ECE 550D
Fundamentals of Computer Systems and Engineering

Fall 2023

Memory Hierarchy

Xin Li & Dawei Liu

Duke Kunshan University

Slides are derived from work by
Andrew Hilton, Tyler Bletsch and Rabih Younes (Duke)

Cache Miss Paper Simulation

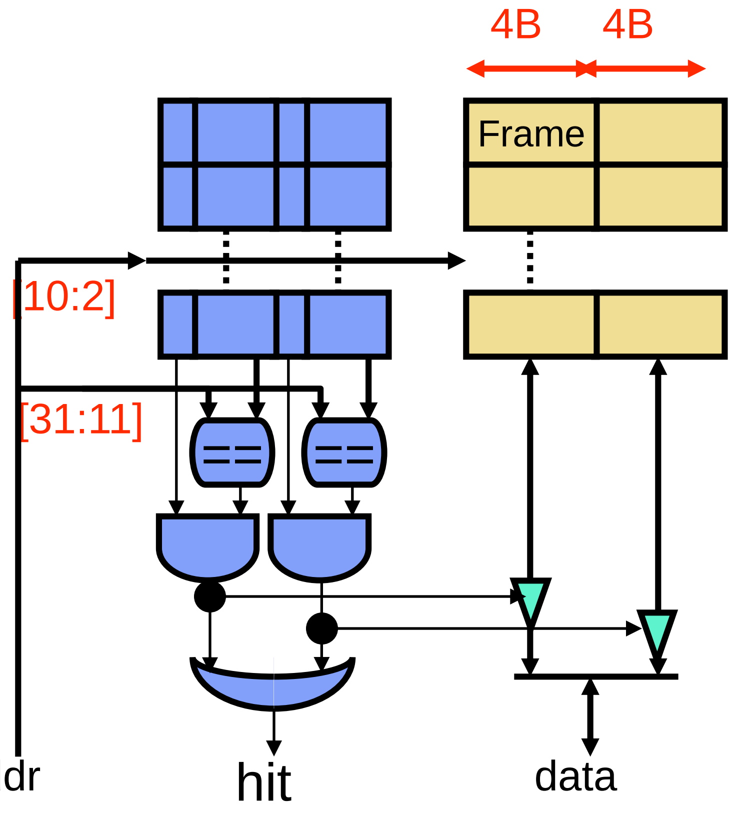
- 8B cache, 4B blocks -> 2 sets

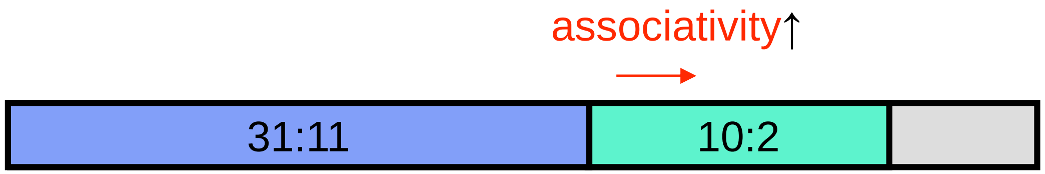
Address	Tag	Index	Offset	Set 0	Set 1	Result
C	1	1	0	invalid	0	Miss
E	1	1	2	invalid	1	Hit
8	1	0	0	invalid	1	Miss
3	0	0	3	1	1	**Miss**
8	1	0	0	0	1	**Miss**
0	0	0	0	1	1	Miss
8	1	0	0	0	1	Miss
4	0	1	0	1	1	Miss
6	0	1	2	1	0	**Hit**

- 8 (1000) and 3 (0011): same set
- 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 a 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: 16b address, 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

Tag = 123

Index = 1

Access address 0x1234 = 0001 0010 0011 01 00

Offset = 0

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

Access address 0x2234 = 0001 0010 0011 01 00

Tag = 223

Index = 1

Offset = 0

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

Access address 0x1234 = 0001 0010 0011 0100 0000

Tag = 123

Index = 1

Offset = 0

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

- 4b address, 8B cache, 2B blocks, 2 ways, 2 sets

Offset: $\log_2 2 = 1$ bit

Index: $\log_2 2 = 1$ bit

Tag: $4 - 1 - 1 = 2$ bit

- What happens for each request?

Cache Miss Paper Simulation

- 4b address, 8B cache, 2B blocks, 2 ways, 2 sets

				Set 0		Set 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?

Cache structure math summary

- Given capacity, block_size, ways (associativity), and word_size.
- Cache parameters:
  - num_frames = capacity / block_size
  - sets = num_frames / ways = capacity / block_size / ways
- Address bit fields:
  - offset_bits = $\log_2$(block_size)
  - index_bits = $\log_2$(sets)
  - tag_bits = word_size - index_bits - offset_bits
- Numeric way to get offset/index/tag from address:
  - block_offset = addr % block_size
  - index = (addr / block_size) % sets
  - tag = addr / (sets*block_size)

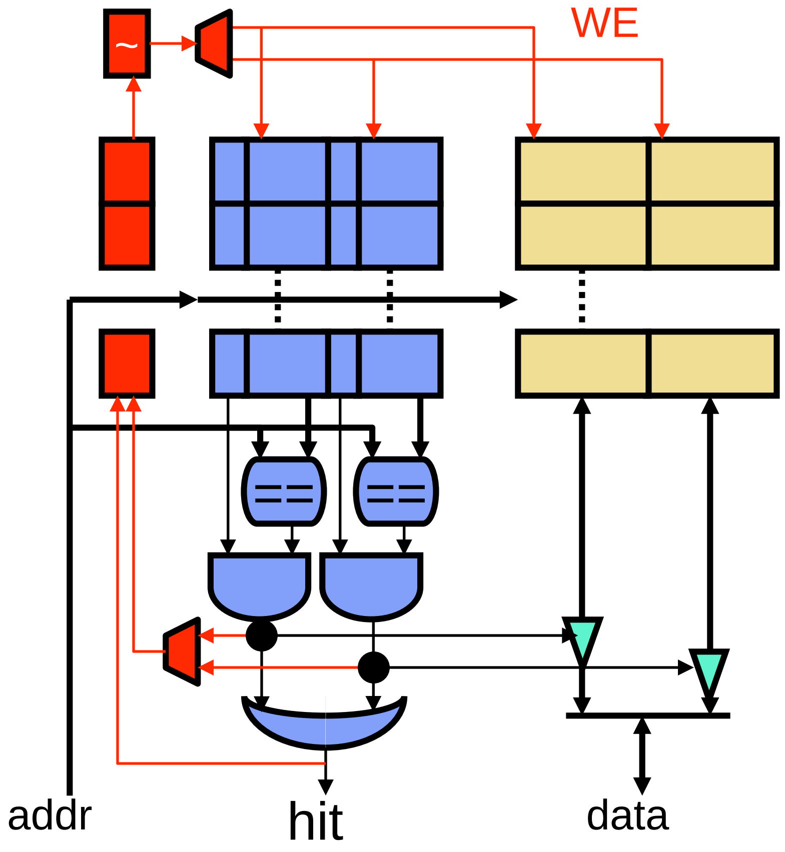
Replacement Policies

- 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

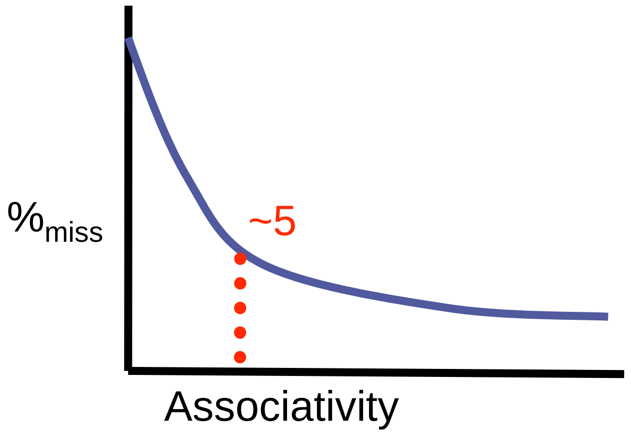
NMRU Implementation

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



Associativity And Performance

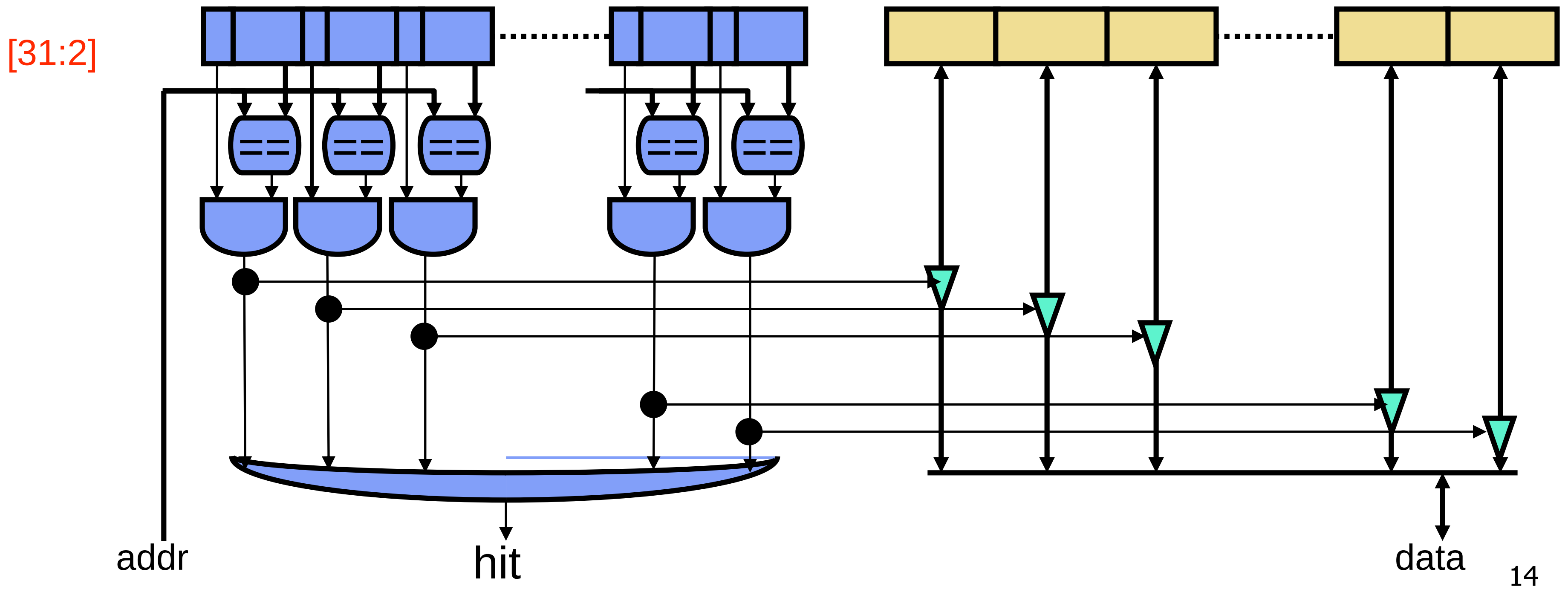
- The associativity game
  - + Higher associative caches have lower $\%_{miss}$
  - – t_{hit} increases
    - Additional logic at output
    - But not much for low associativities (2,3,4,5)
  - t_{avg}?



- 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

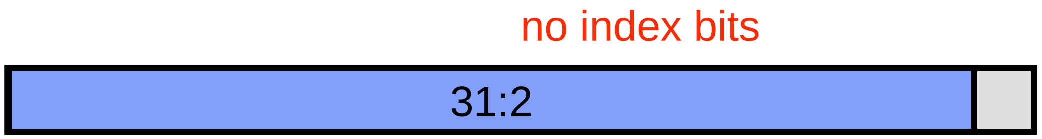
Full Associativity

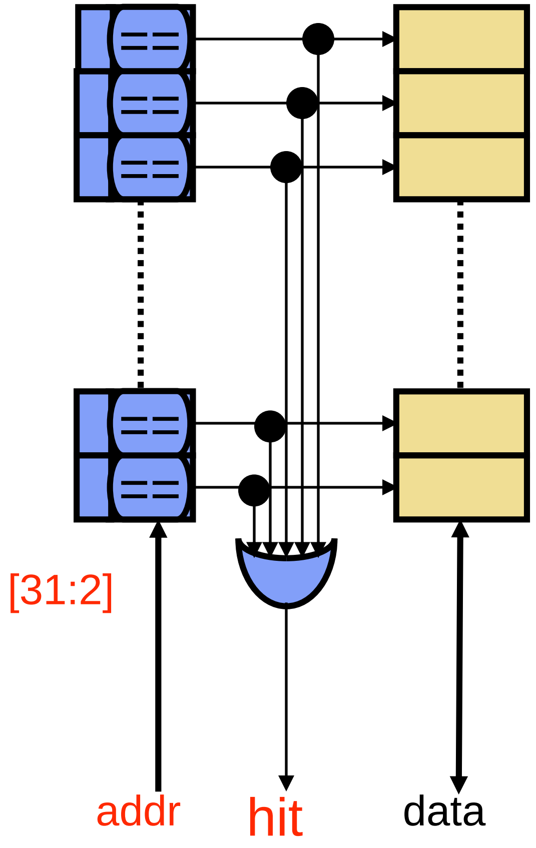
- How to implement full (or at least high) associativity?
  - This way is terribly inefficient
  - 1K matches are unavoidable, but 1K data reads + 1K-to-1 mux?



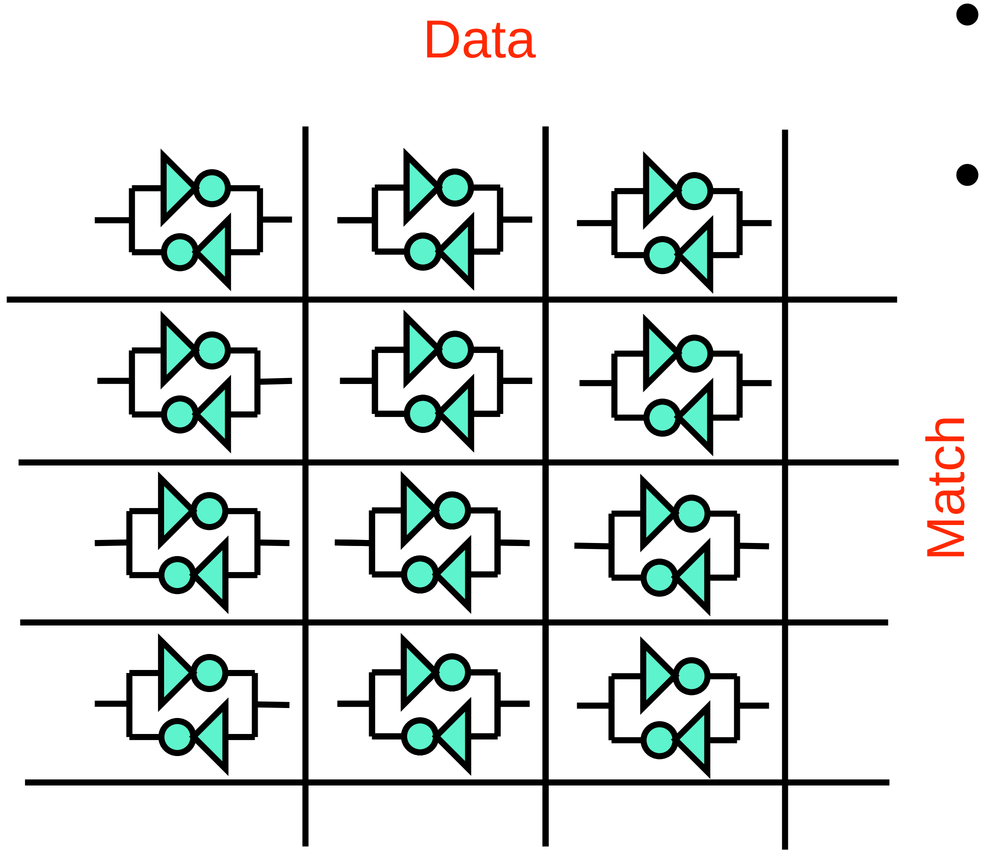
Full-Associativity with CAMs

- **CAM**: content-addressable memory
  - Array of words with built-in comparators
  - Input is data (tag)
  - Output is 1H encoding of matching slot
- Fully associative 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



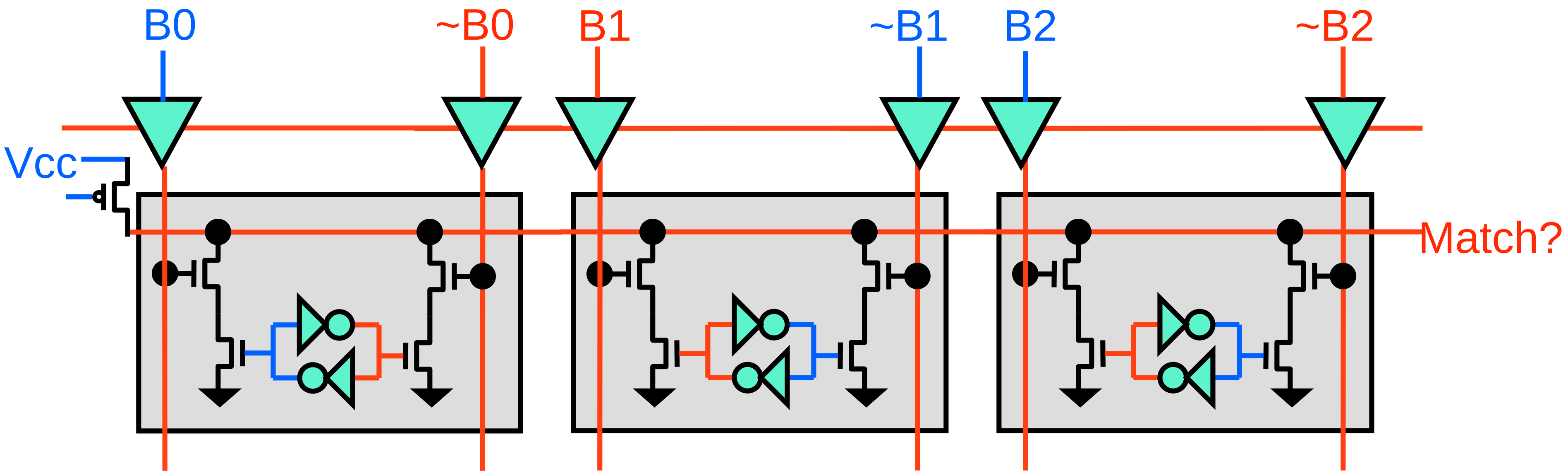


CAM -> Content Addressable Memory



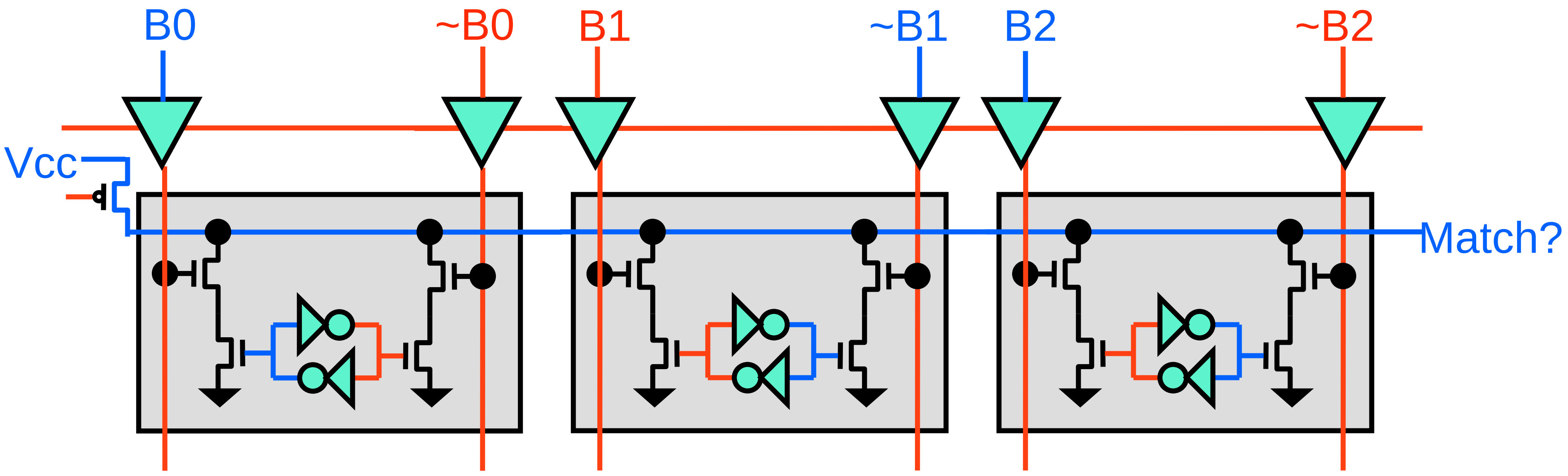
- Input: Data to match
  - (example on left: 3 bits)
- Output: matching entries
  - (example on left: 4 entries)

CAM circuit



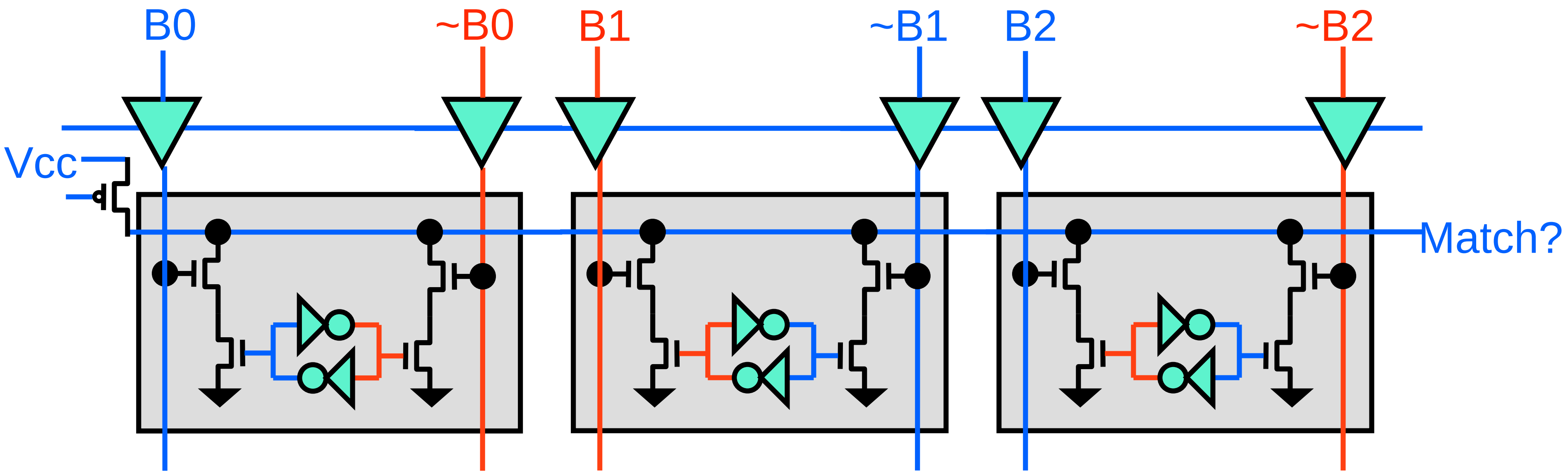
- CAM match port looks different from RAM r/w port
- Cells look similar
  - Note: Bit stored on right, ~Bit on left (opposite of inputs)

CAM circuit



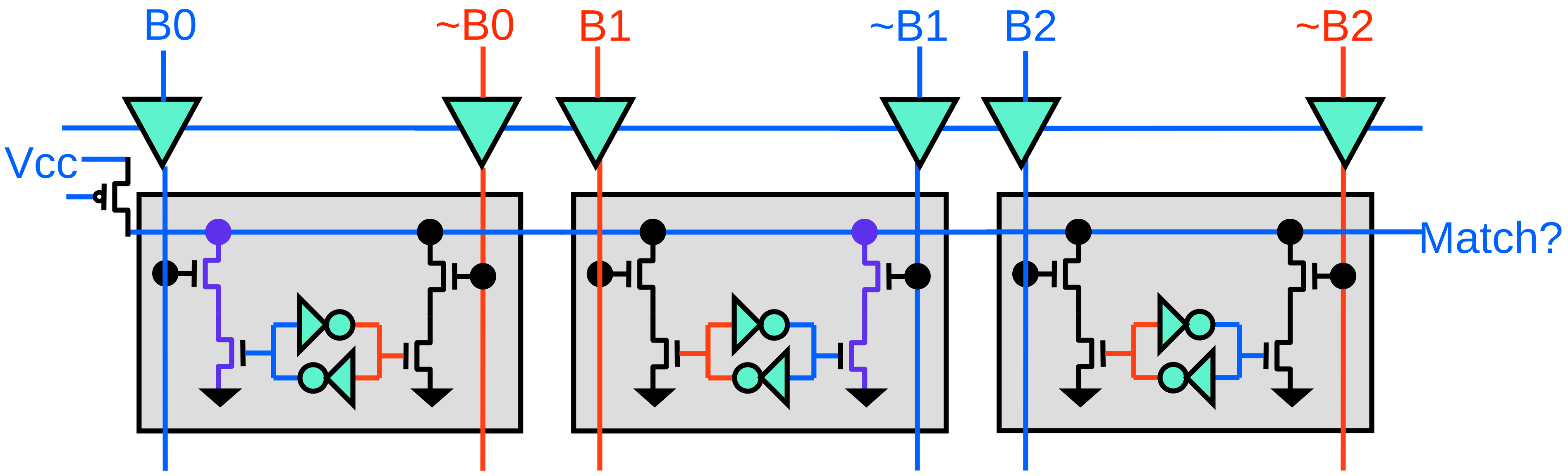
- CAM match port looks different from RAM r/w port
- Cells look similar
  - Note: Bit stored on right, ~Bit on left (opposite of inputs)
- Step 1: Precharge match line to 1 (first half of cycle)

CAM circuit



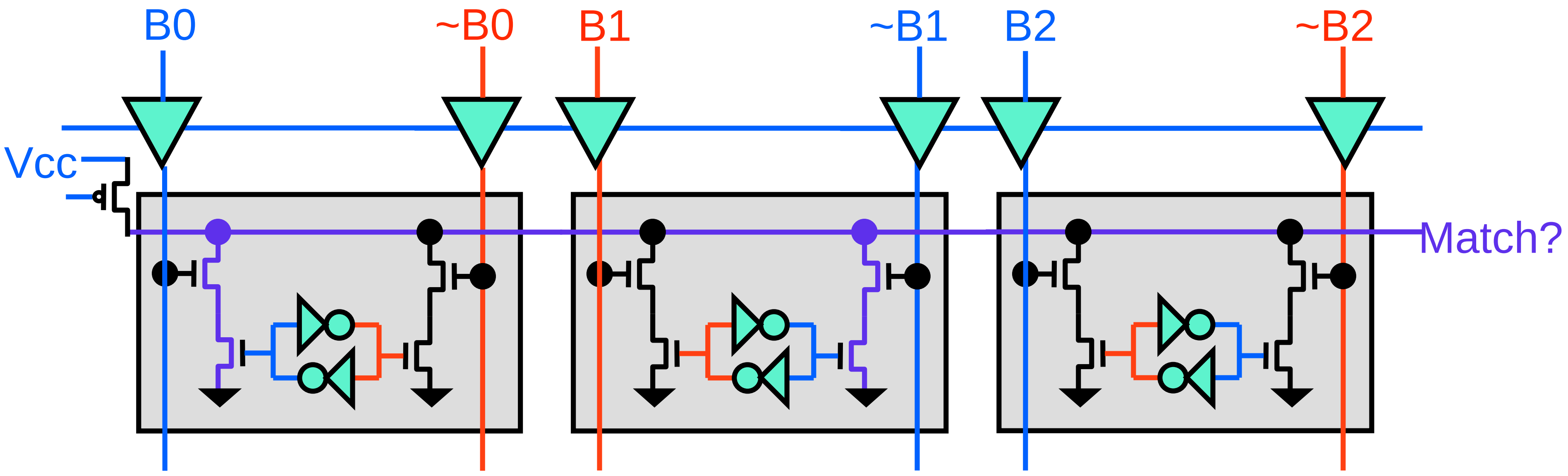
- CAM match port looks different from RAM r/w port
- Cells look similar
  - Note: Bit stored on right, ~Bit on left (opposite of inputs)
- Step 1: Precharge match line to 1 (first half of cycle)
- Step 2: Send data/~data down bit lines

CAM circuit



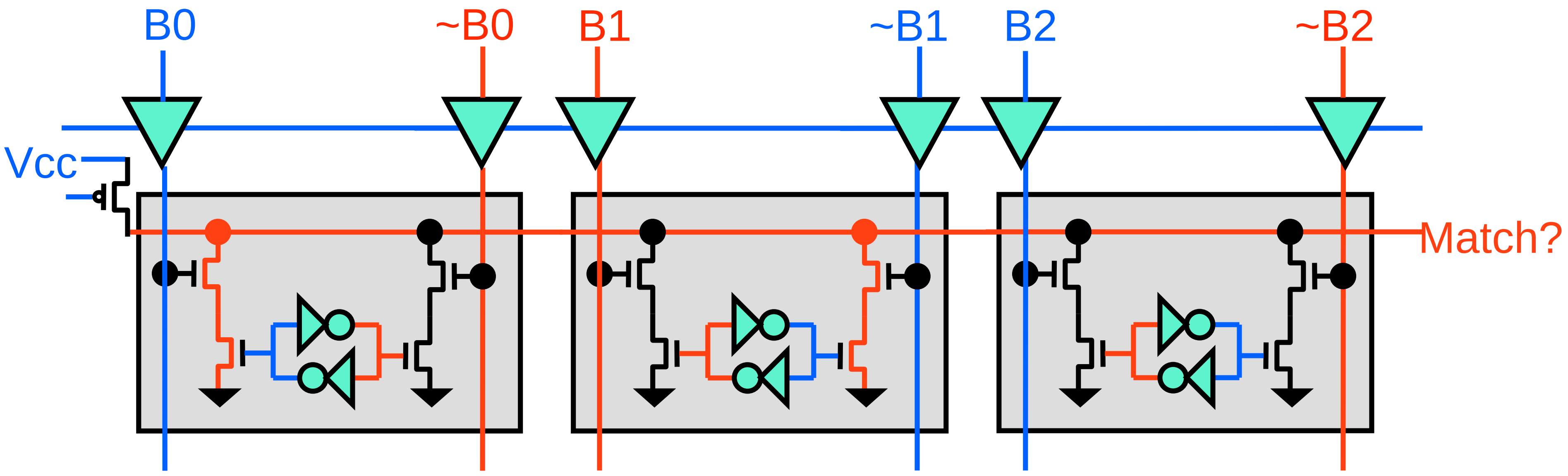
- CAM match port looks different from RAM r/w port
- Cells look similar
  - Note: Bit stored on right, ~Bit on left (opposite of inputs)
- Step 1: Precharge match line to 1 (first half of cycle)
- Step 2: Send data/~data down bit lines
  - Two 1s on same side (bit line != data) open NMOS path -> gnd

CAM circuit



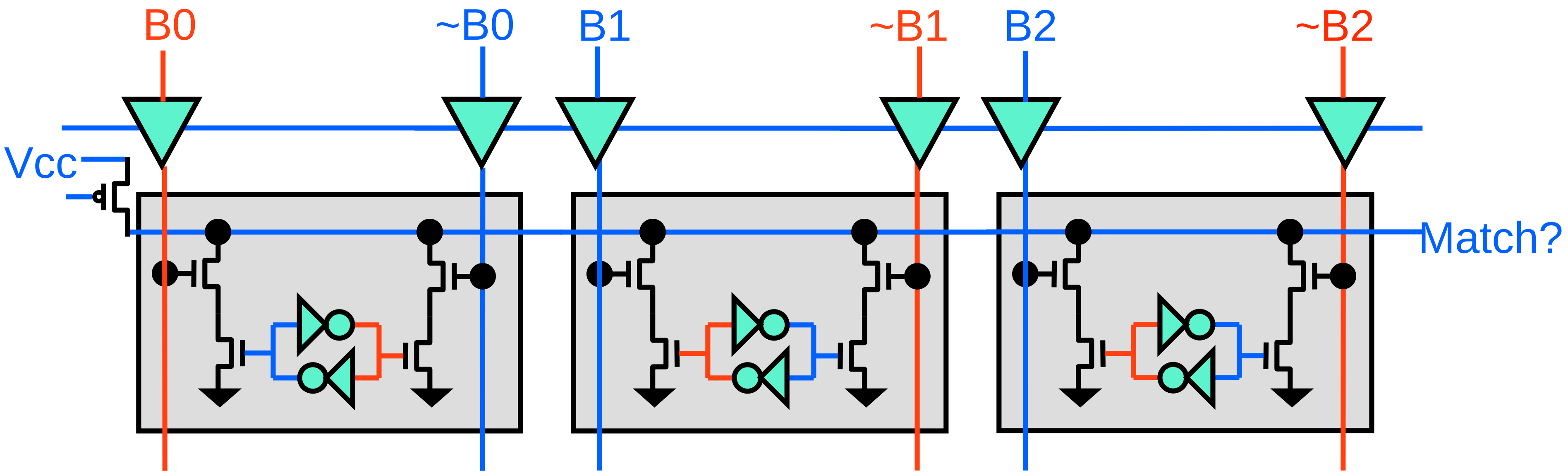
- CAM match port looks different from RAM r/w port
- Cells look similar
  - Note: Bit stored on right, ~Bit on left (opposite of inputs)
- Step 1: Precharge match line to 1 (first half of cycle)
- Step 2: Send data/~data down bit lines
  - Two 1s on same side (bit line != data) open NMOS path -> gnd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

CAM circuit



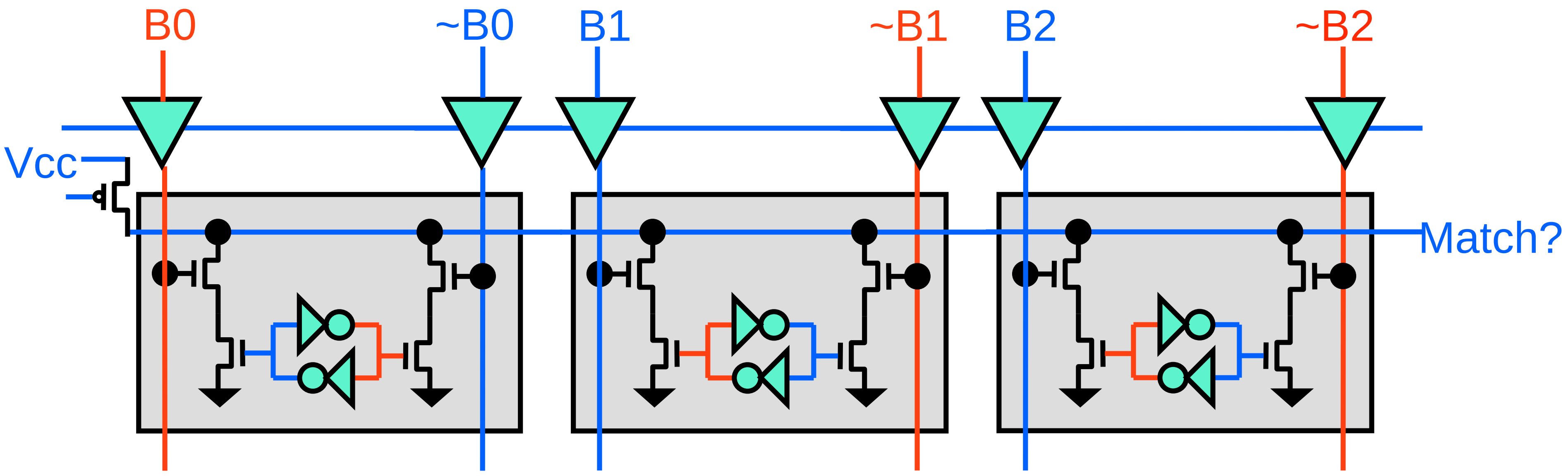
- CAM match port looks different from RAM r/w port
- Cells look similar
  - Note: Bit stored on right, ~Bit on left (opposite of inputs)
- Step 1: Precharge match line to 1 (first half of cycle)
- Step 2: Send data/~data down bit lines
  - Two 1s on same side (bit line != data) open NMOS path -> gnd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

*What are all these **blue** words? We'll see...*

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:
  - 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
    - Capacity/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 (256B instead of 16B)
  - Higher associativity? Fixes problem
    - Can have both 0 and 128 in set 0 at same time (different ways)

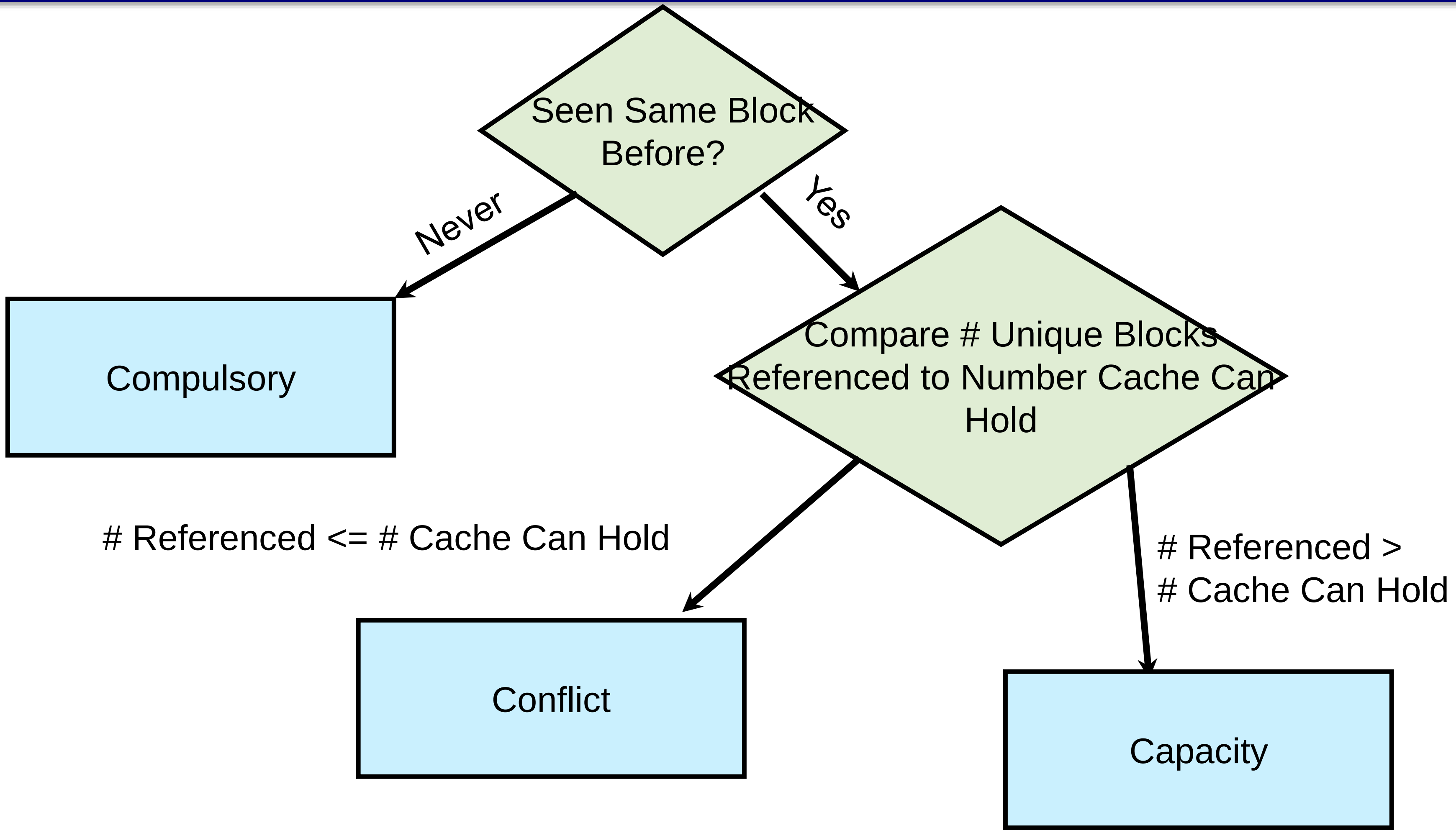
- Recognizing conflict misses:
  - Count unique blocks between last access and miss (inclusive)
  - Number of unique blocks <= number of blocks in cache? Conflict
    - Enough space to hold them all...
    - Just must be having set conflict

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, 20, 40, 60: kick out 0 for 48
    - Kick out 20 for 36
    - Kick out 40 for 24...
  - 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)
  - Number of unique blocks > number of blocks in cache? Capacity
    - Just can’t hold them all

Miss Categorization Flow Chart


Seen Same Block Before?
Never
Yes
Compulsory
Compare # Unique Blocks Referenced to Number Cache Can Hold
# Referenced <= # Cache Can Hold
# Referenced > # Cache Can Hold
Conflict
Capacity

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...

Two Optimizations

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

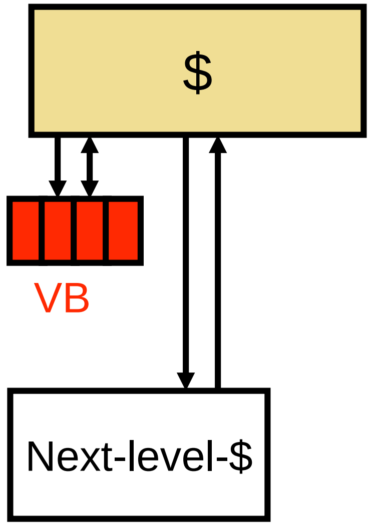
- **Prefetching**: for capacity/compulsory misses

Victim Buffer

- Conflict misses: not enough associativity
  - High associativity is expensive, but also rarely needed

- **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

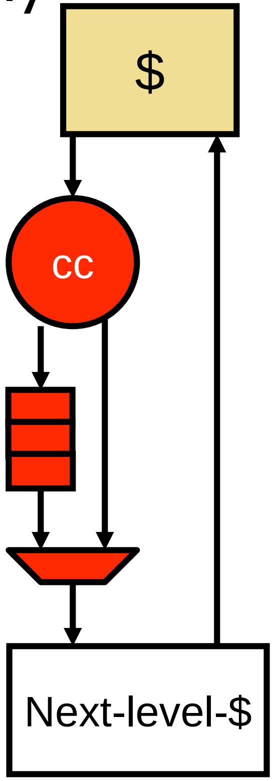


Prefetching

- **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


$
CC
Next-level-$