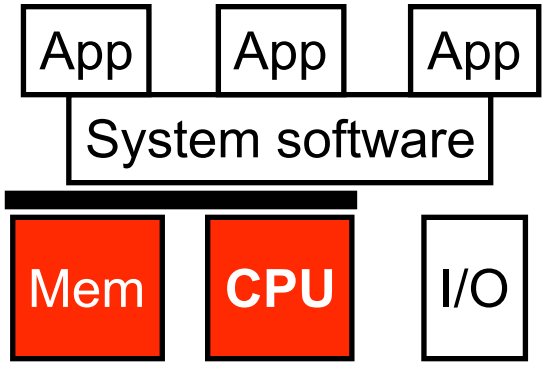
ECE 550
Fundamentals of Computer Systems and Engineering

Memory Hierarchy

Memory Hierarchy

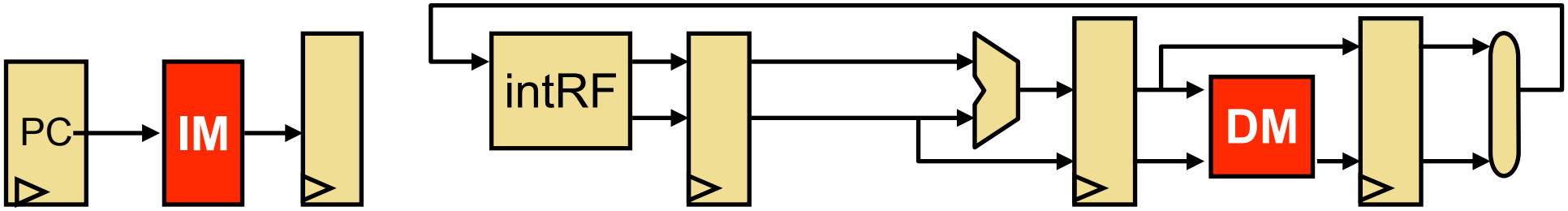


- Basic concepts
- Technology background
- Organizing a single memory component
  - ABC
  - Write issues
  - Miss classification and optimization
- Organizing an entire memory hierarchy
- Virtual memory
  - Highly integrated into real hierarchies, but…
  - …won't talk about until later

Admin

- Homework 3: Due Oct 18
- Homework 4
  - Back to VHDL
  - Put a datapath together
- Midterm: Nov 1
- Reading: Ch 5

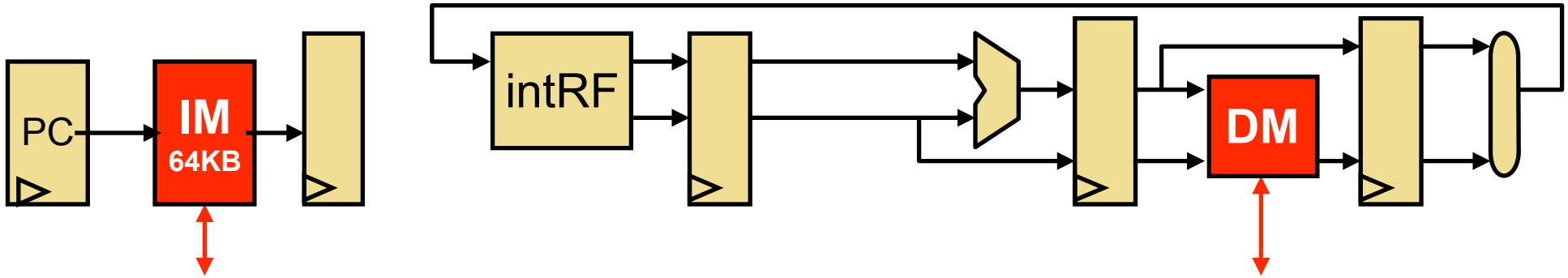
How Do We Build Insn/Data Memory?



- Register file? Just a multi-ported SRAM
  - 32 32-bit registers → 1Kb = 128B
  - Multiple ports make it bigger and slower but still OK

- Insn/data memory? Just a single-ported SRAM?
  - Uh, umm… **it's 2^{32}B = 4GB!!!!**
  - It would be huge, expensive, and pointlessly slow
  - And we can't build something that big on-chip anyway
  - Good news: most ISAs now 64-bit → memory is 2^{64}B = 16EB

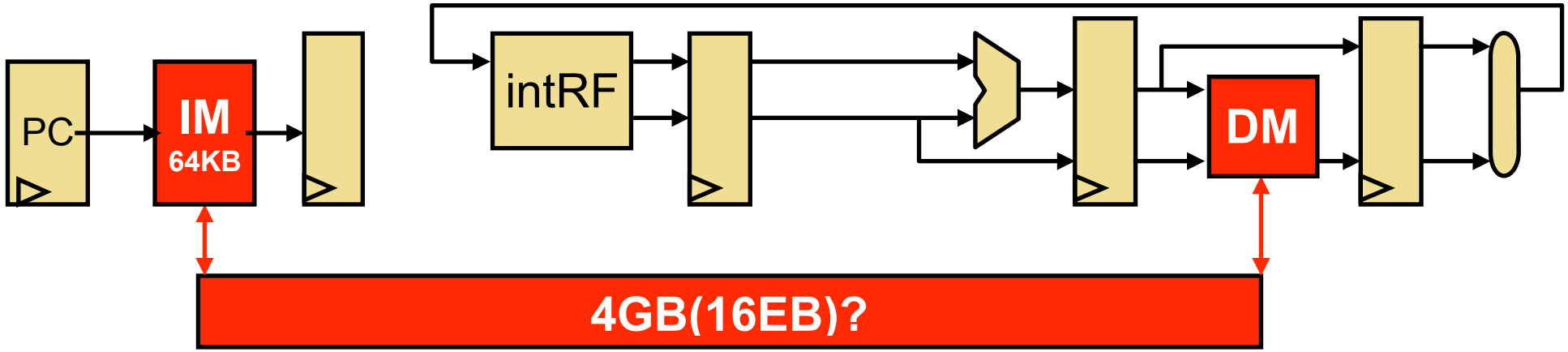
So What Do We Do? Actually…



- "Primary" insn/data memory are single-ported SRAMs…
  - "primary" = "in the datapath"
  - Key 1: they contain only a dynamic subset of "memory"
    - Subset is small enough to fit in a reasonable SRAM
  - Key 2: missing chunks fetched on demand (transparent to program)
    - From somewhere else… (next slide)

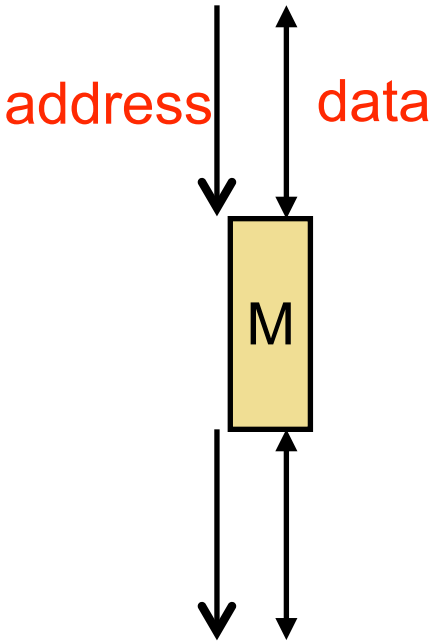
  - Program has illusion that all 4GB (16EB) of memory is physically there
  - Just like it has the illusion that all insns execute atomically

But…



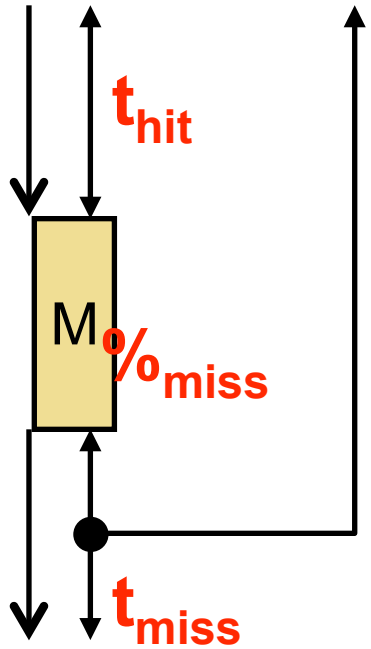
- If requested insn/data not found in primary memory
  - Doesn't the place it comes from have to be a 4GB (16EB) SRAM?
  - And won't it be huge, expensive, and slow? And can we build it?

Memory Overview



- Functionality
  - "Like a big array…"
  - N-bit **address** bus (on N-bit machine)
  - **Data** bus: typically read/write on same bus
  - Can have multiple **ports**: address/data bus pairs
- Access time:
  - Access latency ~ **#bits * #ports^2**

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 component
    - No notion of "miss" in register file
  - **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

- Performance metric: average access time

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

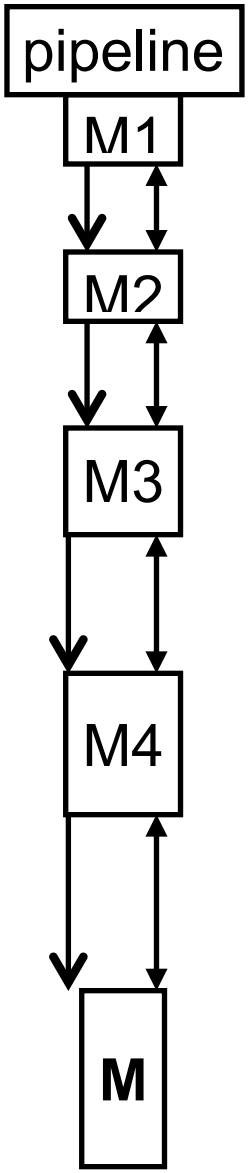
Memory Hierarchy

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

- 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
  - Known from the very beginning

> "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,VonNeumann
>
> "Preliminary discussion of the logical design of an electronic computing instrument"
>
> IAS memo **1946**

Abstract Memory Hierarchy

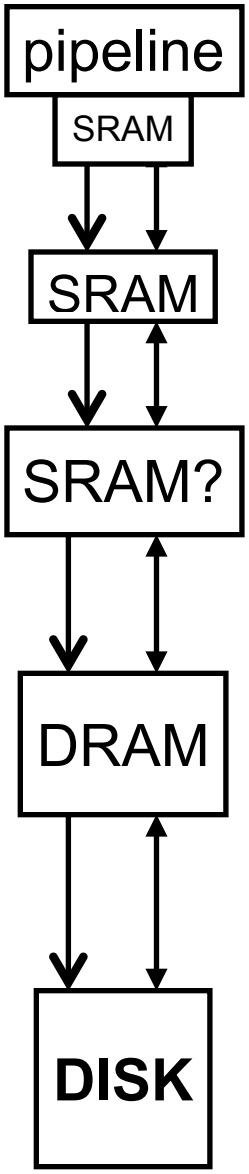


- Hierarchy of memory components
  - Upper levels: small → low t_{hit}, high $\%_{miss}$
  - Going down: larger → higher t_{hit}, lower $\%_{miss}$
- Connected by buses
  - Ignore for the moment
- Make average access time close to M1's
  - How?
  - Most frequently accessed data in M1
  - M1 + next most frequently accessed in M2, etc.
  - **Automatically** move data up-down hierarchy

Why Memory Hierarchy Works

- **10/90 rule (of thumb)**
  - 10% of static insns/data account for 90% of accessed insns/data
    - Insns: inner loops
    - Data: frequently used globals, inner loop stack variables
- **Temporal locality**
  - Recently accessed insns/data likely to be accessed again soon
    - Insns: inner loops (next iteration)
    - Data: inner loop local variables, globals
  - Hierarchy can be **"reactive"**: move things up when accessed
- **Spatial locality**
  - Insns/data near recently accessed insns/data likely accessed soon
    - Insns: sequential execution
    - Data: elements in array, fields in struct, variables in stack frame
  - Hierarchy is **"proactive"**: move things up speculatively

Exploiting Heterogeneous Technologies

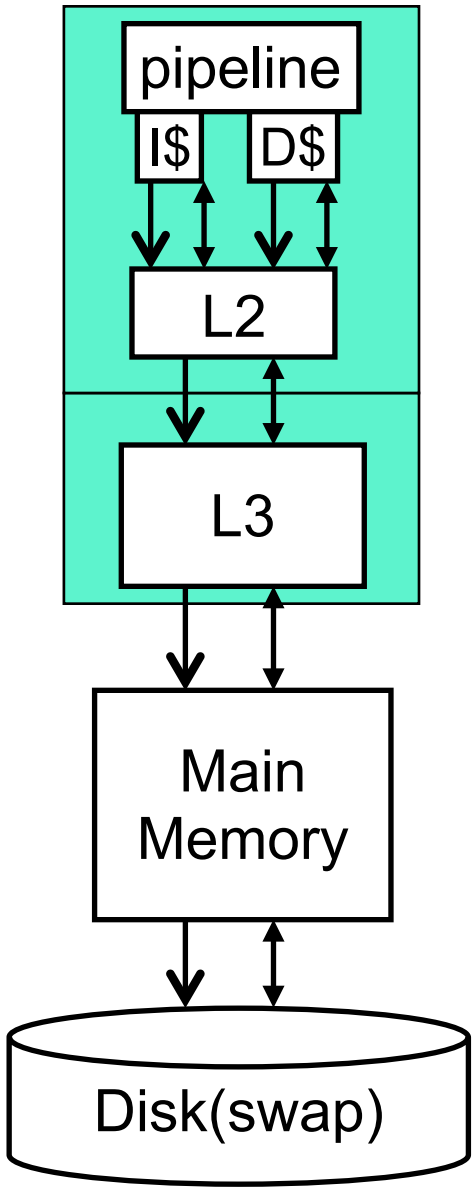


- Apparent problem
  - Lower level components must be huge
  - Huge SRAMs are difficult to build and expensive
- Solution: don't use SRAM for lower levels
  - **Cheaper, denser storage technologies**
  - Will be slower than SRAM, but that's OK
    - Won't be accessed very frequently
    - We have no choice anyway
  - Upper levels: SRAM → expensive (/B), fast
  - Going down: DRAM, Disk → cheaper (/B), fast

Memory Technology Overview

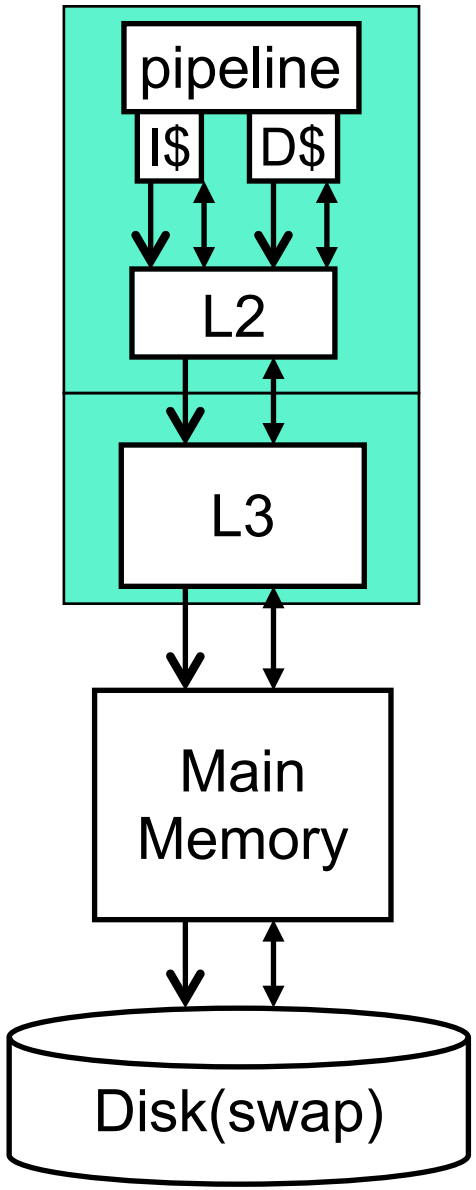
- **Latency**
  - SRAM: <1 to 5ns (on chip)
  - DRAM: ~100ns — 100x or more slower than SRAM
  - Disk: 10,000,000ns or 10ms — 100,000x slower than DRAM
  - Flash: ~200ns — 2x slower than SRAM (read, much slower writes)
- **Bandwidth**
  - SRAM: 10-100GB/sec
  - DRAM: ~1GB/sec — 10x less than SRAM
  - Disk: 100MB/sec (0.1 GB/sec) — sequential access only
  - Flash: about same as DRAM for read (much less for writes)
- **Cost**: what can $300 buy today?
  - SRAM: 4MB
  - DRAM: 1,000MB (1GB) — 250x cheaper than SRAM
  - Disk: 400,000MB (400GB) — 400x cheaper than DRAM
  - Flash: 4,000 MB (4GB) — 4x cheaper than DRAM

(Traditional) Concrete Memory Hierarchy



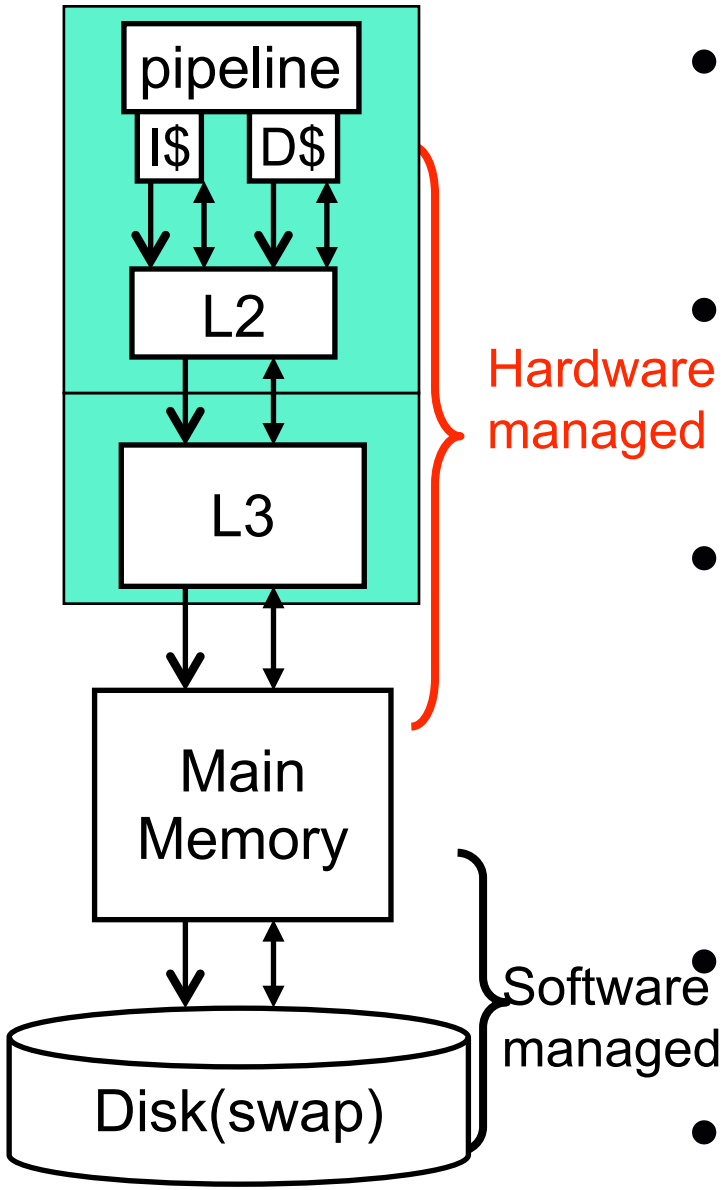
- 1st level: **I$**, **D$** (insn/data caches)
- 2nd level: **L2** (cache)
  - On-chip, certainly on-package (with CPU)
  - Made of SRAM (embedded DRAM?)
- 3rd level: **L3** (cache)
  - Same as L2, may be off-chip
  - Starting to appear
- N-1 level: **main memory**
  - Off-chip
  - Made of DRAM
- N level: **disk (swap space)**
  - Electrical-mechanical

Virtual Memory Teaser



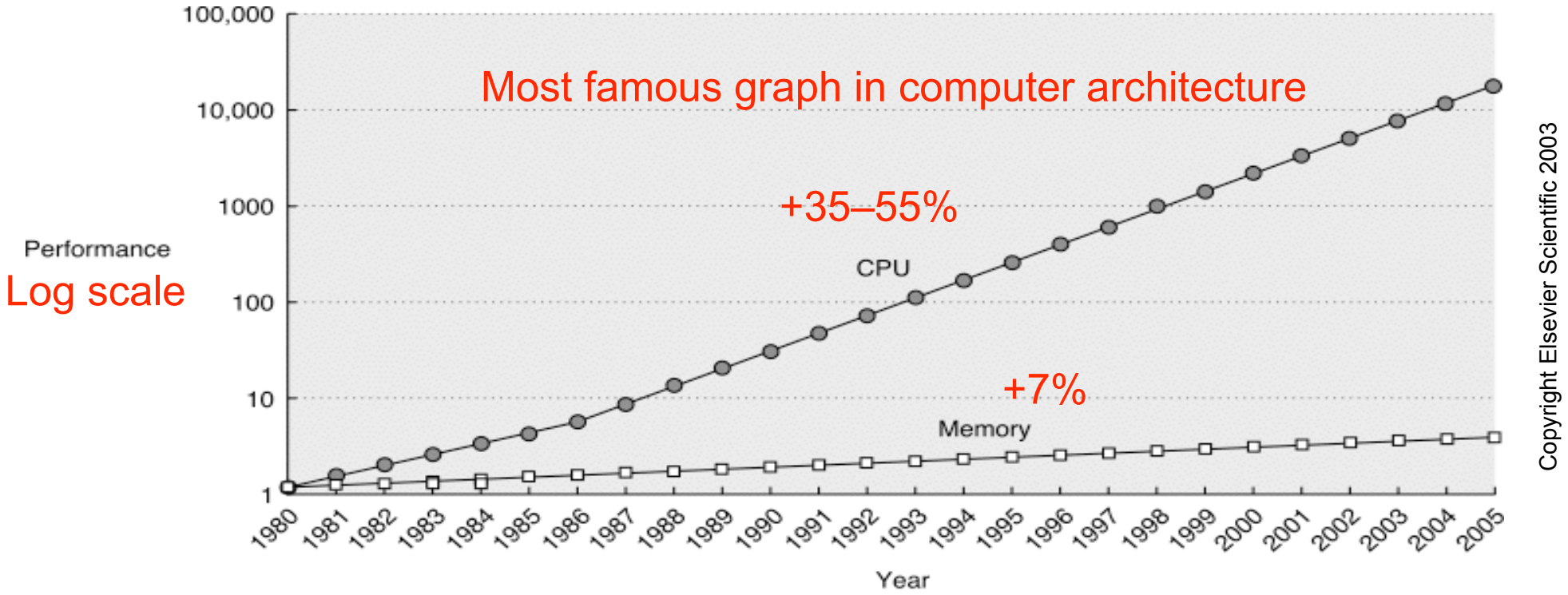
- For 32-bit ISA
  - 4GB disk is easy
  - Even 4GB main memory is common
- For 64-bit ISA
  - 16EB main memory is right out
  - Even 16EB disk is extremely difficult
- Virtual memory
  - Never referenced addresses don't have to physically exist anywhere!
  - Next week…

Start With "Caches"



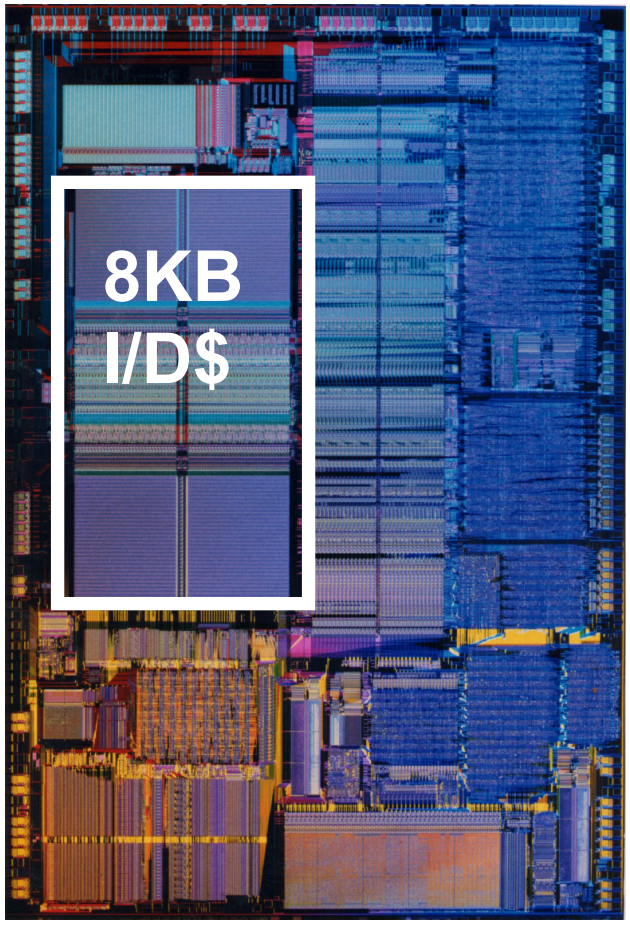
- "Cache": hardware managed
  - Missing chunks retrieved by hardware
- SRAM technology
  - Technology basis of latency
- Cache organization
  - ABC
  - Miss classification & optimization
  - What about writes?
- Cache hierarchy organization
- Some example calculations

Why Are There 2-3 Levels of Cache?

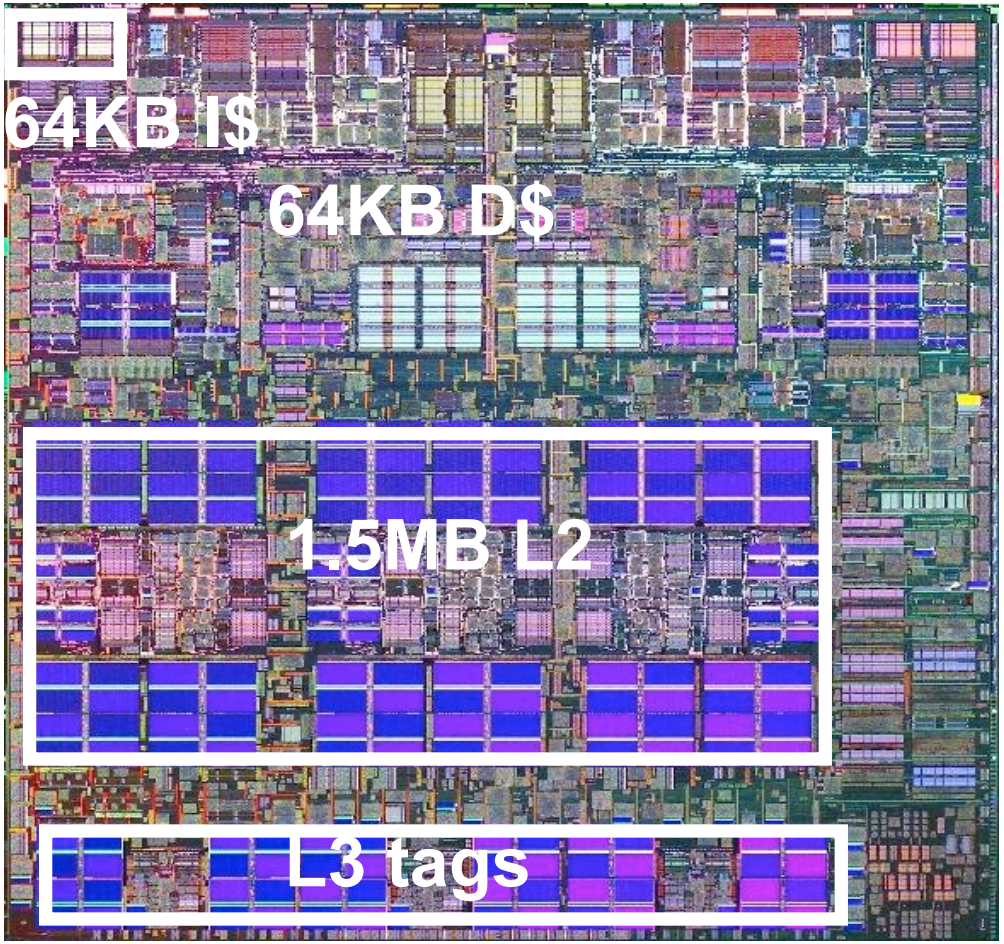


- **"Memory Wall"**: memory 100X slower than primary caches
  - Multiple levels of cache needed to bridge the difference
- **"Disk Wall?"**: disk is 100,000X slower than memory
  - Why aren't there 56 levels of main memory to bridge that difference?
  - Doesn't matter: program can't keep itself busy for 10M cycles
  - So slow, may as well swap out and run another program

Evolution of Cache Hierarchies



Intel 486



IBM Power5 (dual core)

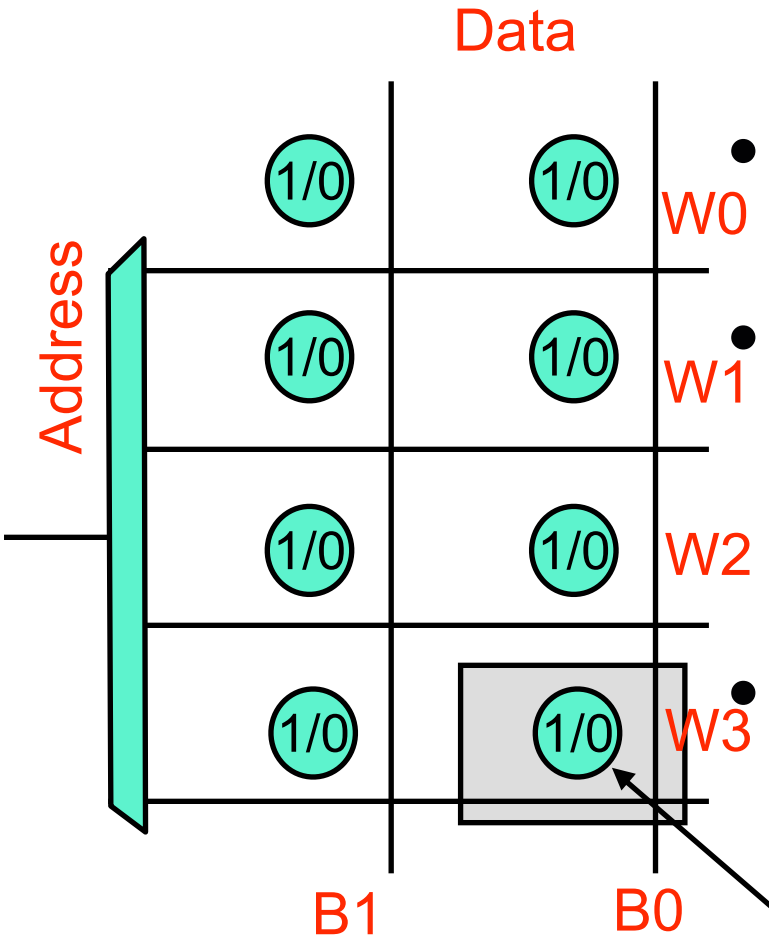
- Chips today are 30–70% cache by area

RAM and SRAM

- Reality: large storage arrays implemented in "analog" way
  - Not as flip-flops (FFs) + giant muxes
- **RAM (random access memory)**
  - Ports implemented as shared buses called wordlines/bitlines
- **SRAM**: **static RAM**
  - Static = bit maintains its value indefinitely, as long as power is on
  - Bits implemented as cross-coupled inverters (CCIs)

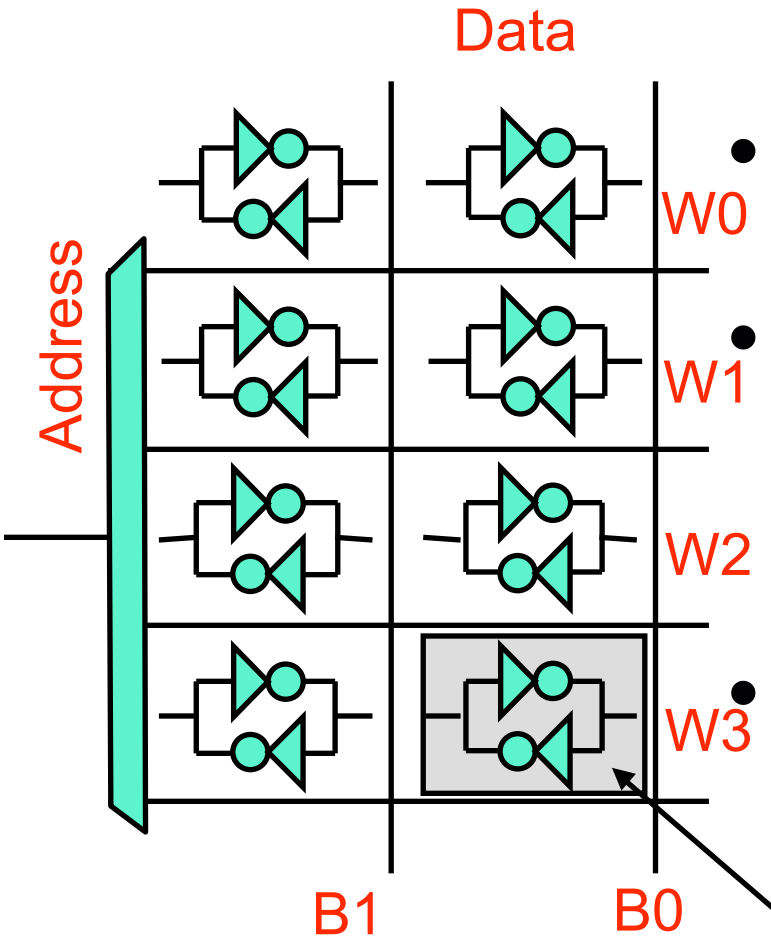
  + 2 gates, 4 transistors per bit
  - All processor storage arrays: regfile, caches, branch predictor, etc.
- Other forms of RAM: Dynamic RAM (DRAM), Flash

Basic RAM



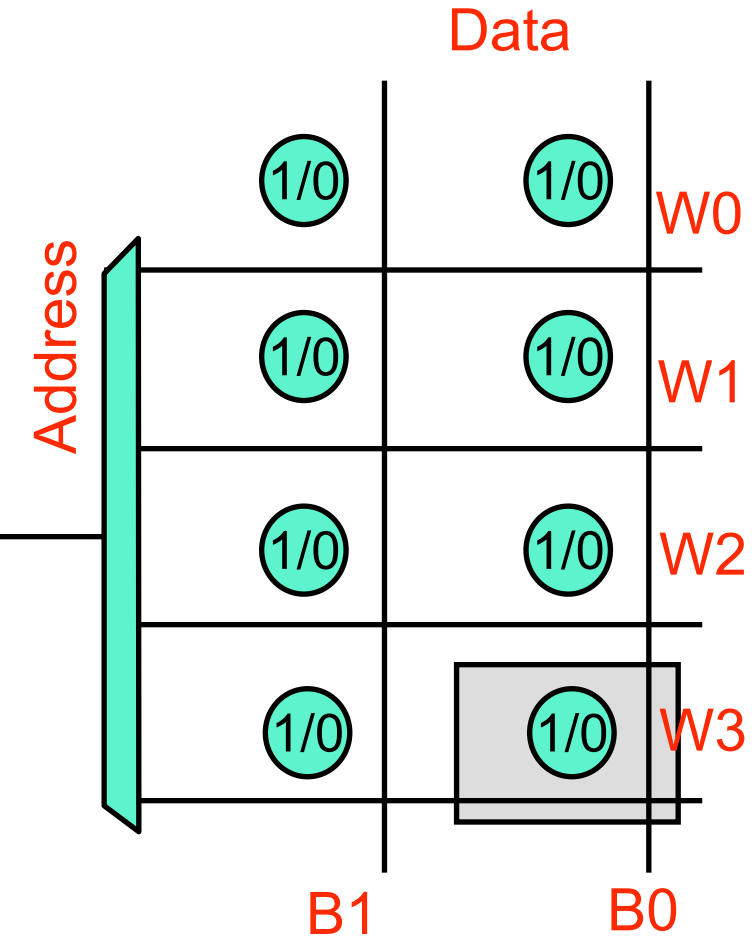
- Storage array
  - M words of N bits each (e.g., 4w, 2b each)
- RAM storage array
  - M by N array of "bits" (e.g., 4 by 2)
- RAM port
  - Grid of wires that overlays bit array
  - **M wordlines**: carry 1H decoded address
  - **N bitlines**: carry data
- RAM port operation
  - Send address → 1 wordline goes high
  - "bits" on this line read/write bitline data
  - Operation depends on bit/W/B connection
    - "Magic" analog stuff

Basic SRAM



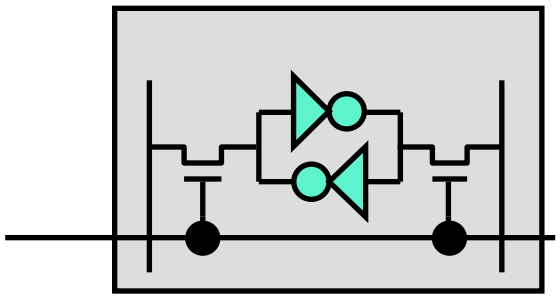
- Storage array
  - M words of N bits each (e.g., 4w, 2b each)
- SRAM storage array
  - M by N array of CCI's (e.g., 4 by 2)
- SRAM port
  - Grid of wires that overlays CCI array
  - **M wordlines**: carry 1H decoded address
  - **N bitlines**: carry data
- SRAM port operation
  - Send address → 1 wordline goes high
  - CCIs on this line read/write bitline data
  - Operation depends on CCI/W/B connection
    - "Magic" analog stuff

ROMS:



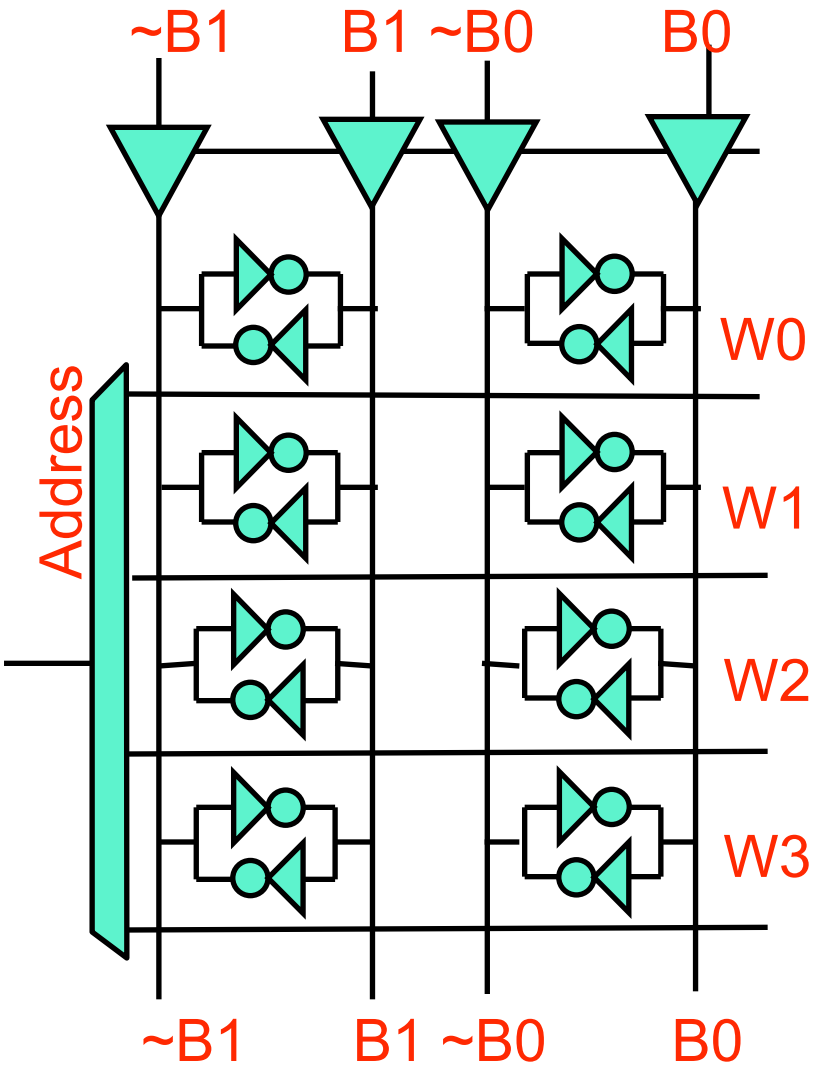
- ROMs = Read Only memory
- Similar layout (wordlines, bitlines) to RAMs
  - Except not writeable: fixed connections to Power/Gnd instead of CCI
- Also EPROMs
  - Programmable once electronically
- And EEPROMs
  - Eraseable and re-programable (very slow)

SRAM Read/Write Port



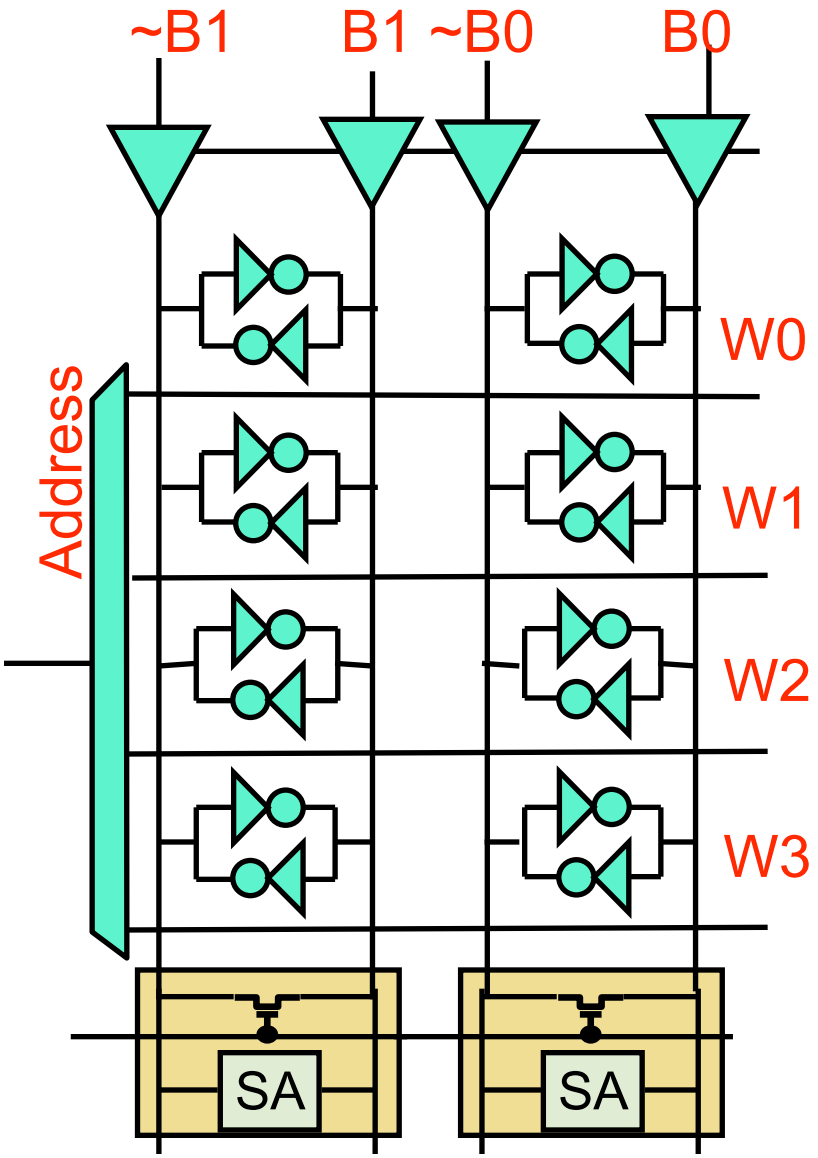
- Cache: read/write on same port
  - Not at the same time
  - Trick: write port with additional bitline
    - "Double-ended" or "differential" bitlines
  - Smaller → faster than separate ports

SRAM Read/Write



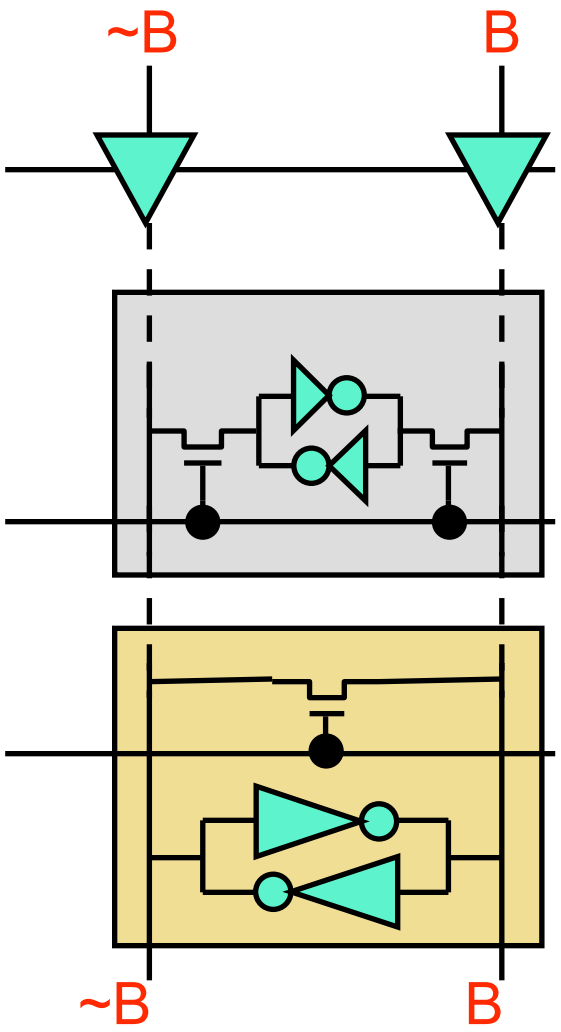
- Some extra logic on the edges
  - To write: tristates "at the top"
  - Drive write data when appropriate

SRAM Read/Write



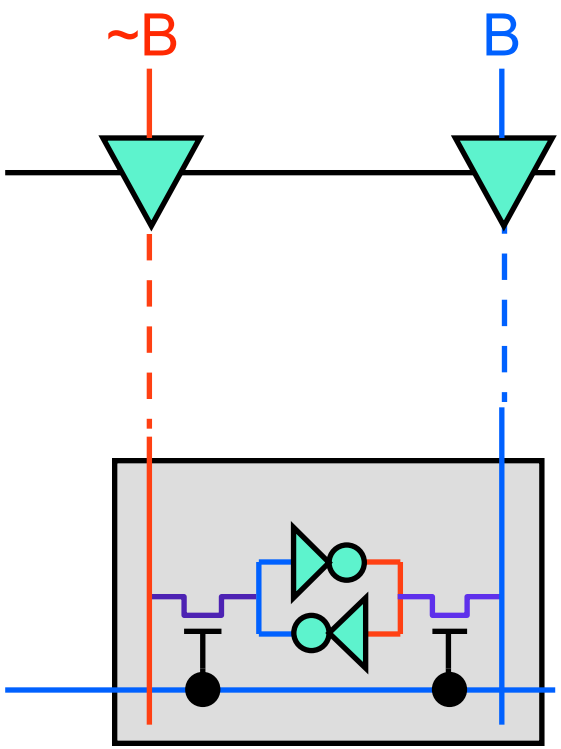
- Some extra logic on the edges
  - To write: tristates "at the top"
  - Drive write data when appropriate
  - To read: 2 things at the bottom
    - Ability to equalize bit lines
    - Sense amps

SRAM Read/Write Port



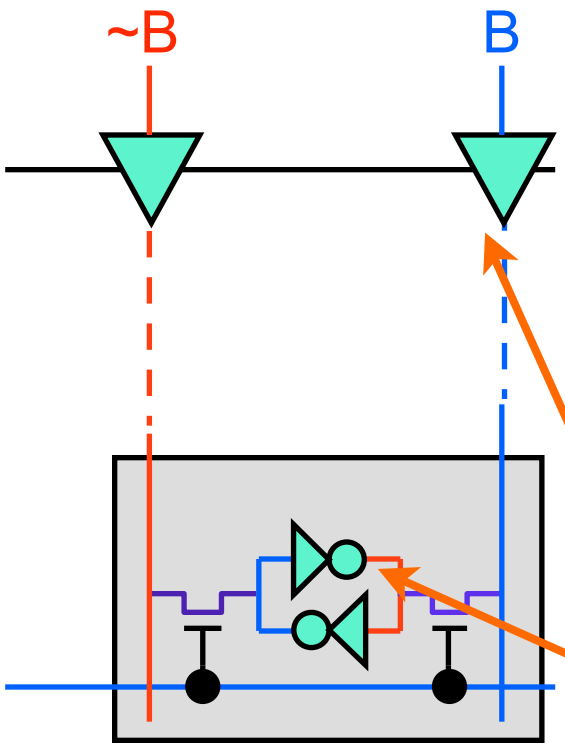
- Write process
  - Drive B and ~B onto bit lines
  - Open transistors to access CCI
  - Done for "row" by one-hot word line

SRAM Read/Write Port



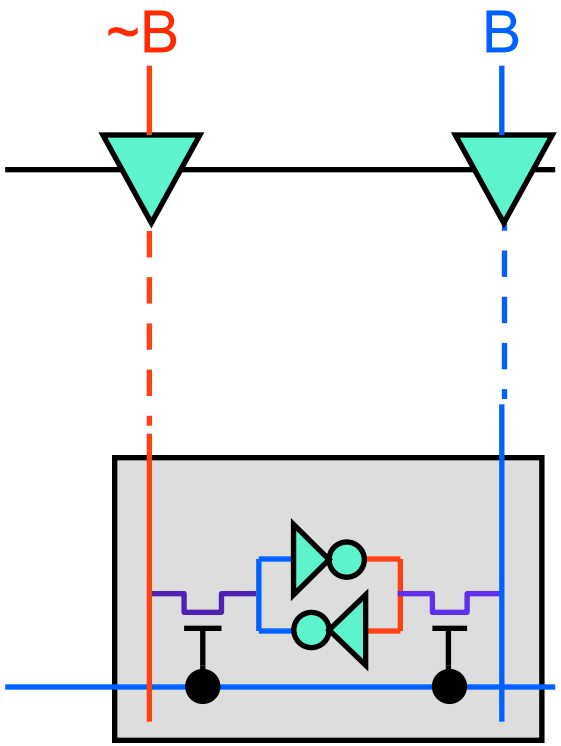
- Write process
  - Drive B and ~B onto bit lines
  - Open transistors to access CCI
  - Done for "row" by one-hot word line
- Example on left:
  - Storing a 0 ("B" side CCI has 0)
  - Writing a 1 ("B" side bit-line is 1)

SRAM Read/Write Port



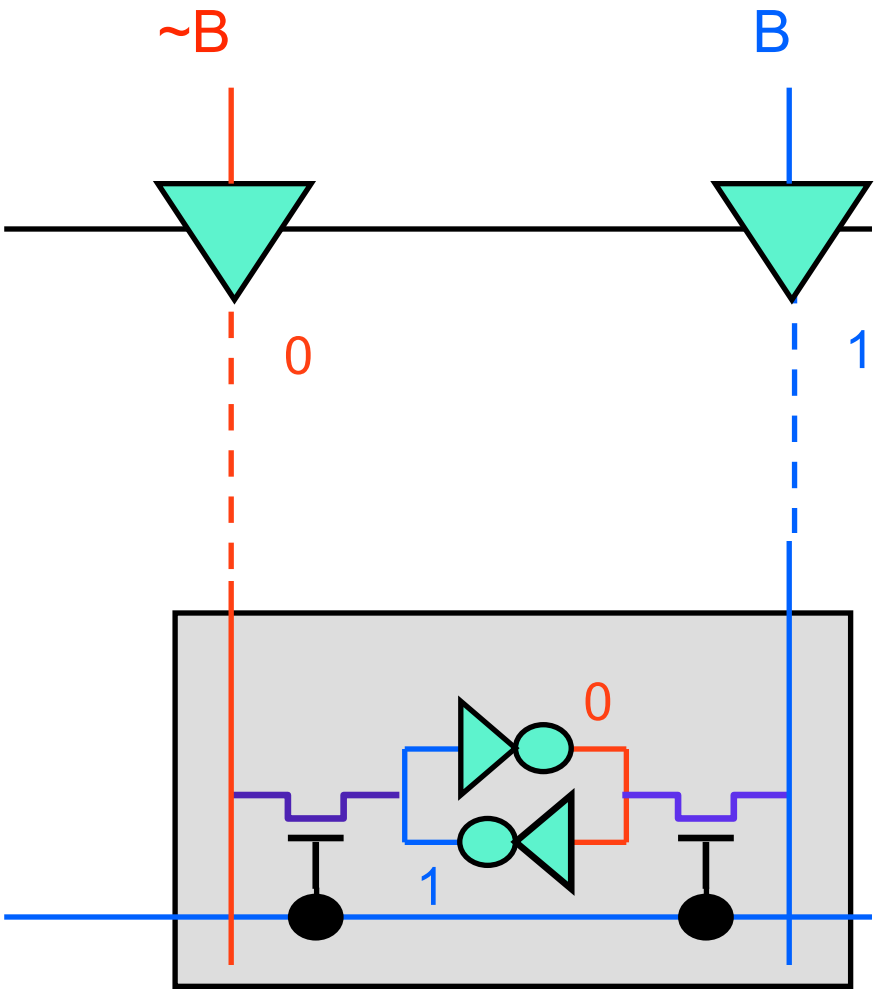
- Write process
  - Drive B and ~B onto bit lines
  - Open transistors to access CCI
  - Done for "row" by one-hot word line
- Example on left:
  - Storing a 0 ("B" side CCI has 0)
  - Writing a 1 ("B" side bit-line is 1)
- Short Circuit??
  - CCI is driving a 0
  - Tri-state is driving a 1
  - (Opposite on ~B side)

SRAM Read/Write Port

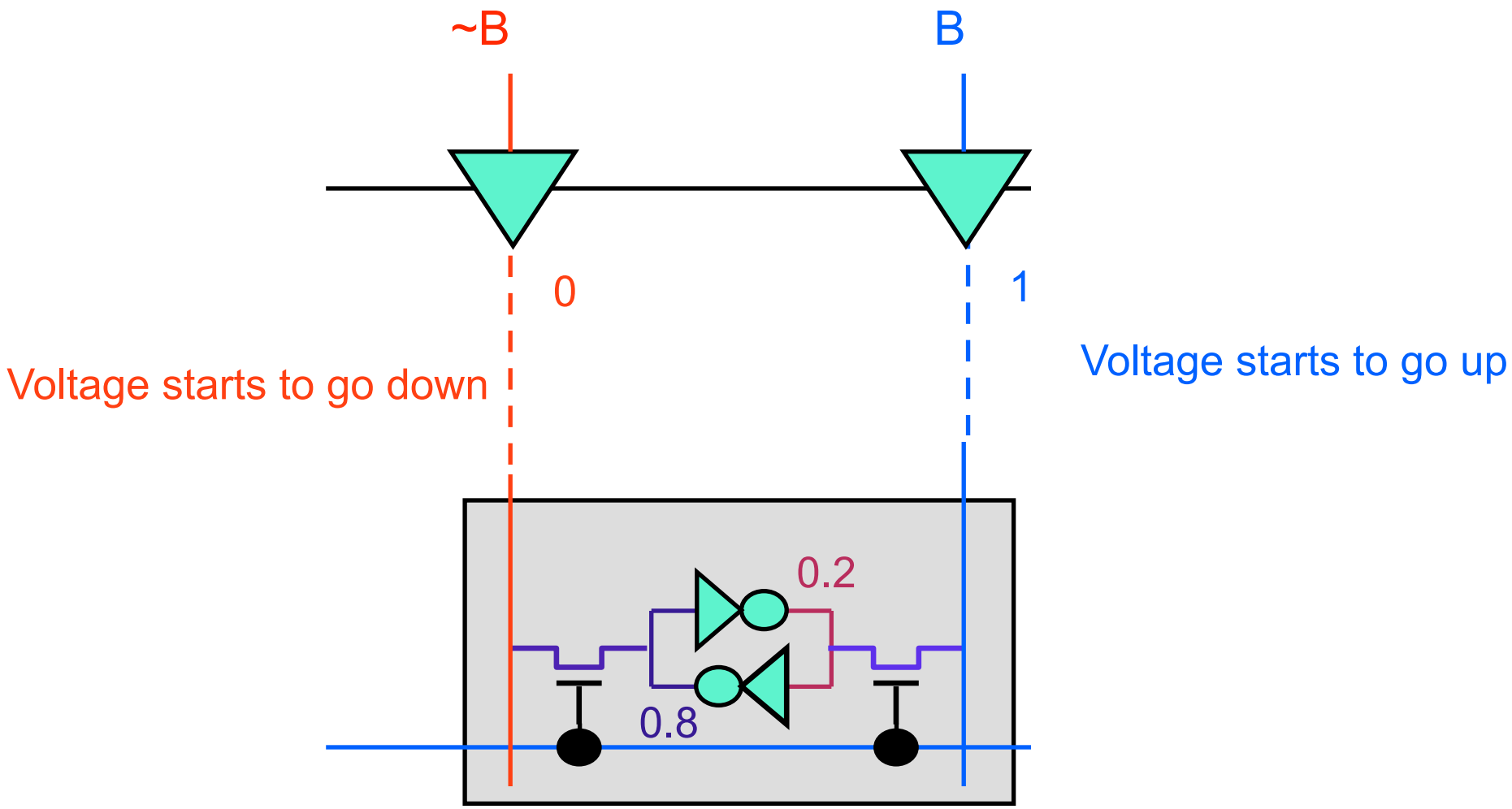


- Write process
  - Drive B and ~B onto bit lines
  - Open transistors to access CCI
  - Done for "row" by one-hot word line
- Example on left:
  - Storing a 0 ("B" side CCI has 0)
  - Writing a 1 ("B" side bit-line is 1)
- Short Circuit??
  - CCI is driving a 0
  - Tri-state is driving a 1
  - (Opposite on ~B side)
- Yes, briefly, but its ok...

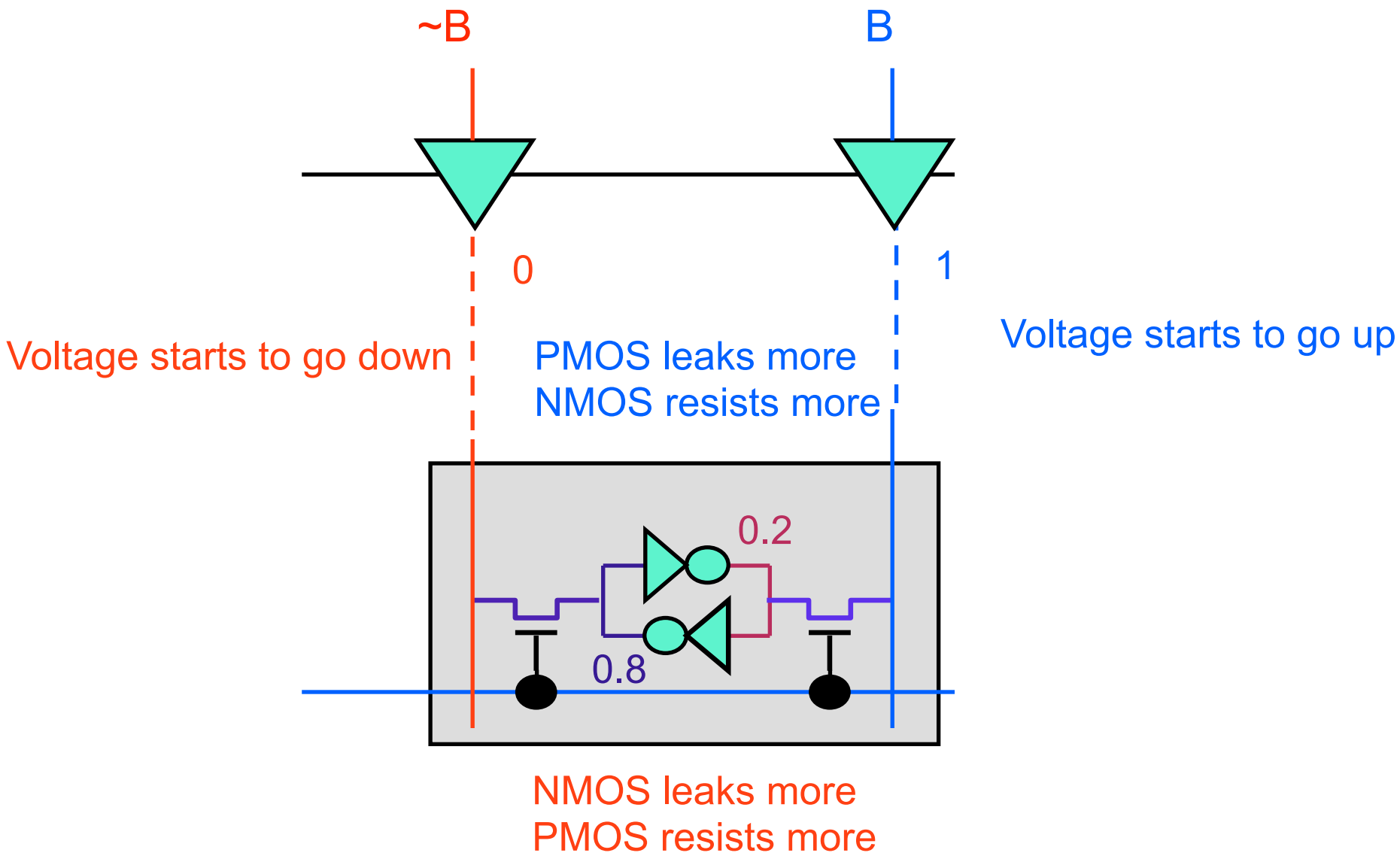
SRAM Read/Write Port



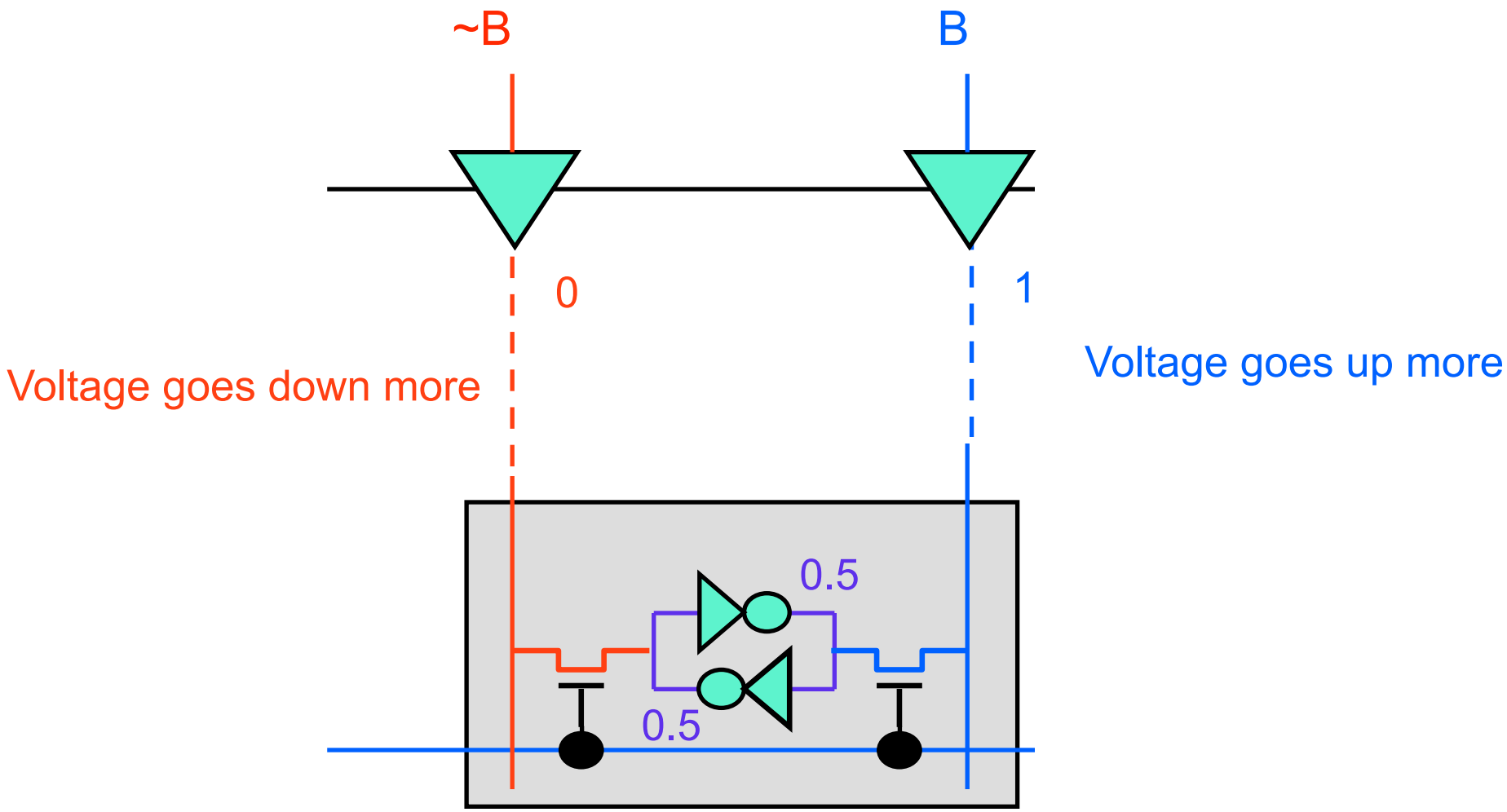
SRAM Read/Write Port



SRAM Read/Write Port

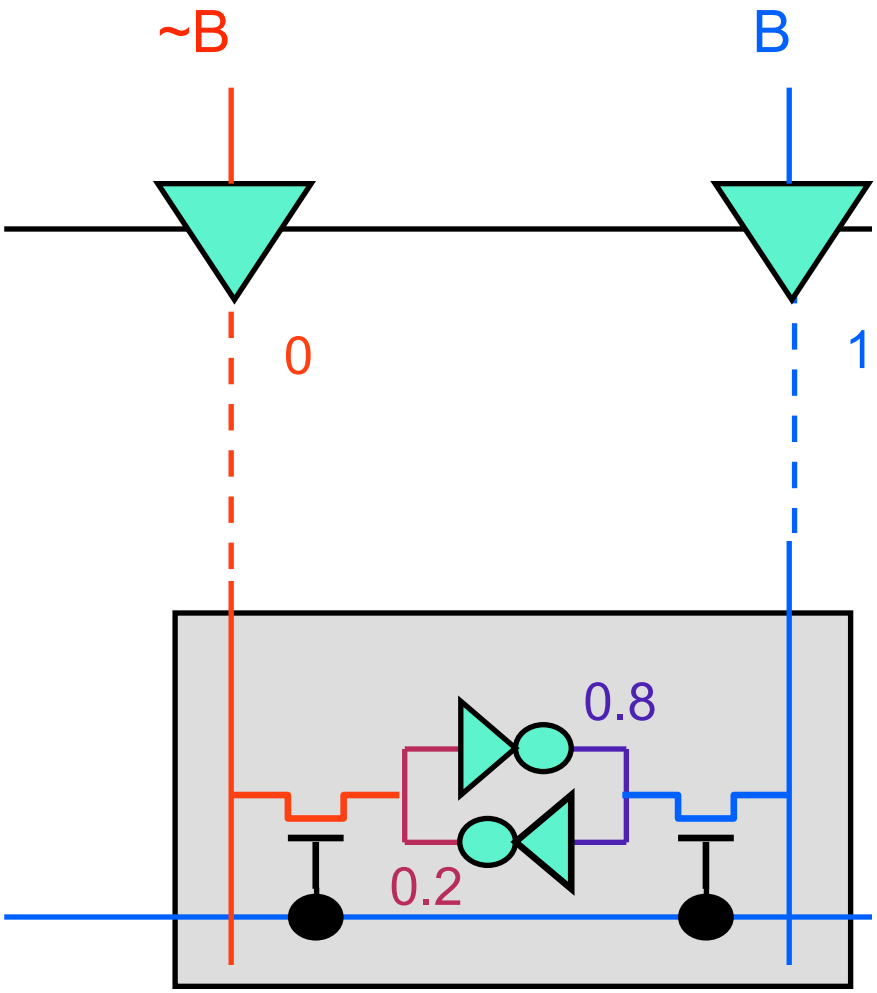


SRAM Read/Write Port



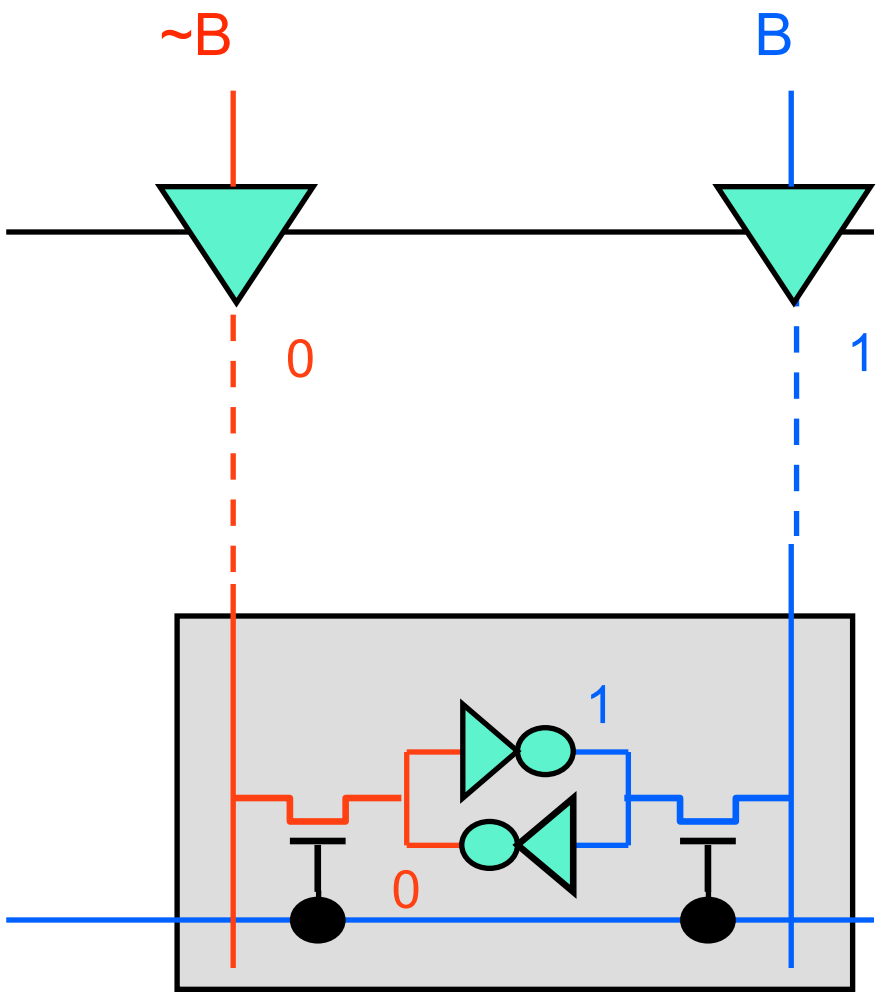
Inverters are “inverting” 0.5 -> 0.5

SRAM Read/Write Port



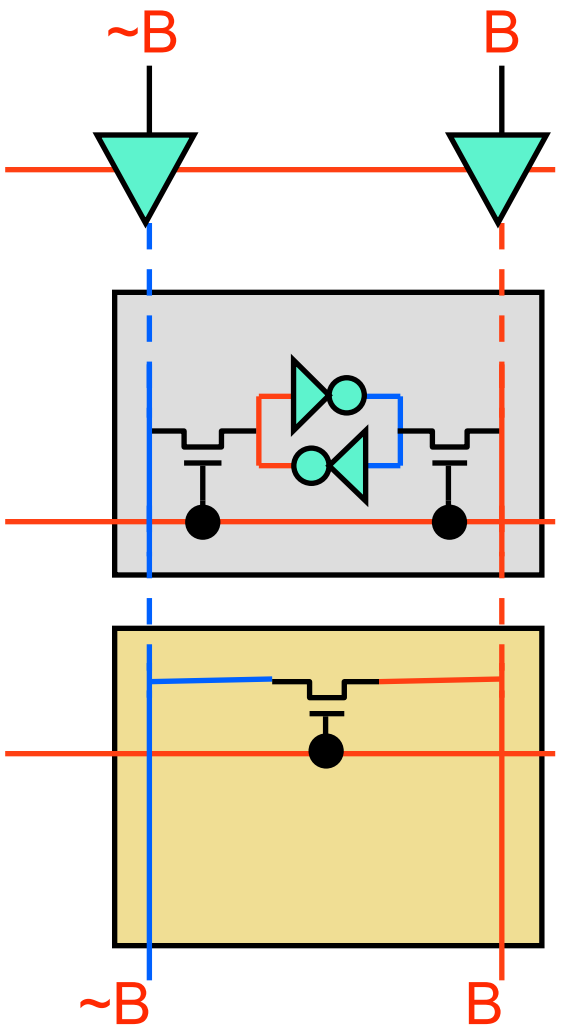
Past half-way point, inverters are “helping”

SRAM Read/Write Port



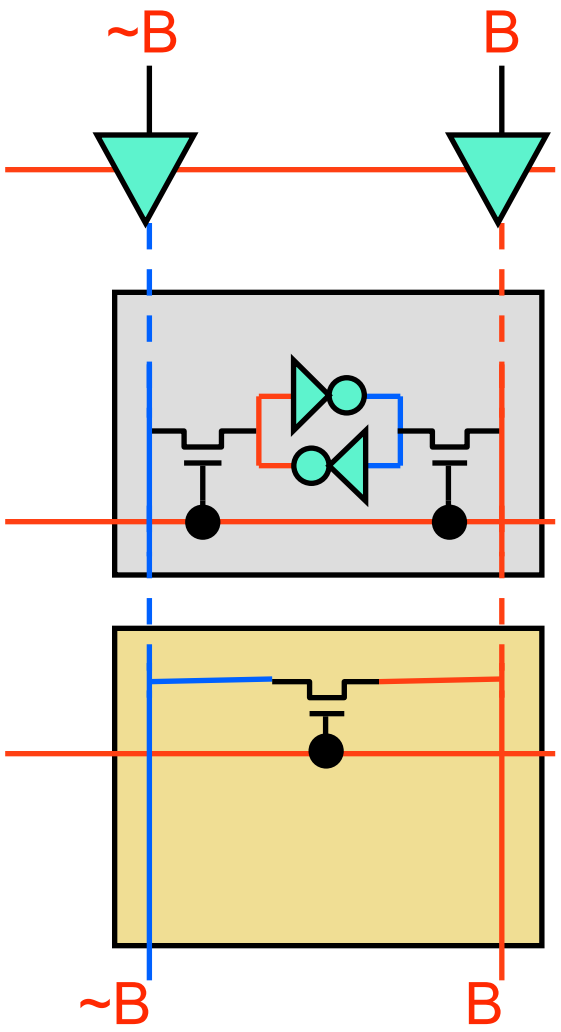
Eventually (hundreds of pico-sec) reach 0/1 state

SRAM Read/Write Port



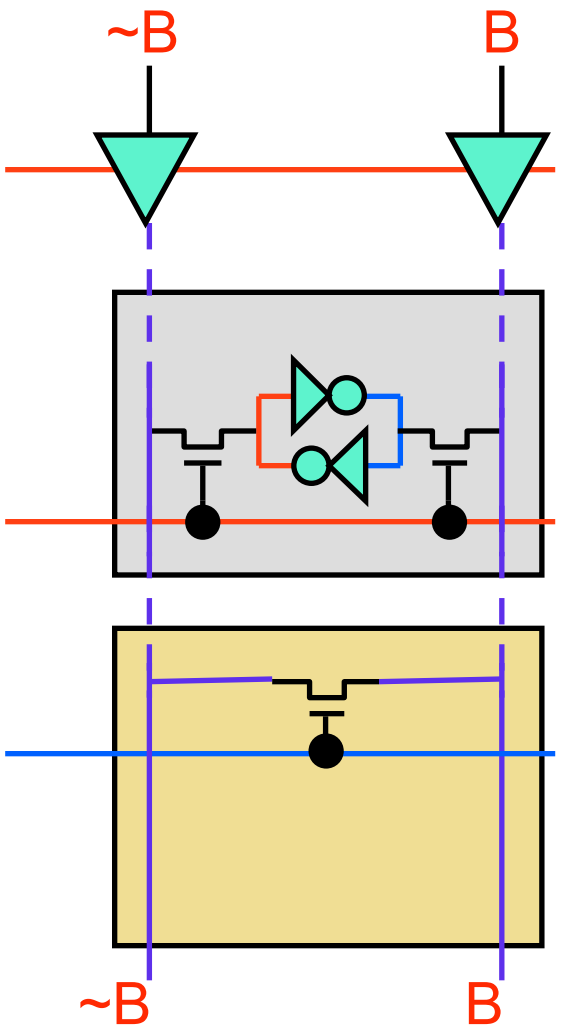
- Now, reading: bit we will read holds 1
- Last activity wrote or read 0

SRAM Read/Write Port



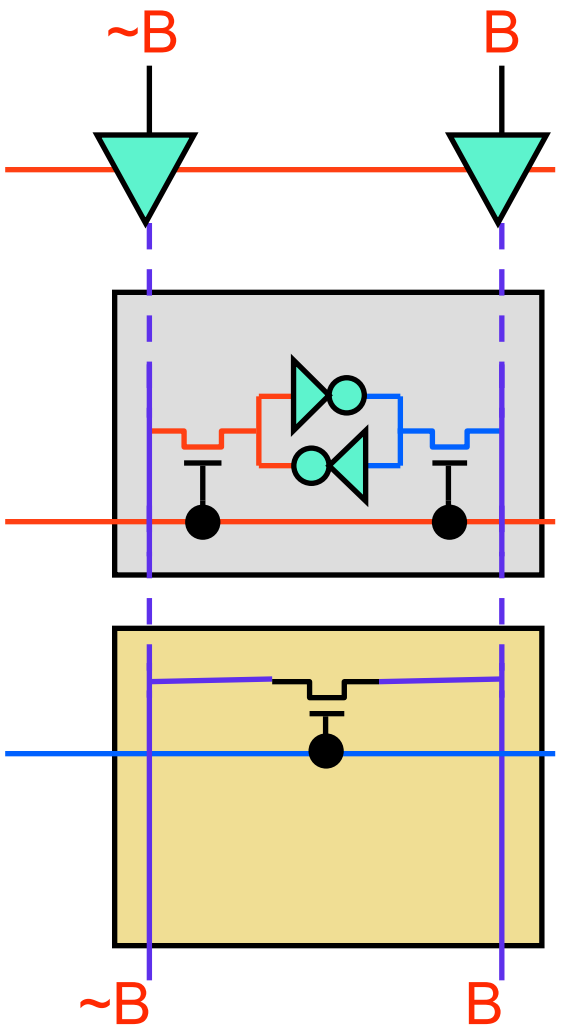
- Now, reading: bit we will read holds 1
- Last activity wrote or read 0
- Two steps to read: half clock each

SRAM Read/Write Port



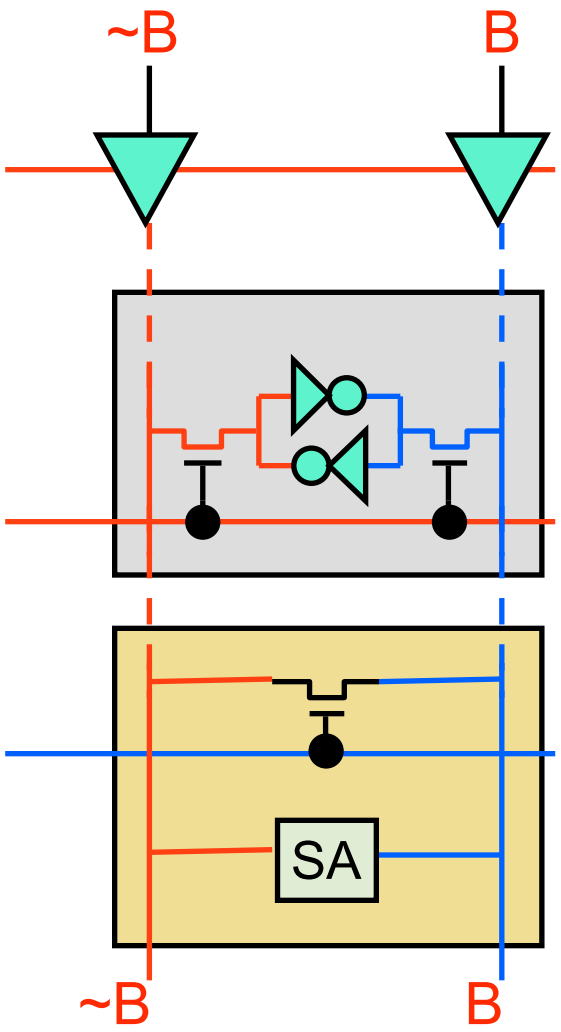
- Now, reading: bit we will read holds 1
- Last activity wrote or read 0
- Two steps to read: half clock each
  - Low clock: equalize bit lines to **0.5**

SRAM Read/Write Port



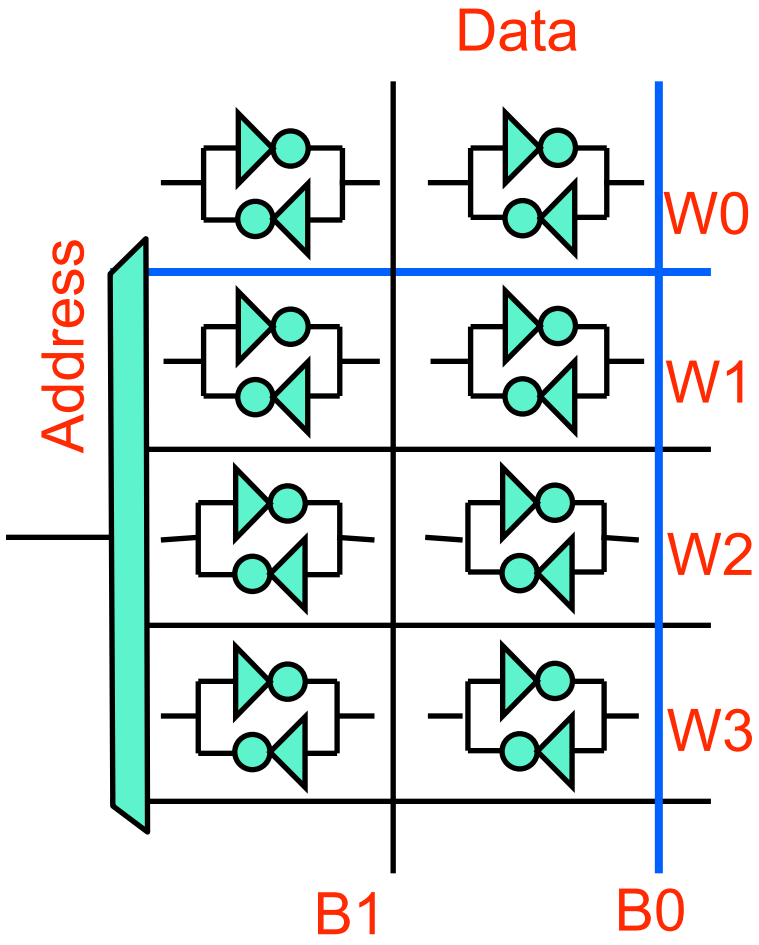
- Now, reading: bit we will read holds 1
- Last activity wrote or read 0
- Two steps to read: half clock each
  - Low clock: equalize bit lines to **0.5**
  - High clock:
    - Close equalizing transistor
    - Open r/w transistor
      - CCI is now driving bit-line
      - Bit-line slowly gets to right value

SRAM Read/Write Port



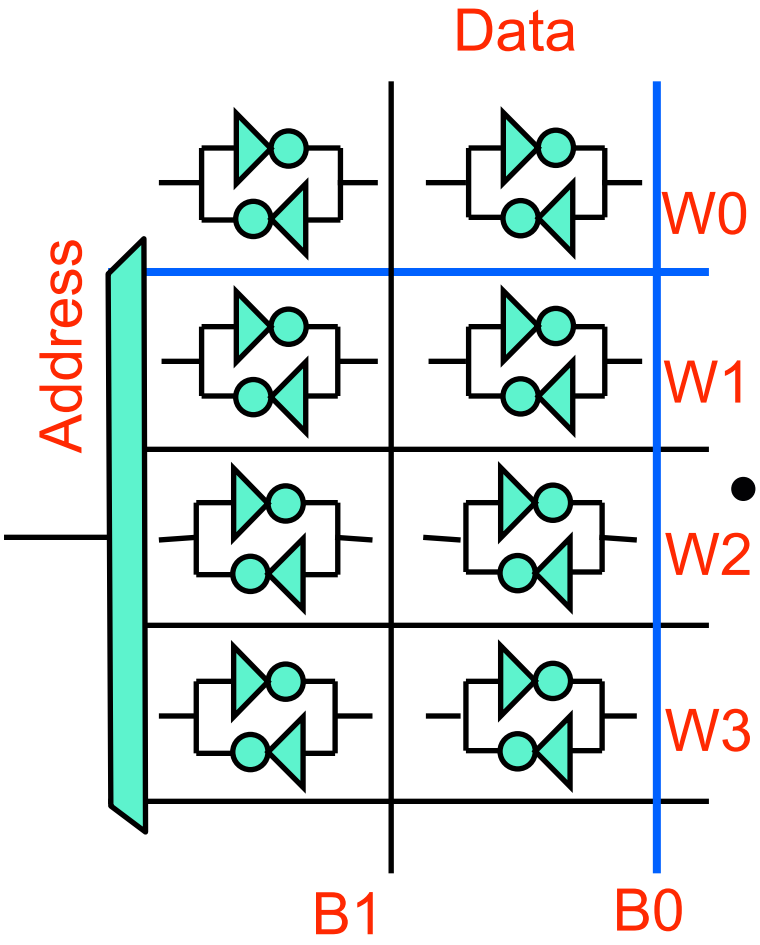
- Now, reading: bit we will read holds 1
- Last activity wrote or read 0
- Two steps to read: half clock each
  - Low clock: equalize bit lines to **0.5**
  - High clock:
    - Close equalizing transistor
    - Open r/w transistor
      - CCI is now driving bit-line
      - Bit-line slowly gets to right value
    - Sense-amps speedup up process
      - Amplify voltage swing

SRAM Latency



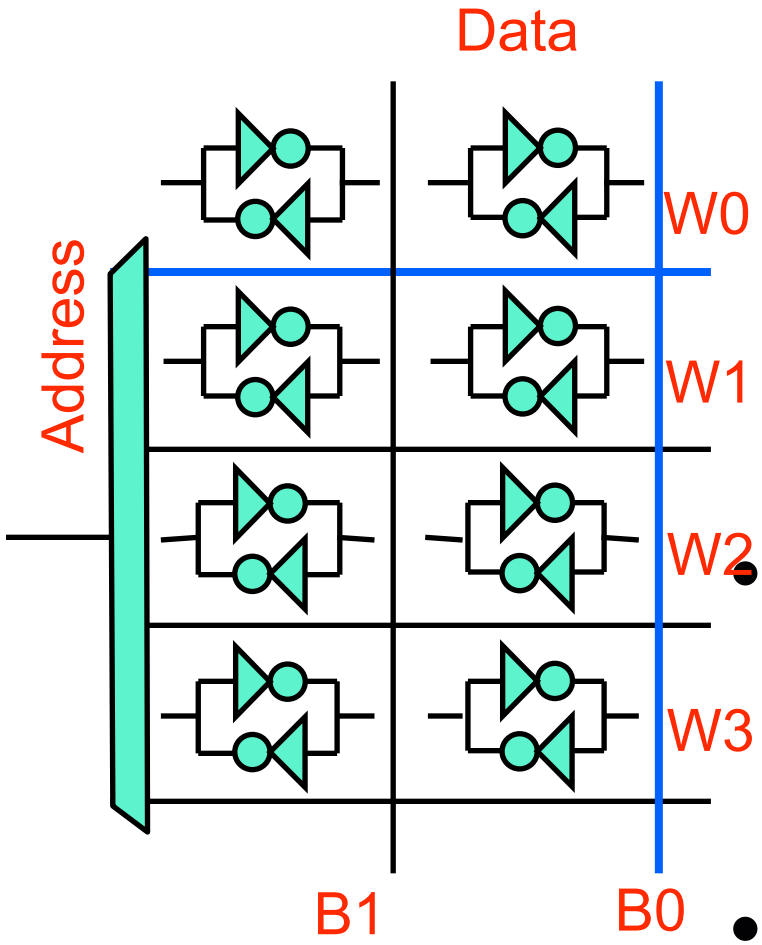
- Physics 102:
  - Delay ~= Resistance * Capacitance
  - Wires have capacitance
    - Proportional to their length
  - Wires have resistance
    - Proportional to their length

SRAM Latency



- Physics 102:
  - Delay ~= Resistance * Capacitance
  - Wires have capacitance
    - Proportional to their length
  - Wires have resistance
    - Proportional to their length
- SRAM latency:
  - Word line latency + bit line latency
  - Word line length: "width"
  - Bit line length: "height"

SRAM Latency



- Physics 102:
  - Delay ~= Resistance * Capacitance
  - Wires have capacitance
    - Proportional to their length
  - Wires have resistance
    - Proportional to their length
  - Delay ~= (Length)^2
- SRAM latency:
  - Word line latency + bit line latency
  - Word line length: "width"
  - Bit line length: "height"
- Word line length: columns * ports
- Bit line length: rows * ports

SRAM Latency

- Physics 102:
  - Delay ~= R * C
  - Wires have capacitance
    - Proportional to their length
  - Wires have resistance
    - Proportional to their length
  - Delay ~= (Length)^2
- SRAM latency:
  - Word line latency + bit line latency
  - Word line length: "width"
  - Bit line length: "height"
- Word line length: columns * ports
- Bit line length: rows * ports

$$(c * p)^2 + (r * p)^2$$

$$c^2 p^2 + r^2 p^2$$

$$p^2 (c^2 + r^2)$$

SRAM Latency

- Physics 102:
  - Delay ~= R * C
  - Wires have capacitance
    - Proportional to their length
  - Wires have resistance
    - Proportional to their length
  - Delay ~= (Length)^2
- SRAM latency:
  - Word line latency + bit line latency
  - Word line length: "width"
  - Bit line length: "height"
- Word line length: columns * ports
- Bit line length: rows * ports

$$(c * p)^2 + (r * p)^2$$

$$c^2 p^2 + r^2 p^2$$

$$p^2 (c^2 + r^2)$$

Optimize RAM with c = r

SRAM Latency

- Physics 102:
  - Delay ~= R * C
  - Wires have capacitance
    - Proportional to their length
  - Wires have resistance
    - Proportional to their length
  - Delay ~= (Length)^2
- SRAM latency:
  - Word line latency + bit line latency
  - Word line length: "width"
  - Bit line length: "height"
- Word line length: columns * ports
- Bit line length: rows * ports

$$(c * p)^2 + (r * p)^2$$

$$c^2 p^2 + r^2 p^2$$

$$p^2 (c^2 + r^2)$$

Optimize RAM with c = r

$$p^2 \ (2c^2)$$

SRAM Latency

- Physics 102:
  - Delay ~= R * C
  - Wires have capacitance
    - Proportional to their length
  - Wires have resistance
    - Proportional to their length
  - Delay ~= (Length)^2
- SRAM latency:
  - Word line latency + bit line latency
  - Word line length: "width"
  - Bit line length: "height"
- Word line length: columns * ports
- Bit line length: rows * ports

$$(c * p)^2 + (r * p)^2$$

$$c^2 p^2 + r^2 p^2$$

$$p^2 (c^2 + r^2)$$

Optimize RAM with c = r

$$p^2 \ (2c^2)$$

c^2 is "number of bits" (square of side c)

SRAM Latency

- Physics 102:
  - Delay ~= R * C
  - Wires have capacitance
    - Proportional to their length
  - Wires have resistance
    - Proportional to their length
  - Delay ~= (Length)^2
- SRAM latency:
  - Word line latency + bit line latency
  - Word line length: "width"
  - Bit line length: "height"
- Word line length: columns * ports
- Bit line length: rows * ports

$(c * p)^2 + (r * p)^2$

$c^2 p^2 + r^2 p^2$

$p^2 (c^2 + r^2)$

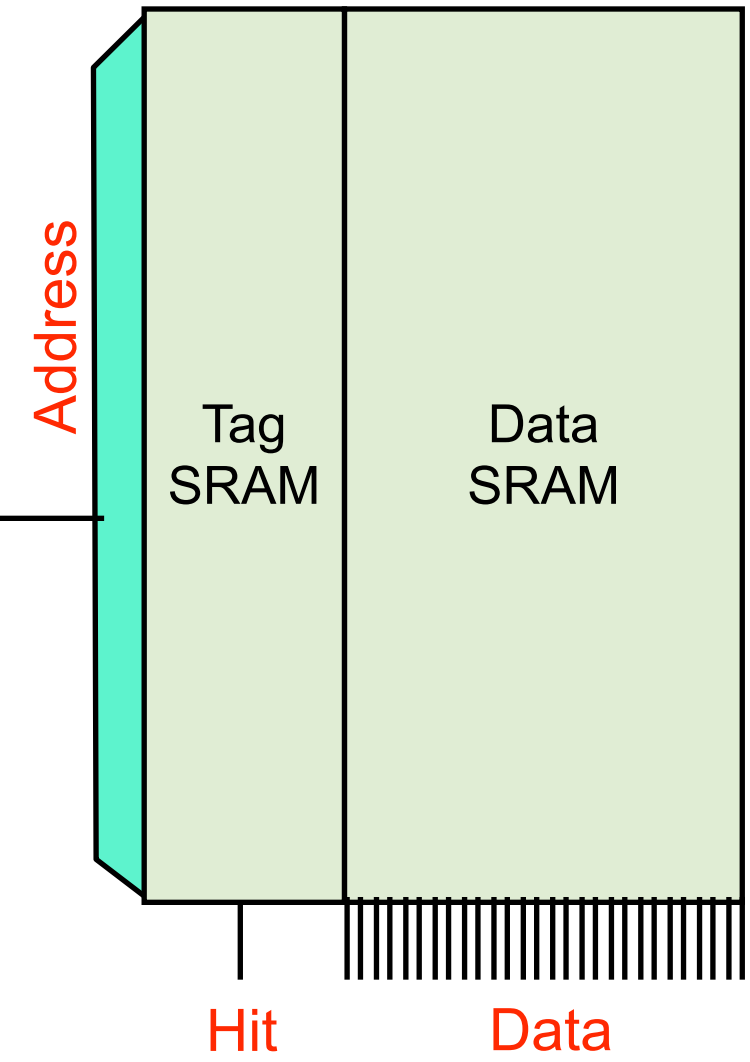
Optimize RAM with c = r

$p^2 \ (2c^2)$

c^2 is "number of bits" (square of side c)

$ports^2 * numBits$

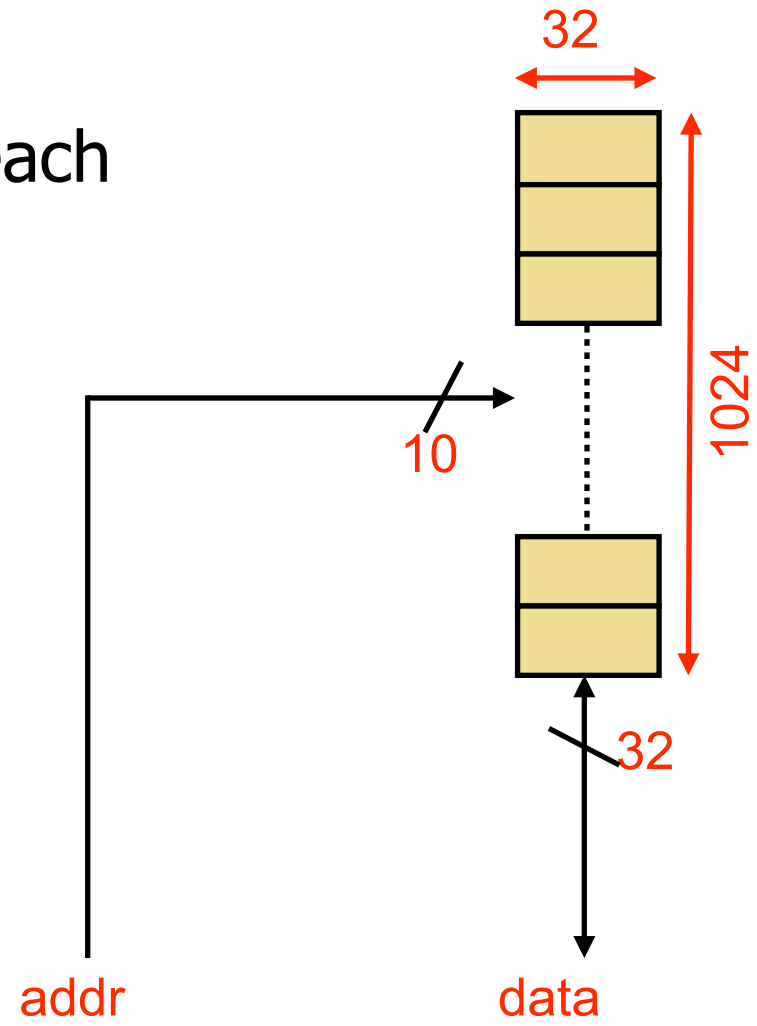
SRAMS -> Caches



- Use SRAMs to make caches
  - Hold a sub-set of memory
- Reading:
  - Input: Address to read (32 or 64 bits)
  - Output:
    - Hit? 1-bit: was it there?
    - Data: if there, requested value

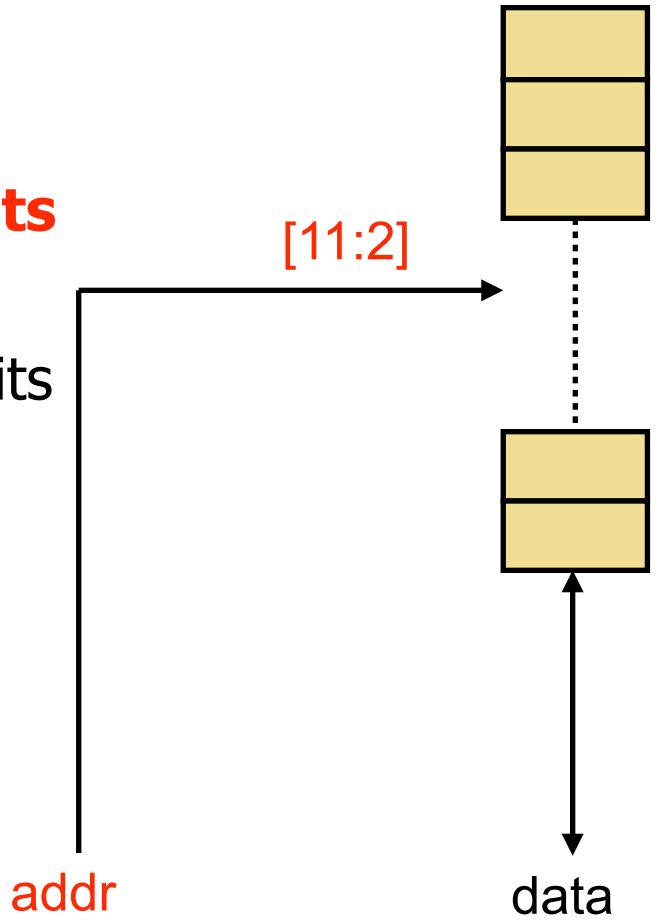
Step 1: Data Basics

- 32-bit addresses
  - 4 Byte words only (to start)
- Start with **blocks** that are 1 word each
  - 4KB, organized as 1K 4B blocks
  - Block: granularity cache manages data
- Physical cache implementation
  - 1K (1024) by 4B (32) **SRAM**
  - Called **data array**
  - 10-bit address input
  - 32-bit data input/output



Looking Up A Block

- Q: which 10 of the 32 address bits to use?
- A: bits [11:2]
  - 2 LS bits [1:0] are the **offset bits**
    - Locate byte within word
    - Don't need these to locate word
  - Next 10 LS bits [11:2] are the **index bits**
    - These locate the word
    - Nothing says index must be these bits
    - But these work best in practice
      - Why? (think about it)

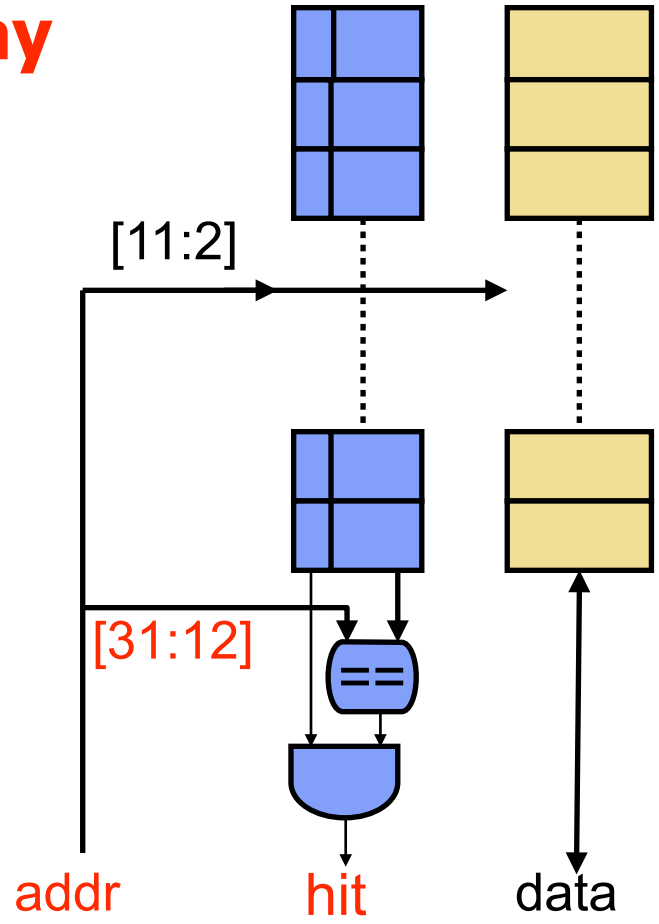


Knowing that You Found It

- Hold a subset of memory
  - How do we know if we have what we need?
  - 2^{20} different addresses map to one particular block
- Build separate and parallel **tag array**
  - 1K by 21-bit SRAM
  - 20-bit (next slide) **tag** + 1 **valid bit**
- Lookup algorithm
  - Read tag indicated by index bits
  - (Tag matches & valid bit set)

    ? Hit → data is good

    : Miss → data is garbage, wait…



Cache Use of Addresses

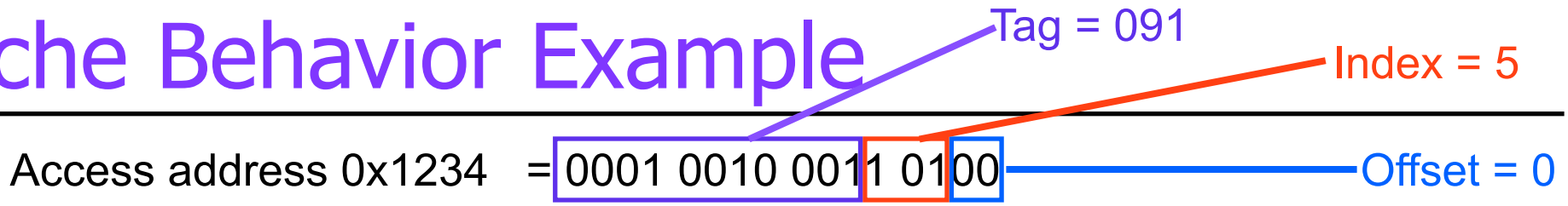
- Split address into three parts:
  - Offset: least-significant $\log_2$(block-size)
  - Index: next $\log_2$(number-of-sets)
  - Tag: everything else

Cache Behavior Example

Cache starts empty (valid = 0). 8 sets, 16 bit address for example

Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	0	000	00 00 00 00
6	0	000	00 00 00 00
7	0	000	00 00 00 00

Cache Behavior Example

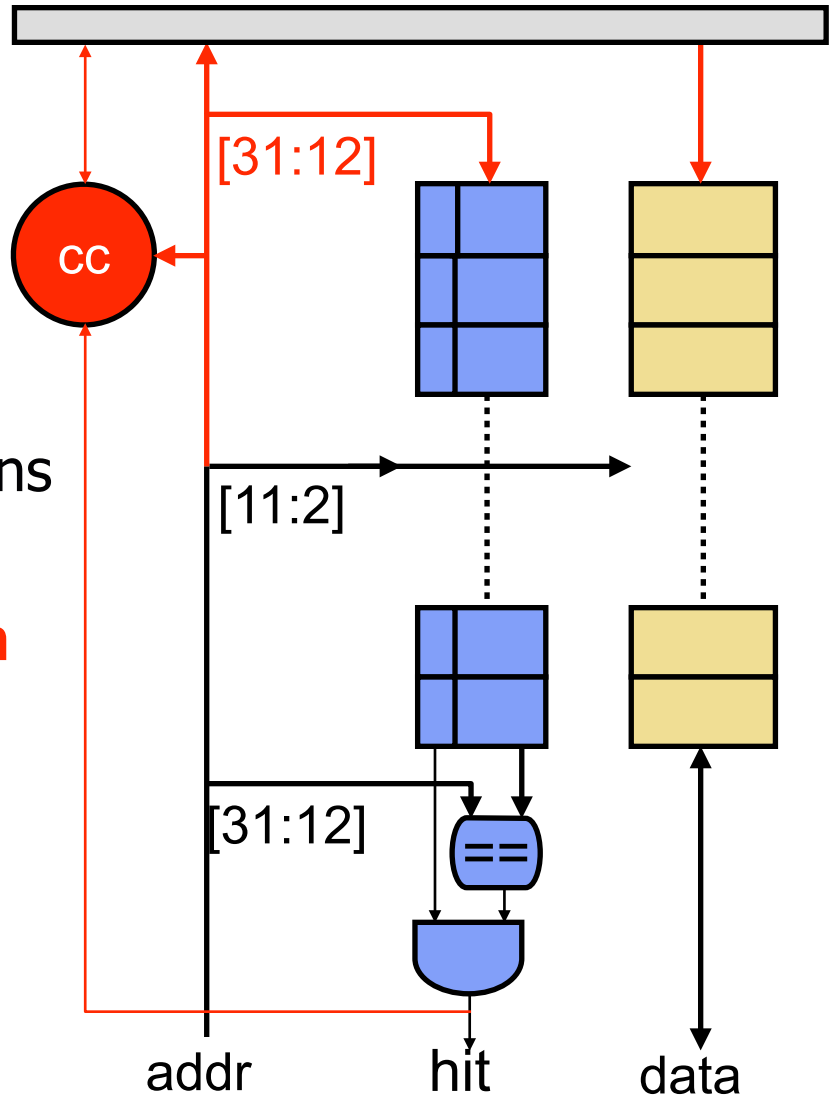


Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	0	000	00 00 00 00
6	0	000	00 00 00 00
7	0	000	00 00 00 00

Not valid: miss

Handling a Cache Miss

- What if requested word isn't in the cache?
  - How does it get in there?
- **Cache controller**: FSM
  - Remembers miss address
  - Asks next level of memory
  - Waits for response
  - Writes data/tag into proper locations
  - All of this happens on the **fill path**
  - Sometimes called **backside**



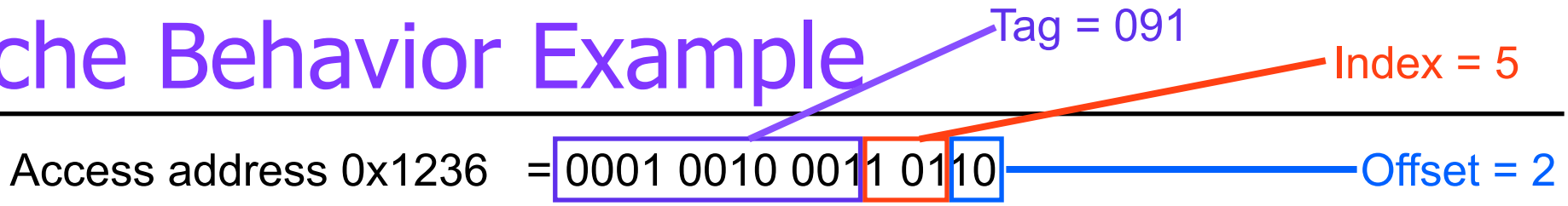
Cache Behavior Example

Access address 0x1234 = 0001 0010 0011 0100

Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	1	091	0F 1E 39 EC
6	0	000	00 00 00 00
7	0	000	00 00 00 00

```
lb: 00 00 00 EC
lh: 00 00 39 EC
lw: 0F 1E 39 EC
```

Cache Behavior Example

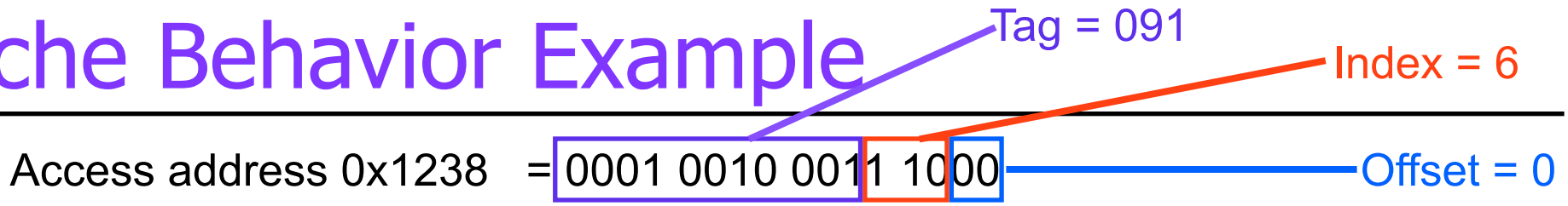


Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	1	091	0F 1E 39 EC
6	0	000	00 00 00 00
7	0	000	00 00 00 00

Valid && Tag match -> hit

```
lb: 00 00 00 1E
lh: 00 00 0F 1E
lw: (unaligned)
```

Cache Behavior Example



Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	1	091	0F 1E 39 EC
6	0	000	00 00 00 00
7	0	000	00 00 00 00

Not valid: miss

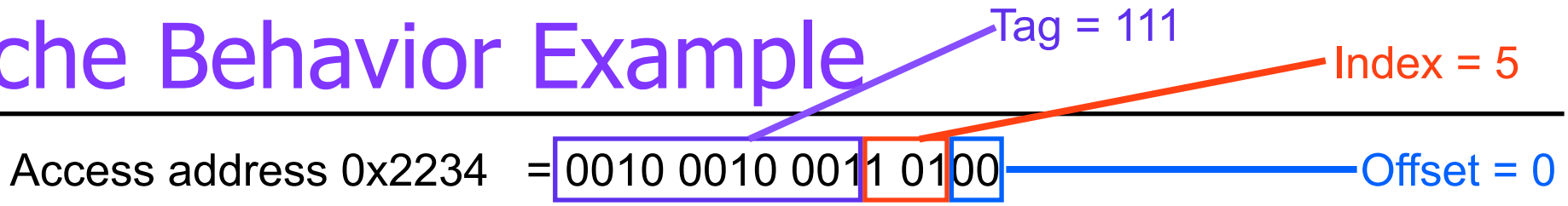
Cache Behavior Example

Access address 0x1238 = 0001 0010 0011 1000

Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	1	091	0F 1E 39 EC
6	**1**	**091**	00 00 00 00
7	0	000	00 00 00 00

Make request to next level... wait for it....

Cache Behavior Example



Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	1	091	0F 1E 39 EC
6	1	091	3C 99 11 12
7	0	000	00 00 00 00

Valid, but tag does not match: miss

Cache Behavior Example

Access address 0x2234 = 0010 0010 0011 0100

Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	1	**111**	0F 1E 39 EC
6	1	091	3C 99 11 12
7	0	000	00 00 00 00

Make request to next level... wait for it....

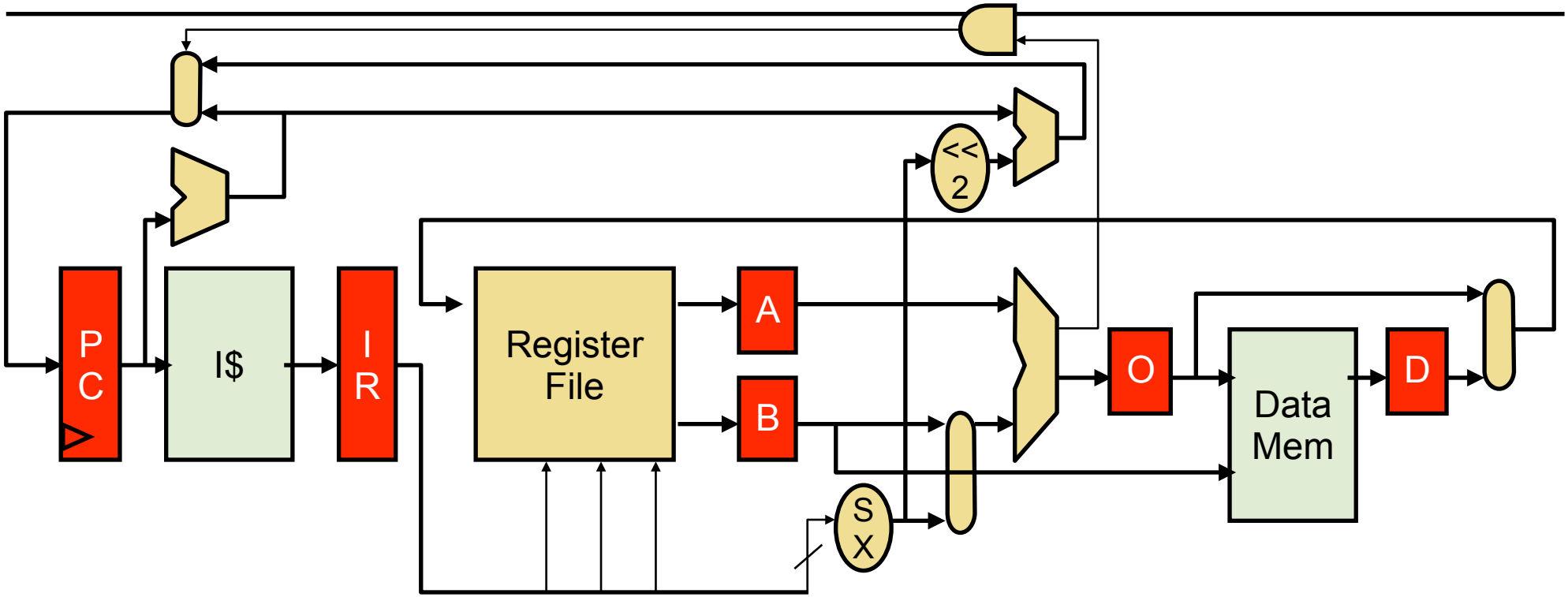
Cache Behavior Example

Access address 0x2234 = 0010 0010 0011 0100

Set #	Valid	Tag	Data
0	0	000	00 00 00 00
1	0	000	00 00 00 00
2	0	000	00 00 00 00
3	0	000	00 00 00 00
4	0	000	00 00 00 00
5	1	111	01 CF D0 87
6	1	091	3C 99 11 12
7	0	000	00 00 00 00

Note that now, 0x1234 is gone replaced by 0x2234

Cache Misses and CPI



- I$ and D$ misses stall datapath (multi-cycle or pipeline)
  - Increase CPI
  - Cache hits built into "base" CPI
    - E.g., Loads = 5 cycles in multi-cycle includes t_{hit}
    - Some loads may take more cycles...
      - Need to know latency of "average" load (t_{avg})

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 (since Pentium)
    - Performance Simulator
    - Paper simulation (like we just did)
      - Only works for small caches
      - Small number of requests (would not do for 1M accesses)

Cache Miss Paper Simulation

- 8B cache, 2B blocks -> 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C				invalid	0	0	1	
E								
8								
3								
8								
0								
8								
4								
6								

- Tag, index, offset?

Cache Miss Paper Simulation

- 8B cache, 2B blocks -> 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C				invalid	0	0	1	
E								
8								
3								
8								
0								
8								
4								
6								

- Tag: 1 bit, Index: 2 bits, Offset: 1 bit

Cache Miss Paper Simulation

- 8B cache, 2B blocks -> 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C				invalid	0	0	1	
E								
8								
3								
8								
0								
8								
4								
6								

- What happens for each request?

Cache Miss Paper Simulation

- 8B cache, 2B blocks -> 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C	1	2	0	invalid	0	0	1	Miss
E				invalid	0	1	1	
8								
3								
8								
0								
8								
4								
6								

- What happens for each request?

Cache Miss Paper Simulation

- 8B cache, 2B blocks -> 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C	1	2	0	invalid	0	0	1	Miss
E	1	3	0	invalid	0	1	1	Hit
8				invalid	0	1	1	
3								
8								
0								
8								
4								
6								

- What happens for each request?

Cache Miss Paper Simulation

- 8B cache, 2B blocks -> 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C	1	2	0	invalid	0	0	1	Miss
E	1	3	0	invalid	0	1	1	Hit
8	1	0	0	invalid	0	1	1	Miss
3				1	0	1	1	
8								
0								
8								
4								
6								

- What happens for each request?

Cache Miss Paper Simulation

- 8B cache, 2B blocks -> 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C	1	2	0	invalid	0	0	1	Miss
E	1	3	0	invalid	0	1	1	Hit
8	1	0	0	invalid	0	1	1	Miss
3	0	1	1	1	0	1	1	Hit
8				1	0	1	1	
0								
8								
4								
6								

- What happens for each request?

Cache Miss Paper Simulation

- 8B cache, 2B blocks -> 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C	1	2	0	invalid	0	0	1	Miss
E	1	3	0	invalid	0	1	1	Hit
8	1	0	0	invalid	0	1	1	Miss
3	0	1	1	1	0	1	1	Hit
8	1	0	0	1	0	1	1	Hit
0				1	0	1	1	
8								
4								
6								

- What happens for each request?

Cache Miss Paper Simulation

- 8B cache, 2B blocks -> 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C	1	2	0	invalid	0	0	1	Miss
E	1	3	0	invalid	0	1	1	Hit
8	1	0	0	invalid	0	1	1	Miss
3	0	1	1	1	0	1	1	Hit
8	1	0	0	1	0	1	1	Hit
0	0	0	0	1	0	1	1	Miss
8				0	0	1	1	
4								
6								

- What happens for each request?

Cache Miss Paper Simulation

- 8B cache, 2B blocks -> 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C	1	2	0	invalid	0	0	1	Miss
E	1	3	0	invalid	0	1	1	Hit
8	1	0	0	invalid	0	1	1	Miss
3	0	1	1	1	0	1	1	Hit
8	1	0	0	1	0	1	1	Hit
0	0	0	0	1	0	1	1	Miss
8	1	0	0	0	0	1	1	Miss
4				1	0	1	1	
6								

- What happens for each request?

Cache Miss Paper Simulation

- 8B cache, 2B blocks -> 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C	1	2	0	invalid	0	0	1	Miss
E	1	3	0	invalid	0	1	1	Hit
8	1	0	0	invalid	0	1	1	Miss
3	0	1	1	1	0	1	1	Hit
8	1	0	0	1	0	1	1	Hit
0	0	0	0	1	0	1	1	Miss
8	1	0	0	0	0	1	1	Miss
4	0	2	0	1	0	1	1	Miss
6				1	0	0	1	

- What happens for each request?

Cache Miss Paper Simulation

- 8B cache, 2B blocks -> 4 sets

Address	Tag	Index	Offset	Set 0	Set 1	Set 2	Set3	Result
C	1	2	0	invalid	0	0	1	Miss
E	1	3	0	invalid	0	1	1	Hit
8	1	0	0	invalid	0	1	1	Miss
3	0	1	1	1	0	1	1	Hit
8	1	0	0	1	0	1	1	Hit
0	0	0	0	1	0	1	1	Miss
8	1	0	0	0	0	1	1	Miss
4	0	2	0	1	0	1	1	Miss
6	0	3	0	1	0	0	1	Miss

- What happens for each request?

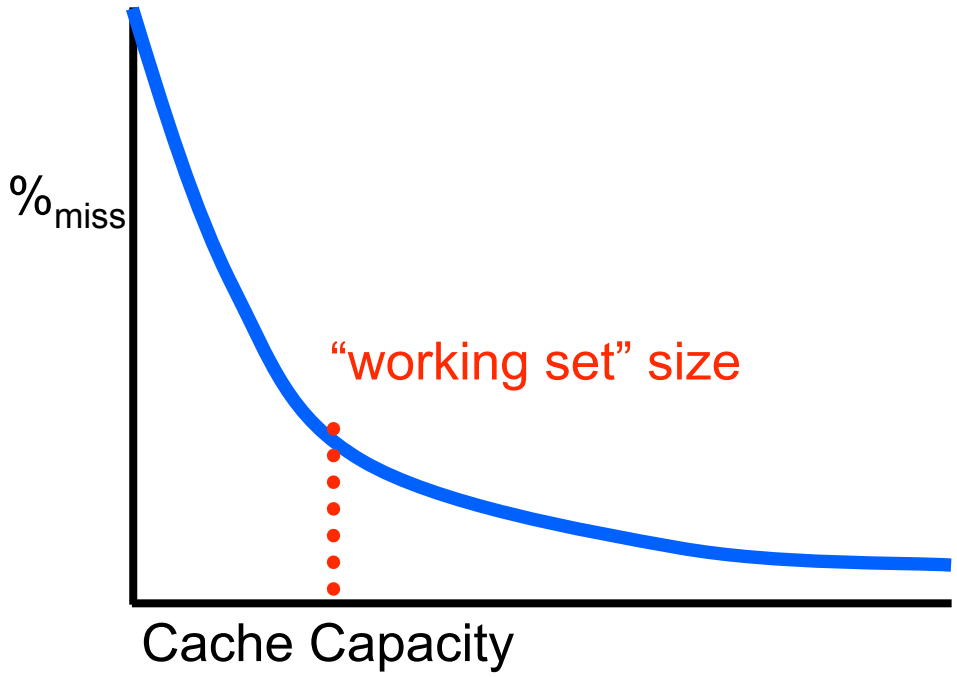
Cache Miss Paper Simulation

- %miss: 6 / 9 = 66%
  - Not good...
  - How could we improve it?

Result
Miss
Hit
Miss
Hit
Hit
Miss
Miss
Miss
Miss

Capacity and Performance

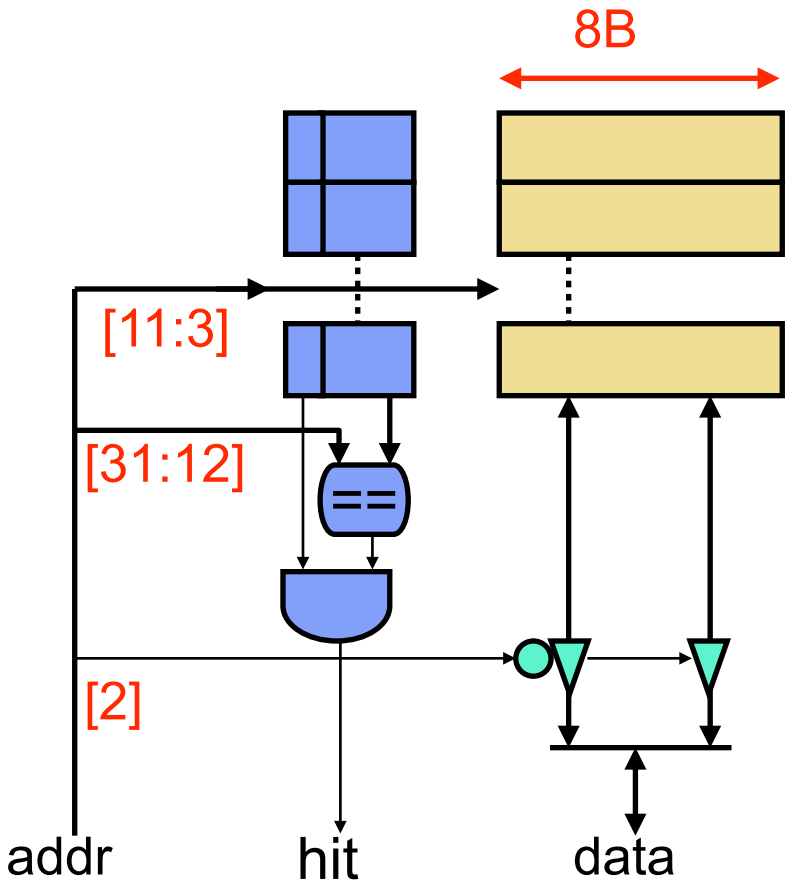
- Simplest way to reduce $\%_{miss}$: increase capacity
  - + Miss rate decreases monotonically
    - **"Working set"**: insns/data program is actively using
  - – t_{hit} increases
  - t_{avg} ?

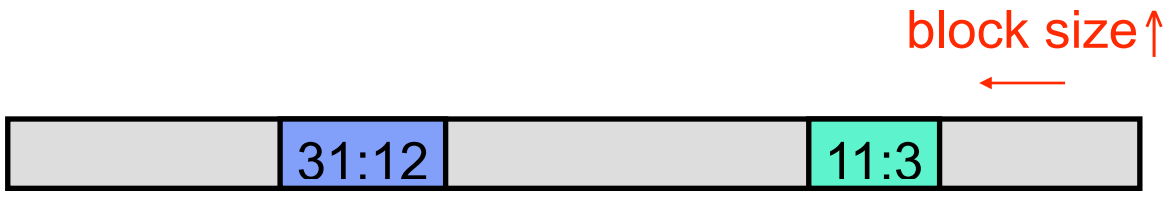


- Given capacity, manipulate $\%_{miss}$ by changing **organization**

Block Size

- One possible re-organization: increase **block size**
  - + Exploit **spatial locality**
  - – Caveat: increase conflicts too
  - – Increases $t_{hit:}$ need word select mux
    - By a little, not too bad
  - + Reduce tag overhead





Tag Overhead

- "4KB cache" means cache holds 4KB of data (**capacity**)
  - Tag storage is considered overhead
    - Valid bit usually not counted
  - Tag overhead = tag size / data size

- 4KB cache with 4B blocks?
  - 4B blocks → 2-bit offset
  - 4KB cache / 4B blocks → 1024 blocks → 10-bit index
  - 32-bit address – 2-bit offset – 10-bit index = 20-bit tag
  - 20-bit tag / 32-bit block = **63% overhead**

Block Size and Tag Overhead

- 4KB cache with 1024 4B blocks?
  - 4B blocks → 2-bit offset, 1024 frames → 10-bit index
  - 32-bit address – 2-bit offset – 10-bit index = 20-bit tag
  - 20-bit tag / 32-bit block = 63% overhead
- 4KB cache with 512 8B blocks
  - 8B blocks → 3-bit offset, 512 frames → 9-bit index
  - 32-bit address – 3-bit offset – 9-bit index = 20-bit tag
  - **20-bit tag** / **64-bit block** = **32% overhead**
  - Notice: tag size is same, but data size is twice as big
- A realistic example: 64KB cache with 64B blocks
  - 16-bit tag / 512-bit block = **~ 2% overhead**

Cache Miss Paper Simulation

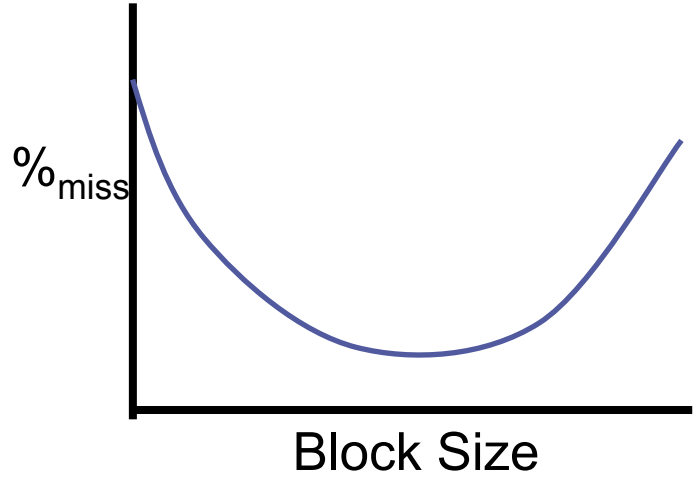
- 8B cache, 4B blocks -> 2 sets

Address	Tag	Index	Offset	Set 0	Set 1	Result
C	1	1	0	invalid	0	Miss
E	1	1	2	invalid	1	Hit
8	1	0	0	invalid	1	Miss
3	0	0	3	1	1	**Miss**
8	1	0	0	0	1	**Miss**
0	0	0	0	1	1	Miss
8	1	0	0	0	1	Miss
4	0	1	0	1	1	Miss
6	0	1	2	1	0	**Hit**

- 8/3: new conflicts (fewer sets)
- 4/6: spatial locality

Block Size and Miss Rate Redux

+ **Spatial prefetching**
  - For blocks with adjacent addresses
  - Turns miss/miss pairs into miss/hit pairs
  - Example: `4,6`



– **Conflicts**
  - For blocks with non-adjacent addresses (but in adjacent frames)
  - Turns hits into misses by disallowing simultaneous residence
  - Example: `8, 3`

- Both effects always present to some degree
  - Spatial prefetching dominates initially (until 64–128B)
  - Conflicts dominate afterwards
  - Optimal block size is 32–256B (varies across programs)
  - Typical: 64B

Block Size and Miss Penalty

- Does increasing block size increase t_{miss}?
  - Don't larger blocks take longer to read, transfer, and fill?
  - They do, but…
- t_{miss} of an isolated miss is not affected
  - **Critical Word First / Early Restart (CRF/ER)**
  - Requested word fetched first, pipeline restarts immediately
  - Remaining words in block transferred/filled in the background
- t_{miss}'es of a cluster of misses will suffer
  - Reads/transfers/fills of two misses cannot be overlapped
  - Latencies start to pile up
  - This is technically a bandwidth problem (more later)

Cache Miss Paper Simulation

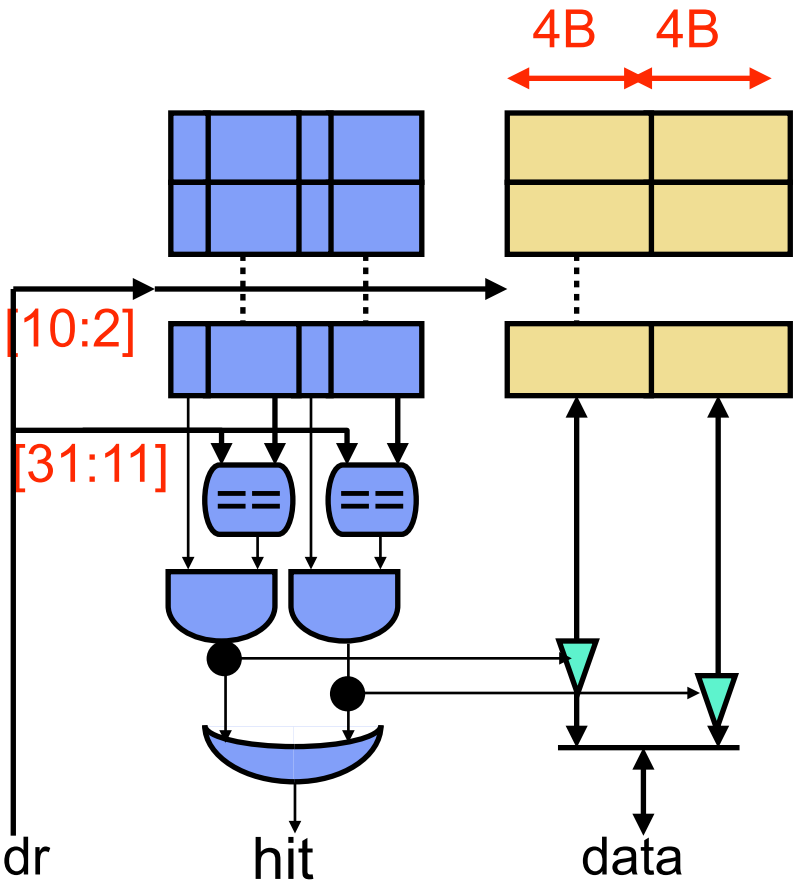
- 8B cache, 4B blocks -> 2 sets

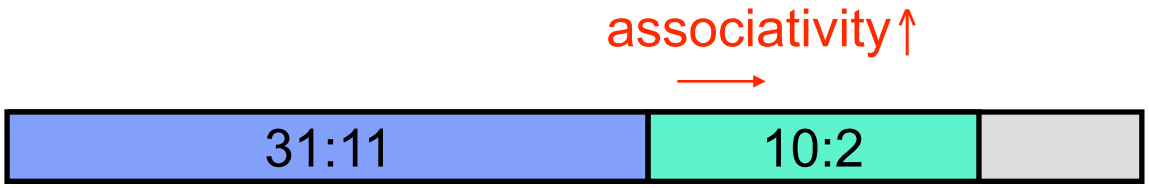
Address	Tag	Index	Offset	Set 0	Set 1	Result
C	1	1	0	invalid	0	Miss
E	1	1	2	invalid	1	Hit
8	1	0	0	invalid	1	Miss
3	0	0	3	1	1	**Miss**
8	1	0	0	0	1	**Miss**
0	0	0	0	1	1	Miss
8	1	0	0	0	1	Miss
4	0	1	0	1	1	Miss
6	0	1	2	1	1	**Hit**

- 8 (1000) and 0 (0000): same set for any $ < 16B
- Can we do anything about this?

Associativity

- New organizational dimension: **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)**
- Lookup algorithm
  - Use index bits to find set
  - Read data/tags in all frames in parallel
  - **Any** (match && valid bit) ? Hit : Miss



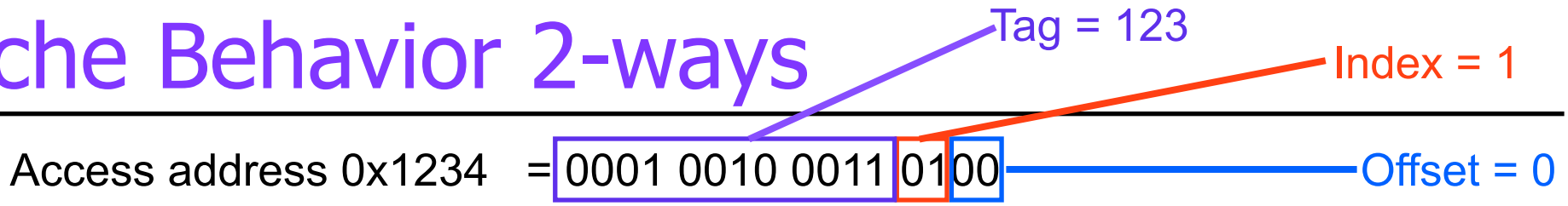


Cache Behavior 2-ways

Cache: 4 sets, 2 ways, 4B blocks

Set #	Way 0			Way 1		
	V	Tag	Data	V	Tag	Data
0	0	000	00 00 00 00	0	000	00 00 00 00
1	0	000	00 00 00 00	0	000	00 00 00 00
2	0	000	00 00 00 00	0	000	00 00 00 00
3	0	000	00 00 00 00	0	000	00 00 00 00

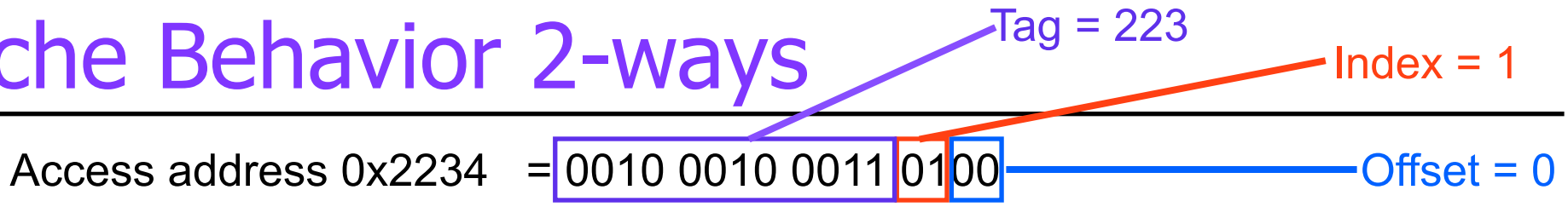
Cache Behavior 2-ways



Set #	Way 0			Way 1		
	V	Tag	Data	V	Tag	Data
0	0	000	00 00 00 00	0	000	00 00 00 00
1	**1**	**123**	00 00 00 00	0	000	00 00 00 00
2	0	000	00 00 00 00	0	000	00 00 00 00
3	0	000	00 00 00 00	0	000	00 00 00 00

Miss. Request from next level. Wait...

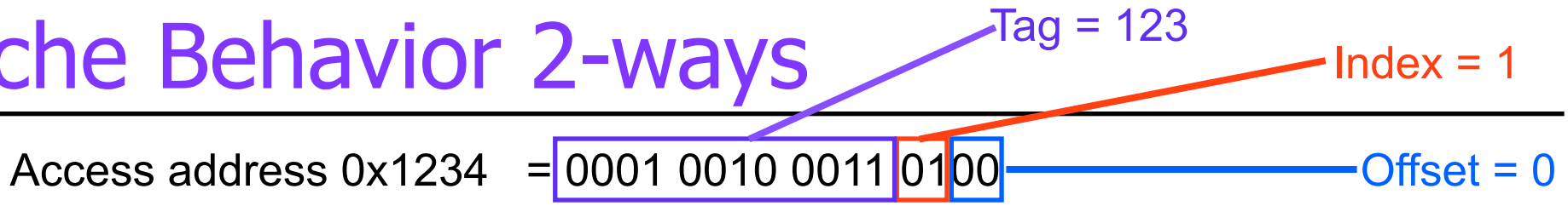
Cache Behavior 2-ways



Set #	Way 0			Way 1		
	V	Tag	Data	V	Tag	Data
0	0	000	00 00 00 00	0	000	00 00 00 00
1	1	123	0F 1E 39 EC	**1**	**223**	00 00 00 00
2	0	000	00 00 00 00	0	000	00 00 00 00
3	0	000	00 00 00 00	0	000	00 00 00 00

Miss. Request from next level. Wait...

Cache Behavior 2-ways



Set #	Way 0			Way 1		
	V	Tag	Data	V	Tag	Data
0	0	000	00 00 00 00	0	000	00 00 00 00
1	1	123	0F 1E 39 EC	1	223	01 CF D0 87
2	0	000	00 00 00 00	0	000	00 00 00 00
3	0	000	00 00 00 00	0	000	00 00 00 00

Hit. In Way 0

Cache Miss Paper Simulation

- 8B cache, 2B blocks, 2 ways -> 2 sets

				Set 0		Way 1		
Address	Tag	Index	Offset	Way0	Way1	Way0	Way1	Result
C	3	0	0	invalid	0	0	1	Miss
E	3	1	0	3	0	0	1	**Miss**
8	2	0	0	3	0	0	3	Miss
3	0	1	1	3	2	0	3	Hit
8	2	0	0	3	2	0	3	Hit
0	0	0	0	3	2	0	3	Miss
8	2	0	0	0	2	0	3	**Hit**
4	1	0	0	0	2	0	3	Miss
6	1	1	0	1	2	0	3	Miss

- What happens for each request?

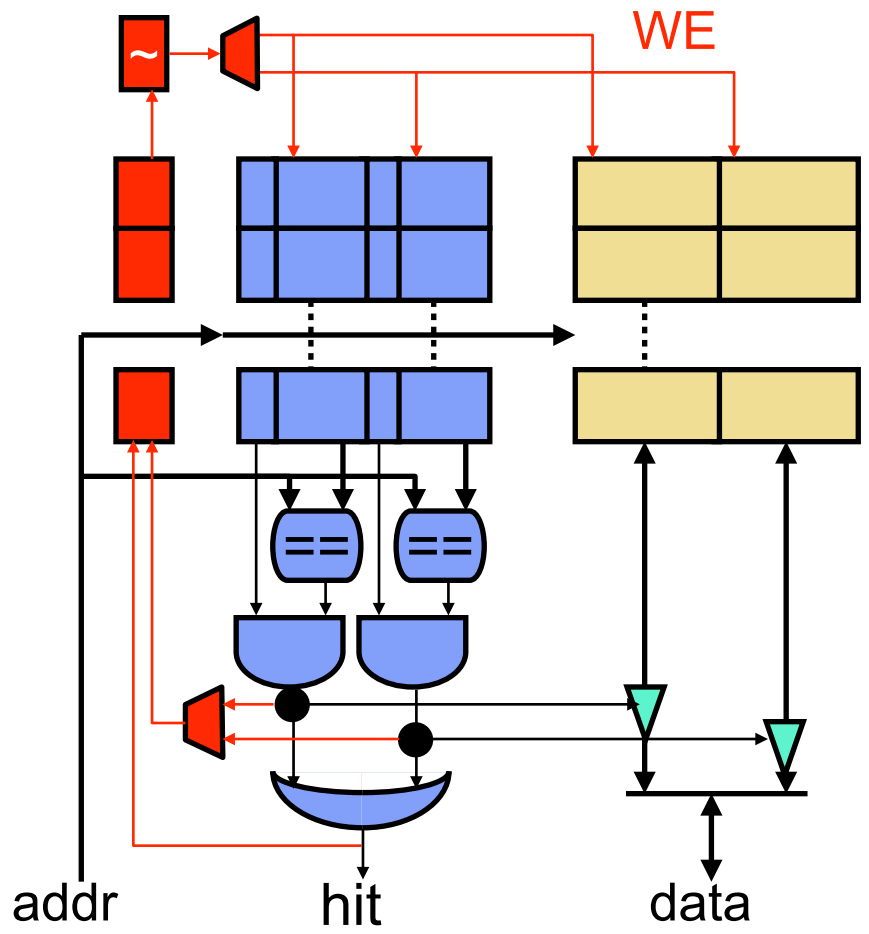
Replacement Policies

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

- Belady's (oracle): block that will be used furthest in future
- Random
- FIFO (first-in first-out)
- 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
  - Equal to LRU for 2-way SA caches

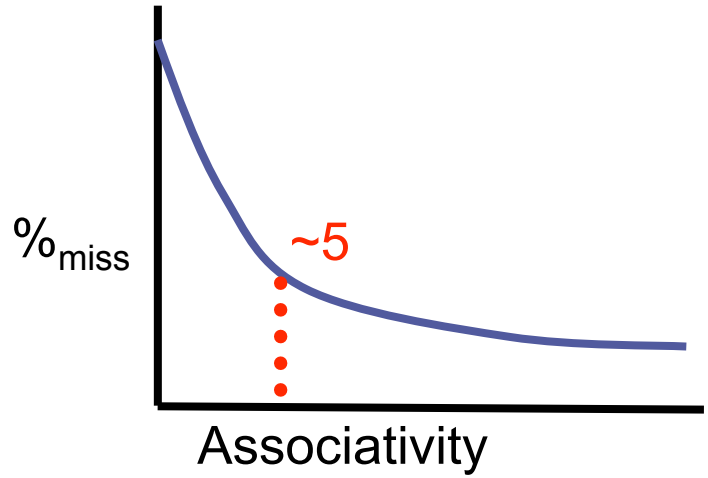
NMRU Implementation

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



Associativity And Performance

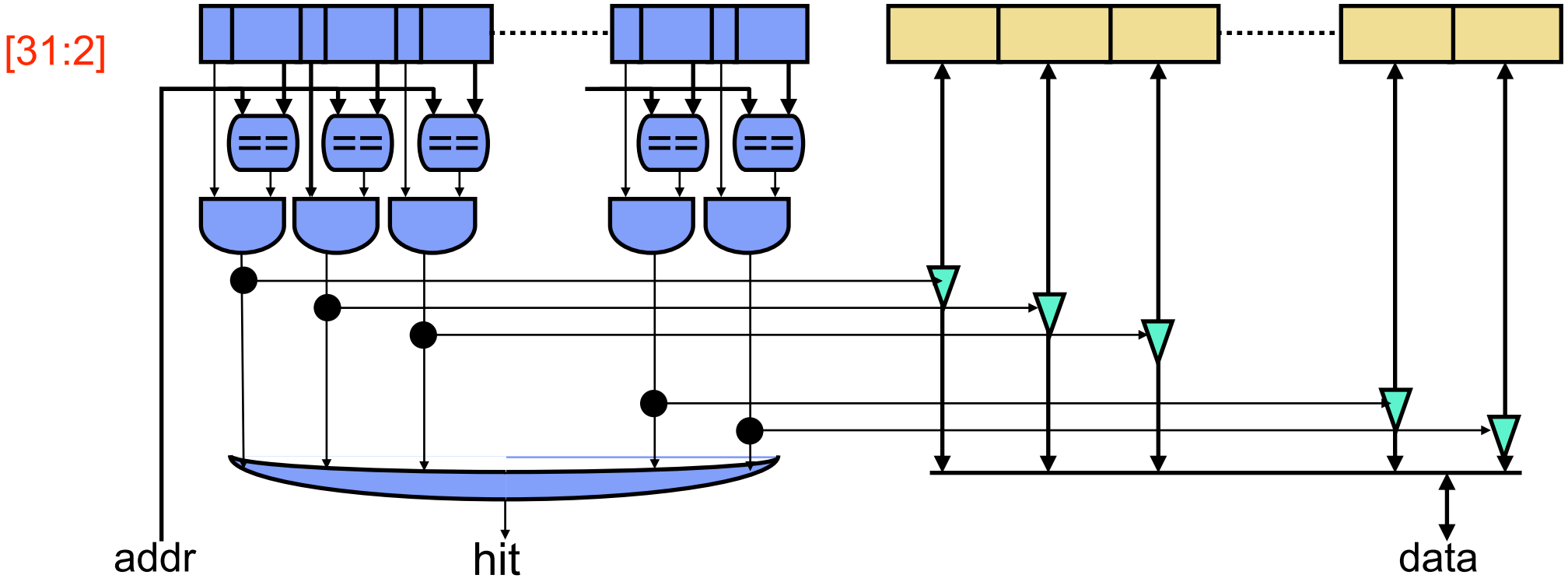
- The associativity game
  - + Higher associative caches have lower $\%_{miss}$
  - – t_{hit} increases
    - But not much for low associativities (2,3,4,5)
  - t_{avg}?



- Block-size and number of sets should be powers of two
  - Makes indexing easier (just rip bits out of the address)
- 5-way set-associativity? No problem

Full Associativity

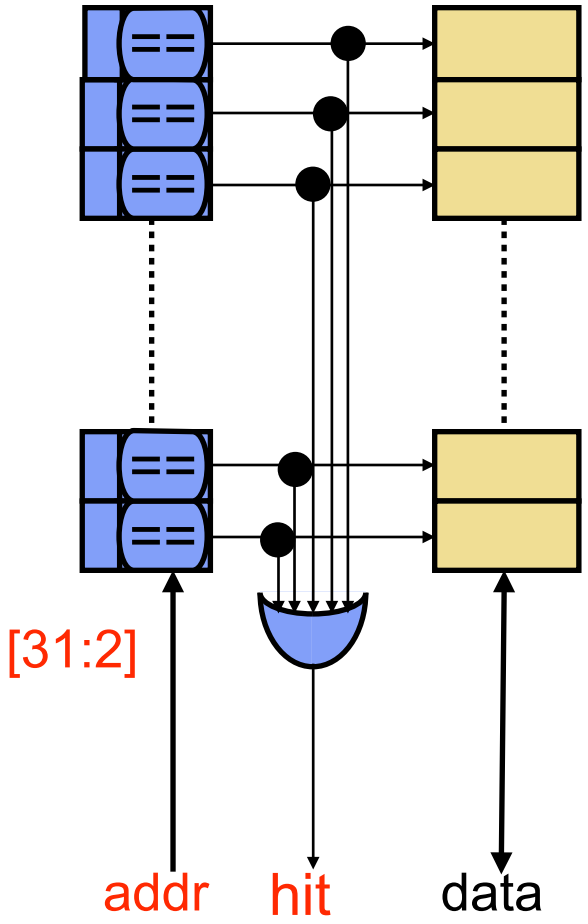
- How to implement full (or at least high) associativity?
  - This way is terribly inefficient
  - 1K matches are unavoidable, but 1K data reads + 1K-to-1 mux?



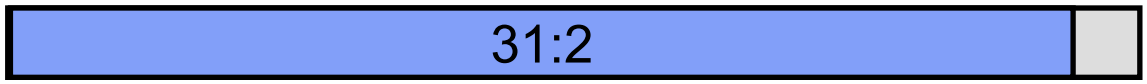
Full-Associativity with CAMs

- **CAM**: content associative memory
  - Array of words with built-in comparators
  - Input is data (tag)
  - Output is 1H encoding of matching slot
- FA cache
  - Tags as CAM, data as RAM

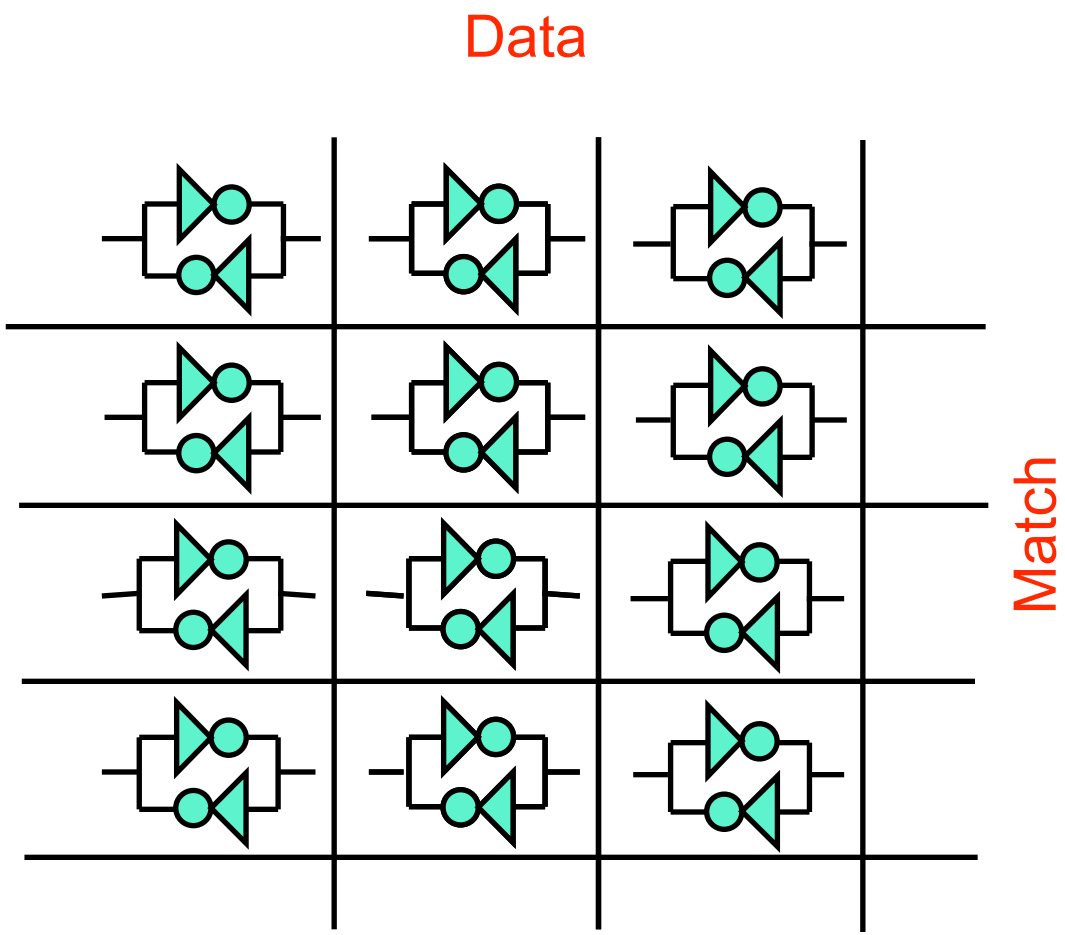
  - Effective but expensive (EE reasons)
  - Upshot: used for 16-/32-way associativity
  - – No good way to build 1024-way associativity
  - \+ No real need for it, either



look ma, no index bits

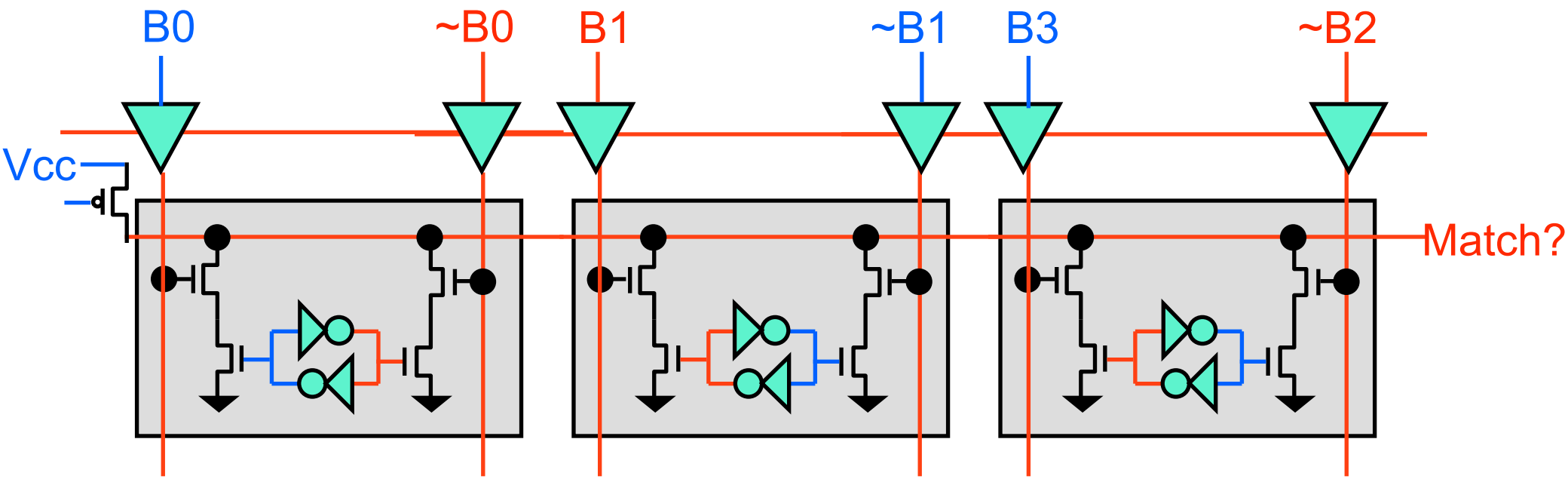


CAM -> Content Associative Memory



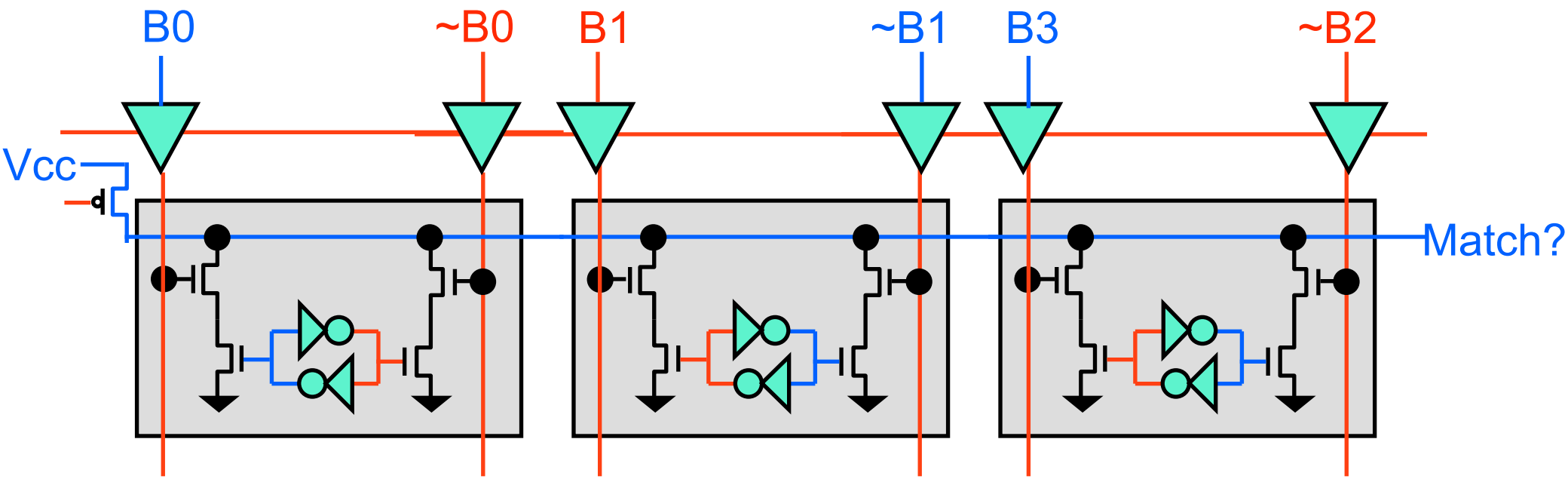
- Input: Data to match
  - (ex on left: 3 bits)
- Output: matching entries
  - (ex on left: 4 entries)

CAM circuit



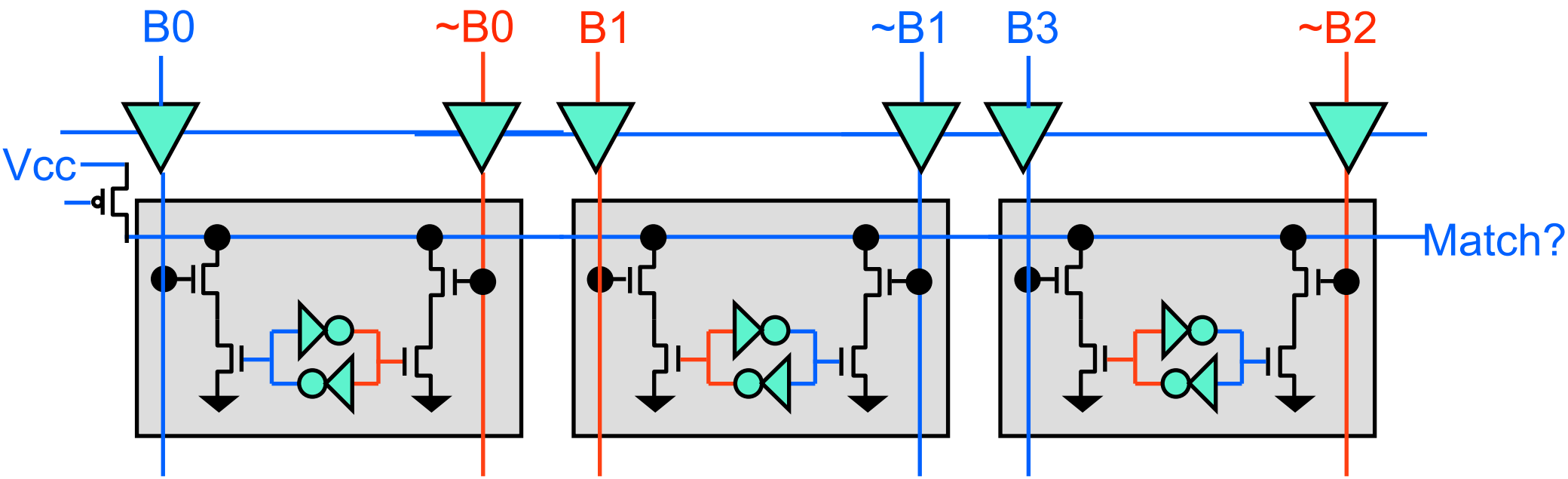
- CAM match port looks different from RAM r/w port
- Cells look similar
  - Note: Bit stored on right, ~Bit on left (opposite of inputs)

CAM circuit



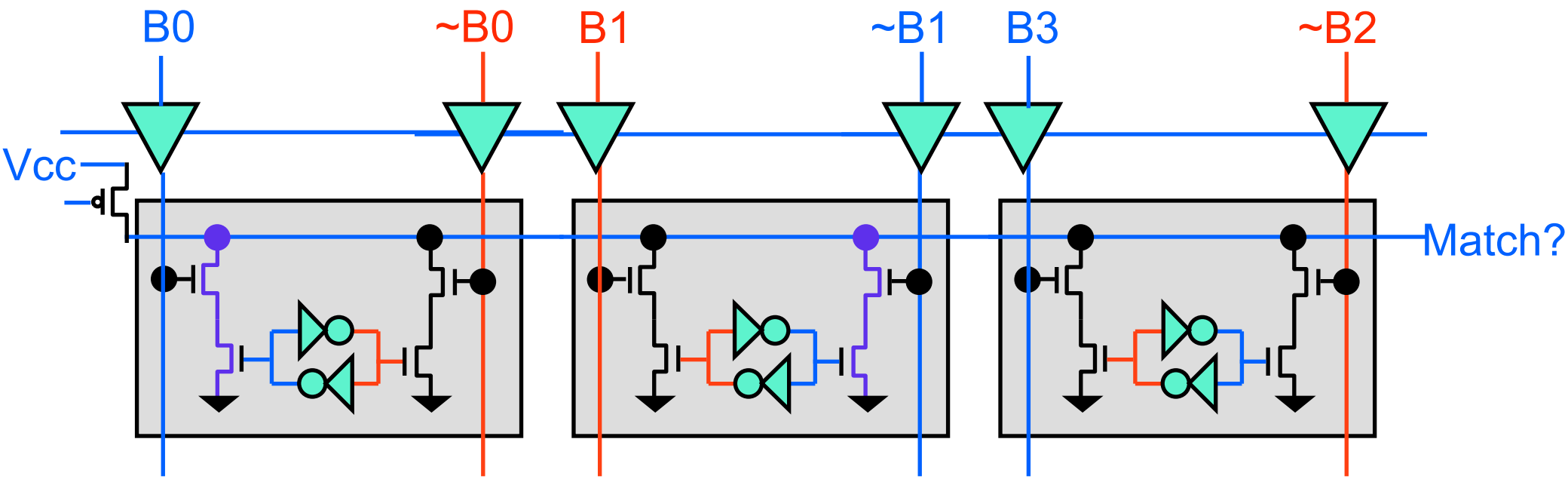
- CAM match port looks different from RAM r/w port
- Cells look similar
  - Note: Bit stored on right, ~Bit on left (opposite of inputs)
- Step 1: Precharge match line to 1 (first half of cycle)

CAM circuit



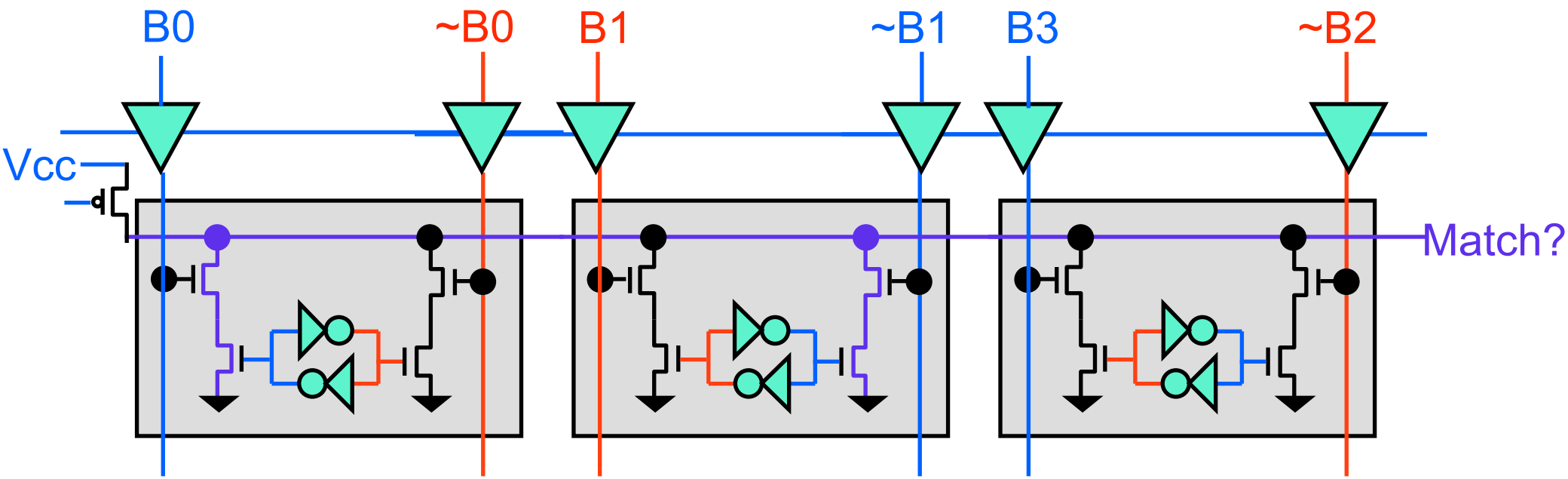
- CAM match port looks different from RAM r/w port
- Cells look similar
  - Note: Bit stored on right, ~Bit on left (opposite of inputs)
- Step 1: Precharge match line to 1 (first half of cycle)
- Step 2: Send data/~data down bit lines

CAM circuit



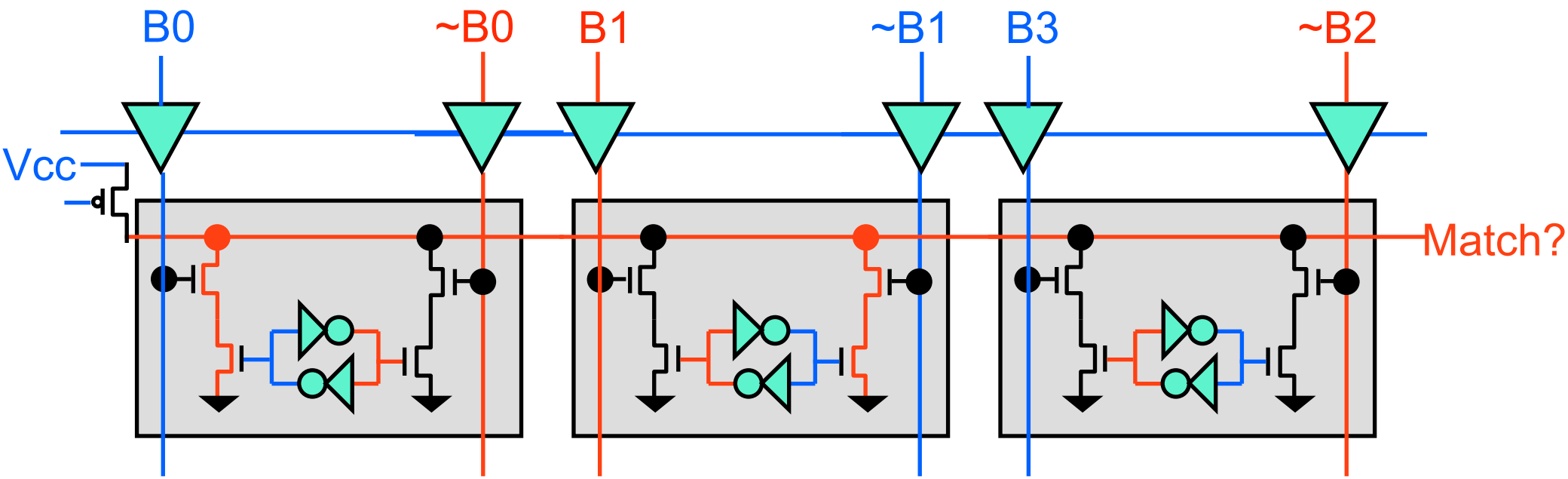
- CAM match port looks different from RAM r/w port
- Cells look similar
  - Note: Bit stored on right, ~Bit on left (opposite of inputs)
- Step 1: Precharge match line to 1 (first half of cycle)
- Step 2: Send data/~data down bit lines
  - Two 1s on same side (bit line != data) open NMOS path -> gnd

CAM circuit



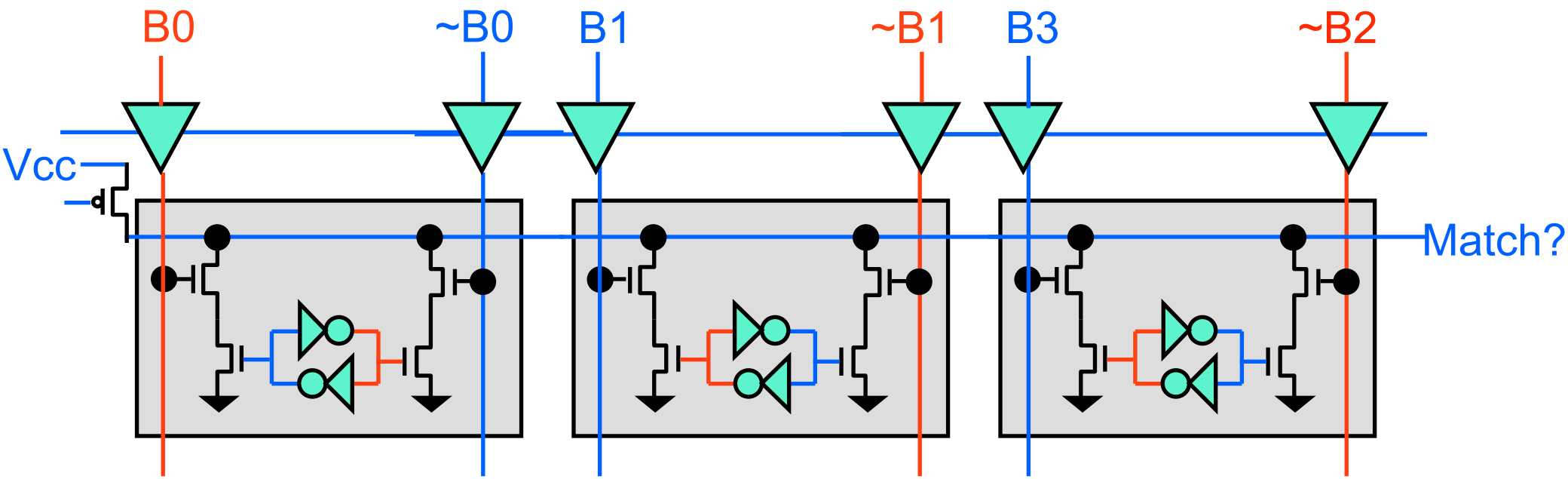
- CAM match port looks different from RAM r/w port
- Cells look similar
  - Note: Bit stored on right, ~Bit on left (opposite of inputs)
- Step 1: Precharge match line to 1 (first half of cycle)
- Step 2: Send data/~data down bit lines
  - Two 1s on same side (bit line != data) open NMOS path -> gnd
    - Drains match line 1->0

CAM circuit



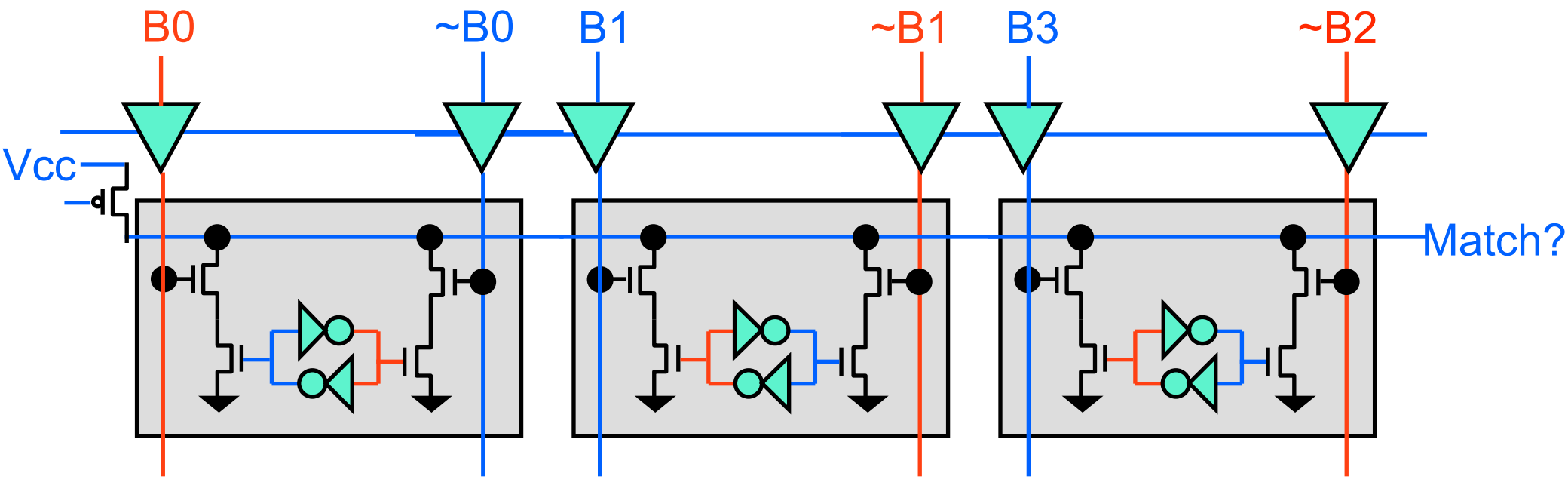
- CAM match port looks different from RAM r/w port
- Cells look similar
  - Note: Bit stored on right, ~Bit on left (opposite of inputs)
- Step 1: Precharge match line to 1 (first half of cycle)
- Step 2: Send data/~data down bit lines
  - Two 1s on same side (bit line != data) open NMOS path -> gnd
    - Drains match line 1->0

CAM circuit



- Note that if all bits match, each side has a 1 and a 0
  - One NMOS in the path from Match -> Gnd is closed
  - No conductive path -> Match keeps its charge @ 1

CAMs: Slow and High Power..



- CAMs are slow and high power
  - Pre-charge all, discharge most match lines every search
    - Pre-charge + discharge take time: capacitive load of match line
  - Bit lines have high capacitive load: Driving 1 transistor per row

ABC

- **Capacity**
  - + Decreases capacity misses
  - – Increases t_{hit}
- **Associativity**
  - + Decreases conflict misses
  - – Increases t_{hit}
- **Block size**
  - – Increases conflict misses
  - + Decreases compulsory misses
  - ± Increases or decreases capacity misses
  - Little effect on t_{hit}, may exacerbate t_{miss}
- How much they help depends...

Different Problems -> Different Solutions

- Suppose we have 16B, direct-mapped cache w/ 4B blocks
  - 4 sets
  - Examine some access patterns and think about what would help
  - Misses in red
- Access pattern A:
  - As is: 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26
  - 8B blocks? 0, 2, **4**, 6, 8, 10, **12**, 14, 16, 18, **20**, 22, 24, 26
  - 2-way assoc? 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26
- Access pattern B:
  - As is: 0, 128, 1, 129, 2, 130, 3, 131, 4, 132, 5, 133, 6
  - 8B blocks? 0, 128, 1, 129, 2, 130, 3, 131, 4, 132, 5, 133, 6
  - 2-way assoc? 0, 128, **1, 129, 2, 130, 3, 131,** 4, 132, **5, 133, 6**
- Access pattern C (All 3):
  - 0,20,40,60,48,36,24,12,1,21,41,61,49,37,25,13,2,22,42,62,50,38,…

Analyzing Misses: 3C Model (Hill)

- Divide cache misses into categories based on cause
  - **Compulsory**: block size is too small (i.e., address not seen before)
  - **Capacity**: capacity is too small
  - **Conflict**: associativity is too low

Different Problems -> Different Solutions

- Access pattern A: Compulsory misses
  - 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26
  - For misses, have not accessed that block
    - Size/associativity won't help (never had it)
    - Larger block -> include more data in one block -> more hits

- Recognizing compulsory misses
  - Never seen the **block** before

Different Problems -> Different Solutions

- Access pattern B: Conflict misses
  - 0, 128, 1, 129, 2, 130, 3, 131, 4, 132, 5, 133, 6
  - 0 and 128 map to same set (set 0): kick each other out ("conflict")
  - Larger block? No help
  - Larger cache? Only helps if MUCH larger (256 B instead of 16B)
  - Higher associativitiy? Fixes problem
    - Can have both 0 and 128 in set 0 at same time (different ways)

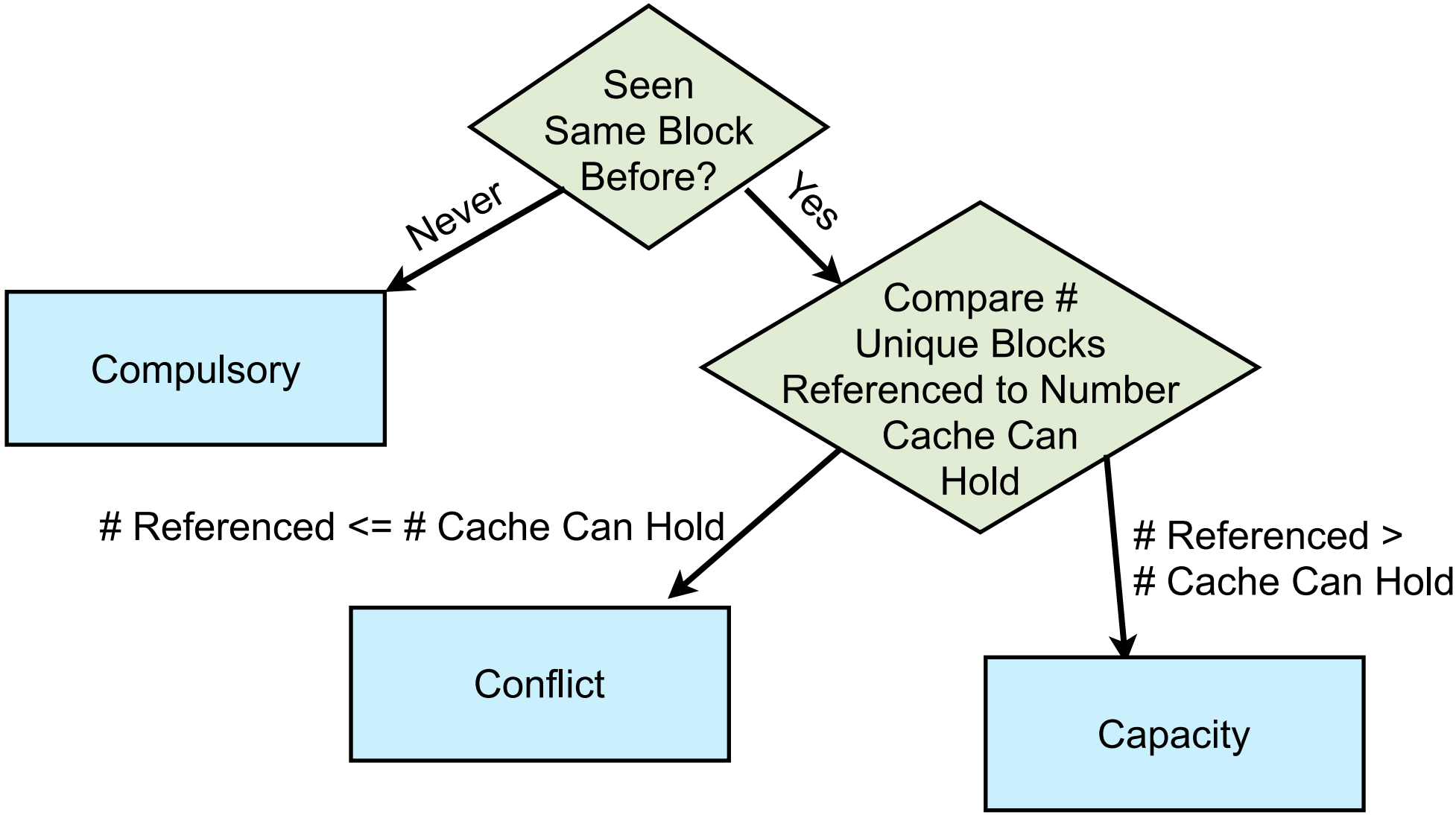
- Recognizing conflict misses:
  - Count unique blocks between last access and miss (inclusive)
  - Number of unique blocks <= number of blocks in cache? Conflict
    - Enough space to hold them all...
    - Just must be having set conflict

Different Problems -> Different Solutions

- Access pattern C: Capacity Misses
  - 0,20,40,60,48,36,24,12,1,21,41,61,49,37,25,13,2,22,42,62,50,38,…
  - Larger block size? No help
    - Even 16B block (entire cache) won't help
  - Associativity? No help... even at full assoc
    - After 0, 68, 40, 144: kick out 0 for 84
    - Kick out 68 for 152
    - Kick out 40 for 1...
  - Only solution: make cache larger
    - Doubling cache size turns all most misses into hits
      - A few compulsory misses remain
  - 0,20,40,60,48,36,24,12,1,21,41,61,49,37,25,13,2,22,42,62,50,38,…

- Recognizing Capacity Misses
  - Count unique blocks between last access and miss (inclusive)

Miss Categorization Flow Chart



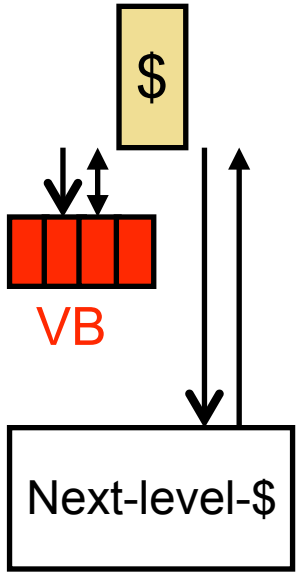
Two Optimizations

- **Victim buffer**: for conflict misses
  - Technically: reduces t_{miss} for these misses, doesn't eliminate them
  - Depends how you do your accounting

- **Prefetching**: for capacity/compulsory misses

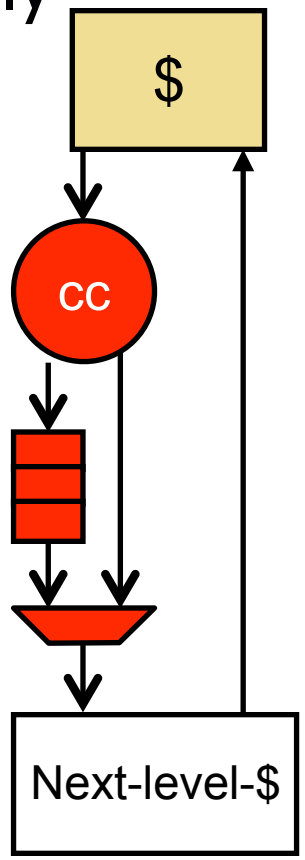
Victim Buffer

- Conflict misses: not enough associativity
  - High associativity is expensive, but also rarely needed
    - 3 blocks mapping to same 2-way set and accessed (XYZ)+
- **Victim buffer (VB)**: small FA cache (e.g., 4 entries)
  - Small so very fast
  - Blocks kicked out of cache placed in VB
  - On miss, check VB: hit ? Place block back in cache
  - 4 extra ways, shared among all sets
    + Only a few sets will need it at any given time
  - On cache fill path: reduces t_{miss}, no impact on t_{hit}
  + Very effective in practice



Prefetching

- **Prefetching**: put blocks in cache proactively/speculatively
  - In software: insert prefetch (non-binding load) insns into code
  - In hardware: cache controller generates prefetch addresses
- Keys: anticipate upcoming miss addresses accurately
  - **Timeliness**: initiate prefetches sufficiently in advance
  - **Accuracy**: don't evict useful data
  - Prioritize misses over prefetches
- Simple algorithm: **next block prefetching**
  - Miss address **X** → prefetch address **X+block-size**
  - Works for insns: sequential execution
    - What about non-sequential execution?
  - Works for data: arrays
    - What about other data-structures?
  - Address prediction is actively researched area

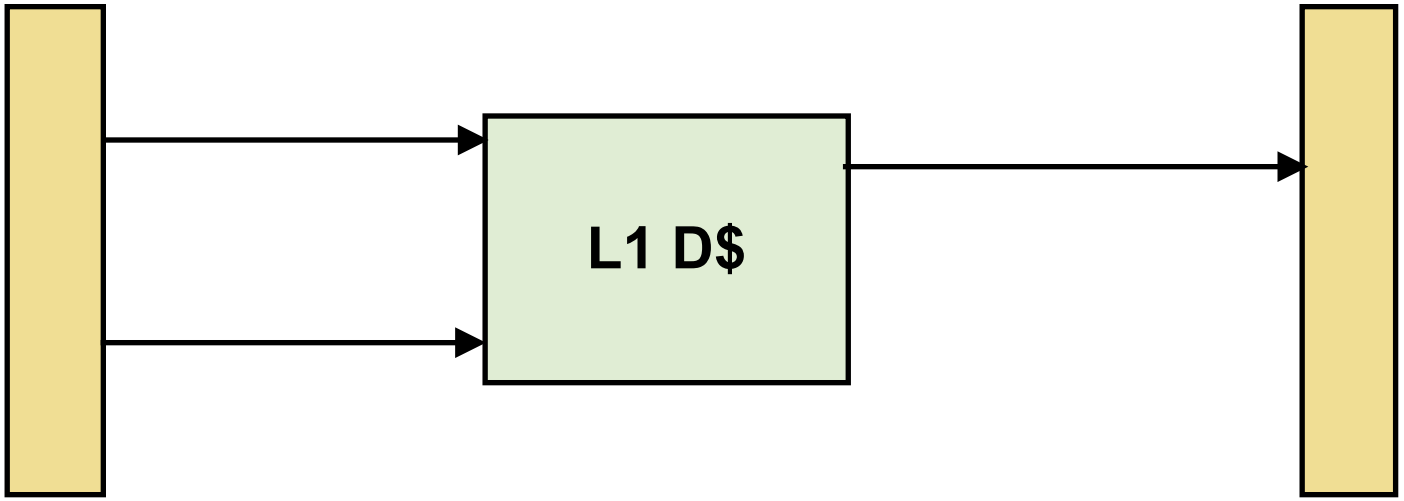


Write Issues

- So far we have looked at reading from cache
  - Insn fetches, loads
- What about writing into cache
  - Stores, not an issue for insn caches (why they are simpler)

- Several new issues
  - Must read tags first **before** writing data
    - Cannot be in parallel
  - Cache may have **dirty** data
    - Data which has been updated in this cache, but not lower levels
    - Must be written back to lower level before eviction

Recall Data Memory Stage of Datapath



- So far, have just assume D$ in Memory Stage...
  - Actually a bit more complex for a couple reasons...

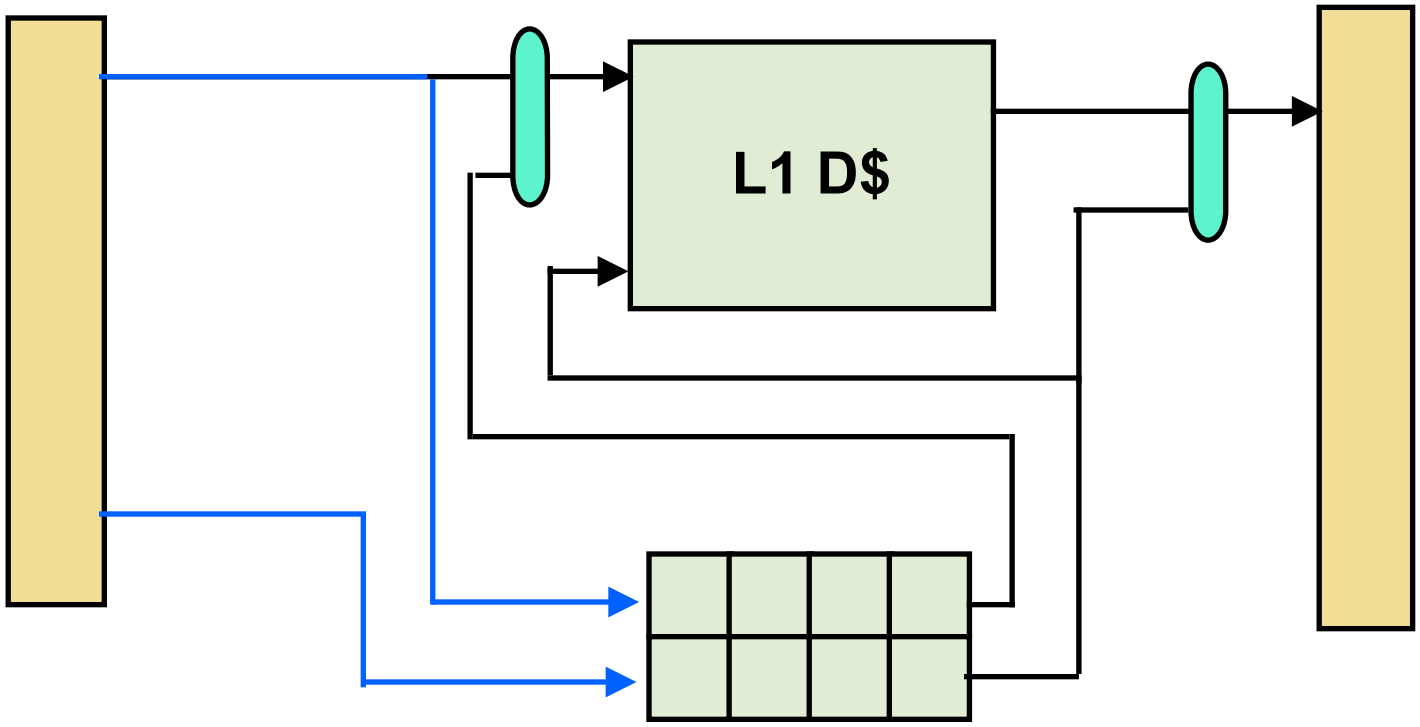
Reason 1: Store Misses

- Load instruction misses D$:
  - Have to stall datapath
  - Need missing data to complete instruction
  - (Fancier: stall at first consumer rather than load)
- Store instruction misses D$:
  - Stall?
  - Would really like not to
  - Store is writing the data
  - Need rest of block because we cannot have part of a block
    - Generally do not support "these bytes are valid, those are not"
  - How to avoid?

Reason 2: Serial Tag/Data Access

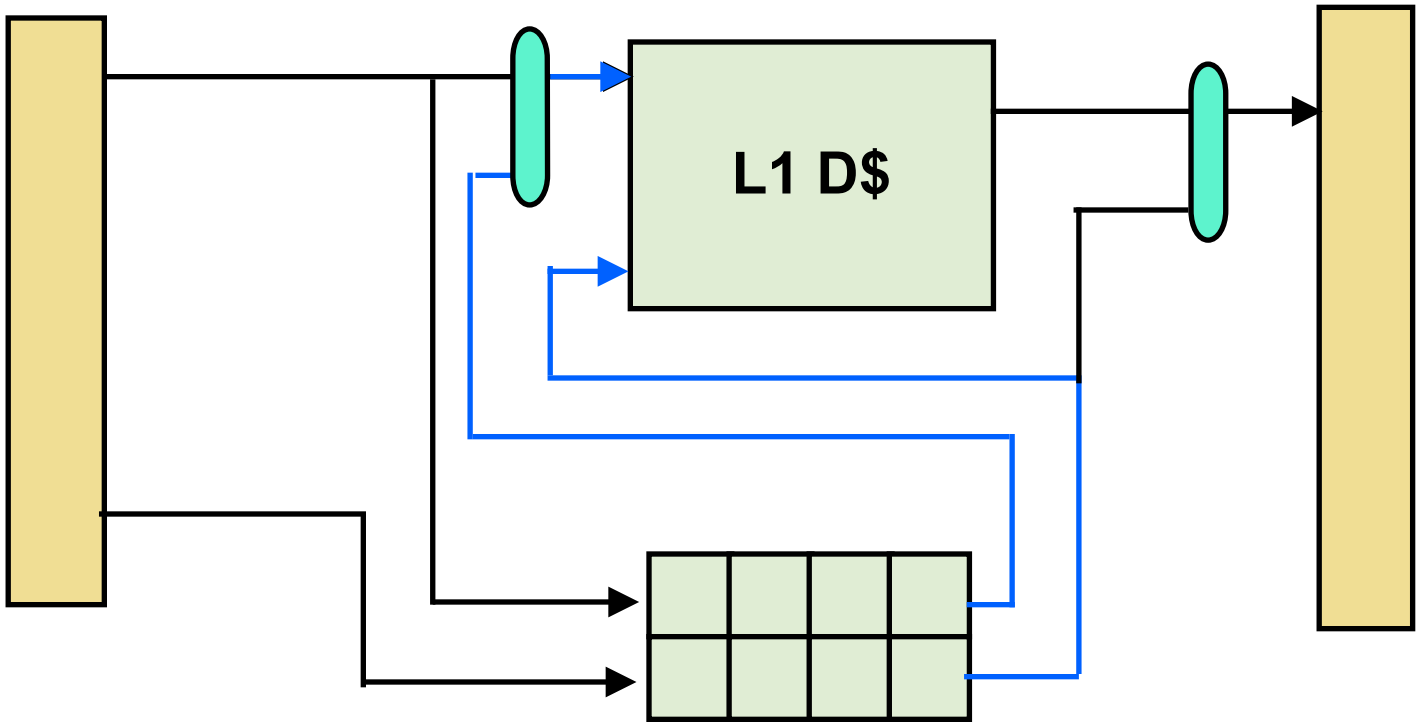
- Load can read tags/data in parallel
  - Read both SRAMs
  - Compare Tags -> Select proper way (if any)
- Stores cannot write tags/data in parallel
  - Read tags/write data array at same time??
    - How to know which way?
    - Or even if its a hit?
    - Incorrect -> overwrote data from somewhere else..
- Multi-cycle data-path:
  - Stores take an extra cycle? Increase CPI
- Pipelined data-path:
  - Tags in one stage, Data in the next?
  - Works for stores, but loads serialize tags/data -> higher CPI

Store Buffer



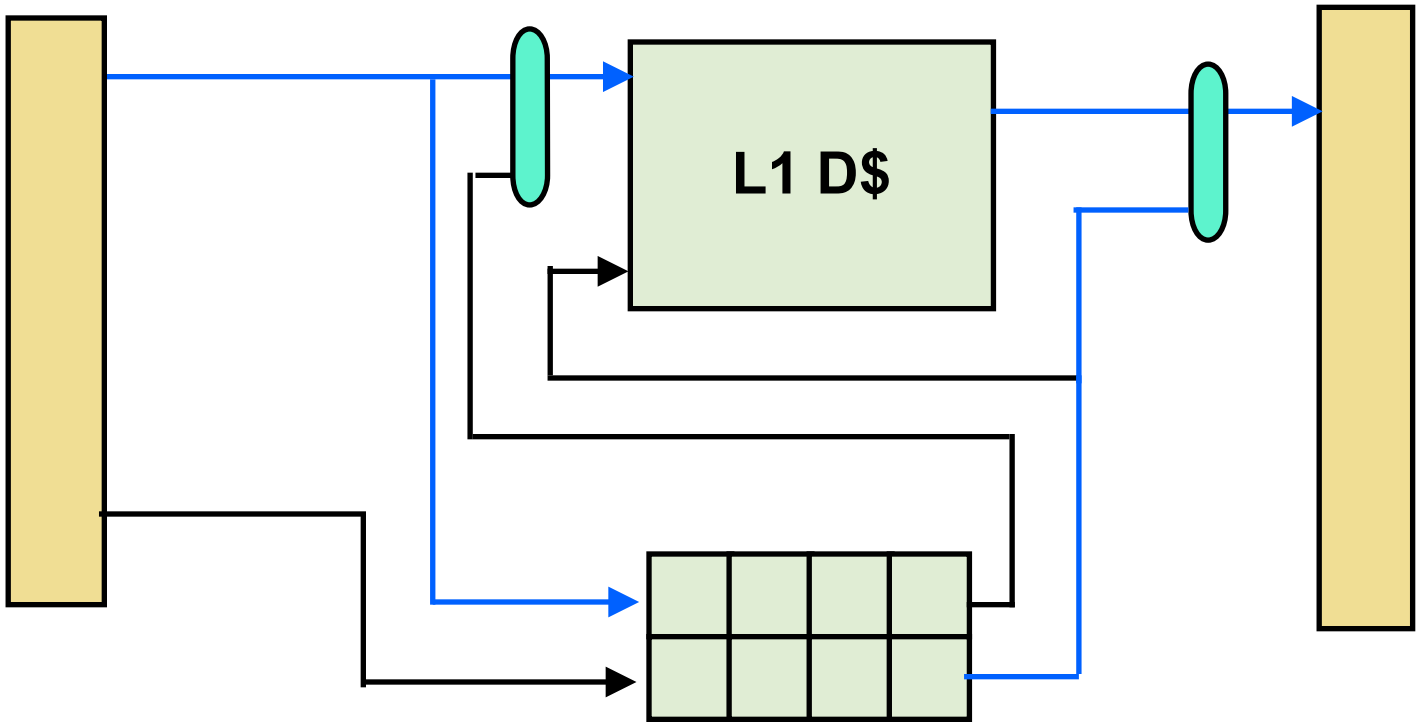
- Stores write into a **store buffer**
  - Holds address, size, data, of stores

Store Buffer



- Stores write into a **store buffer**
  - Holds address, size, data, of stores
  - Store data written from store buffer into cache
    - Miss? Data stays in buffer until hit

Store Buffer



- Loads search store buffer for matching store
  - Match? **Forward** data from the store
  - No match: Use data from D$
- Addresses are CAM: allow search for match

Store Buffer

- How does this resolve our issues?

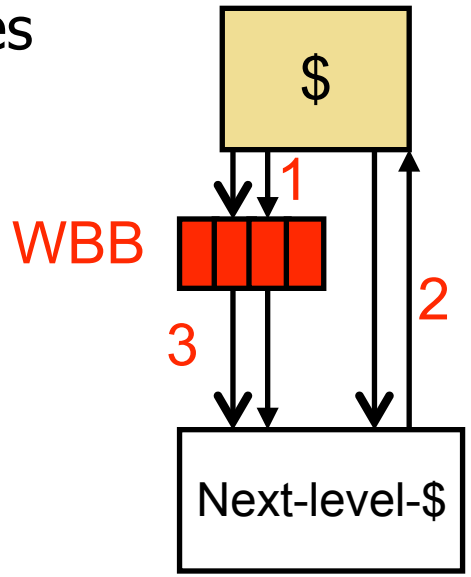
- Issue 1: Store misses
  - Stores write to store buffer and are done
  - FSM writes stores into D$ from store buffer
  - Misses stall store buffer -> D$ write (but not pipeline)
  - Pipeline will stall on full store buffer
- Issues 2: Tags -> Data
  - FSM that writes stores to D$ can check tags... then write data
  - Decoupled from data path's normal execution
    - Can happen whenever loads are not using the D$

Write Propagation

- When to propagate new value to (lower level) memory?
- **Write-thru**: immediately
  - Requires additional bus bandwidth
  - Not common
- **Write-back**: when block is replaced
  - Blocks may be **dirty** now
  - Dirty bit (in tag array)
    - Cleared on fill
    - Set by a store to the block

Write Back: Dirty Misses

- Writeback caches may have **dirty misses**:
  - Victim block (one to be replaced) is dirty
    - Must first writeback to next level
    - Then request data for miss
    - Slower :(
  - Solution:
    - Add a buffer on back side of cache: **writeback buffer**
      - Small full associative buffer, holds a few lines
    - Request miss data immediately
    - Put dirty line in WBB
    - Writeback later



Brief History of DRAM

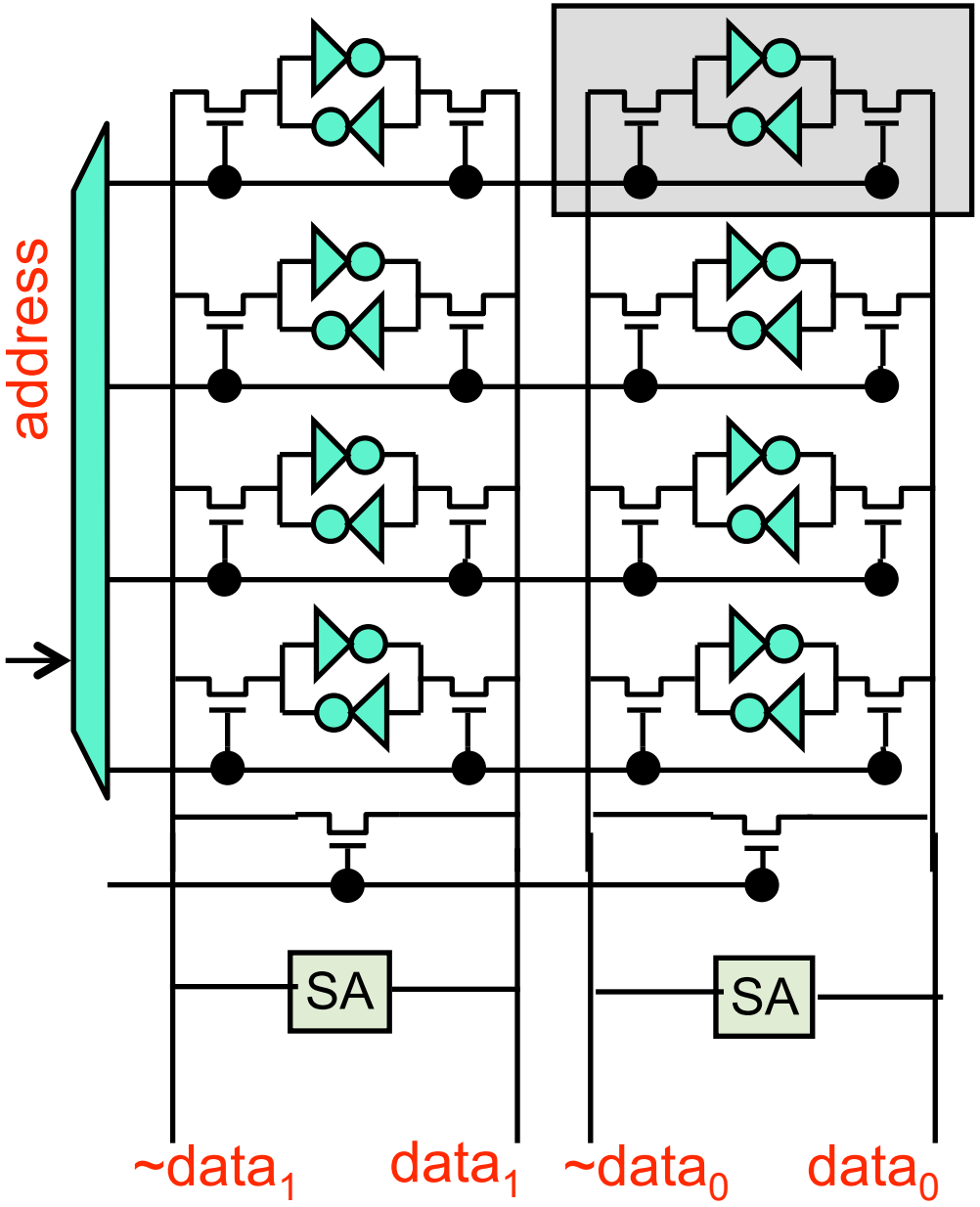
- DRAM (memory): a major force behind computer industry
  - Modern DRAM came with introduction of IC (1970)
  - Preceded by magnetic "core" memory (1950s)
    - More closely resembles today's disks than memory
  - And by mercury delay lines before that (ENIAC)
    - Re-circulating vibrations in mercury tubes

"the one single development that put computers on their feet was the invention of a reliable form of memory, namely the core memory… It's cost was reasonable, it was reliable, and because it was reliable it could in due course be made large"

Maurice Wilkes

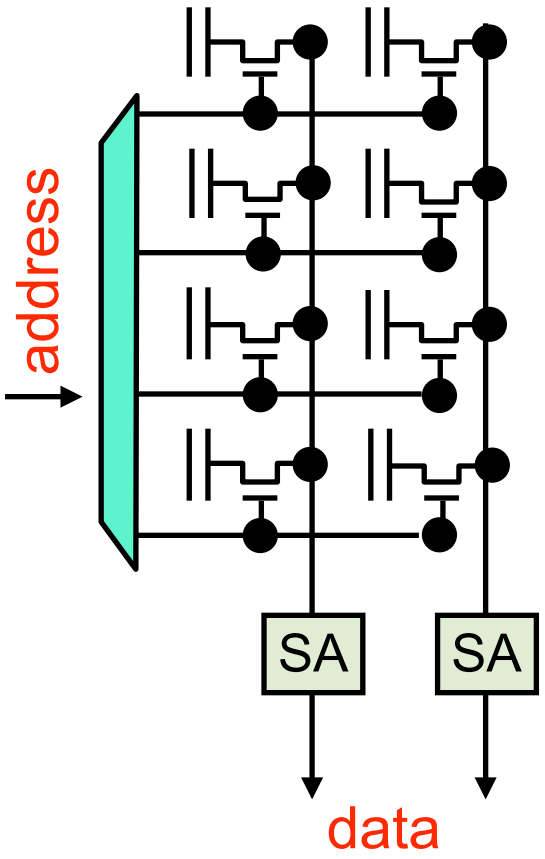
Memoirs of a Computer Programmer, 1985

SRAM



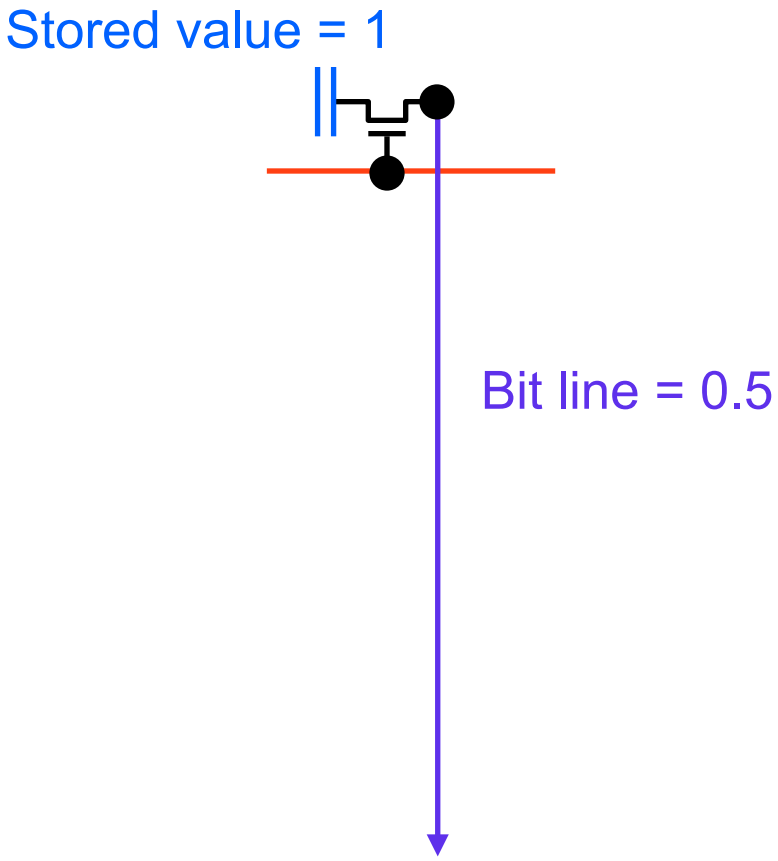
- SRAM: "6T" cells
  - 6 transistors per bit
    - 4 for the CCI
    - 2 access transistors
- **Static**
  - CCIs hold state
- To read
  - Equalize, swing, amplify
- To write
  - Overwhelm

DRAM



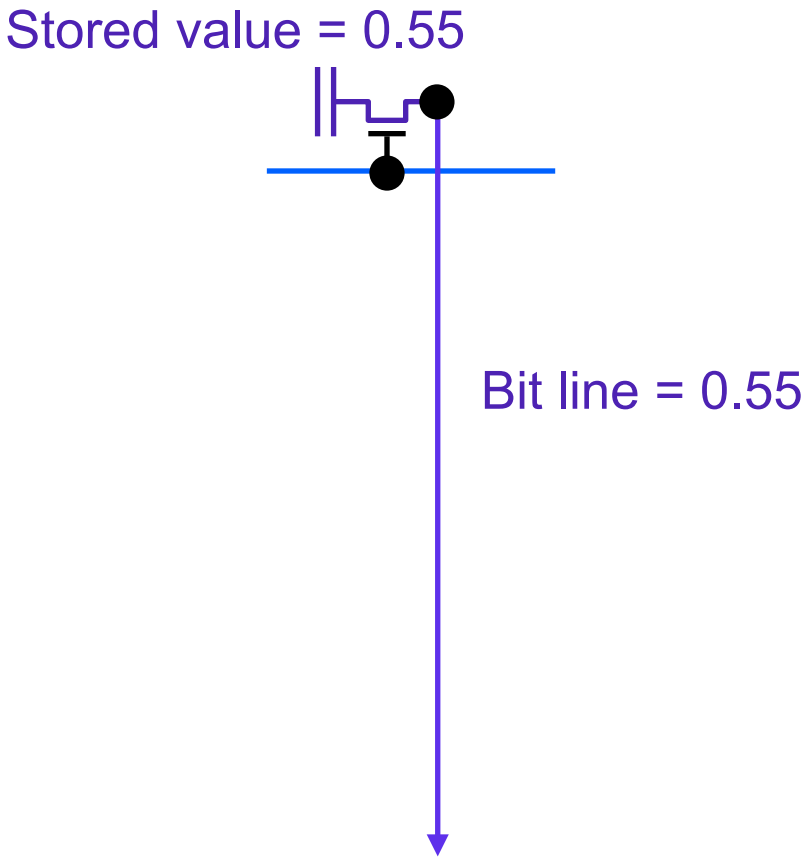
- **DRAM**: dynamic RAM
  - Bits as capacitors
  - Transistors as ports
  - "1T" cells: one access transistor per bit
- **"Dynamic"** means
  - Capacitors not connected to pwr/gnd
  - Stored charge decays over time
  - Must be explicitly refreshed
- Designed for density
  - \+ ~6–8X denser than SRAM
  - – But slower too

DRAM Read (simplified version)



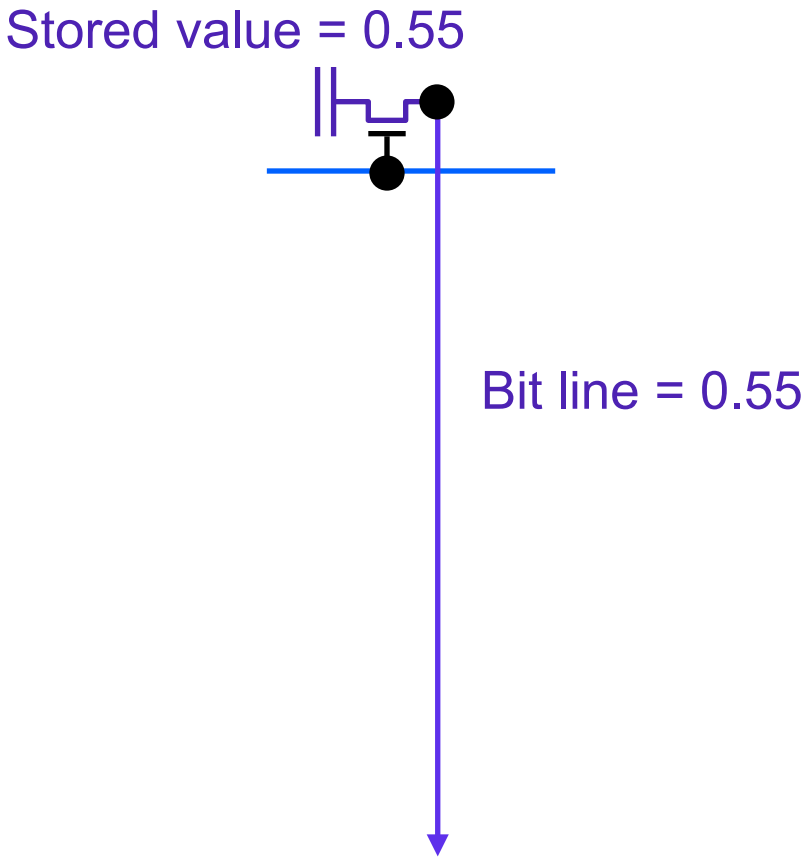
- Bit line pre-charged to 0.5 (think: pipe half full)
- Storage at 1 (think: tank full of water)

DRAM Read (simplified version)



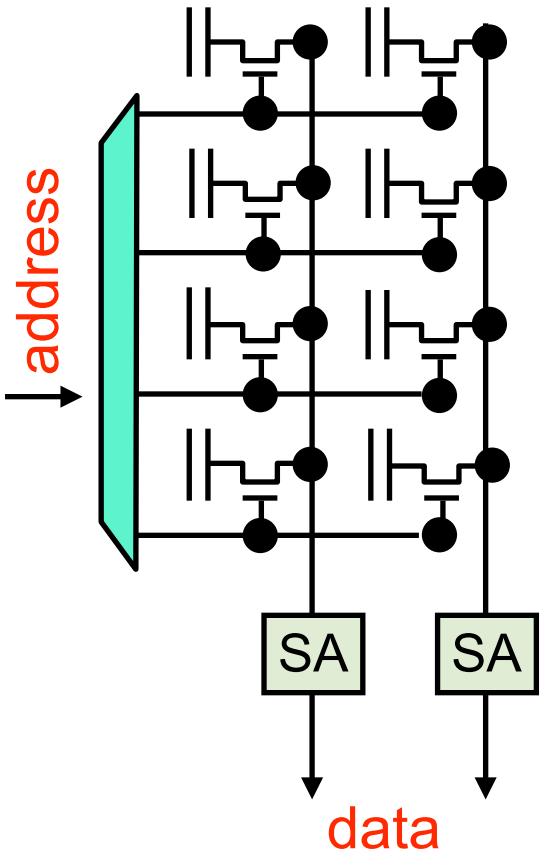
- Bit-line and capacitor equalize
  - Think: opening valve between pipe + tank
- Settle out a bit above 0.5 if 1 was stored
  - A bit less if 0 was stored

DRAM Read (simplified version)



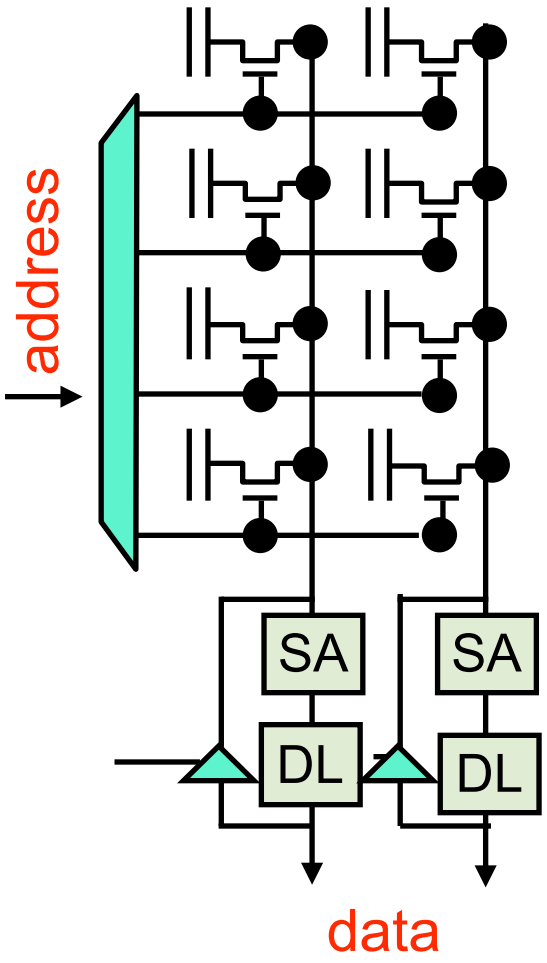
- Destroyed the stored value in the process
  - Could not read this again: change too small to detect

DRAM Operation I



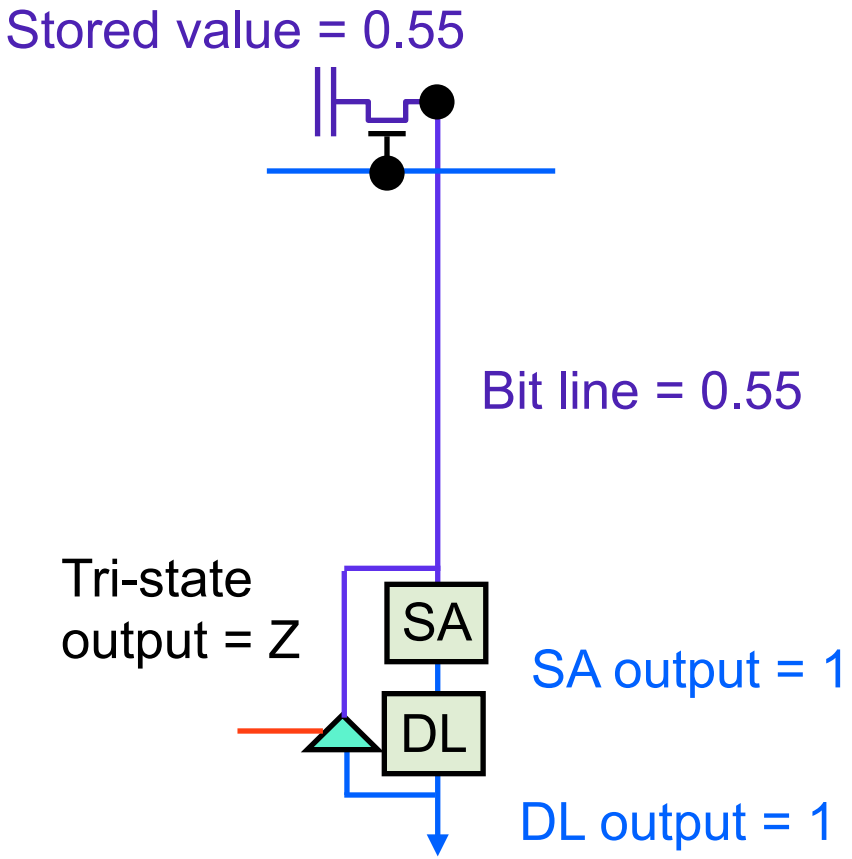
- Sense amps detect small swing
  - Amplify into 0 or 1
- This read: very slow
  - Why? No Vcc/Gnd connection in storage
- Need to deal with destructive reads:
  - Might want to read again...
- Also need to be able to write

DRAM Operation I



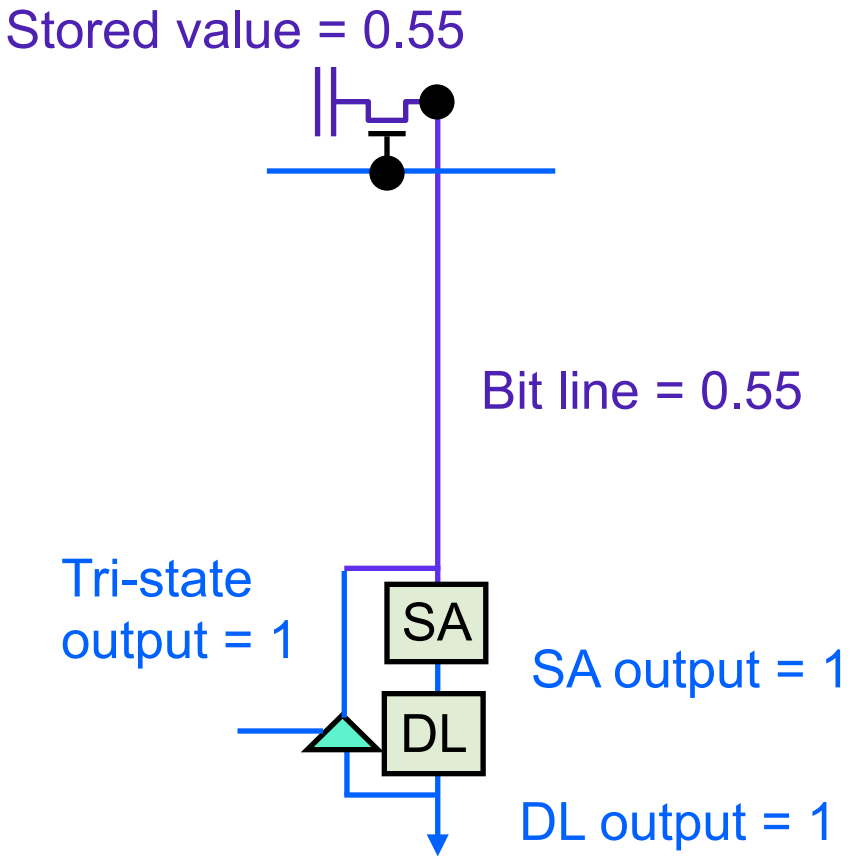
- Add some d-latches (**row buffer**)
  - Ok to use d-latches, not DFFs
  - No path from output->input when enabled
- Also add a tri-state path back
  - From the d-latch to the bit-line
  - Can drive the output of the d-latch onto bit lines
    - After we read, drive the value back
    - "Refill" (or re-empty) the capacitor

DRAM Read (better version)



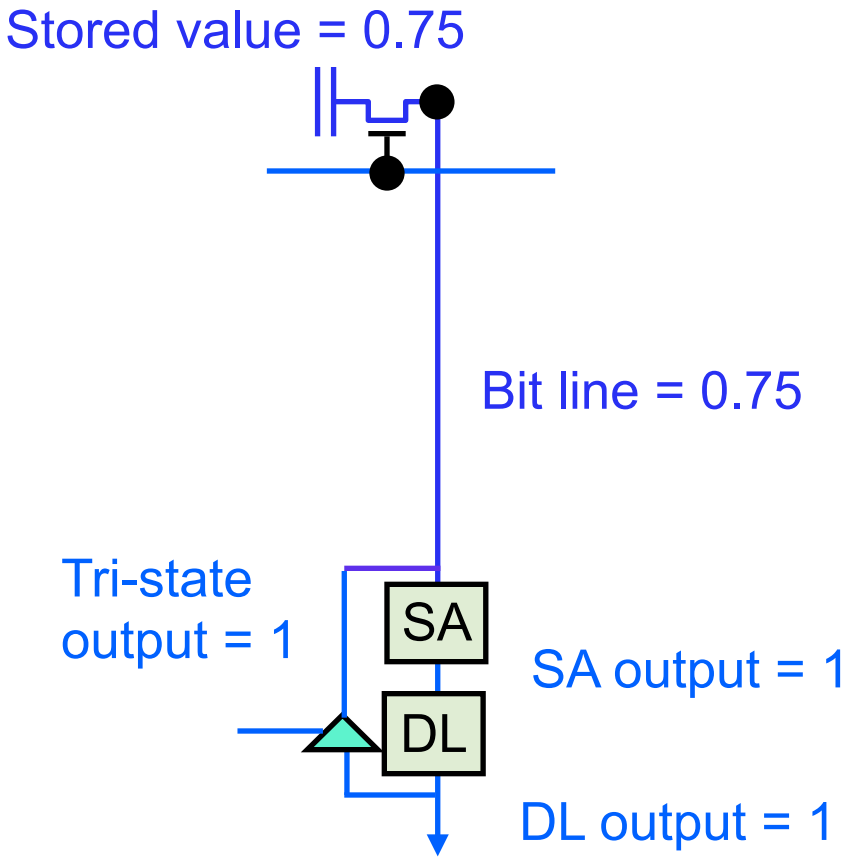
- SA amplifies 0.55 -> 1
- DL is enabled: latches the 1
- Tri-state disabled

DRAM Read (better version)



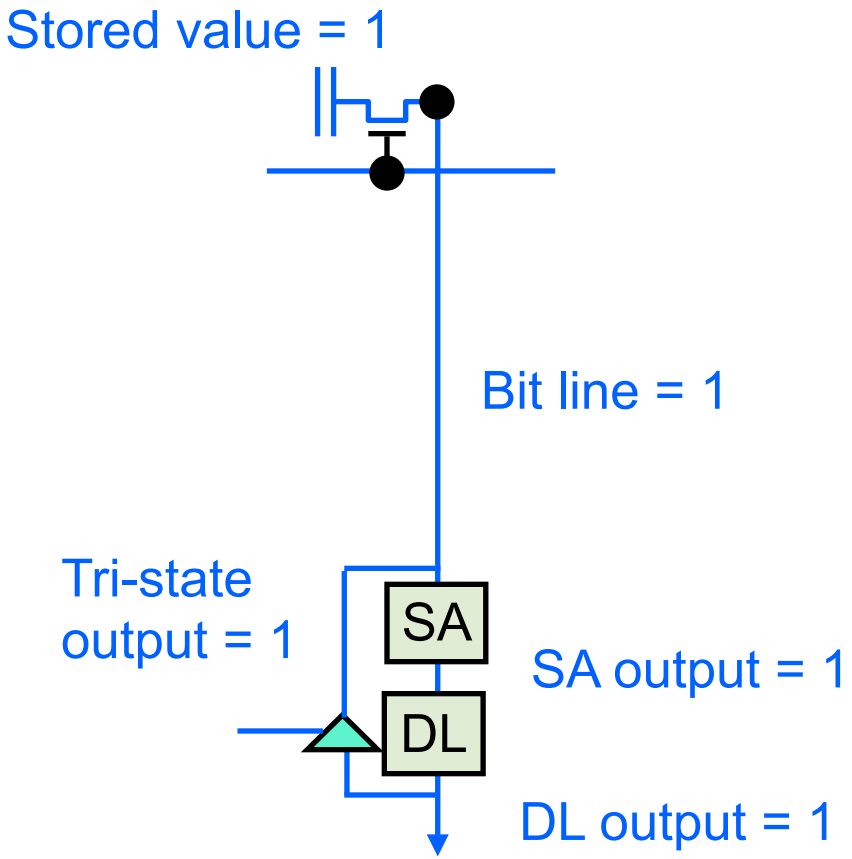
- Enable tri-state
  - Drives 1 back up bit-line

DRAM Read (better version)



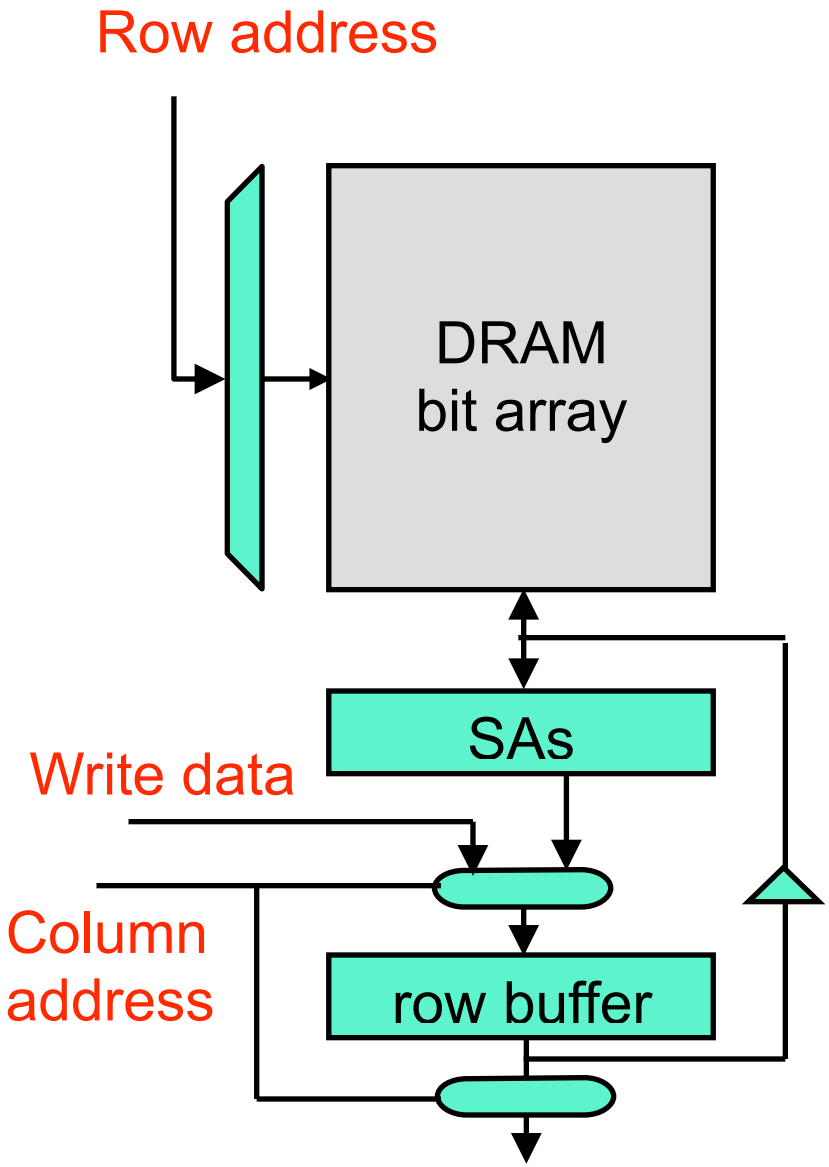
- Enable tri-state
  - Drives 1 back up bit-line
  - Starts to push value back up towards 1 (takes time)

DRAM Read (better version)



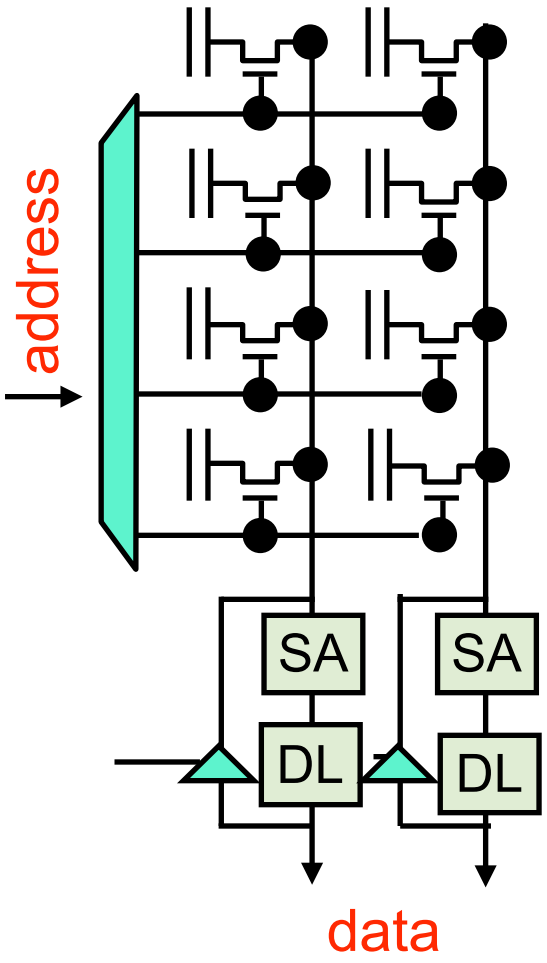
- Enable tri-state
  - Drives 1 back up bit-line
  - Starts to push value back up towards 1 (takes time)
  - Eventually restores value.

DRAM Operation



- Open row (read bits -> row buffer)
- Read "columns"
  - Mux selects right part of RB
  - Send data on bus -> processor
- Write "columns"
  - Change values in dlatches
- May read/write multiple columns
- Close row
  - Close access transistors
  - Pre-charge bit lines
- Row must remain open long enough
  - Must fully restore capacitors

DRAM Refresh



- DRAM periodically refreshes all contents
  - Loops through all rows
    - Open row (read -> RB)
    - Leave row open long enough
    - Close row
  - 1–2% of DRAM time occupied by refresh

So a DRAM array walks into a bar...

Aside: Non-Volatile CMOS Storage

- Before we leave the subject of CMOS storage technology…
- Another important kind: **flash**
  - "Floating gate": no conductor/semi-conductor
    - Quantum tunneling involved in writing it
    - Effectively no leakage (key feature)
    - **Non-volatile**: remembers state when power is off
  - Slower than DRAM
  - Wears out with writes
    - Eventually writes just do not work

Memory Bus

- **Memory bus**: connects CPU package with main memory
  - Has its own clock
    - Typically slower than CPU internal clock: 100–500MHz vs. 3GHz
    - SDRAM operates on this clock
  - Is often itself internally pipelined
    - Clock implies bandwidth: 100MHz → start new transfer every 10ns
    - Clock doesn't imply latency: 100MHz !→ transfer takes 10ns
  - Bandwidth is more important: determines peak performance

Memory Latency and Bandwidth

- Nominal **clock frequency** applies to CPU and caches

- Careful when doing calculations
  - Clock frequency increases don't reduce memory or bus latency
  - May make misses come out faster
    - At some point memory bandwidth may become a **bottleneck**
    - Further increases in clock speed won't help at all

Clock Frequency Example

- Baseline setup
  - Processor clock: 1GHz.
    - 20% loads, 15% stores, 20% branches, 45% ALU
    - Branches: 3, ALU/stores 4, Loads 5 + misses
  - L1 D$: t_{hit} = 1 cycle, 10% miss $t_{avg} = 1 + 0.10 * 30 = 4$
  - L2$: t_{hit} = 20 cycles, 5% miss $t_{avg} = 20 + 0.05 * 200 = 30$
  - Memory: 200 cycles

Average load latency = 8

CPI = 0.2 * 8 + 0.15 * 4 + 0.2 * 3 + 0.45 * 4 = 4.6

Performance = 217 MIPS

Clock Frequency Example

- Baseline setup
  - Processor clock: **2**GHz.
    - 20% loads, 15% stores, 20% branches, 45% ALU
    - Branches: 3, ALU/stores 4, Loads 5 + misses
  - L1 D$: t_{hit} = 1 cycle, 10% miss $t_{avg} = 1 + 0.10 * \mathbf{40} = \mathbf{5}$
  - L2$: t_{hit} = 20 cycles, 5% miss $t_{avg} = 20 + 0.05 * \mathbf{400} = \mathbf{40}$
  - Memory: **4**00 cycles

Average load latency = **9**

CPI = 0.2 * **9** + 0.15 * 4 + 0.2 * 3 + 0.45 * 4 = **4.8**

Performance = **417** MIPS (91% speedup, for 100% freq increase)

Clock Frequency Example

- Baseline setup
  - Processor clock: **4**GHz.
    - 20% loads, 15% stores, 20% branches, 45% ALU
    - Branches: 3, ALU/stores 4, Loads 5 + misses
  - L1 D$: t_{hit} = 1 cycle, 10% miss $t_{avg} = 1 + 0.10 * \mathbf{60} = \mathbf{7}$
  - L2$: t_{hit} = 20 cycles, 5% miss $t_{avg} = 20 + 0.05 * \mathbf{800} = \mathbf{60}$
  - Memory: **8**00 cycles

Average load latency = **11**

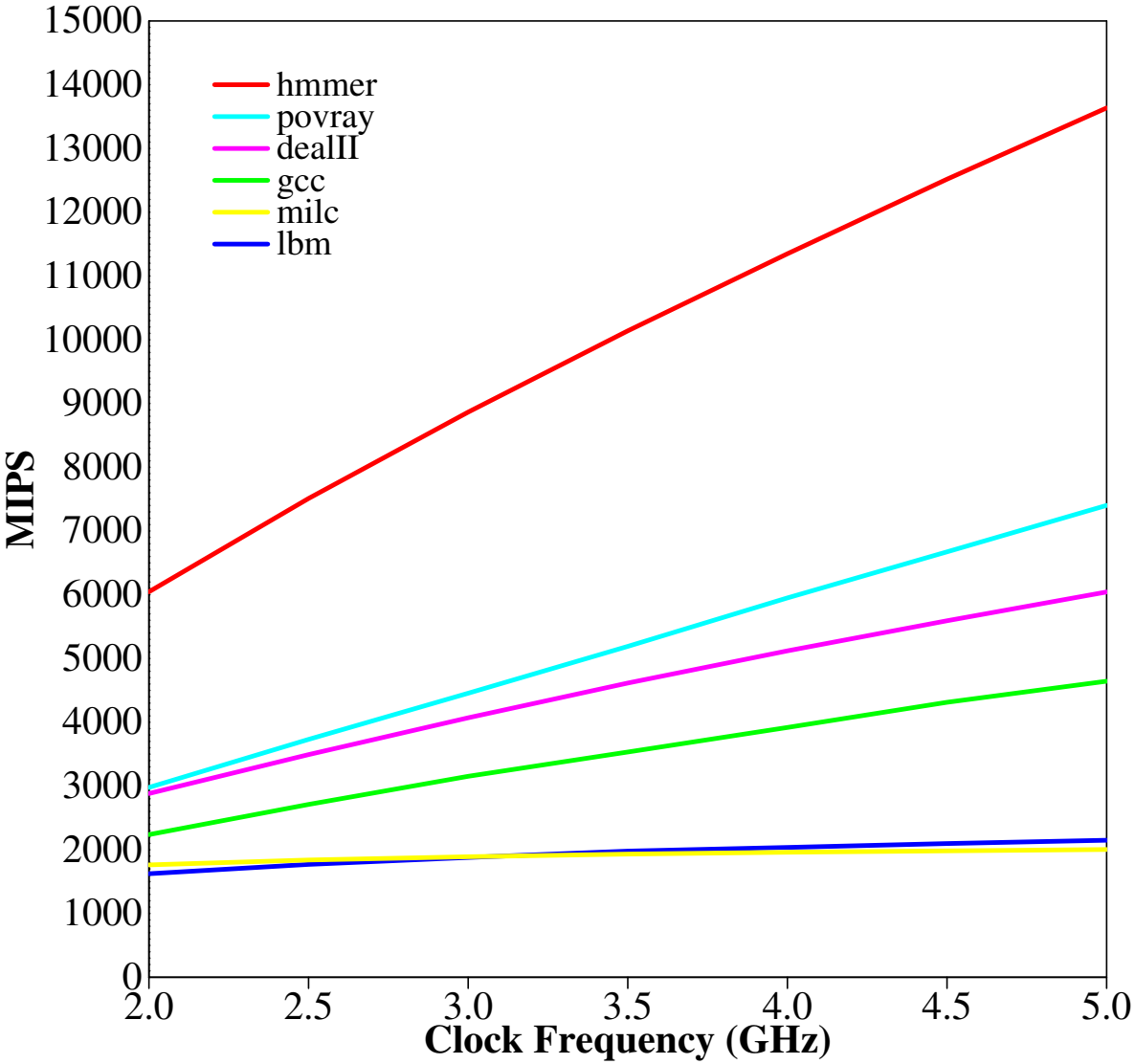
CPI = 0.2 * **11** + 0.15 * 4 + 0.2 * 3 + 0.45 * 4 = **5.2**

Performance = **769** MIPS (84% speedup, for 100% freq increase)

Actually a Bit Worse..

- Only looked at D$ miss impact
  - Ignored store misses: assumed storebuffer can keep up
- Also have I$ misses
- At some point, become bandwidth constrained
  - Effectively makes t_{miss} go up (think of a traffic jam)
  - Also makes things we ignored matter
    - Storebuffer may not be able to keep up as well -> store stalls
  - Data we previously prefetched may not arrive in time
    - Effectively makes %miss go up

Clock Frequency and Real Programs

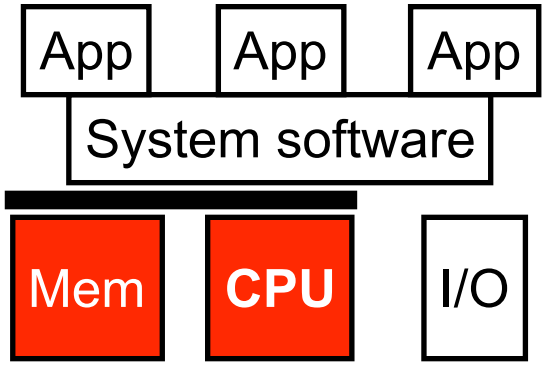


Detailed Simulation Results
- Includes all caches, bandwidth,...
- Has L3 on separate clock
- Real programs
- 2.0 Ghz -> 5.0 Ghz (150% increase)

hmmer:
- Very low %miss
- Good performance for clock
- **125%** speedup

lbm, milc:
- Very high %miss
- Not much performance gained
- lbm: **32%**
- milc: **14%**

Summary



- $t_{avg} = t_{hit} + \%_{miss} * t_{miss}$
  - t_{hit} and $\%_{miss}$ in one component? Difficult
- Memory hierarchy
  - Capacity: smaller, low t_{hit} → bigger, low$\%_{miss}$
    - 10/90 rule, temporal/spatial locality
  - Technology: expensive→cheaper
    - SRAM →DRAM→Disk: reasonable total cost
- Organizing a memory component
  - ABC, write policies
  - 3C miss model: how to eliminate misses?
- Technologies:
  - DRAM, SRAM, Flash