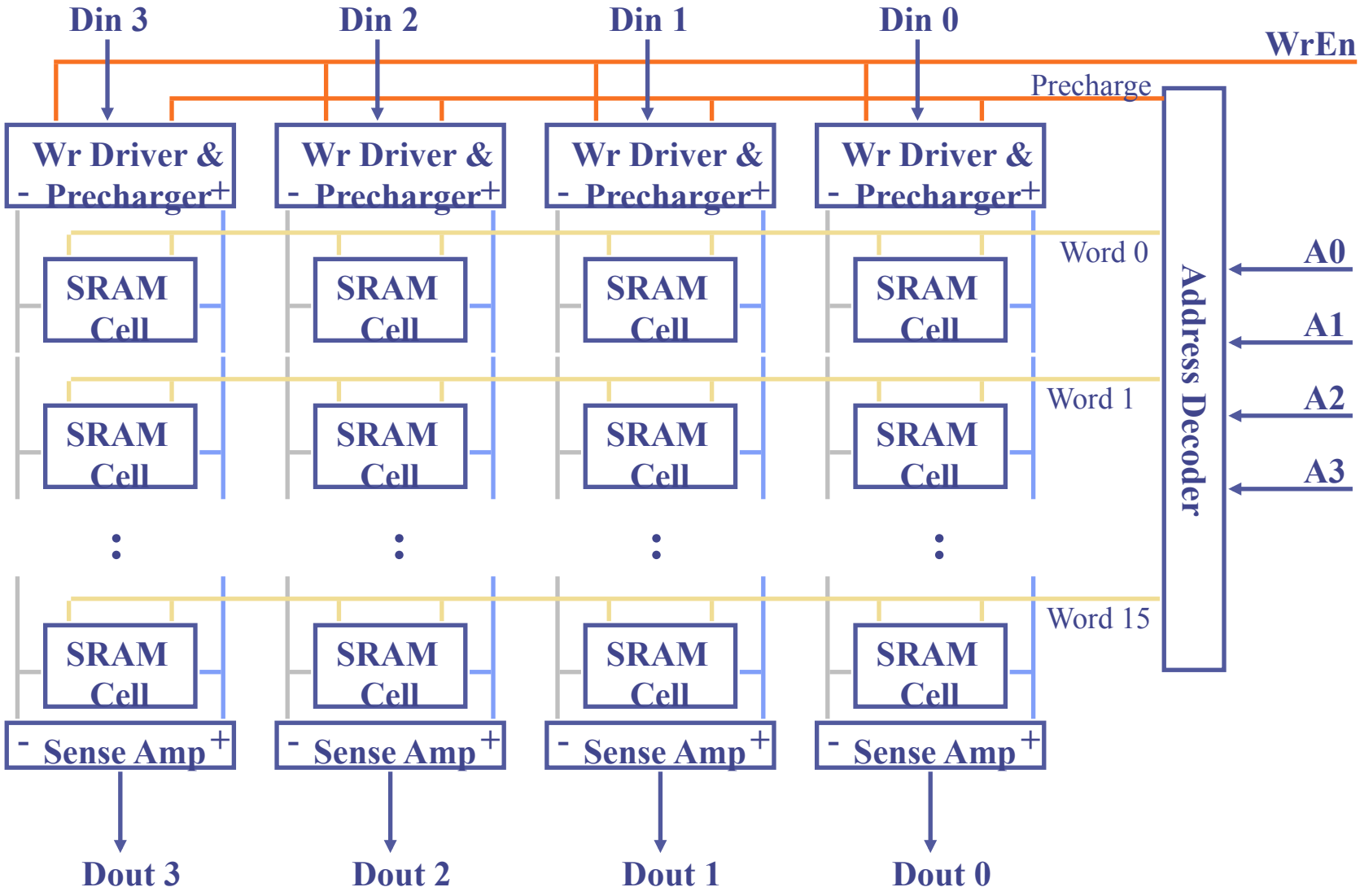
Static RAM Cell



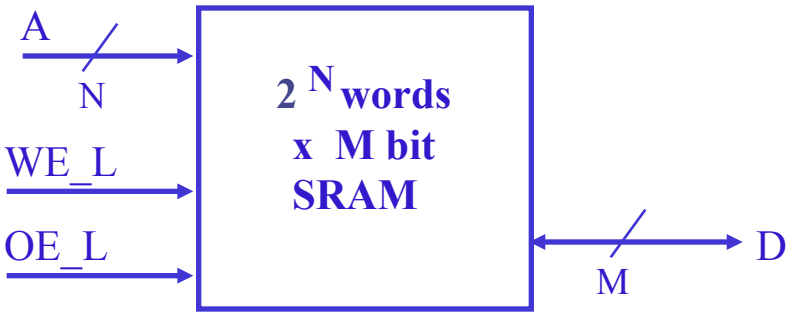


- To write (a 1):
  1. Drive bit lines (bit=1, $\overline{bit}$=0)
  2. Select row
- To read:
  1. Pre-charge bit and $\overline{bit}$ to Vdd (set to 1)
  2. Select row
  3. Cell pulls one line lower (pulls towards 0)
  4. Sense amp on column detects difference between bit and $\overline{bit}$

Typical SRAM Organization: 16-word x 4-bit

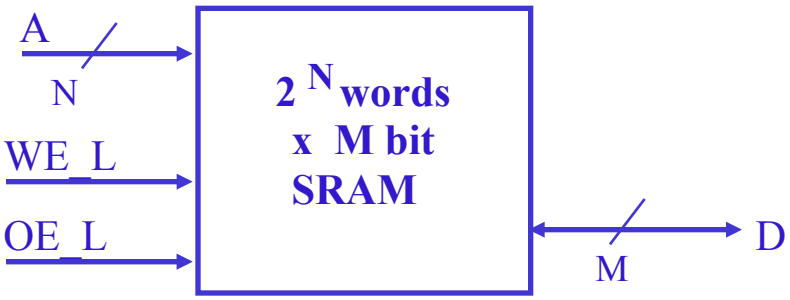


Logic Diagram of a Typical SRAM



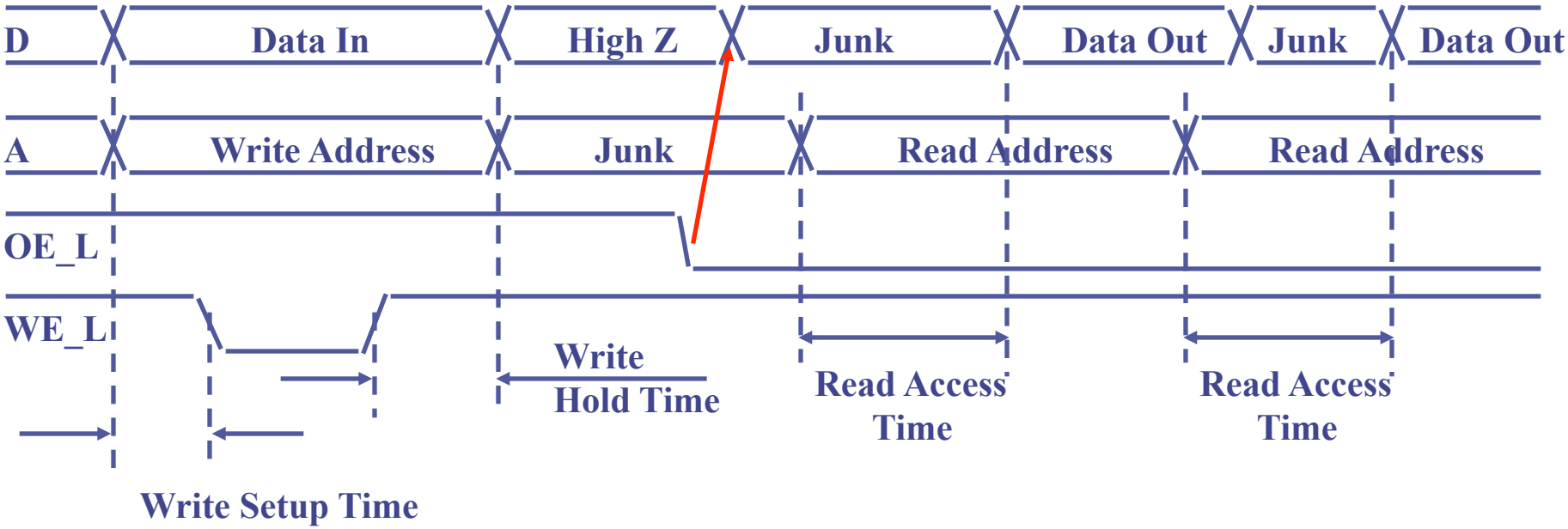
- Write Enable is usually active low (WE_L)
- Din and Dout are combined (D) to save pins:
  - A new control signal, output enable (OE_L) is needed
  - WE_L is asserted (Low), OE_L is de-asserted (High)
    - D serves as the data input pin
  - WE_L is de-asserted (High), OE_L is asserted (Low)
    - D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

Typical SRAM Timing: Write then 2 Reads



Write Timing:

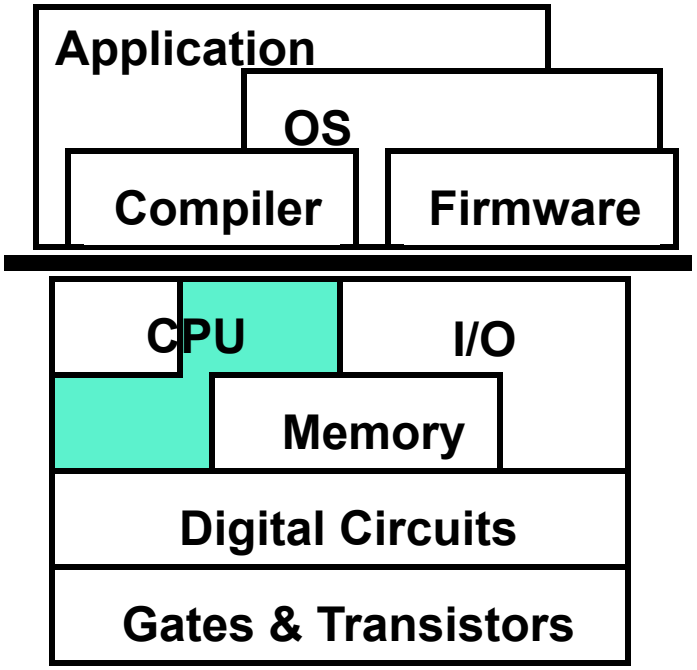
Read Timing:



SRAM 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)

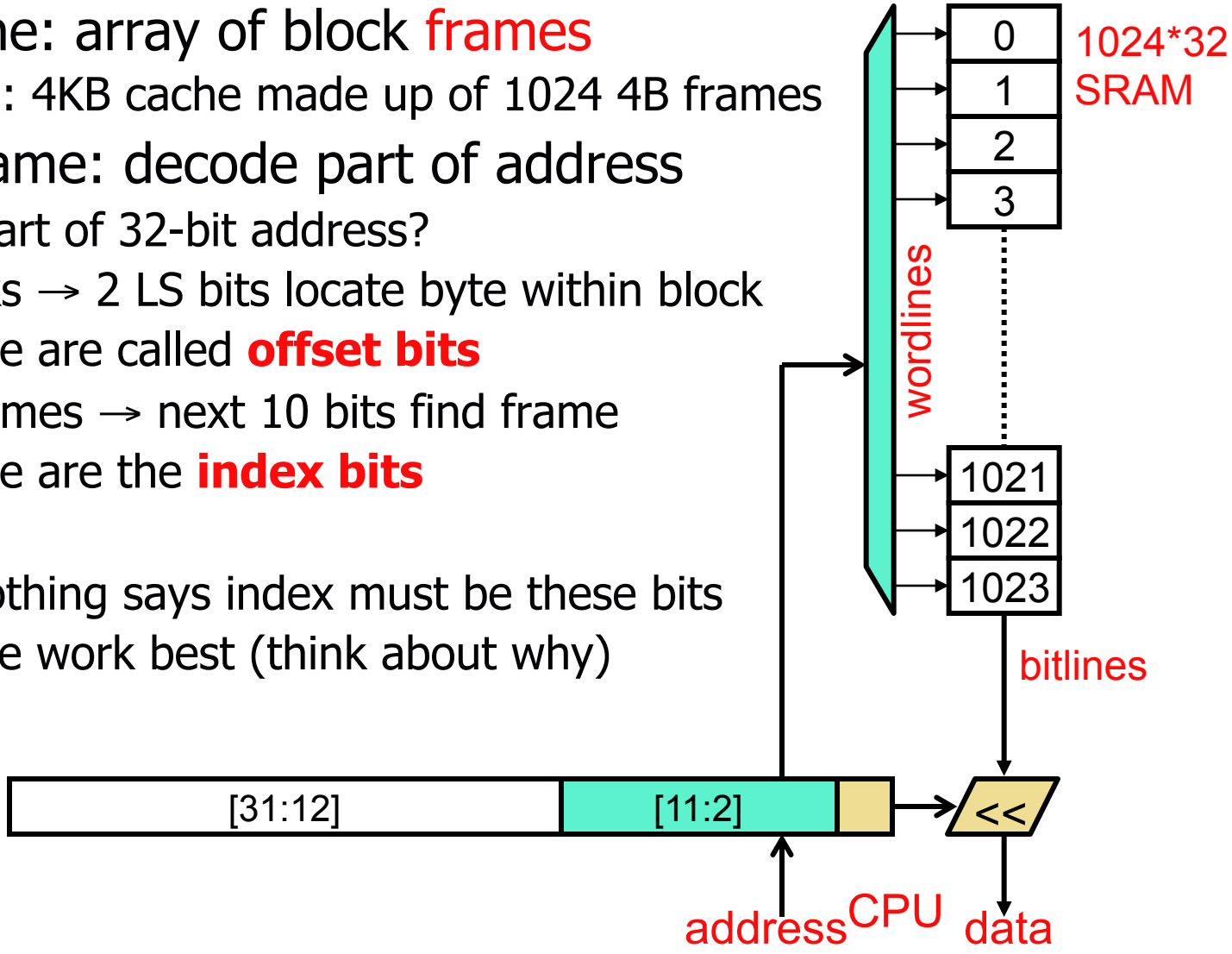
This Unit: Caches and Memory Hierarchies



- Memory hierarchy
  - Basic concepts
- SRAM technology
  - Transistors and circuits
- Cache organization
  - ABCs
  - CAM (content associative memory)
  - Classifying misses
  - Two optimizations
  - Writing into a cache
- Some example calculations

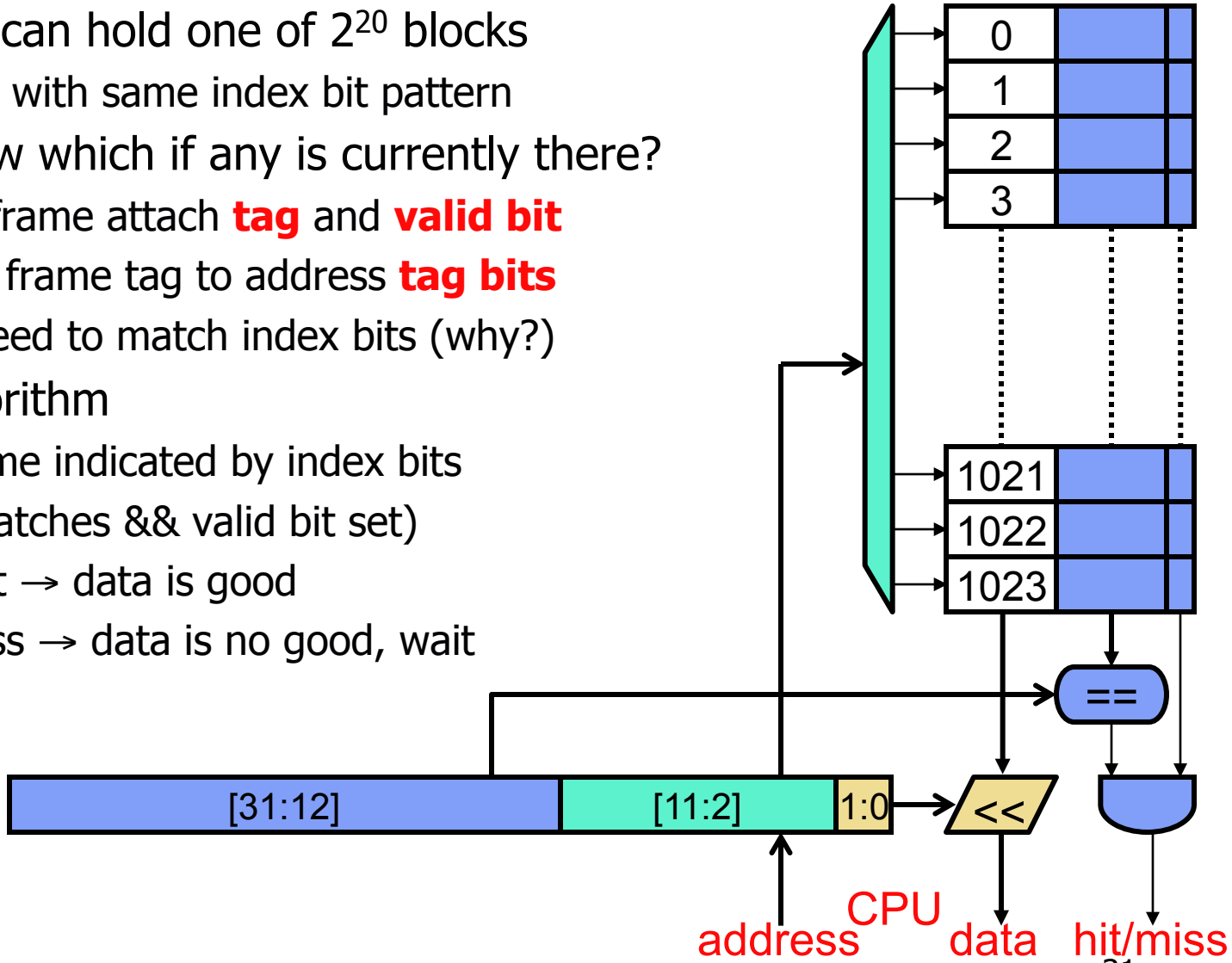
Basic Cache Structure

- Basic cache: array of block frames
  - Example: 4KB cache made up of 1024 4B frames
- To find frame: decode part of address
  - Which part of 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 bit pattern
- How to know which 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



Calculating Tag Size

- "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

Measuring Cache Performance

- Ultimate metric is t_{avg}
  - Cache capacity roughly determines t_{hit}
  - Lower-level memory structures determine t_{miss}
  - Measure $\%_{miss}$
    - Hardware performance counters (Pentium, Sun, etc.)
    - Simulation (write a program that mimics behavior)
    - Hand simulation (next slide)
  - $\%_{miss}$ depends on program that is running
    - Why?

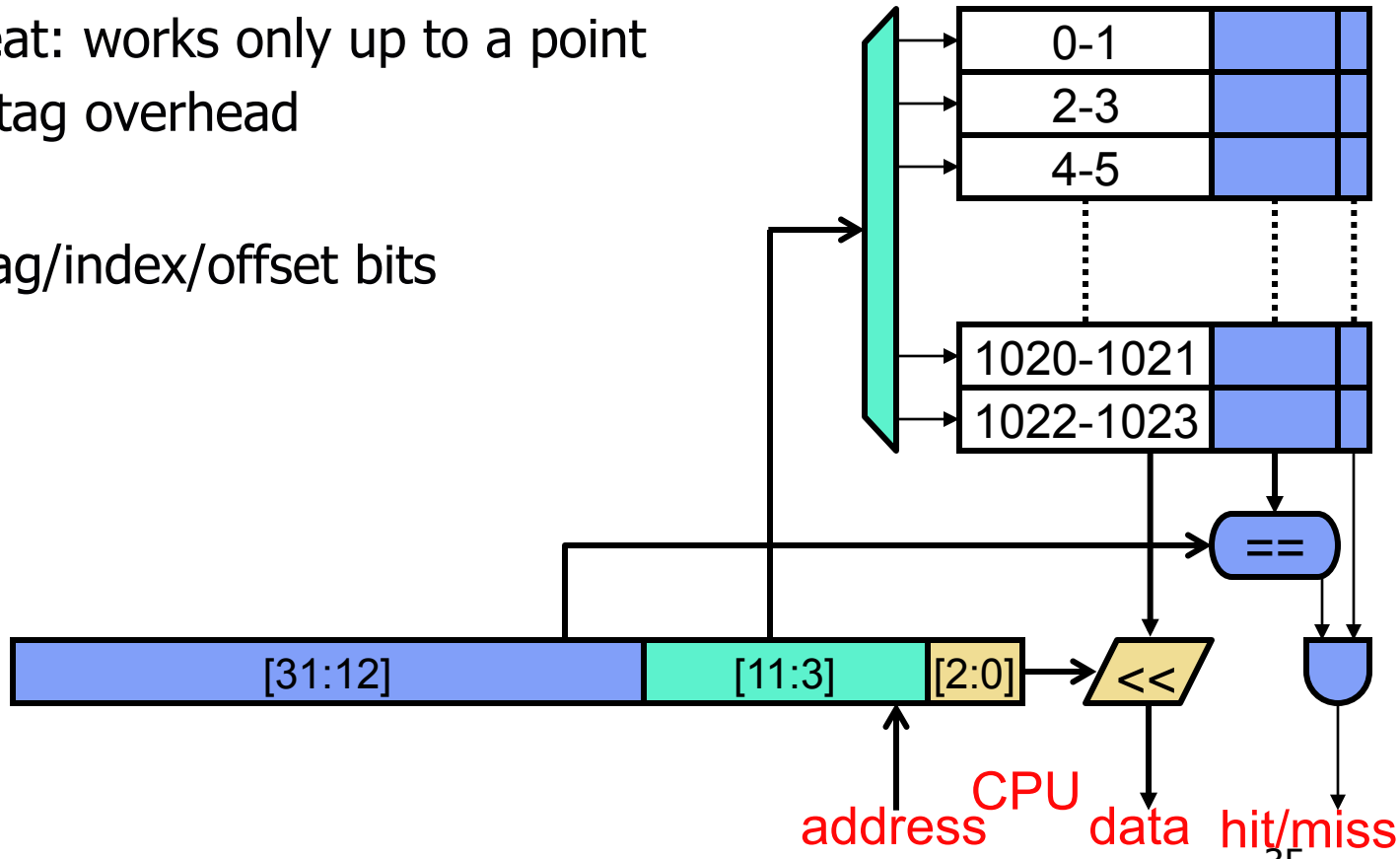
Cache Performance Simulation

- Parameters: 8-bit addresses, 32B cache, 4B blocks
  - Addresses initially in cache : 0, 4, 8, 12, 16, 20, 24, 28
  - To find location in cache, do mod32 arithmetic (why 32?)

Cache contents (prior to access)	Address	Outcome
0, 4, 8, 12, 16, 20, 24, 28	200 (200%32=8)	Miss
0, 4, **200**, 12, 16, 20, 24, 28	204 (204%32=12)	Miss
0, 4, 200, **204**, 16, 20, 24, 28	144 (144%32=16)	Miss
0, 4, 200, 204, **144**, 20, 24, 28	6	Hit
0, 4, 200, 204, 144, 20, 24, 28	8	Miss
0, 4, **8**, 204, 144, 20, 24, 28	12	Miss
0, 4, 8, **12**, 144, 20, 24, 28	20	Hit
0, 4, 8, 12, 144, 20, 24, 28	16	Miss
0, 4, 8, 12, **16**, 20, 24, 28	144	Miss
0, 4, 8, 12, **144**, 20, 24, 28	200	Miss

Block Size

- Given capacity, manipulate $\%_{miss}$ by changing organization
- One option: increase **block size**
  - + Exploit **spatial locality**
    - Caveat: works only up to a point
  - + Reduce tag overhead
  - Notice tag/index/offset bits



Calculating Tag Size

- Calculate tag overhead of 4KB cache with 512 8B frames
  - Not including valid bits
  - 8B frames → 3-bit offset
  - 512 frames → 9-bit index
  - 32-bit address – 3-bit offset – 9-bit index = 20-bit tag
  - 20-bit tag * 512 frames = 10Kb tags = 1.25KB tags

  + 32% overhead

    + Less tag overhead with larger blocks

Cache Performance Simulation

- Parameters: 8-bit addresses, 32B cache, **8B blocks**
  - Addresses in base4 ("nibble") notation
  - Initial contents : 0000(0010), 0020(0030), 0100(0110), 0120(0130)

Cache contents (prior to access)	Address	Outcome
0000(0010), 0020(0030), 0100(0110), 0120(0130)	3020	Miss
0000(0010), **3020(3030)**, 0100(0110), 0120(0130)	3030	**Hit (spatial locality!)**
0000(0010), 3020(3030), 0100(0110), 0120(0130)	2100	Miss
0000(0010), 3020(3030), **2100(2110)**, 0120(0130)	0012	Hit
0000(0010), 3020(3030), 2100(2110), 0120(0130)	0020	Miss
0000(0010), **0020(0030)**, 2100(2110), 0120(0130)	0030	**Hit (spatial locality)**
0000(0010), 0020(0030), 2100(2110), 0120(0130)	0110	**Miss (conflict)**
0000(0010), 0020(0030), **0100(0110)**, 0120(0130)	0100	**Hit (spatial locality)**
0000(0010), 0020(0030), 0100(0110), 0120(0130)	2100	Miss
0000(0010), 0020(0030), **2100(2110)**, 0120(0130)	3020	Miss

Effect of Block Size

- Increasing block size has two effects (one good, one bad)
  - + **Spatial prefetching**
    - For blocks with adjacent addresses
    - Turns miss/miss pairs into miss/hit pairs
    - Example from previous slide: 3020,3030
  - – **Conflicts**
    - For blocks with non-adjacent addresses (but adjacent frames)
    - Turns hits into misses by disallowing simultaneous residence
    - Example: 2100,0110
  - Both effects always present to some degree
  - Spatial prefetching dominates initially (until 64–128B)
  - Interference dominates afterwards
  - Optimal block size is 32–128B (varies across programs)

Conflicts

- What about pairs like 3030/0030, 0100/2100?
  - These will **conflict** in any size cache (regardless of block size)
    - Will keep generating misses
- Can we allow pairs like these to simultaneously reside?
  - Yes, but we have to reorganize cache to do so

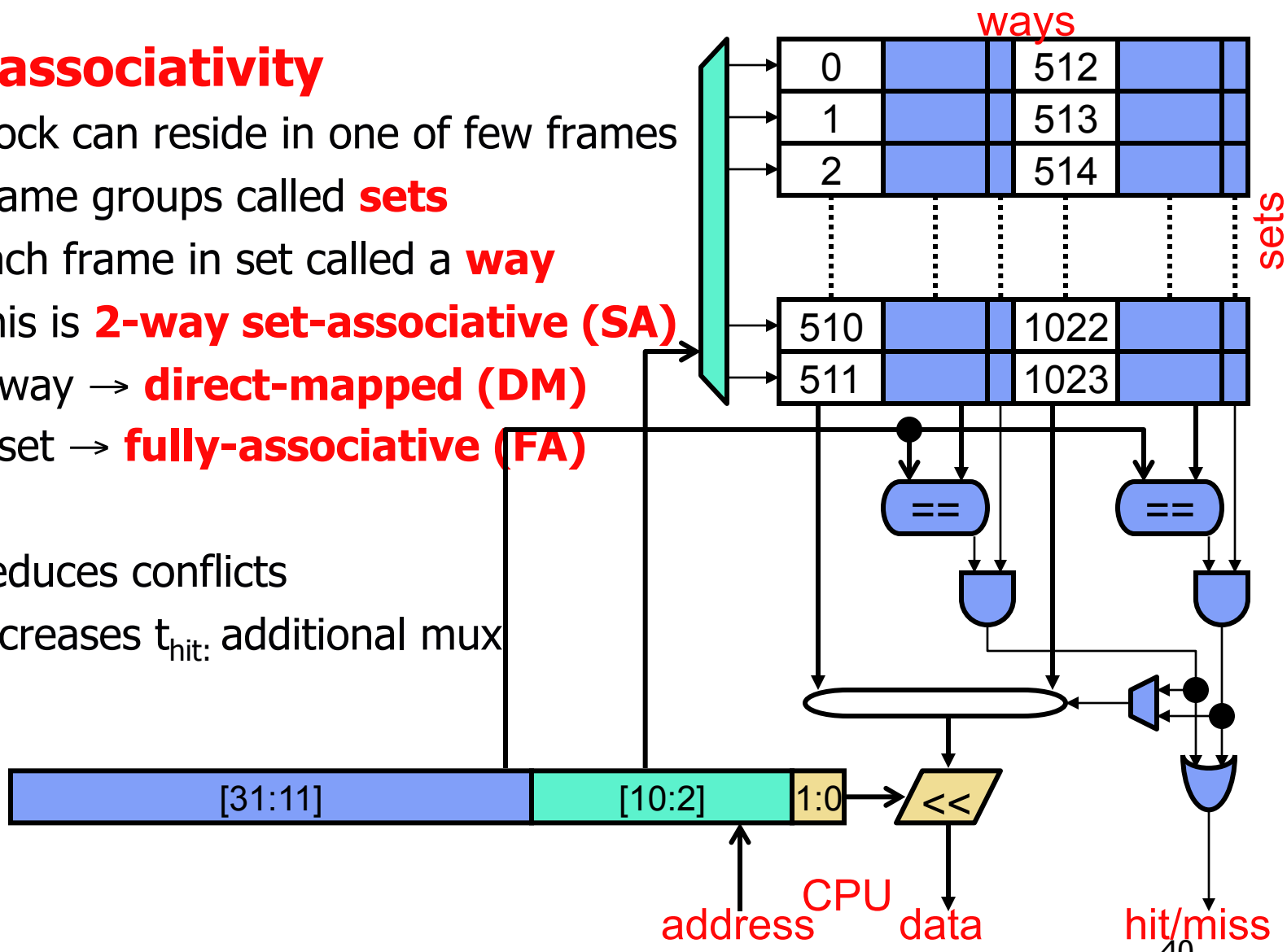
Cache contents (prior to access)	Address	Outcome
0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130	3020	Miss
0000, 0010, 3020, 0030, 0100, 0110, 0120, 0130	**3030**	Miss
0000, 0010, 3020, **3030**, 0100, 0110, 0120, 0130	2100	Miss
0000, 0010, 3020, 3030, 2100, 0110, 0120, 0130	0012	Hit
0000, 0010, 3020, 3030, 2100, 0110, 0120, 0130	0020	Miss
0000, 0010, 0020, 3030, 2100, 0110, 0120, 0130	**0030**	Miss
0000, 0010, 0020, **0030**, 2100, 0110, 0120, 0130	0110	Hit

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)**

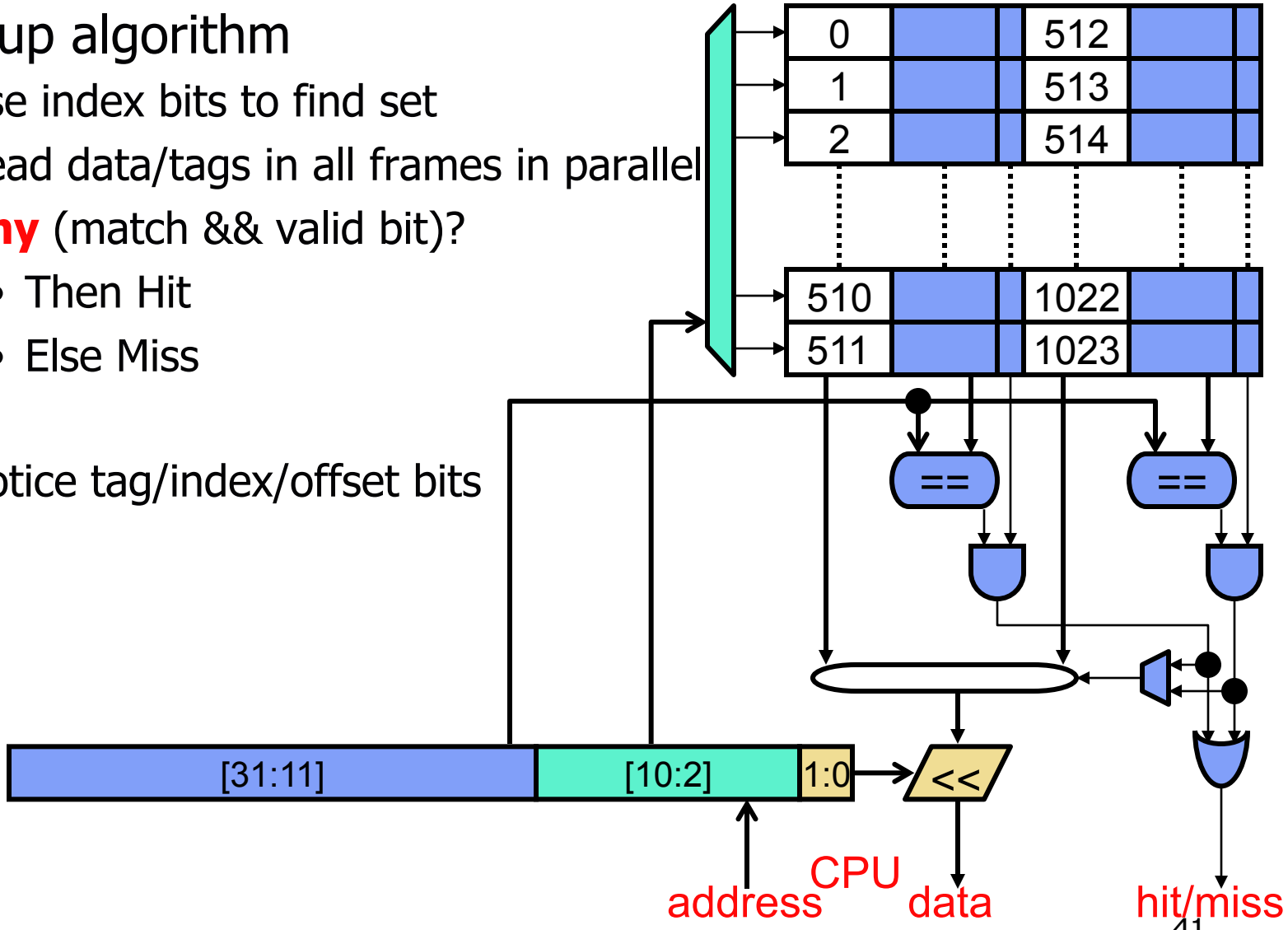
  \+ Reduces conflicts

  – 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



Cache Performance Simulation

- Parameters: 32B cache, 4B blocks, **2-way set-associative**
  - Initial contents : 0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130

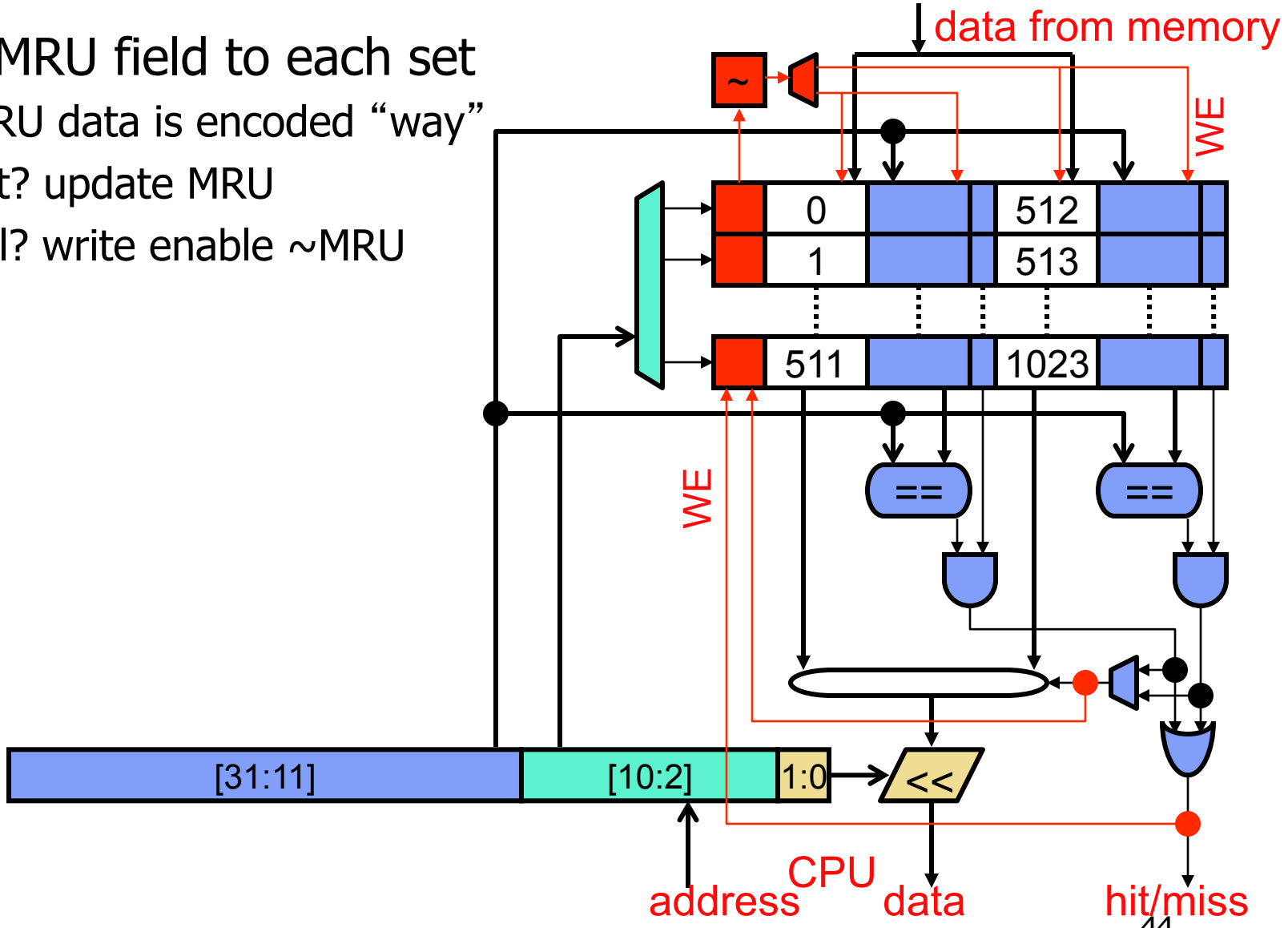
Cache contents	Address	Outcome
[0000,0100], [0010,0110], [0020,0120], [0030,0130]	3020	Miss
[0000,0100], [0010,0110], [0120,**3020**], [0030,0130]	3030	Miss
[0000,0100], [0010,0110], [0120,3020], [0130,**3030**]	2100	Miss
[0100,**2100**], [0010,0110], [0120,3020], [0130,3030]	0012	Hit
[0100,2100], [0010,0110], [0120,3020], [0130,3030]	0020	Miss
[0100,2100], [0010,0110], [3020,**0020**], [0130,3030]	0030	Miss
[0100,2100], [0010,0110], [3020,0020], [3030,**0030**]	0110	Hit
[0100,2100], [0010,0110], [3020,0020], [3030,0030]	0100	**Hit (avoid conflict)**
[2100,0100], [0010,0110], [3020,0020], [3030,0030]	2100	**Hit (avoid conflict)**
[0100,2100], [0010,0110], [3020,0020], [3030,0030]	3020	**Hit (avoid conflict)**

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**
  - **FIFO (first-in first-out)**
    - When is this a good idea?
  - **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
  - **Belady's**: replace block that will be used furthest in future
    - Unachievable optimum (but good for comparisons)
  - Which policy is simulated in previous slide?

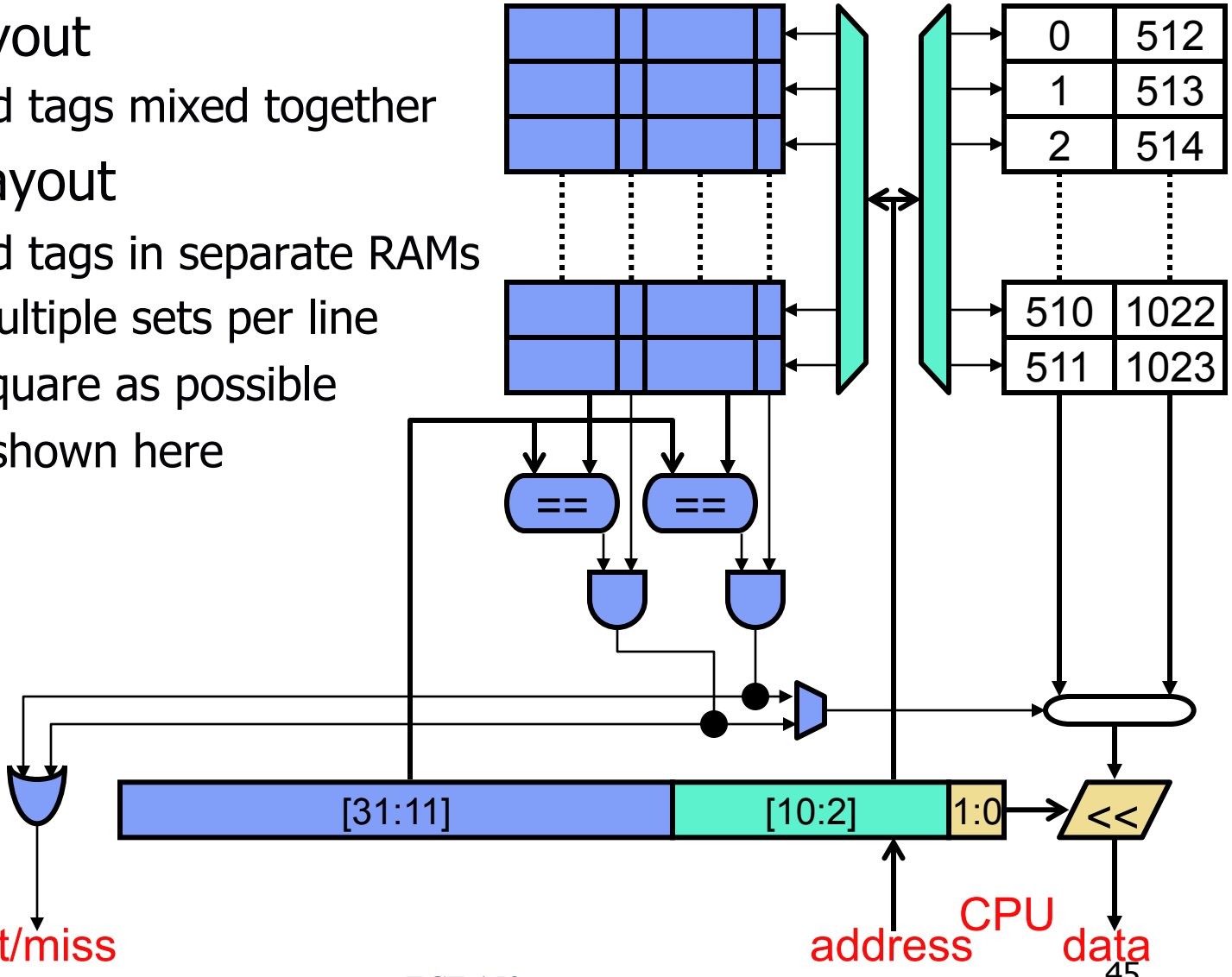
NMRU and Miss Handling

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

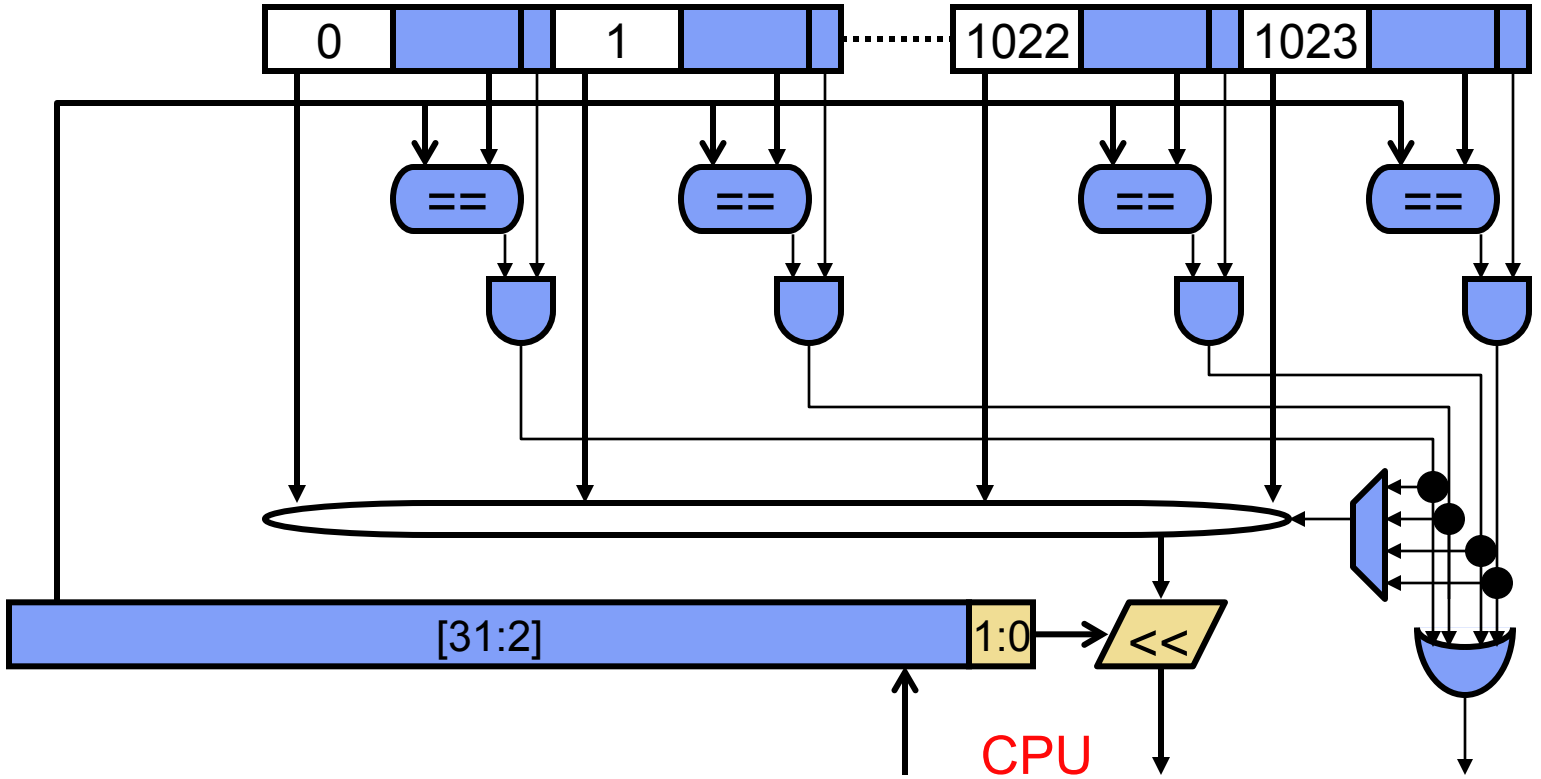


Physical Cache Layout

- 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



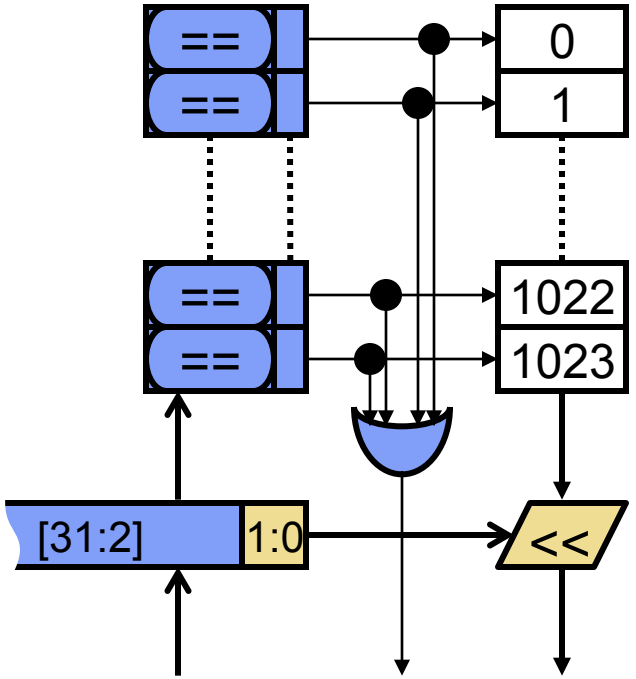
Full-Associativity



- 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

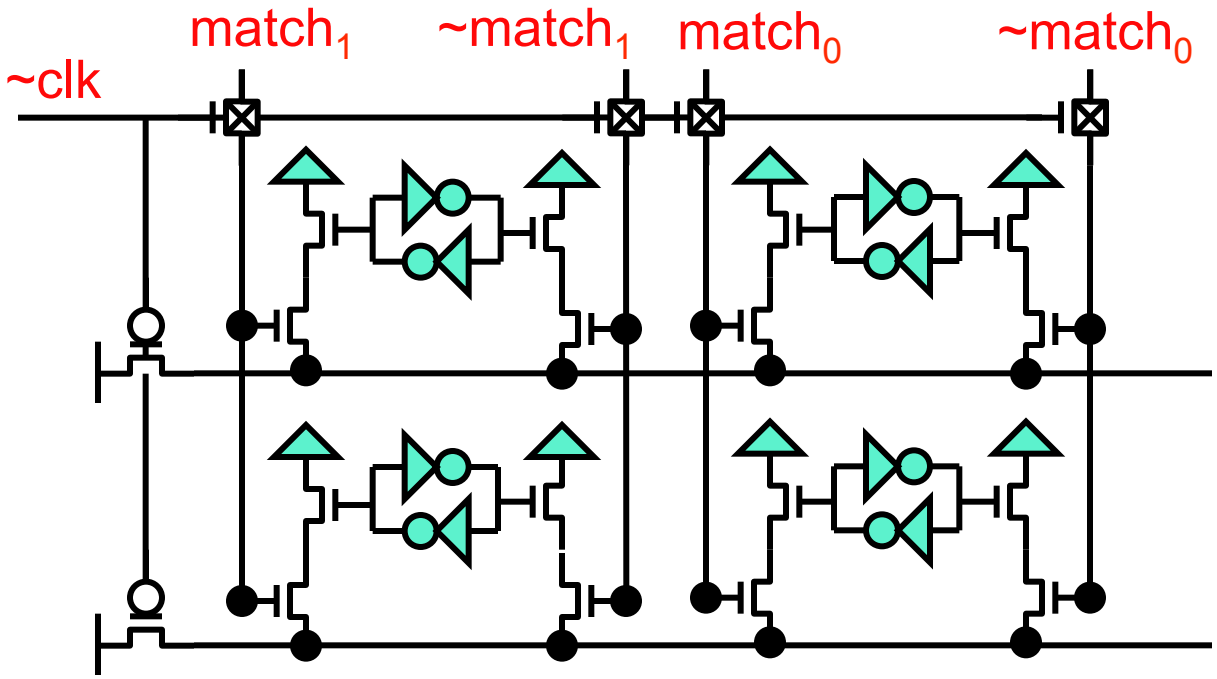
- **CAM**: content addressable memory
  - Array of words with built-in comparators
  - Matchlines instead of bitlines
  - Output is "one-hot" encoding of match



- FA cache?
  - Tags as CAM
  - Data as RAM

- **Hardware is not software**
  - Example I: parallel computation with carry select adder
  - Example II: parallel search with CAM
    - No such thing as software CAM

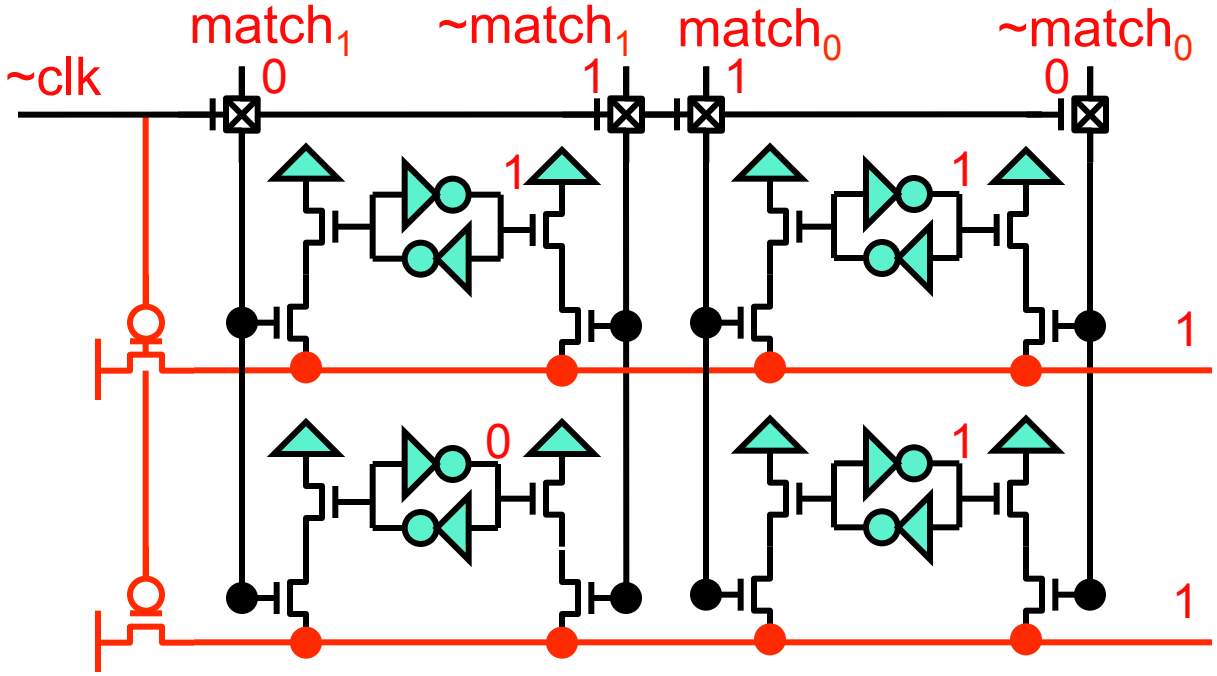
CAM Circuit



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

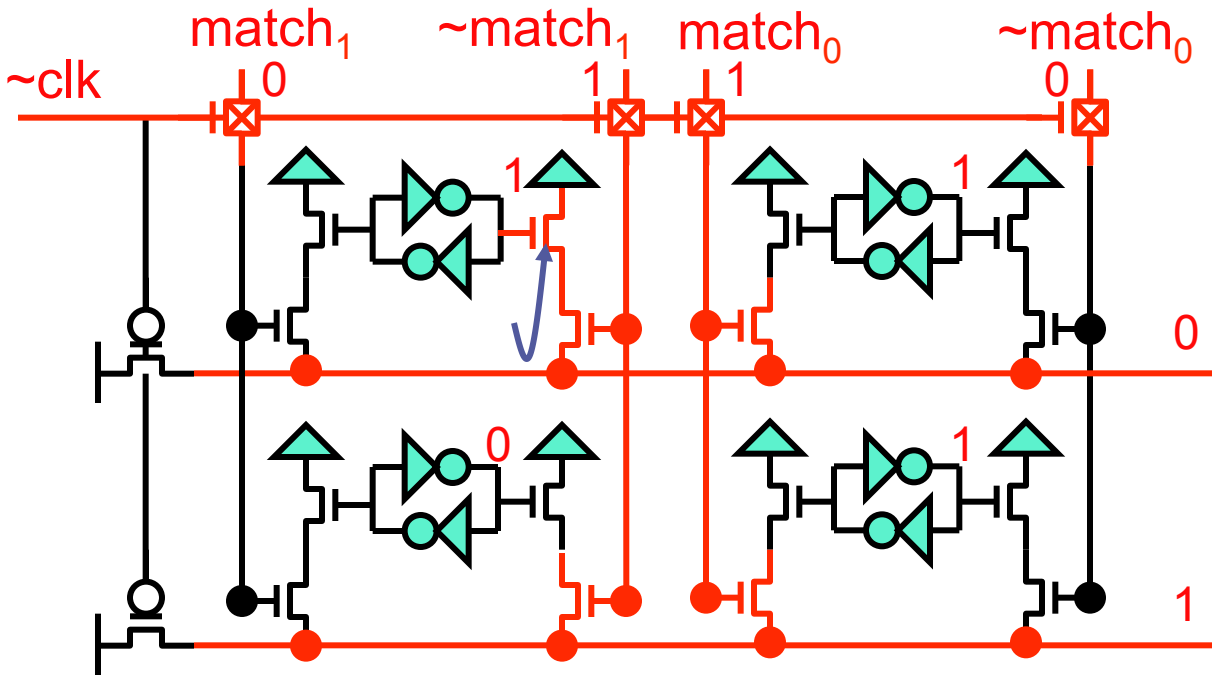
- Two phase match
  - Phase I: clk=1, pre-charge wordlines to 1
  - Phase II: clk=0, enable matchlines, non-matched bits dis-charge wordlines

CAM Circuit In Action



- 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 are flipped)
  - Any non-matching bit discharges entire wordline
    - Implicitly ANDs all bit matches (NORs all bit non-matches)
  - Similar technique for doing a fast OR for hit detection

CAM Upshot

- CAMs are effective but expensive
  - Matchlines are very expensive (for nasty circuit-level reasons)
  - 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

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
    - Consecutive accesses to block separated by accesses to at least N other distinct blocks where N is number of frames in cache
    - Misses that would occur in fully-associative cache of given size
  - **Conflict**: miss caused because cache associativity is too low
    - All other misses

Cache Performance Simulation

- Parameters: 8-bit addresses, 32B cache, 4B blocks
  - Initial contents : 0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130
  - Initial blocks accessed in increasing order

Cache contents	Address	Outcome
0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130	3020	Miss (compulsory)
0000, 0010, **3020**, 0030, 0100, 0110, 0120, 0130	3030	Miss (compulsory)
0000, 0010, 3020, **3030**, 0100, 0110, 0120, 0130	2100	Miss (compulsory)
0000, 0010, 3020, 3030, **2100**, 0110, 0120, 0130	0012	Hit
0000, 0010, 3020, 3030, 2100, 0110, 0120, 0130	0020	Miss (capacity)
0000, 0010, **0020**, 3030, 2100, 0110, 0120, 0130	0030	Miss (capacity)
0000, 0010, 0020, **0030**, 2100, 0110, 0120, 0130	0110	Hit
0000, 0010, 0020, 0030, 2100, 0110, 0120, 0130	0100	Miss (capacity)
0000, 1010, 0020, 0030, **0100**, 0110, 0120, 0130	2100	Miss (conflict)
1000, 1010, 0020, 0030, **2100**, 0110, 0120, 0130	3020	Miss (capacity)

ABC

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

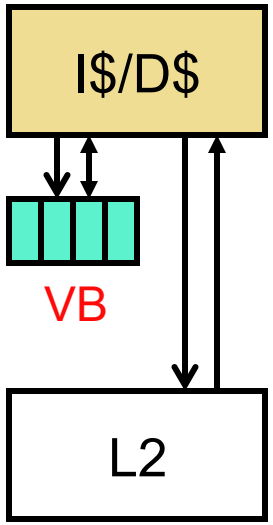
Two (of many possible) Optimizations

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

Victim Buffer

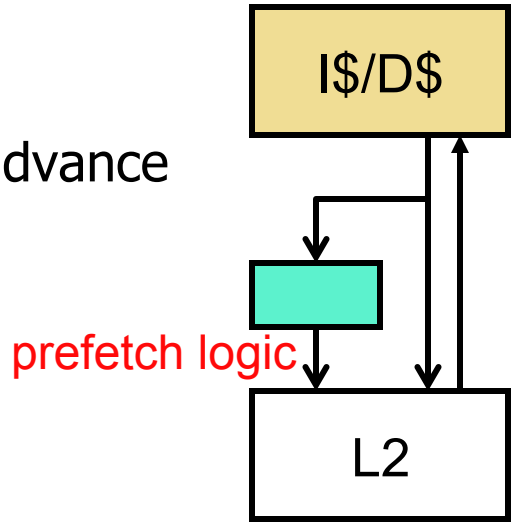
- Conflict misses: not enough associativity
  - High-associativity is expensive, but also rarely needed
    - 3 blocks mapping to same 2-way set and accessed (ABC)*

- **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



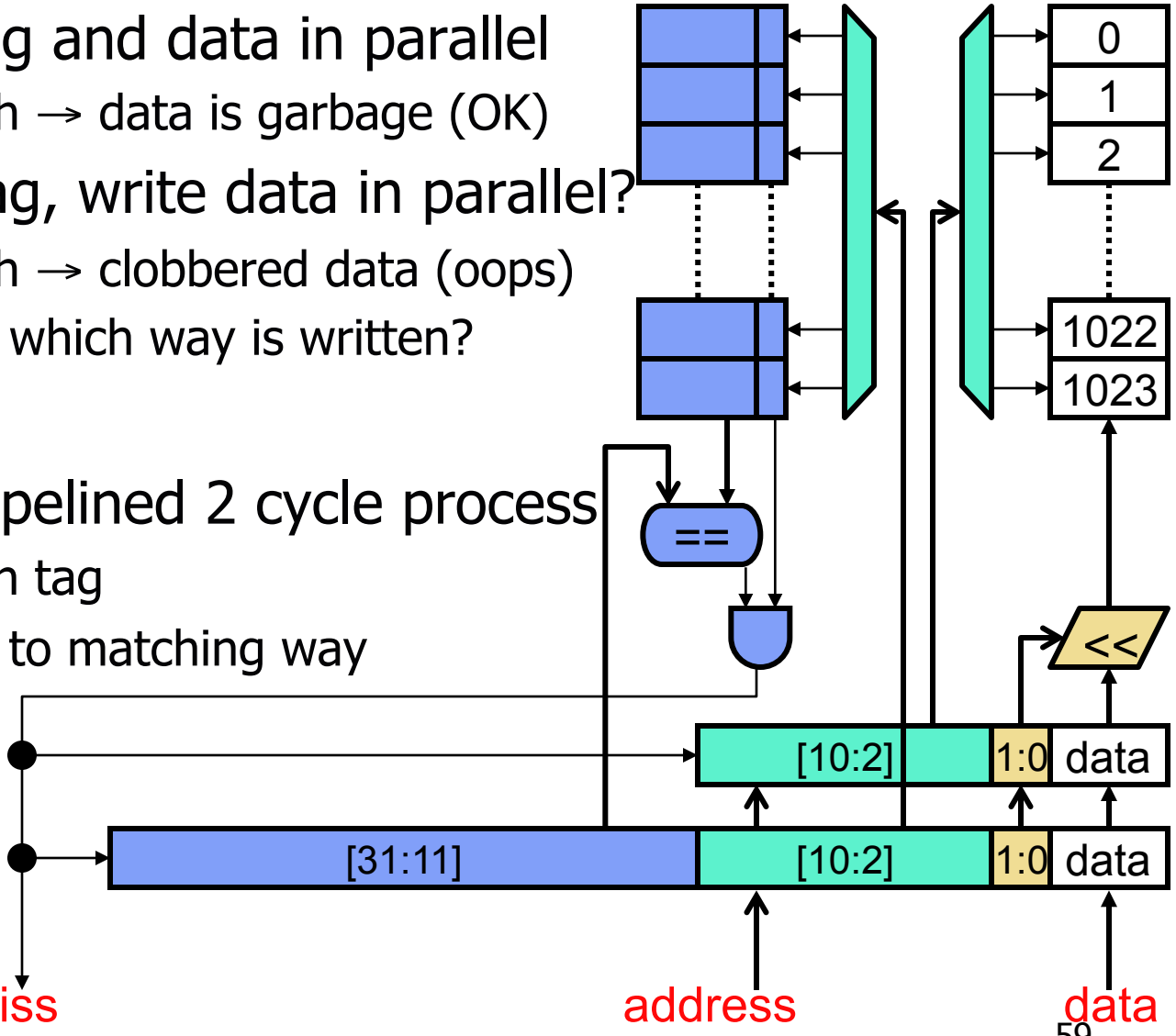
Write Issues

- So far we have looked at reading from cache (loads)
- What about writing into cache (stores)?

- Several new issues
  - Tag/data access
  - Write-through vs. write-back
  - Write-allocate vs. write-not-allocate

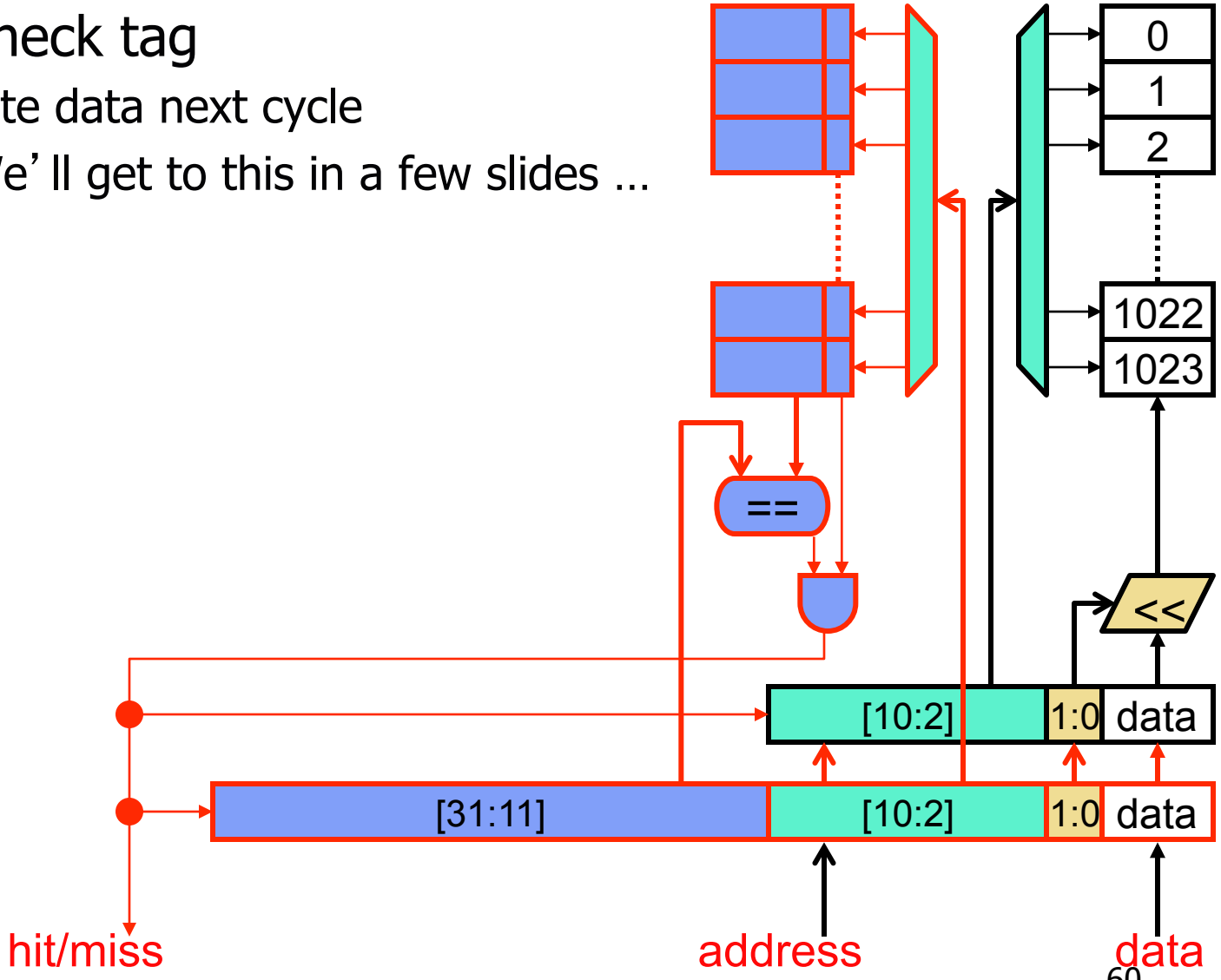
Tag/Data Access

- 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



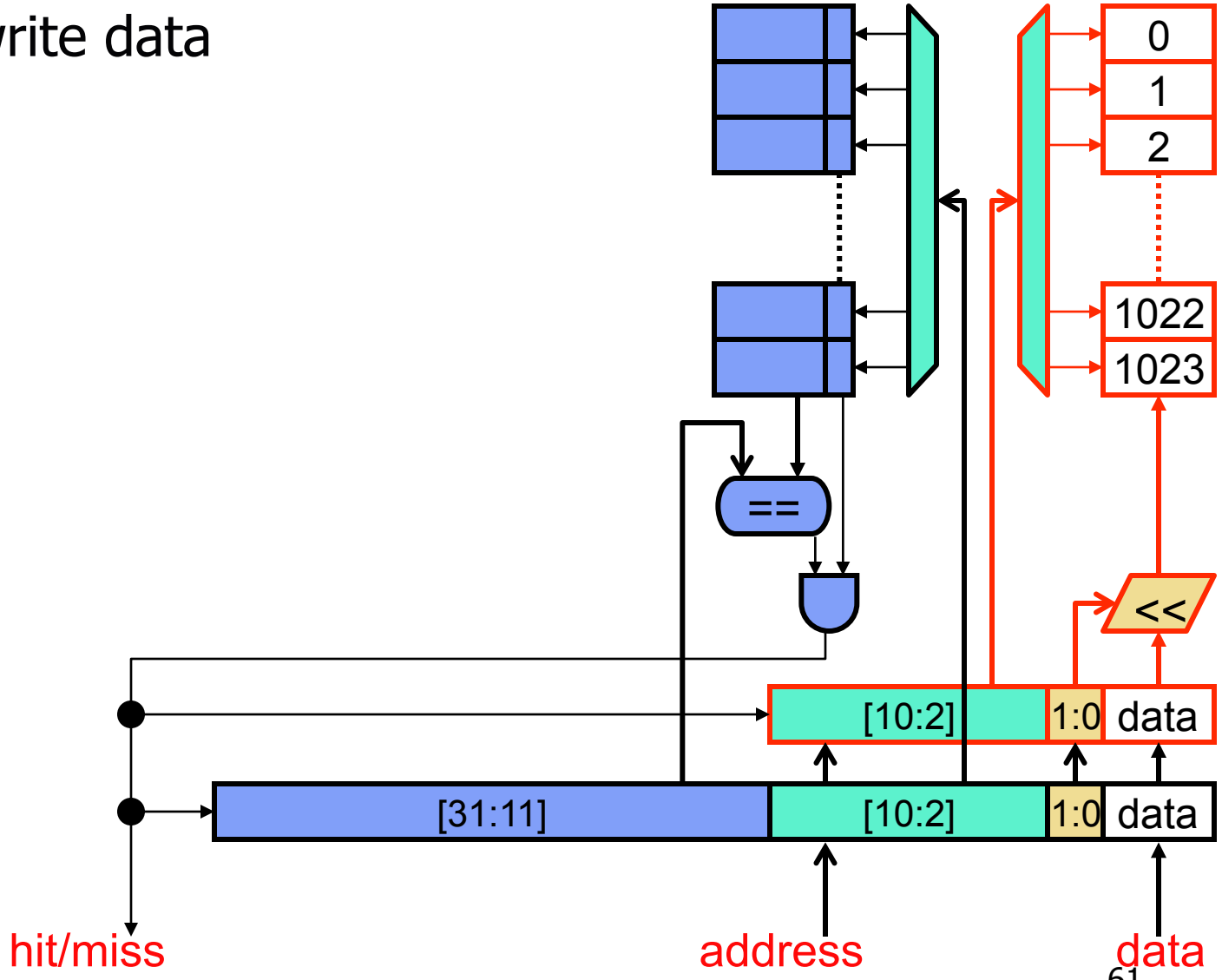
Tag/Data Access

- Cycle 1: check tag
  - Hit? Write data next cycle
  - Miss? We'll get to this in a few slides …



Tag/Data Access

- Cycle 2: write data



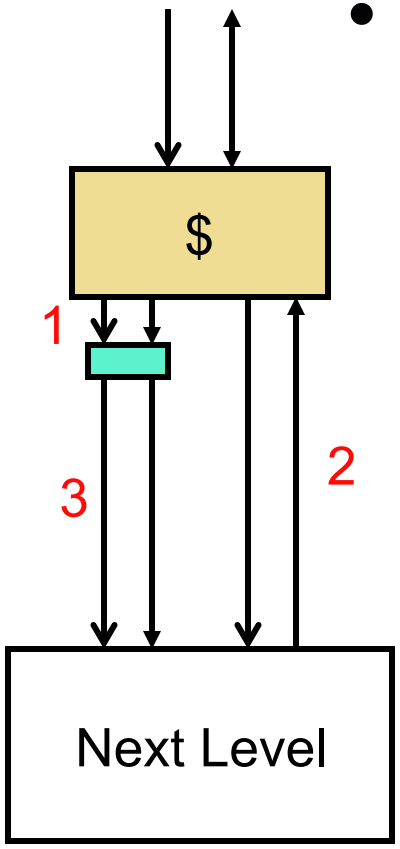
Write-Through vs. Write-Back

- When to propagate new value to (lower level) memory?
  - **Write-through**: immediately
    - \+ Conceptually simpler
    - \+ Uniform latency on misses
    - – Requires additional bus bandwidth
  - **Write-back**: when block is replaced
    - Requires additional "dirty" bit per block
    - \+ Minimal bus bandwidth
      - Only write back dirty blocks
    - – Non-uniform miss latency
      - Clean miss: one transaction with lower level (fill)
      - Dirty miss: two transactions (writeback & fill)

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

Write Buffer



- **Write buffer**: between cache and memory
- Write-through cache? Helps with store misses
  - + Write to buffer to avoid waiting for memory
    - 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 memory to cache
    3. Write buffer contents to memory

Typical Processor Cache Hierarchy

- First level caches: optimized for t_{hit} and parallel access
  - Insns and data in separate caches (**I$**, **D$**)
  - 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: 1–8MB
  - 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
- 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

Summary

- Average access time of a memory component
  - $t_{avg} = t_{hit} + \%_{miss} * t_{miss}$
  - Hard to get low t_{hit} and $\%_{miss}$ in one structure → hierarchy
- Memory hierarchy
  - Cache (SRAM) → memory (DRAM) → swap (Disk)
  - Smaller, faster, more expensive → bigger, slower, cheaper
- SRAM
  - Analog technology for implementing big storage arrays
  - Cross-coupled inverters + bitlines + wordlines
  - Delay ~ √#bits * #ports

Summary, cont'd

- Cache ABCs
  - Capacity, associativity, block size
  - 3C miss model: compulsory, capacity, conflict
- Some optimizations
  - Victim buffer for conflict misses
  - Prefetching for capacity, compulsory misses
- Write issues
  - Pipelined tag/data access
  - Write-back vs. write-through/write-allocate vs. write-no-allocate
  - Write buffer

Next Course Unit: Main Memory