ECE243

Storage

Prof. Enright Jerger

Storage

- A storage mechanism can be *two* of:
  - fast
  - large
  - cheap
- ie., any given storage mechanism is either:
  - slow, small, or expensive
- Examples:
  - fast/small/cheap: register file
  - slow/large/cheap: hard drive
  - fast/large/expensive: on-chip cache

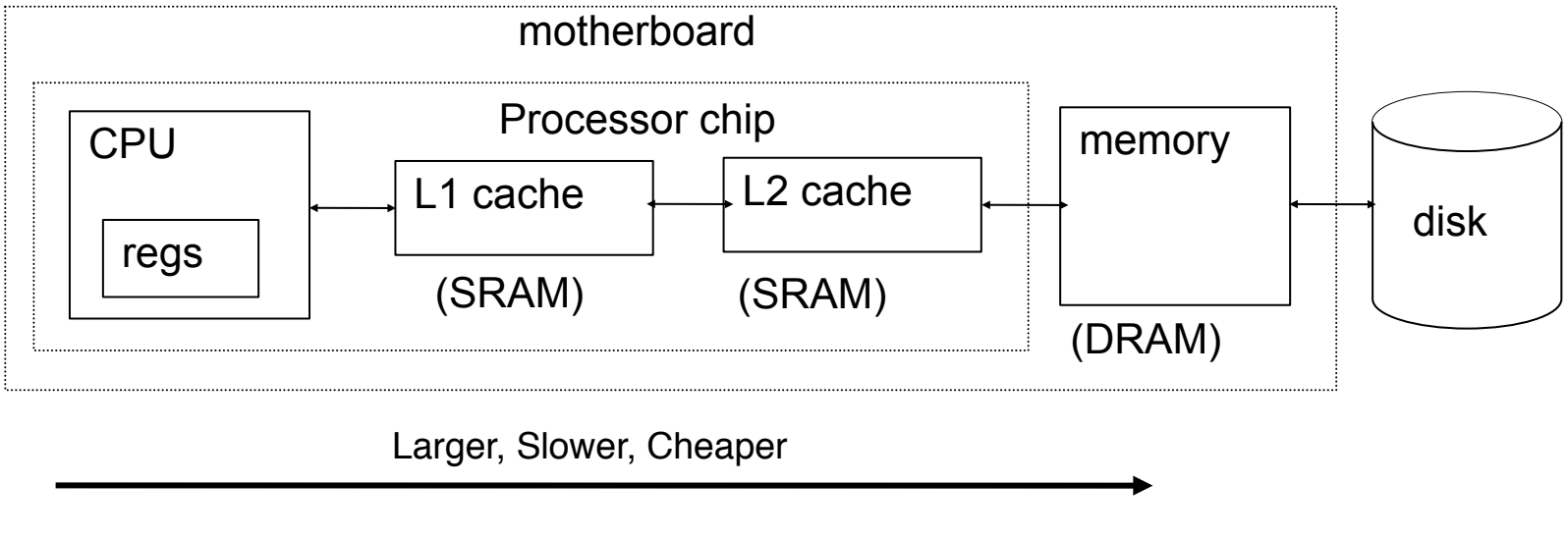
Storage Topics

- Cache Design
- Memory Design

CACHE

- a small, fast storage mechanism
- contains a subset of main memory
  - which subset?
- there can be several “levels” of cache
  - may modern desktops have 3 levels (L1,L2,L3)
- caches are “invisible” to ISAs
  - therefore they are invisible to programs too
  - you know they are there
  - but you do not directly manage them (mostly)

TYPICAL MEMORY HIERARCHY



- L1= level 1, L2 = level 2
- main memory (DRAM)
  - can be **~1000X** larger than an SRAM cache
- DRAM can be ~10X slower than SRAM
- disk can be **~100,000X** slower than DRAM

MEMORY METRICS

- latency:
  - how long it takes from request to response
    - ie, READ location 0xabcd
  - measured in hundreds of CPU cycles
    - or in tens of nanoseconds
- bandwidth:
  - data transfer rate (bits per second)
  - impacted by width and speed of buses
  - eg: DDR SDRAM can do 2.1GB/s
    - data bus: 133MHz, 64 bits wide

WHY CACHES? **LOCALITY**

- programs tend to reuse data and instructions near those they have used recently
- temporal locality: (reuse)

recently referenced items are likely to be

referenced again soon

- spatial locality: (nearby)

  items w/ nearby addresses tend to be referenced close together in time

Locality Example

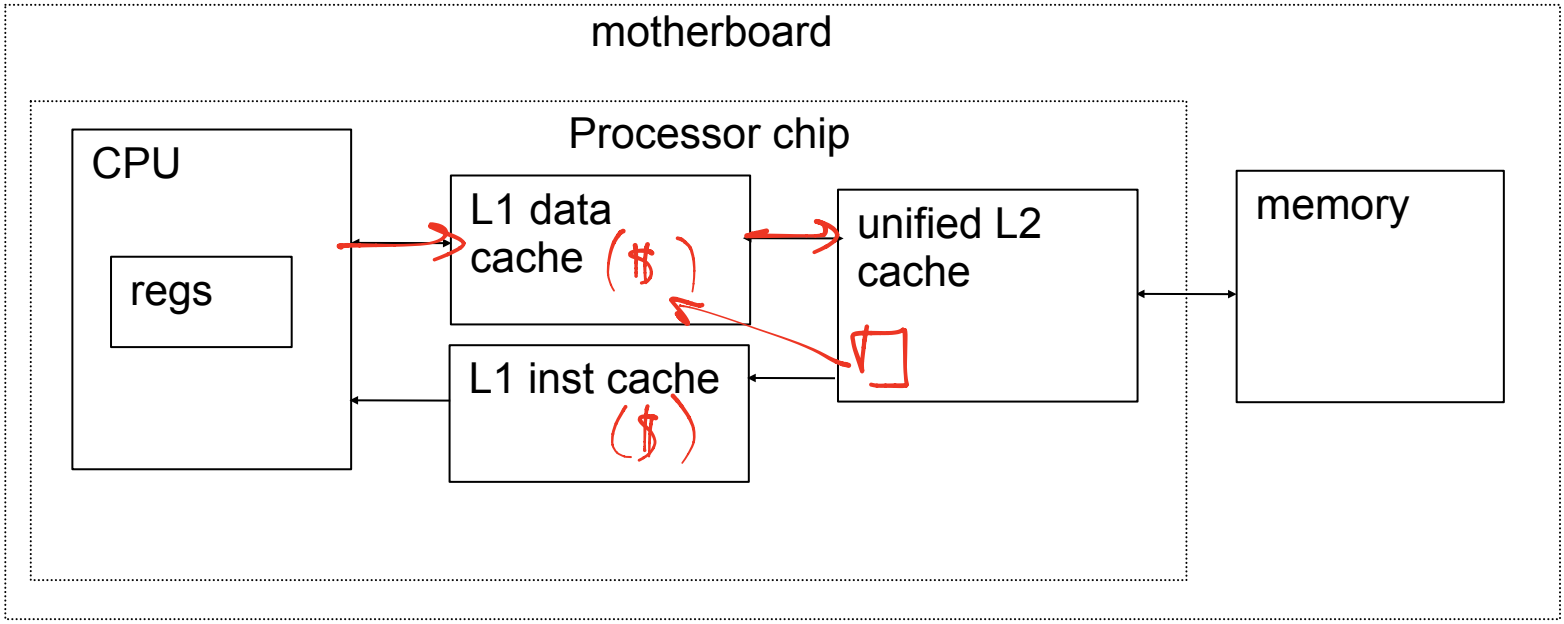
```
for (I=0;I<n;I++){
    sum+=A[I]);
}
```

- What kinds of locality are in this code?

  data: spatial
  access array sequentially

  instrs: temporal
  cycle through loop repeatedly

CACHE REALITY



- a typical processor chip
  - has separate L1 caches for data and insts
    - why?
  - has "unified" L2 cache which holds both
    - data and instructions (mixed together)
- we'll focus mainly on L1 data cache in this lecture

CACHE TERMINOLOGY

- capacity: how much can $ hold (bytes)
- placement: where should new $ blk be placed?
- Identification: how do we find a blk in the $?

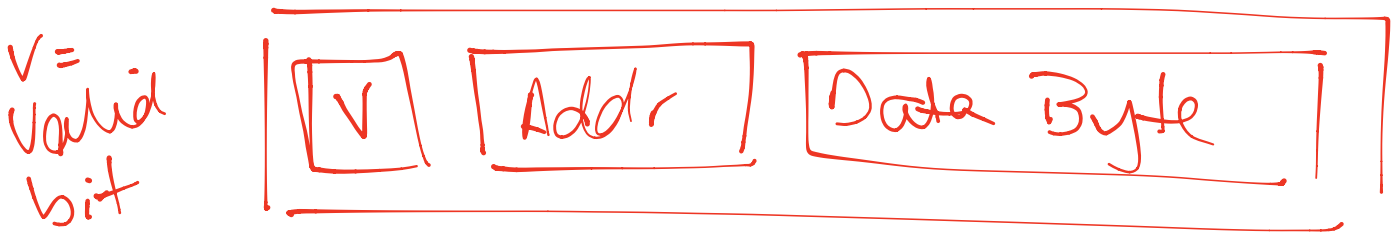
- Cache hit: $ blk we want is present
  - Hit rate = num_hits / num_accesses
- Cache miss: $ blk we want is absent
  - Miss rate: num_misses / num_accesses
    = 1 - hit rate
- Replacement: on miss, must kick blk out to make room for new blk
- Replacement strategy: which blk should we kick out?
- Write strategy: what happens on a write (store)?

HOW A CACHE IS USED

- 1) CPU performs a read for address A
- 2) If it's a "hit"
  - return the value of location A stored in the cache
  - (DONE)
- 3) if it's a "miss"
  - fetch location A from mem (or next level of $)
- 4) place A in the cache
  - replacing an existing value
- 5) return the value of location A

– (DONE)

A 1-Byte Cache (academic only)



- holds the most recent byte accessed
  - pretend only byte loads and stores used
- how do you know if the cache is empty?
  - can't use address==0x0---this is a valid address!
  - need a valid bit
  - V==0 means empty, V==1 means valid
- how do you know what byte is in the cache?
  - must store the address of the byte that is present

1-Byte Cache EXAMPLE

Cache: Mem:

V	Addr	Data
0	① 0x7ace	① 0x78
① 1	③ 0x7ac0	③ 0x56
	④ 0x3b00	④ 0x12
	⑤ 0x5d00	⑤ 0x25

Addr	Value
0x3b00	0x12
…	0x0
0x5d00	0x25
…	0x0
0x7ac0	0x56
…	0x0
0x7ace	0x78

① ldb r8, 0x7ace(r0) # miss!
② ldb r8, 0x7ace(r0) # hit!
③ ldb r8, 0x7ac0(r0) # miss!
④ ldb r8, 0x3b00(r0) # miss!
⑤ ldb r8, 0x5d00(r0) # miss!

HOW CACHES EXPLOIT LOCALITY

- cache stores a subset of memory
  - which subset? the subset likely to be reused
  - to exploit temporal locality
  - our 1-byte cache does this
- cache groups contents into blocks
  - aka “cache lines”
  - to exploit spatial locality
  - motivates a “1-block cache”

A 1-Block Cache (academic only)

Announcements.

Extra lab hours for project

Tues April 1 - Thurs April 3

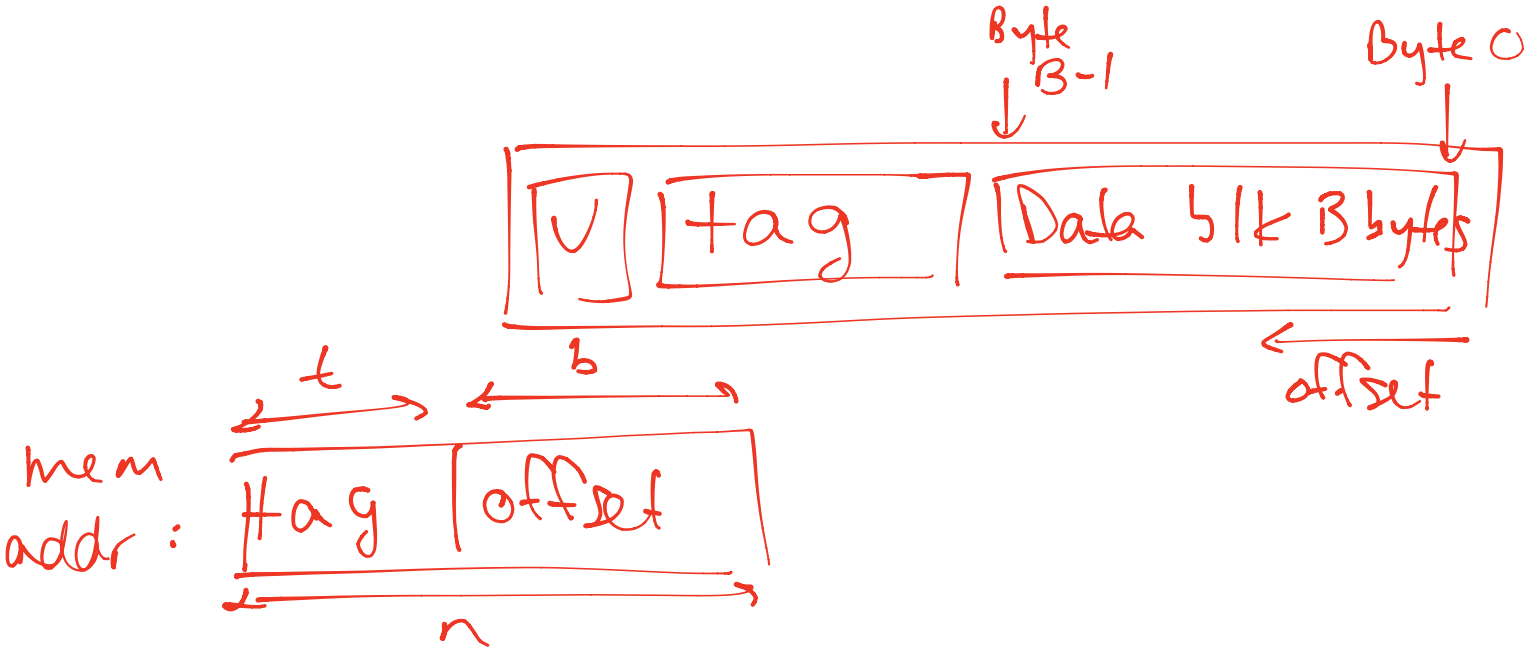
6 - 9pm

BA3135 / 3145

FCFS

Remarked & Uncollected midterms (all sections) available in my office

Office hours: Tues & Fri 12 - 1

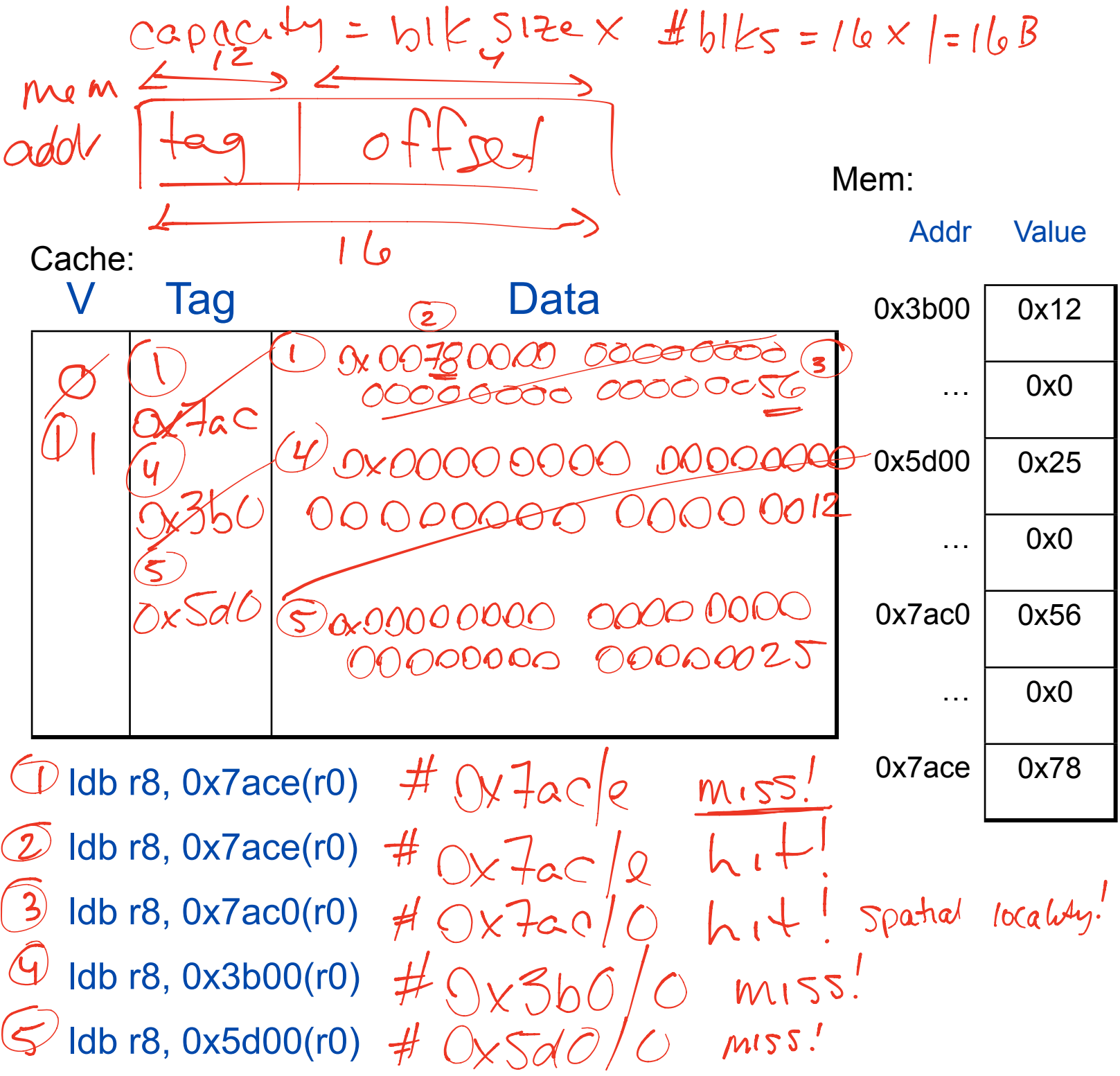


- holds the most recent *block* accessed
  - a block is a group of B consecutive bytes
  - $B = 2^b$
- how do you know what block is in the cache?
  - must store the tag of the block that is present
  - a tag is just a subset of the address bits
- how do you access a specific byte in a block?
  - a portion of the address called the *offset*
  - used as an index into the data block

1-Block-Cache EXAMPLE

- given a 16 bit addr space
- assume little-endian (like NIOS)
- assume a 1-block cache with:
  - t = 12 bits, b=4 bits

blk size: $B = 2^4 = 16$ Bytes



REAL CACHE IMPLEMENTATIONS

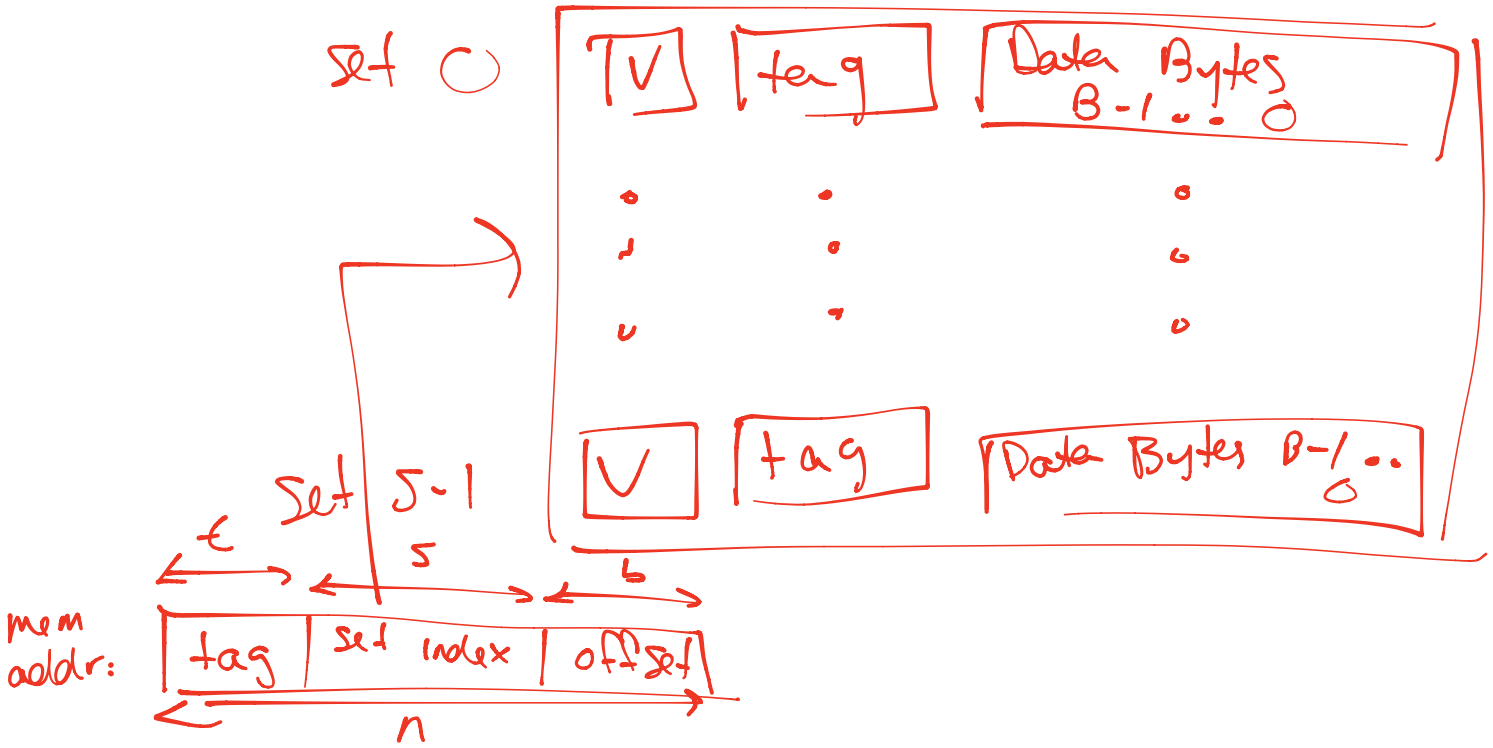
- Store multiple blocks
  - can cache more data at once
- Three main types:

- Direct-Mapped
  - the simplest but inflexible
- Fully-Associative
  - the most complex but most flexible
- Set-Associative
  - the happy medium

DIRECT MAPPED

- each memory location maps to:
  - a specific cache location
- since cache is much smaller than memory:
  - multiple memory blocks map to each cache block
- for a given memory address:
  - the set index indicates how to locate a block
    - since there are multiple blocks in the cache
  - the offset indicates which byte within a cache block
  - the tag identifies the block
    - ie., which memory location does this block corresponds to?
    - since several memory locations map to each cache block

Direct-Mapped Cache



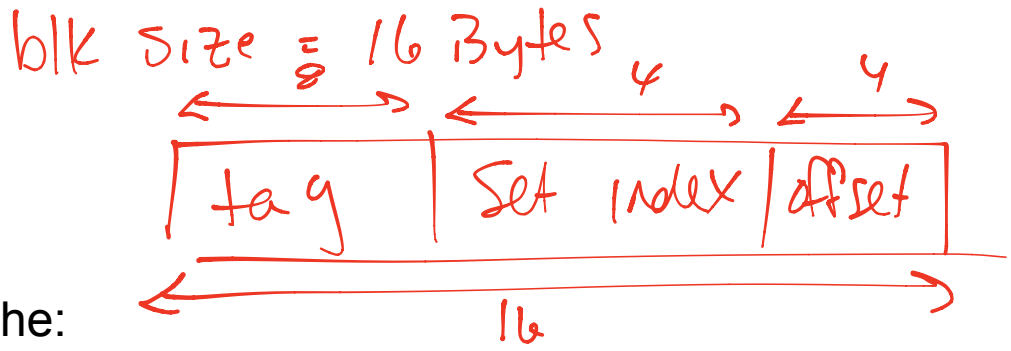
- address space size:
  - 2^n (bytes) (assuming byte addressable)
- block size
  - $B = 2^b$ (bytes)
- number of sets
  - $S = 2^s$ = the total number of blocks in the cache
  - ie, there is 1 block per set for a direct-mapped cache
- capacity:
  - $B*S = 2^{(b+s)}$

Direct-Mapped EXAMPLE

- given a 16 bit addr space
- assume little-endian (like nios)
- assume a direct-mapped cache with:
  - t = 8 bits, s = 4 bits, b=4 bits

capacity = $2^{(4+4)}$ = 256 Bytes

#sets = 2^4 = 16



Cache:

Mem:

Set	V	Tag	Data
0	0	0x35	0x0000000000000012
	1	0x5d	0x0000000000000025
…	0		
12	0	0x7a	0x0078
	1		0x56
…	0		
15	0		

Addr	Value
0x3b00	0x12
…	0x0
0x5d00	0x25
…	0x0
0x7ac0	0x56
…	0x0
0x7ace	0x78

① ldb r8, 0x7ace(r0) # 0x7a|c|e - miss!

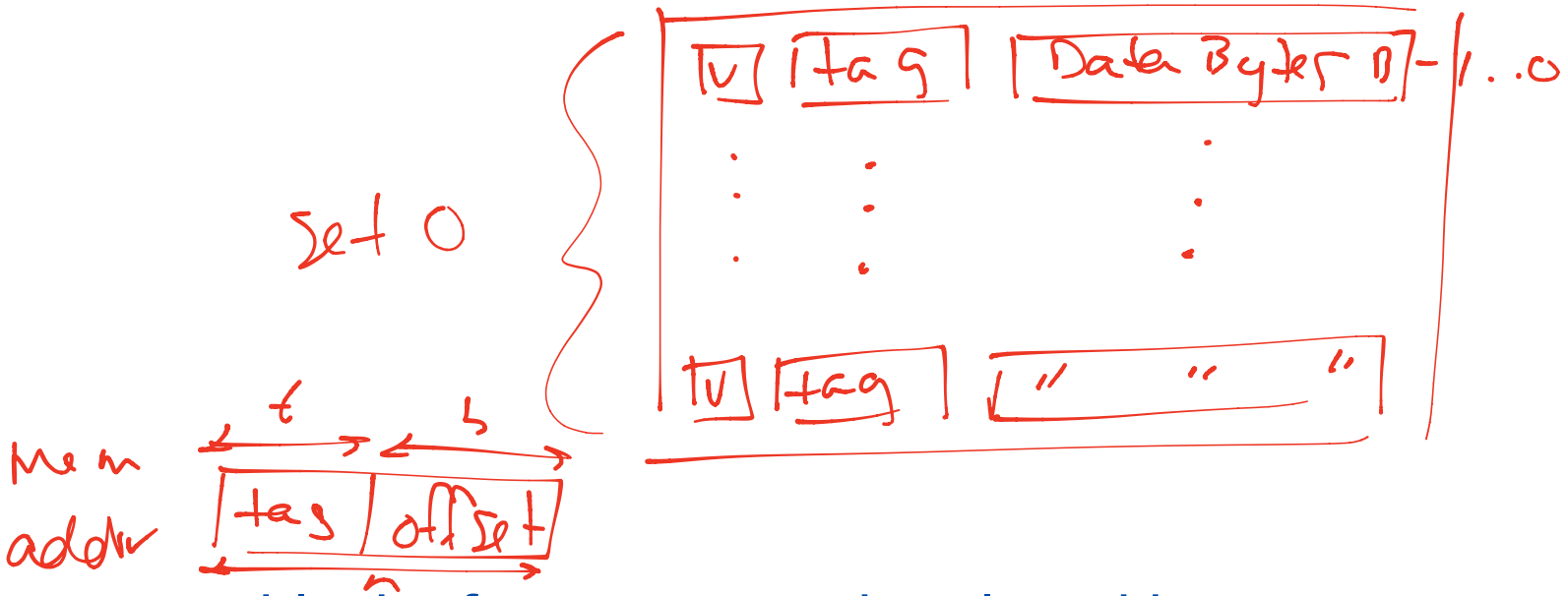
② ldb r8, 0x7ace(r0) # 0x7a|c|e - hit!

③ ldb r8, 0x7ac0(r0) # 0x7a|c|0 - hit!

④ ldb r8, 0x3b00(r0) # 0x3b|0|0 - miss!

⑤ ldb r8, 0x5d00(r0) # 0x5d|0|0 - miss!

FULLY-ASSOCIATIVE CACHE



- any block of memory can be placed in any cache block
  - can think of it as one large set
- good: more flexible
- bad: harder to find a given cache block
  - must search **all** of the tags!

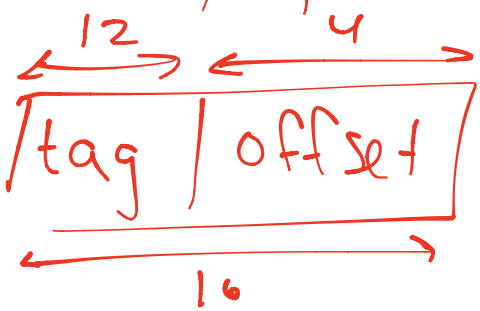
Fully-associative example

- a 16 bit addr space
- 16-byte blocks
- total capacity of 64 bytes

$b = 4$

$n = 16 \quad \therefore \quad t = 12$

blks = capacity / (bytes/blk) = 64/16 = 4 blks



Cache:

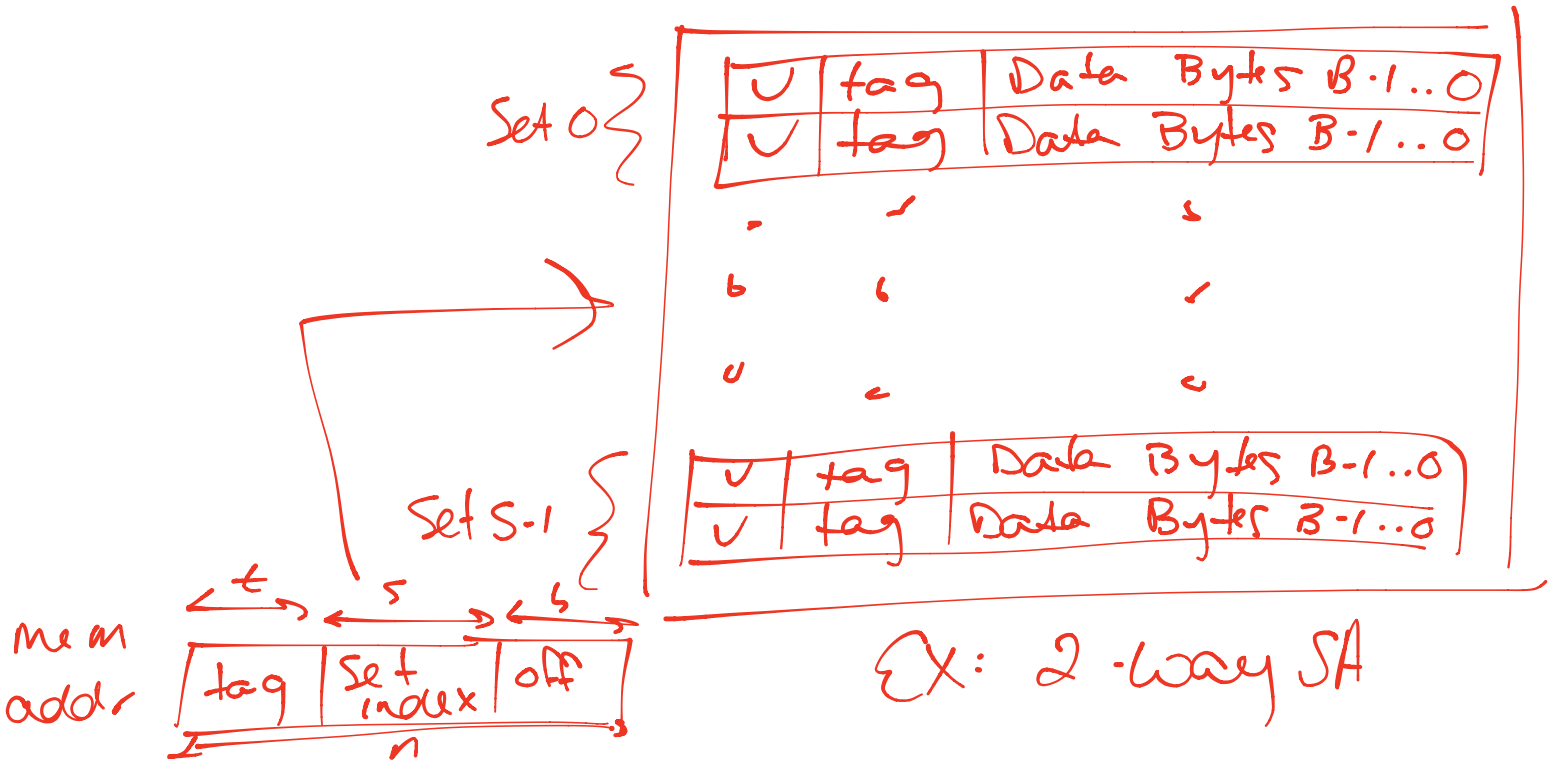
	V	Tag	Data
0	1	0x7ac	0x00780000 00000000 00000000 00000056
1	1	0x3b0	0x00000000 00000000 00000000 00000012
2			
3	1	0x5d0	0x00000000 00000000 00000000 00000025
4			

Mem:

Addr	Value
0x3b00	0x12
…	0x0
0x5d00	0x25
…	0x0
0x7ac0	0x56
…	0x0
0x7ace	0x78

1. ldb r8, 0x7ace(r0) # 0x7ac/e - miss!
2. ldb r8, 0x7ace(r0) # 0x7ac/e - hit!
3. ldb r8, 0x7ac0(r0) # 0x7ac/0 - hit!
4. ldb r8, 0x3b00(r0) # 0x3b0/0 - miss!
5. ldb r8, 0x5d00(r0) # 0x5d0/0 - miss!

SET ASSOCIATIVE



- number of "ways" within a set (W)
  - say "a W-way set-associative cache"
- each block of memory maps to a specific set in the cache
  - but can map to any block within that set
- capacity = B*S*W bytes

SET ASSOCIATIVE Example

- given a 16 bit addr space
- assume a 2-way set-associative cache
- t = 8 bits, s = 4 bits, b=4 bits

sets = 2^4 = 16

blk size = 2^4 = 16 Bytes

capacity = 16 x 16 x 2 = 512 Bytes

Cache:

Set	V	Tag	Data
0	1	0x3b	0x0000000000000012
	1	0x5d	0x0000000000000025
12	0		
	1	0x7a	0x0078000000000000 0x0000000000000056
15	0		
	0		

Mem:

Addr	Value
0x3b00	0x12
…	0x0
0x5d00	0x25
…	0x0
0x7ac0	0x56
…	0x0
0x7ace	0x78

1. ldb r8, 0x7ace(r0) # 0x7a|c|e - miss!
2. ldb r8, 0x7ace(r0) # 0x7a|c|e - hit!
3. ldb r8, 0x7ac0(r0) # 0x7a|c|0 - hit!
4. ldb r8, 0x3b00(r0) # 0x3b|0|0 - miss!
5. ldb r8, 0x5d00(r0) # 0x5d|0|0 - miss!

Real life cache Example:

- Core2:

- L1 Data Cache:
  - 32KB
  - 8-way set-associative
- L2 Cache:
  - 6MB
  - 16-way set associative

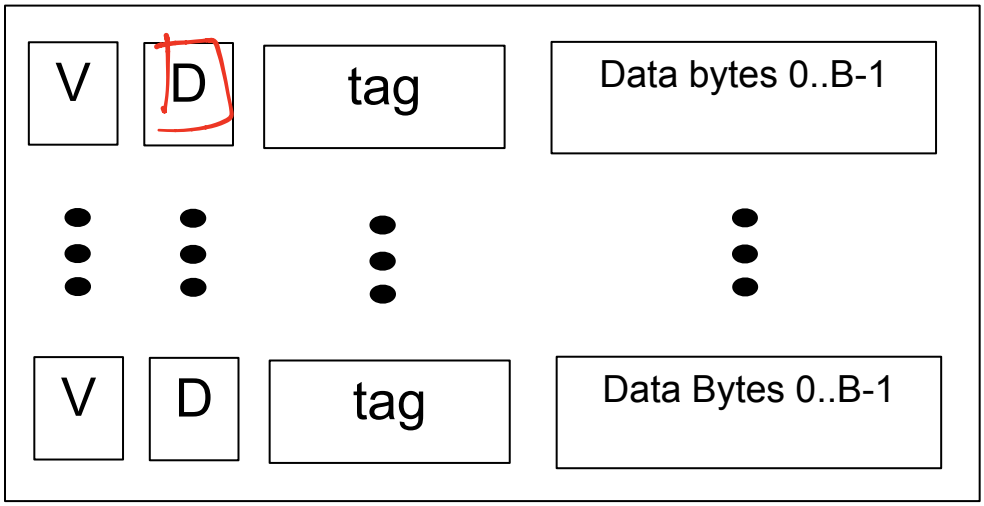
COMPARING CACHE TYPES

	Direct-mapped	Set-associative	Fully-associative
Time to access	Fast	med	slow
Flexibility	none	some	lots

What if you write to a cache block?

- Option1:

- can pass the write to the next level too
  - i.e., to L2 and/or Memory
  - This is called “write through”
  - because the cache writes through to memory
- Option2:
  - Don’t pass the write to the next level
  - only do so when the block is replaced
  - Need a “dirty” bit
  - the block has been written if the dirty bit is 1
  - This is called “write back”
  - because the cache writes the cache block back later



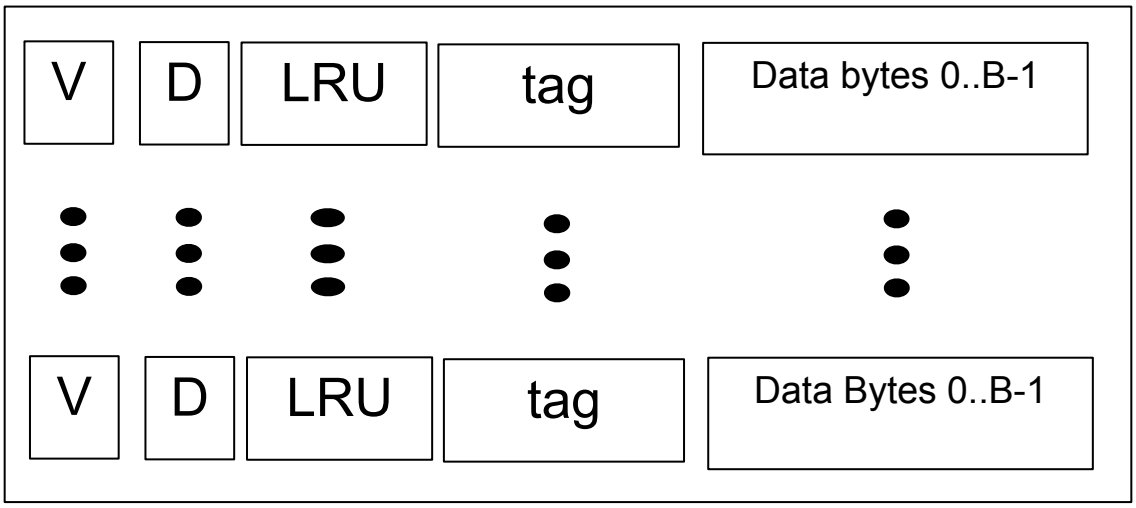
REPLACEMENT STRATEGIES

- For fully and set-associative caches:
  - which cache block do you replace on a miss?
- Ideal algorithm:

- replace block that will not be used for the longest time
- Impossible: must know the future

- Realistic algorithm: Least Recently Used (LRU)
  - Replace the block that has been accessed least recently
    - Provides a good approximation to the ideal algorithm
  - each set tracks the ordering of the blocks within it
    - Each set needs extra bits to store this LRU state

LRU IMPLEMENTATIONS



- num of bits to track LRU for 2-way set-assoc?

  1 bit / set

- for 4-way set-assoc?

  2 bits per blk (way) ⇒ 4 x 2 = 8 bits/set

- for N-way set-assoc?

N x $\log_2 N$ bits/set

Other Replacement Strategies

- Random:
  - pick a random block
  - works pretty good, fast!
- Random-but-not-MRU
  - used in modern processors
  - Track MRU
    - how many bits?

      $\log_2 N$
  - pick a random block
    - if you pick the MRU block then pick another block
  - why is this used?
    - Good compromise: faster than LRU, better than pure random

Total Cache Size

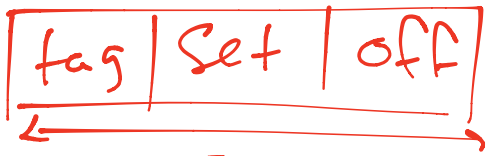
- total_cache_size = capacity + overhead
- overhead = tags + valid_bits + dirty_bits + LRU_bits
- Ex: for 32bit addr space, how many bits for:

⇒ 64 entry DM, write back, 32B blks = B Addr ✓

$s = \log_2(64) = 6$

$b = 5$

$t = 32 - 6 - 5 = 21$ bits



capacity = 64 x 32B/blk = 2048 B

overhead = 64 x (1 valid + 1 dirty + 21 tag)
= 64 x 23

total = 64 x 23 bits + 2048 x 8 bits
= 1472 + 16384 = 17856 = 2232B

TYPES OF CACHE MISSES:

- cold misses:
  - the cache is empty when a program starts, all blocks are invalid
- capacity misses:
  - a program accesses much more data than can fit in the cache
  - ex: access a 1MB array over and over, when cache is only 1KB
- conflict misses:
  - a program accesses two memory blocks that map to the same cache block
  - these two blocks will “conflict” in the cache, causing misses
  - even though there is spare capacity
  - more likely to happen for direct-mapped caches

Ex 4096B capacity, 4-way SA, LRU, writeback, 32B blks, 32 bits addr space

Q: Calculate total size?

$4096/32 = 2^{12}/2^5 = 2^7 = 128$ blks

$128/4 = 32$ sets

$b = 5$

$s = 5$

$t = 32 - 5 - 5 = 22$

overhead = 128 x (1 valid + 1 dirty + 2 LRU + 22 tag)

128 x (26 bits)

$128 \times 26 + 4096 \times 8 =$ total size

LRU for 4-way SA (example from board)

set 0: | LRU | Addr | LRU | Addr | | | | | (C is LRU)
| 00 | A | 01 | B | 11 | C | 10 | D |

Access pattern: C D B A E A (time →)

| 01 | A | 10 | B | 00 | E | 11 | D |

| 00 | A | 10 | B | 01 | E | 11 | D |

Ex: ACCESS PATTERNS

- Assume:
  - 1KB direct-mapped
  - 32B blocks
  - A[i] is 4B
  - cache is cold/empty

```
for (i=0;i<1024;i++){
        sum += A[i];
}
```

Each miss causes 32B to load ⇒ 8 elements
when A[0] is read it loads A[0]..A[7]
∴ 1 in every 8 accesses will miss
misses = 1024/8 = 128 misses
miss rate = 128/1024 = 12.5%
hit rate = 1 − miss rate = 87.5%

Ex: ACCESS PATTERNS

- Assume:
  - 1KB direct-mapped
  - 32B blocks
  - A[i] is 4B
  - cache is cold/empty
  - A[i] maps to same set as B[i]

```
for (i=0;i<1024;i++){
        sum += A[i] + B[i];
}
```

e.g. A[0] addr = 0x200000, B[0] addr = 0x400000
Read A[0] - miss, load A[0]..A[7]
Read B[0] - miss, replace A[0]..A[7] w/ B[0..7]
Read A[1] - miss, replace B[0]..B[7] w/ A[0]..A[7]

Read B[1] - miss, replace A[0]-A[7] w/ B[0]..B[7] etc.
miss rate = 100% - blks constantly conflict
fixes:
1) move B[] so it doesn't map to same set as A[]
2) break into 2 loops { for ( ) { sum += A[i] } for ( ) { sum += B[i] }
3) 2-way SA

CACHE PERFORMANCE

- Average_ Access_Time =
  (hit_rate * hit_time) + (miss_rate * miss_penalty)
- Cache_hit_time = time for a hit
  - ex: 1 or 2 cycles for a hit in the L1
- Miss_penalty = time for a miss
  - ex: 10-20 cycles if misses L1 but hits in L2

Ex: Cache performance

- *Assume:*
  - a miss rate of 12.5%
  - a 1-cycle L1 hit latency
  - a 10-cycle L2 hit latency
  - no access misses the L2 cache.
- What is the average access time?

.875 x 1 cycle + 0.125 x 10 cycles

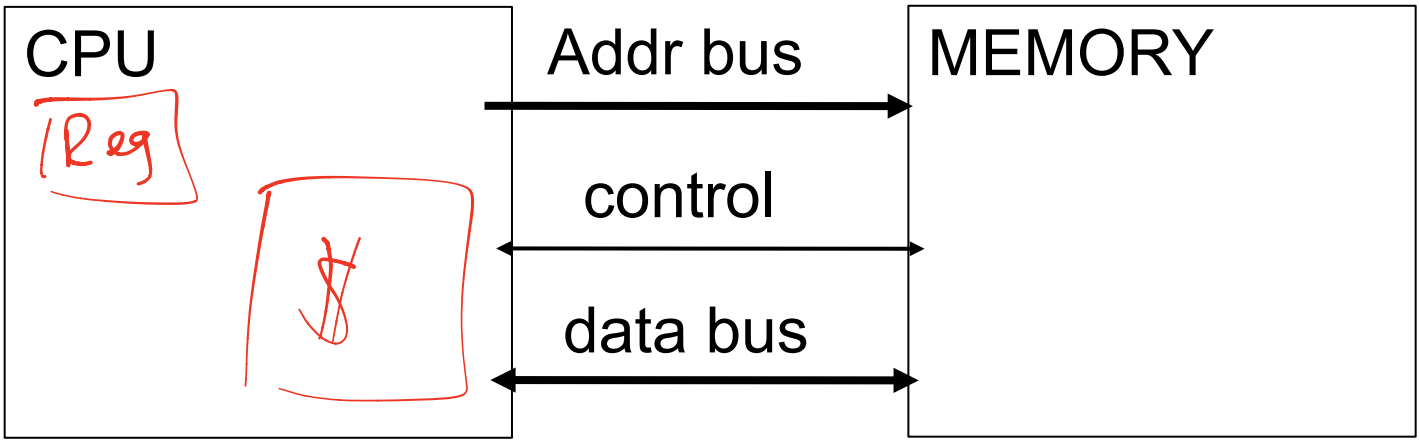
= 0.875 + 1.25

= 2.125 cycles

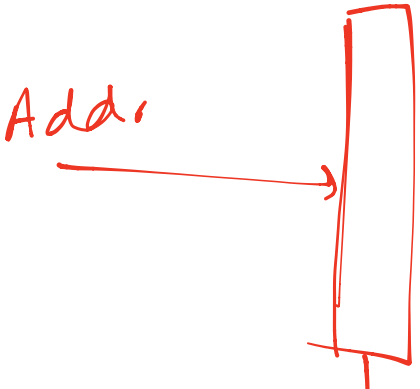
Memory

RAM IMPLEMENTATION DETAILS

- a simple view of memory:

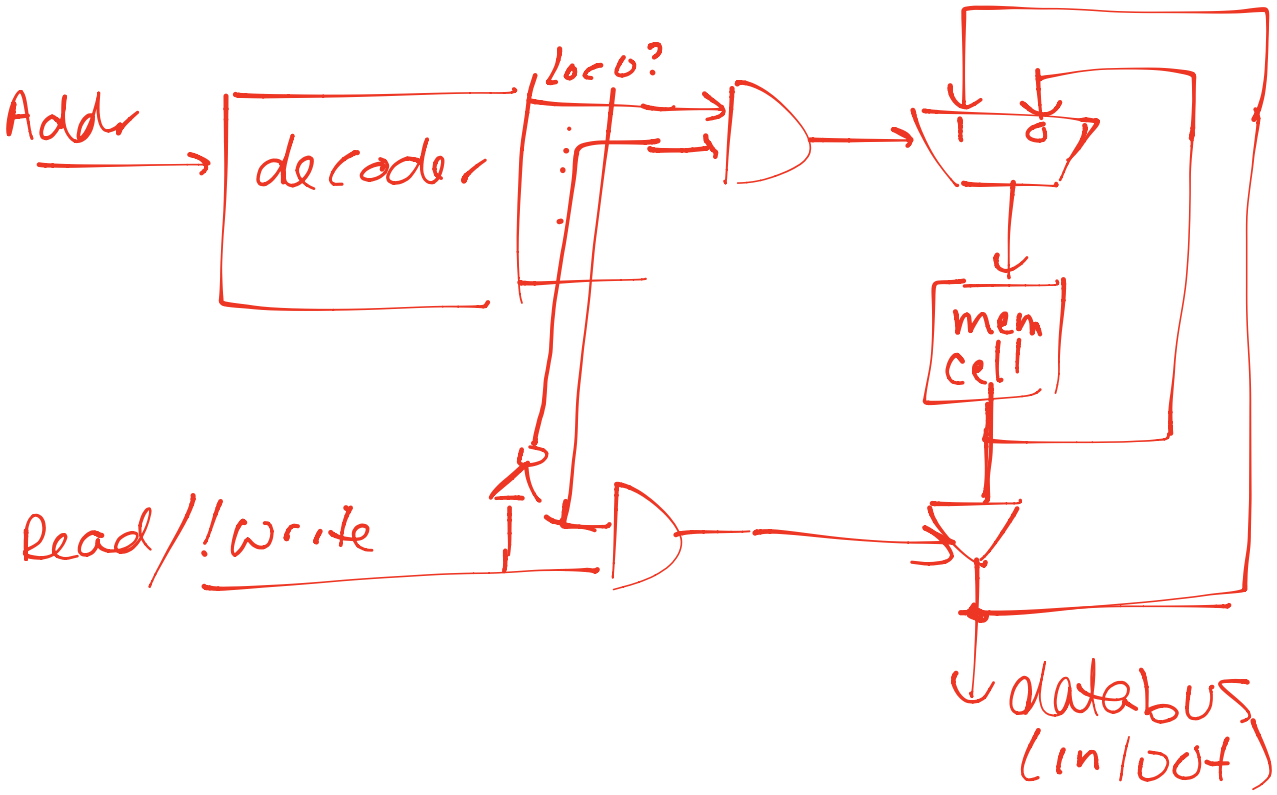


A NAÏVE IMPLEMENTATION



- address in, one bit out

A NAÏVE IMPLEMENTATION: Detailed view

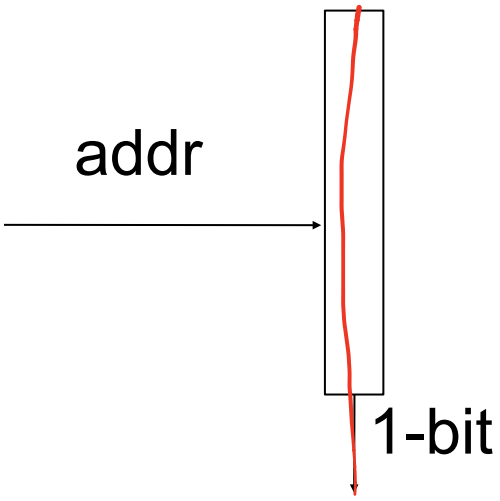


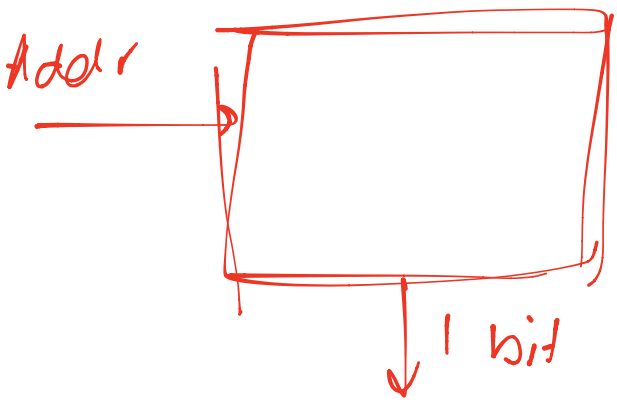
- N copies of this circuit for N bits
- All attach to the data bus via tristates

DESIGNING A BETTER MEMORY

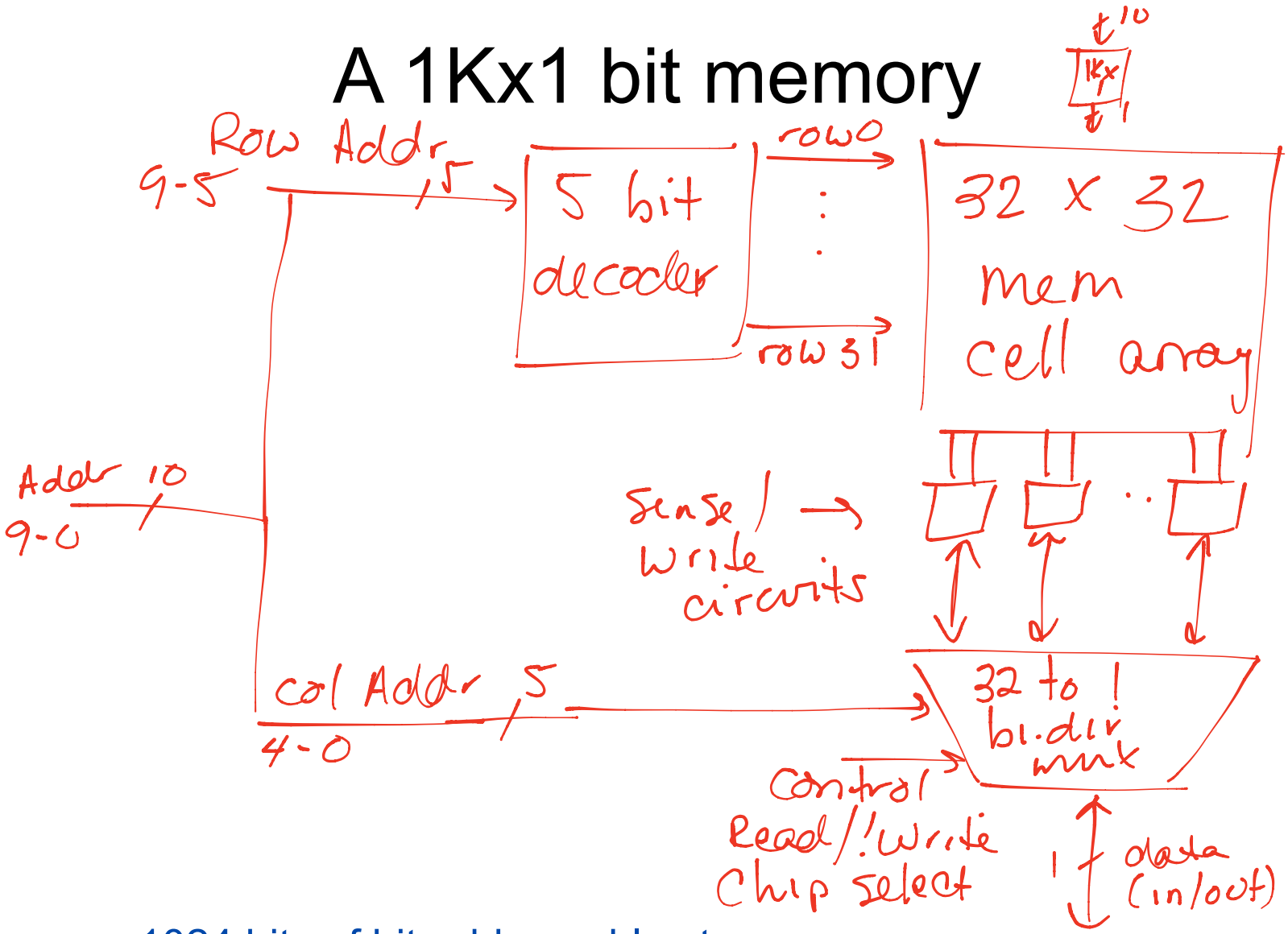
inefficient

more efficient?





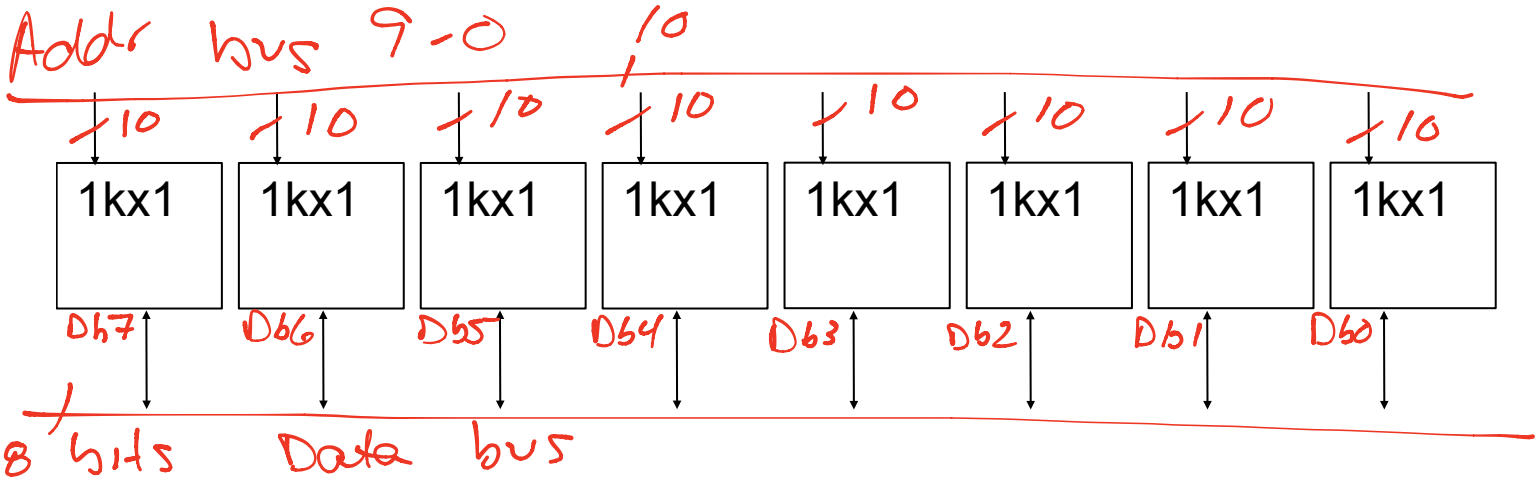
A 1Kx1 bit memory



- 1024 bits of bit-addressable storage
- most efficient as a 32 x 32 grid (square)
- 10 address bits: use 5 addr bits x 5 addr bits to address the grid

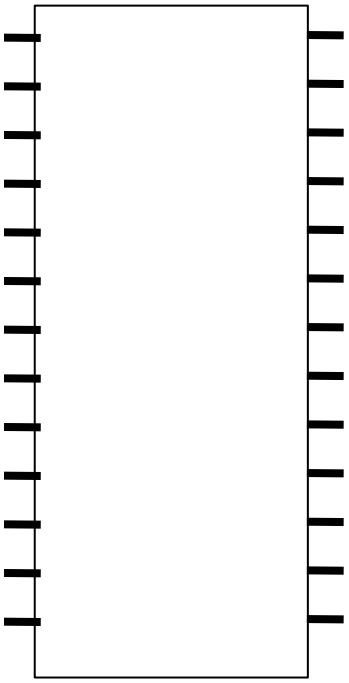
Combining Memories

- Use 8 1Kx1 memories to make:
  - a 1Kx8 memory (byte addressable)

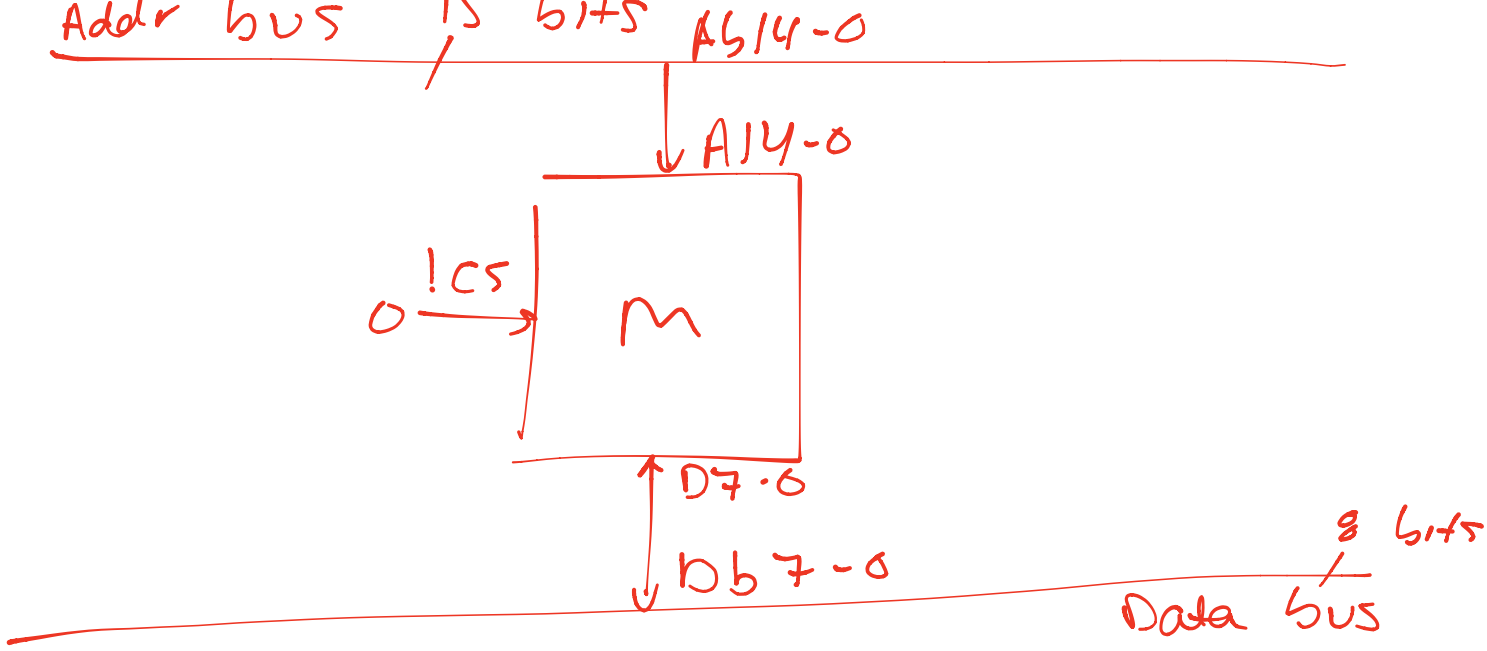


28-PIN MEMORY CHIP

- A14-0: address pins
- D7-0: data pins
- !CS: chip select: 0=enabled, 1=disabled
- !WE: write enable
- !OE: output enable,
- Vcc: power supply
- GND: ground



CONNECTION WITH A BUS



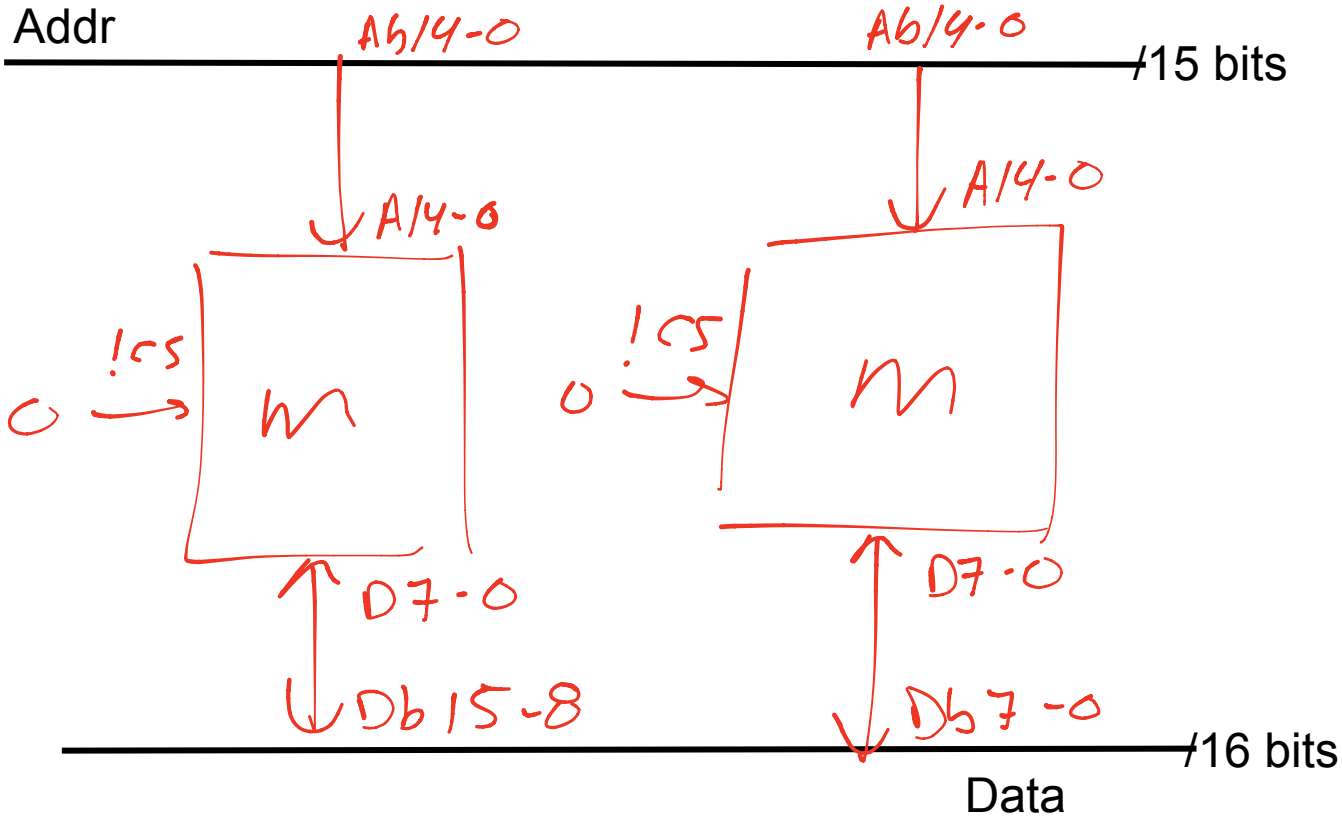
- how much storage does it have?

  15 bit addr space

  2^{15} x 8 bits = 32 kB

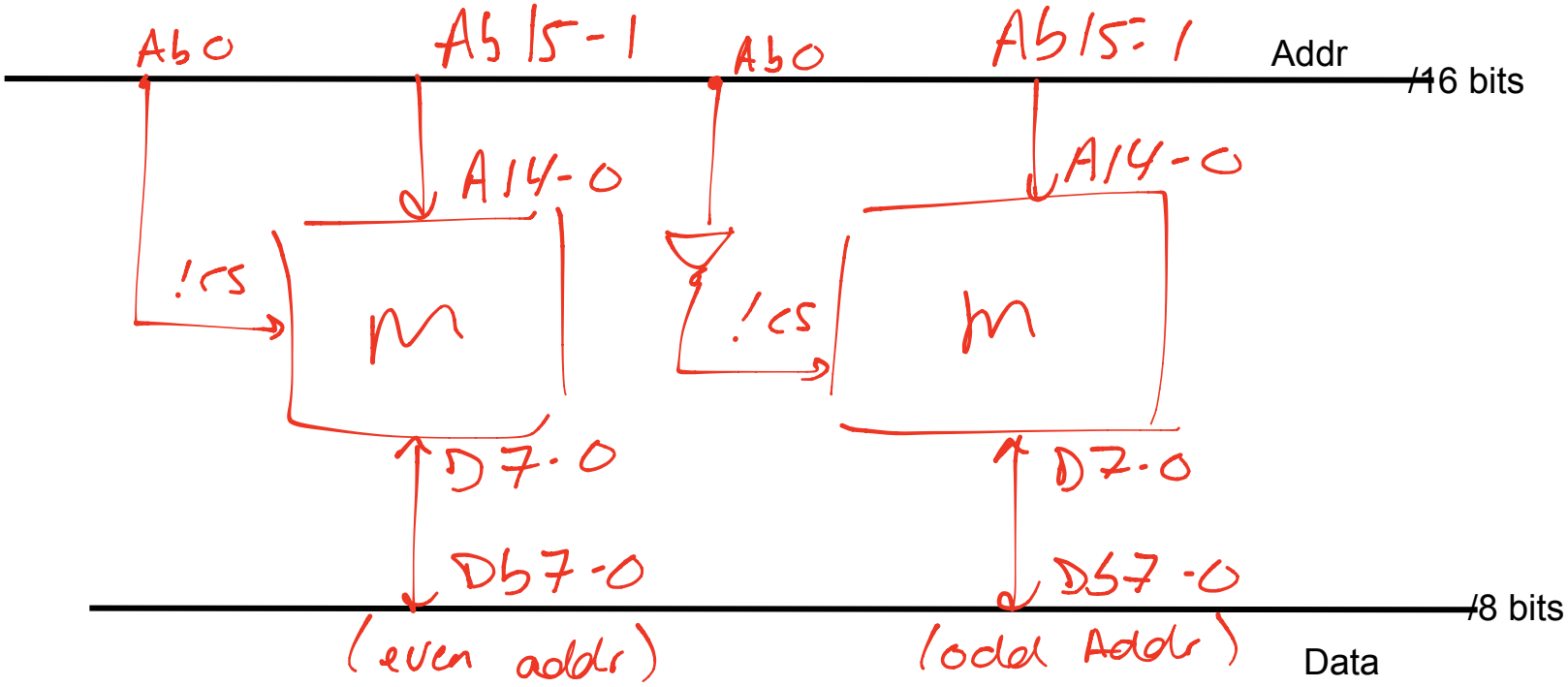
Example question

- build a 16-bit-wide data bus?

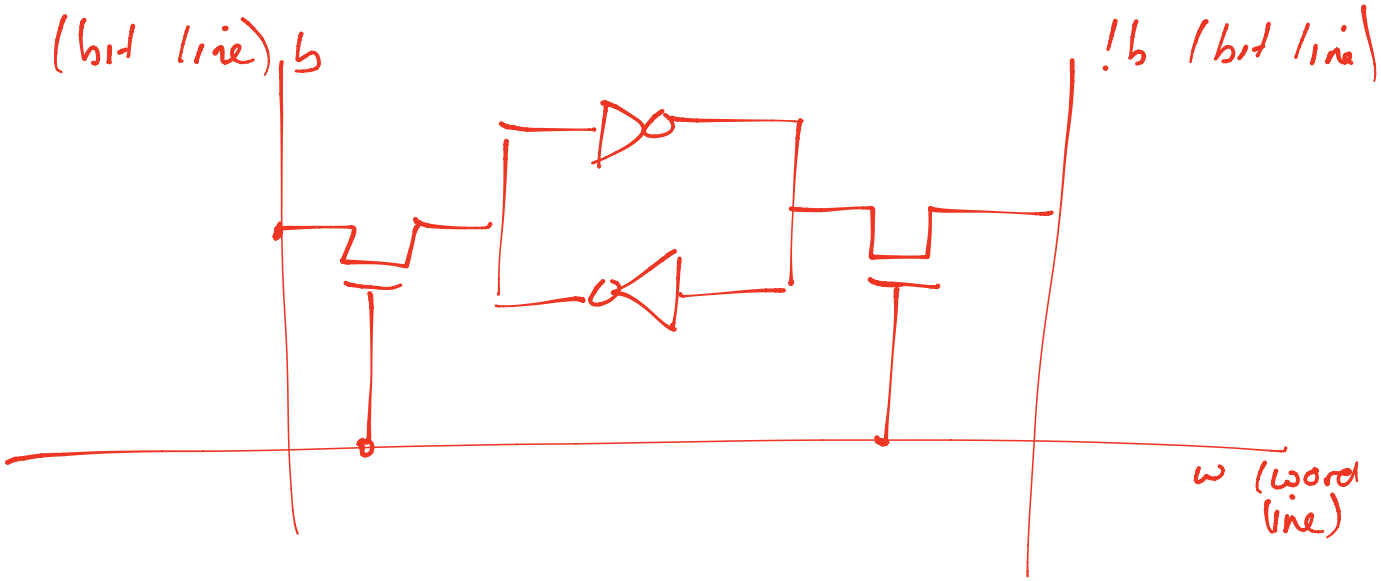


Example Question 2

- build 64KB of storage (8-bit-wide bus)
  - note: 64KB = 2^16 i.e., 16 bit addr space



AN SRAM CELL



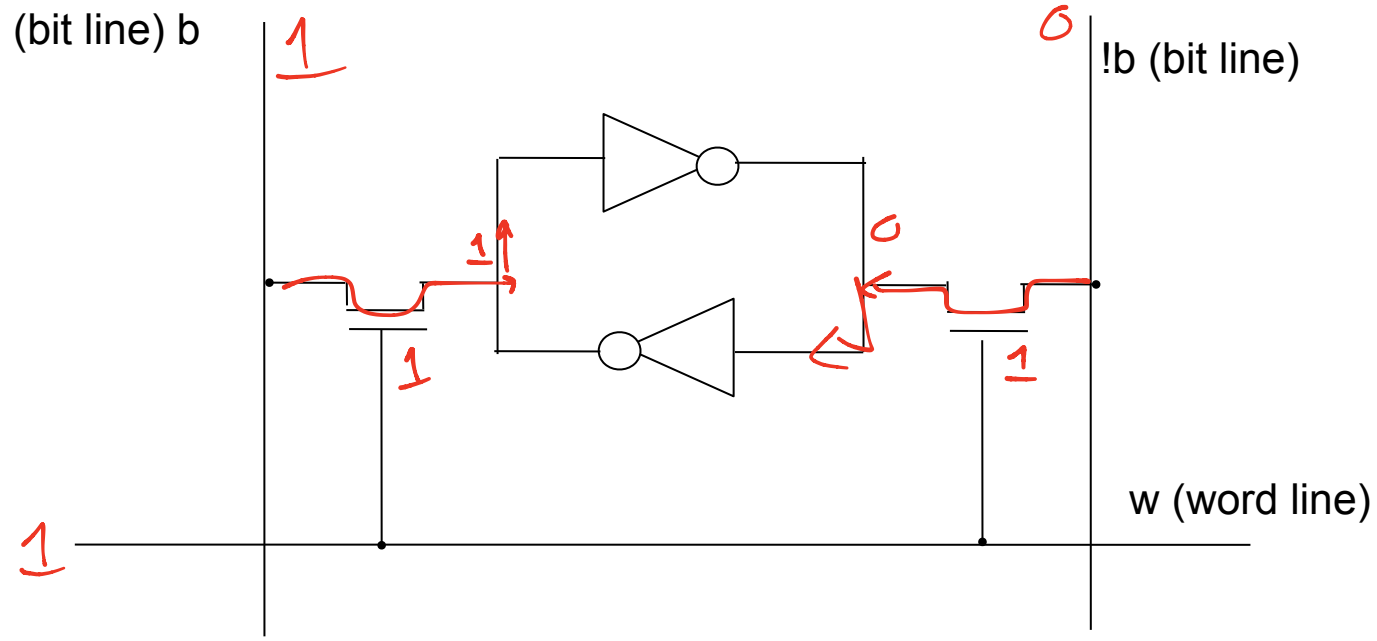
- SRAM: Static RAM
- READ: set w, “sense” b and !b

- WRITE: drive b and !b, then set w to “strobe” in value
  -  cell == 2 transistors, therefore 6 transistors for a SRAM

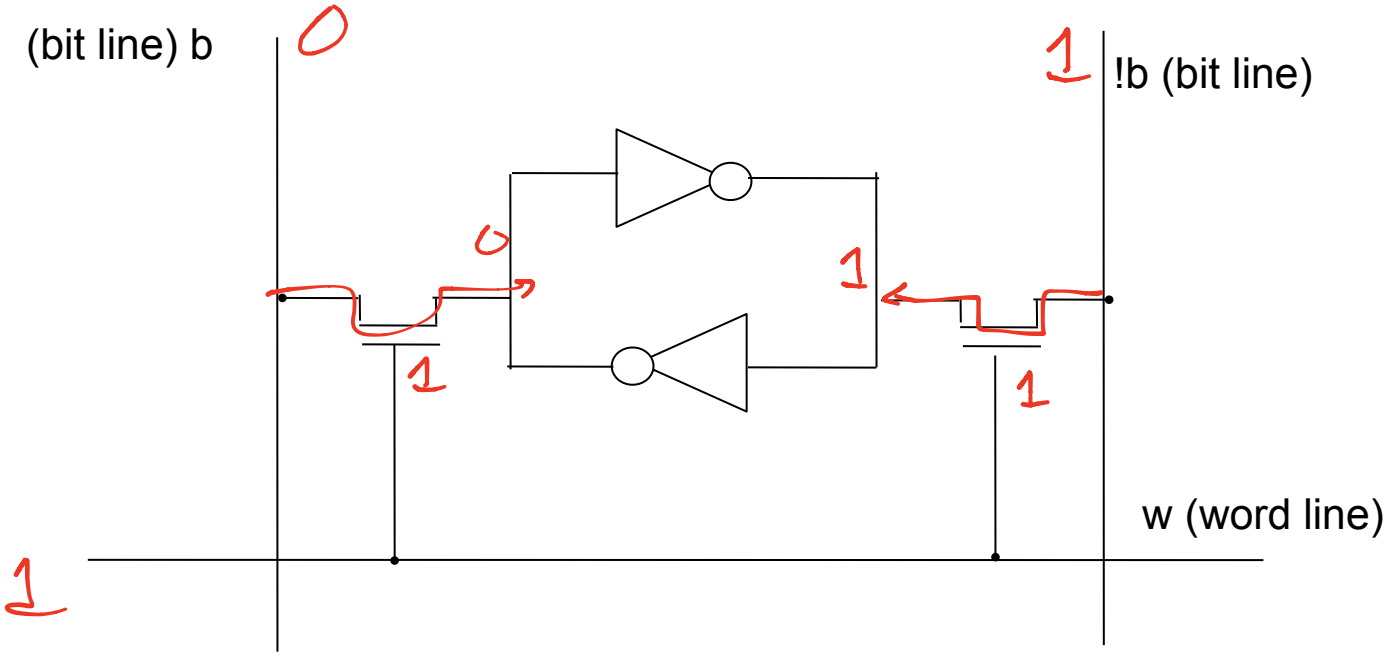
6T

SRAM CELL EXAMPLE:

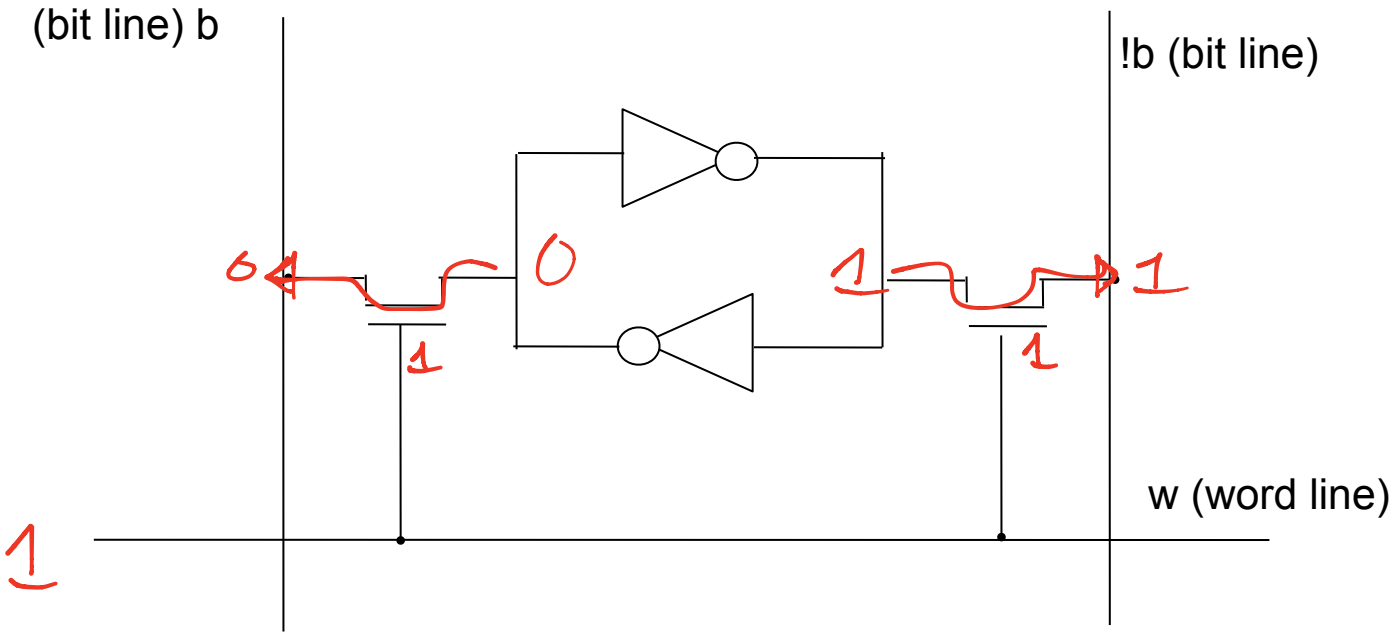
- Write a 1:



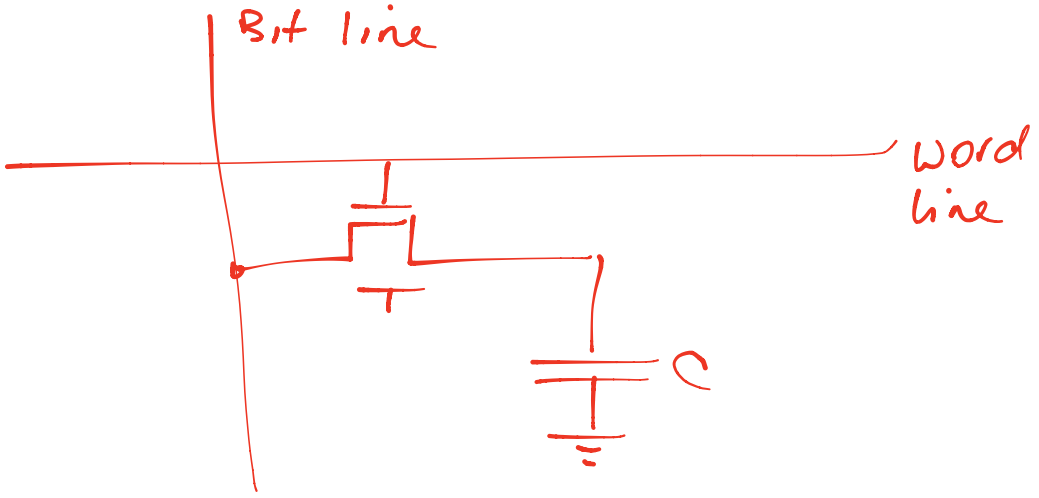
- Write a 0:



- Read (when cell is set to zero):



DRAM CELL

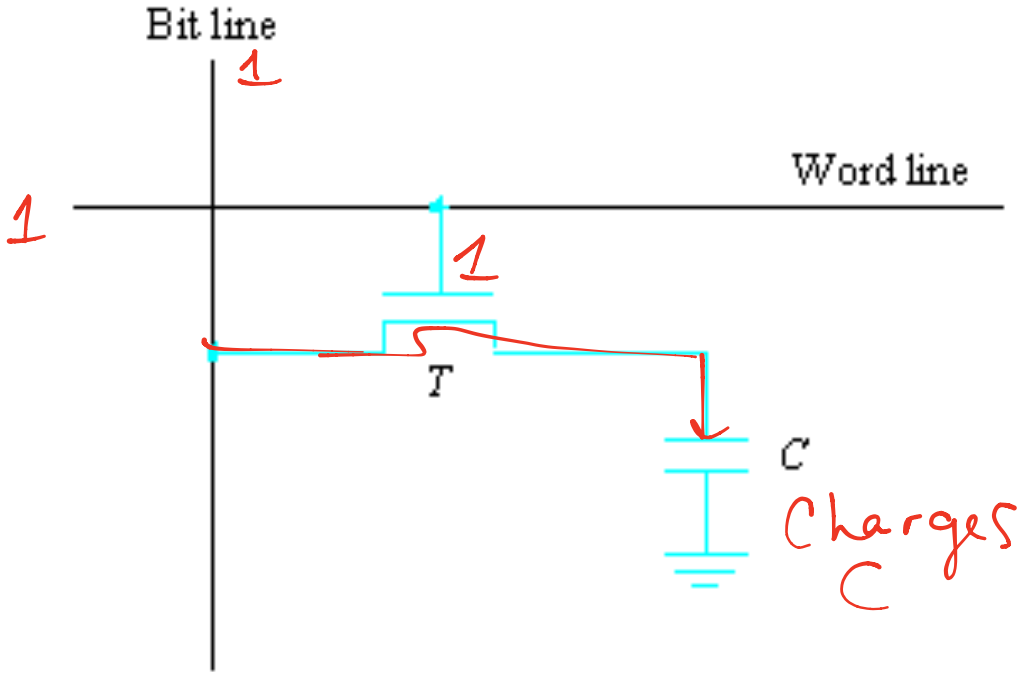


- 0 or 1 stored in capacitor C
- Read: Set word line, look for charge on bitline (sense)
  - This drains the capacitor a little if it is storing a 1
- Write:
  - deliver charge on bitline to write ‘1’
  - drain charge to write‘0’
  - Then set w to “strobe” in value

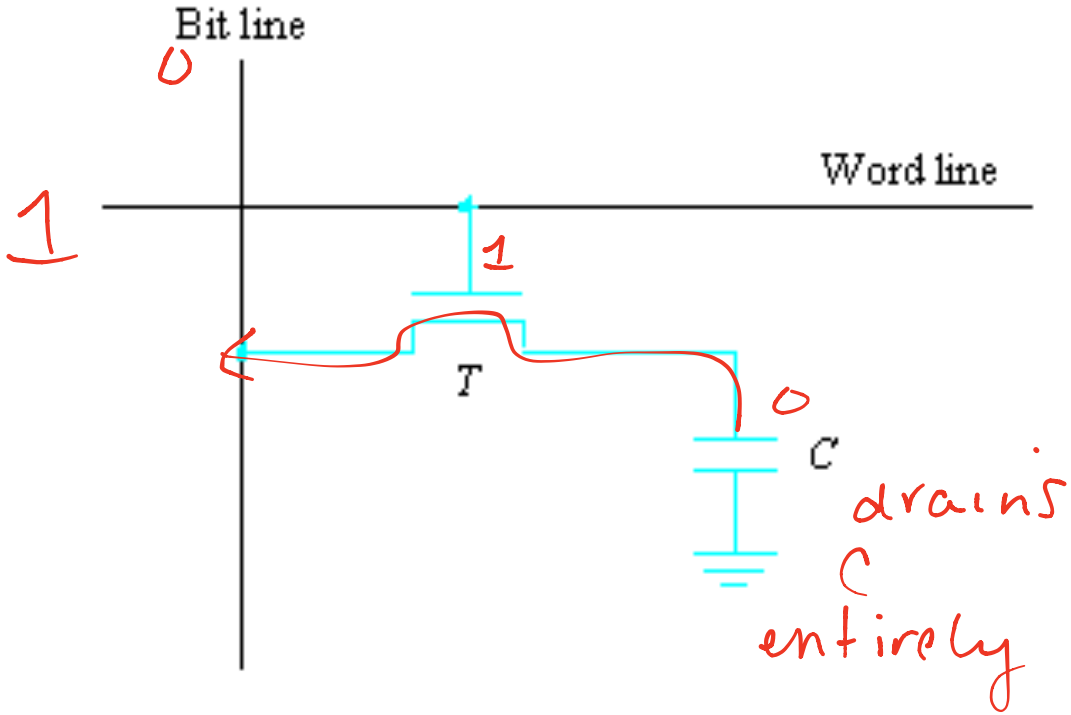
- must frequently recharge the capacitor (because it leaks)
  - called “refresh”, must do this for every cell
  - managed/performed by a dedicated memory controller

DRAM CELL EXAMPLE

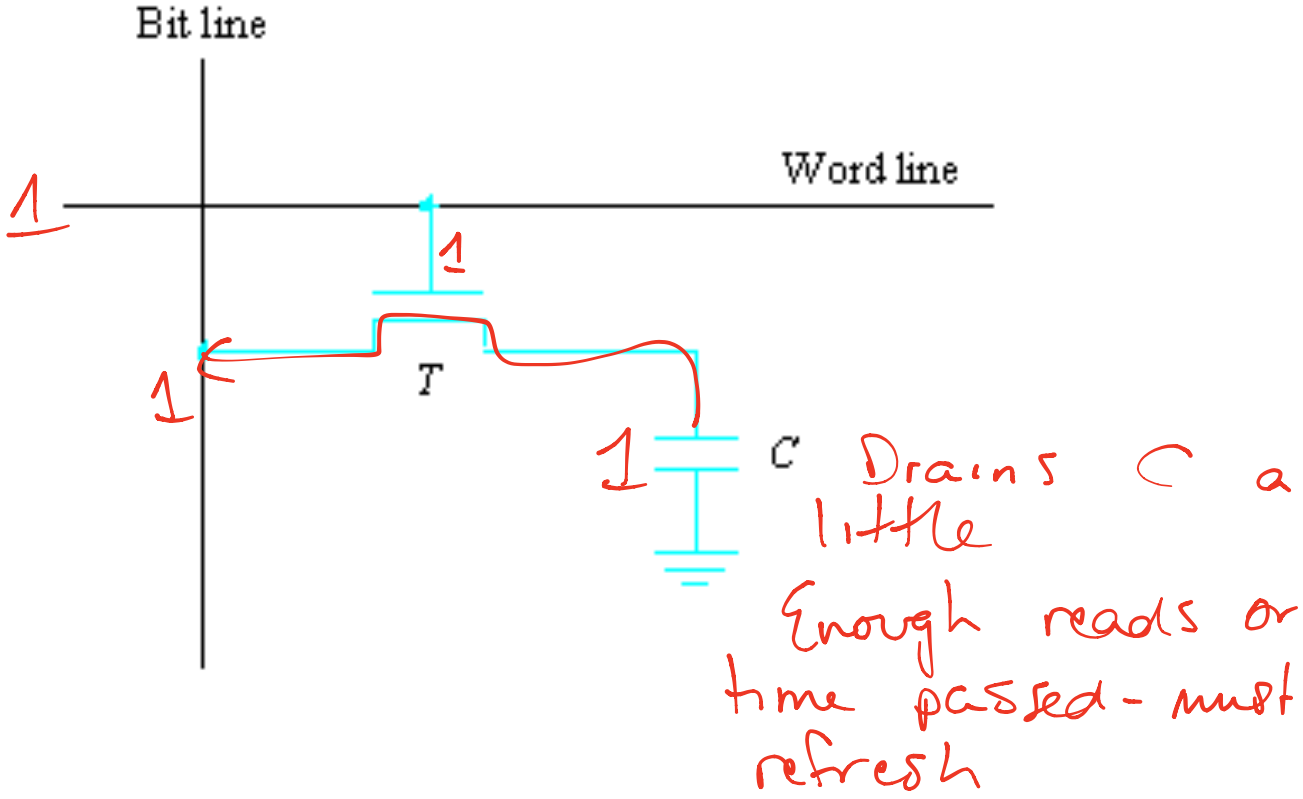
- Write a 1:



- Write a 0:



- Read (when cell set to 1):



COMPARE SRAM TO DRAM

- SRAM:
  - 6 transistors---expensive
  - large but fast
  - no refresh necessary
- DRAM:
  - 1 transistor + 1 capacitor---cheap
  - small but slower
  - refresh necessary

TYPES OF DRAM

- DRAM:

  - dynamic RAM
  - asynchronous, needs to be refreshed
- SDRAM:
  - synchronous DRAM
    - synchronized with a clock signal
  - Can enter a ‘mode’ to do a “block transfer“
    - many bytes with one request from the CPU
- DDR-SDRAM: double-data rate SDRAM
  - Transfer data on both edges of the clock
  - rather than just the rising edge