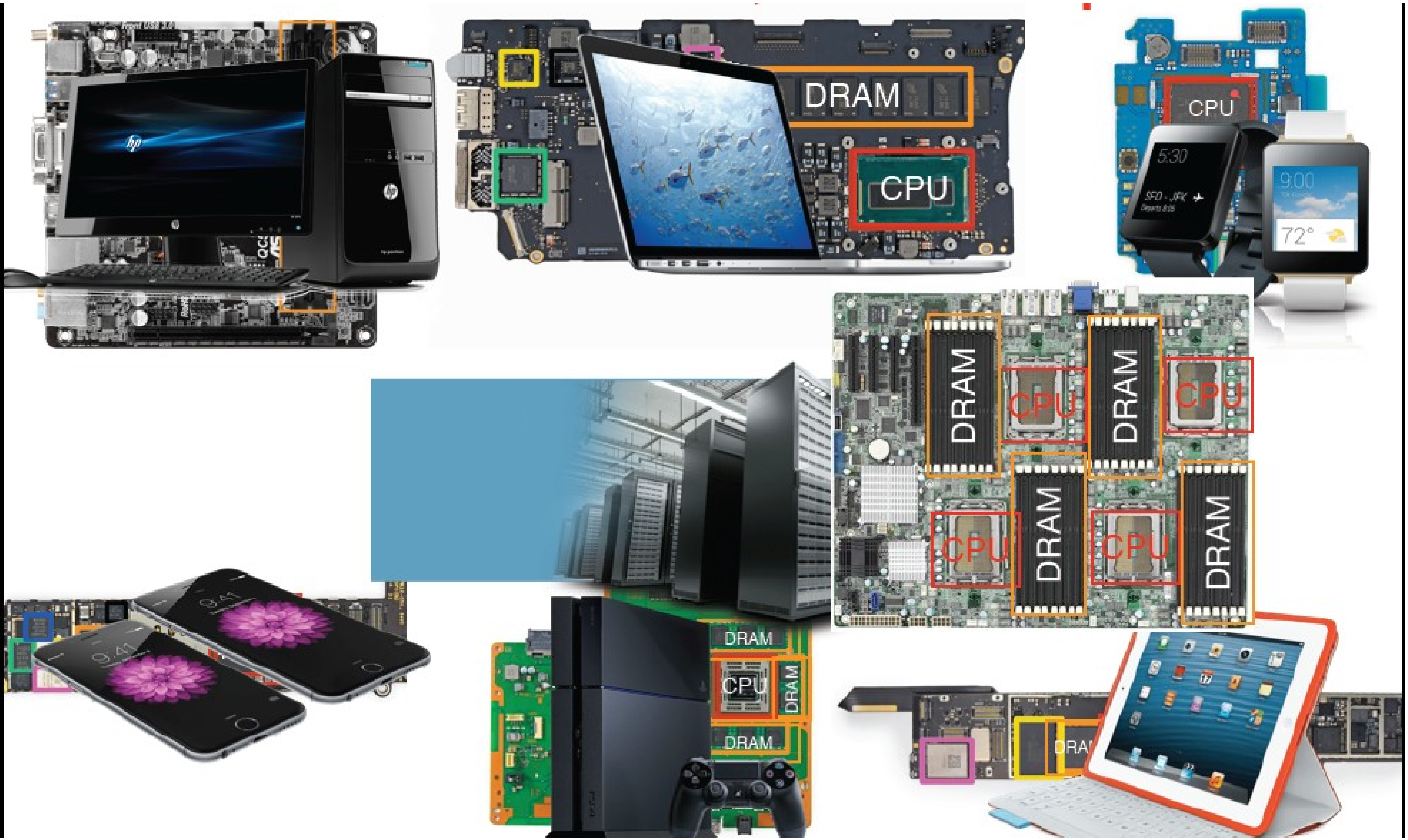
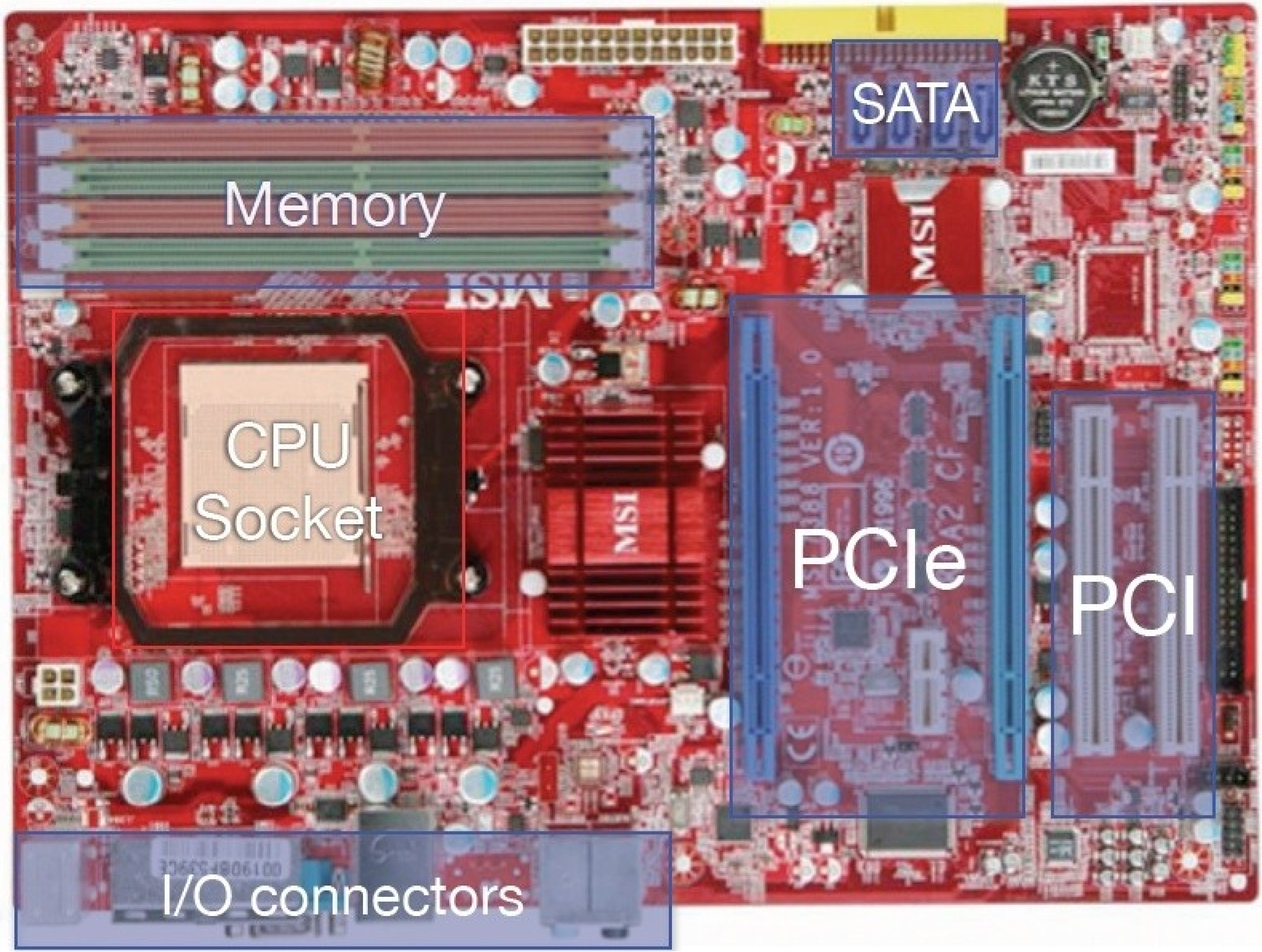
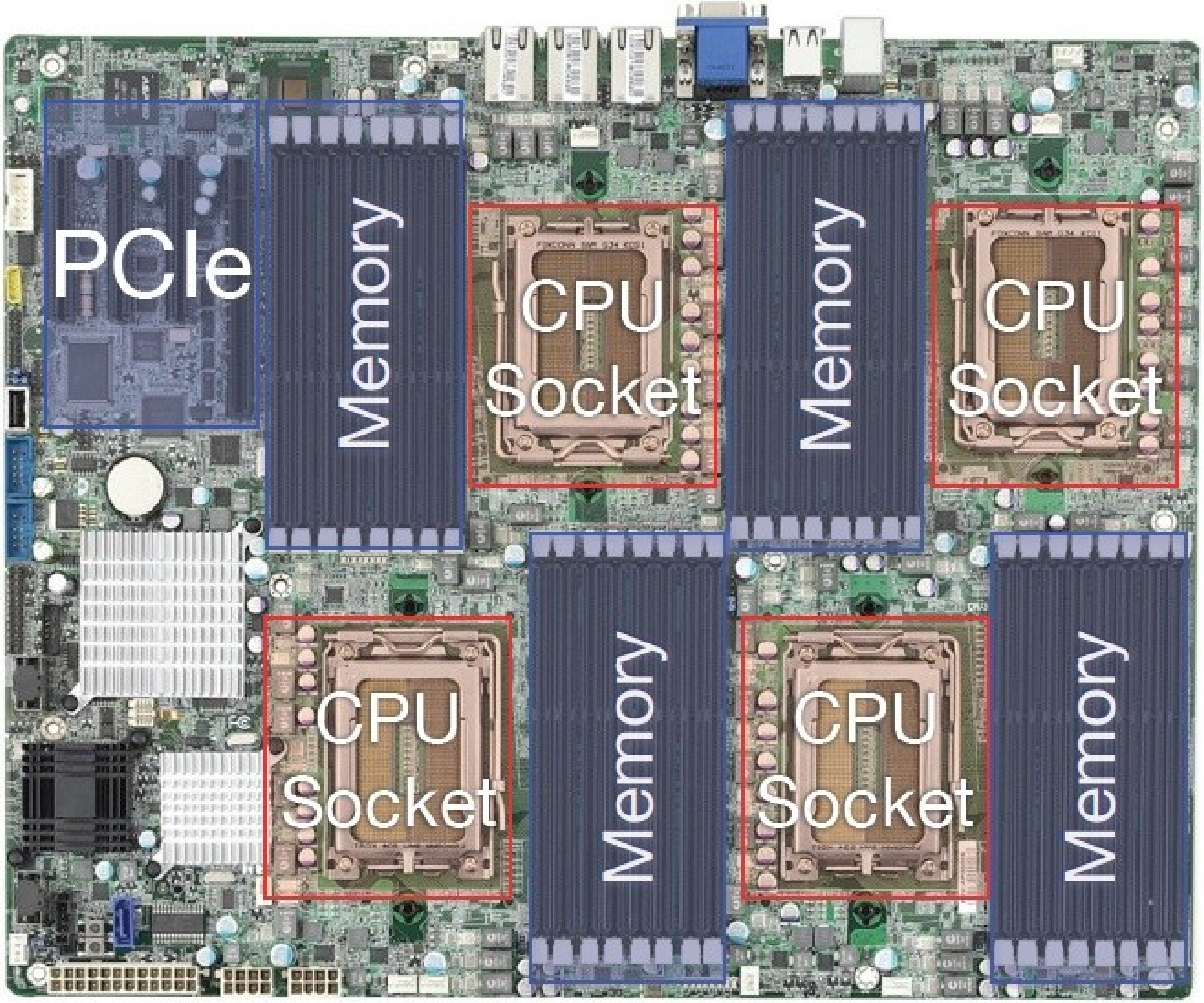
Systems


DRAM
CPU
CPU
5:30
SFD - JFK
9:00
72°
DRAM
CPU
DRAM
CPU
CPU
DRAM
CPU
DRAM
9:41
DRAM
CPU
DRAM
DRAM
DRAI

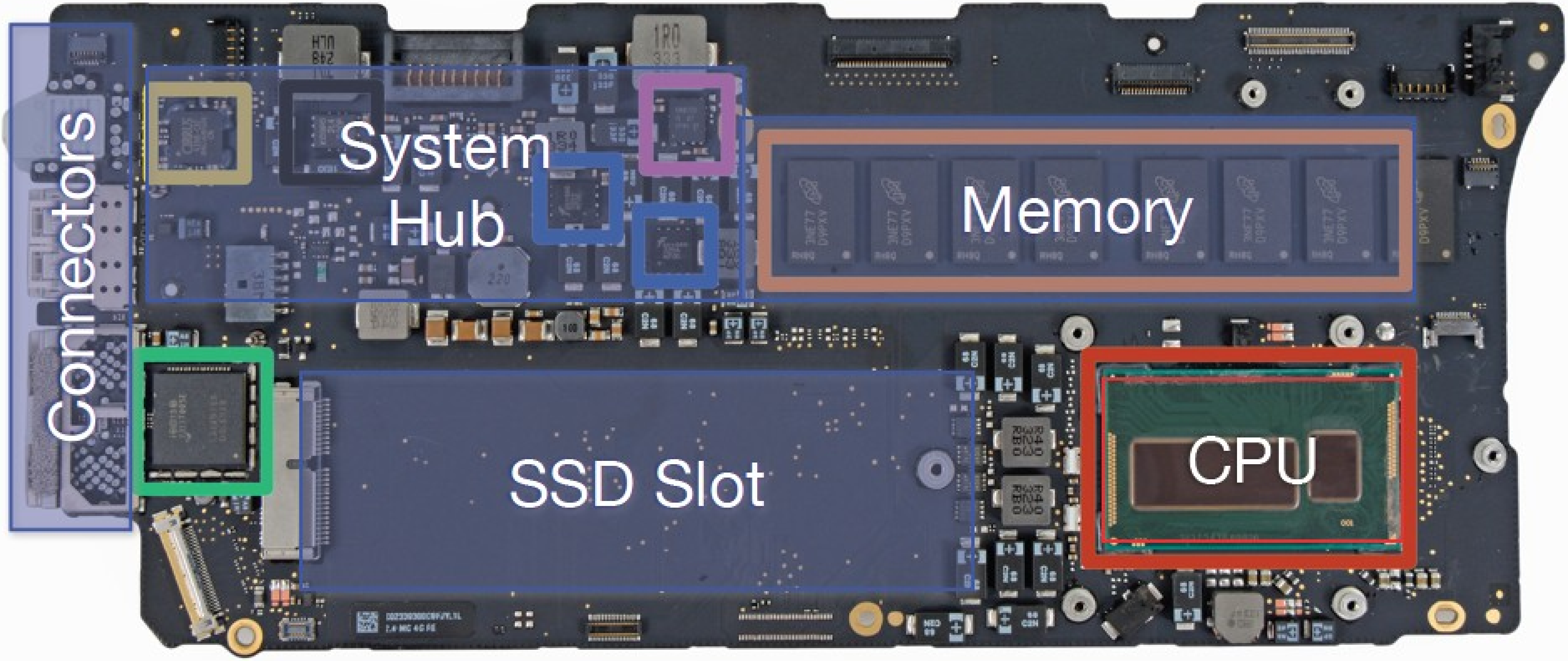
Desktop PC



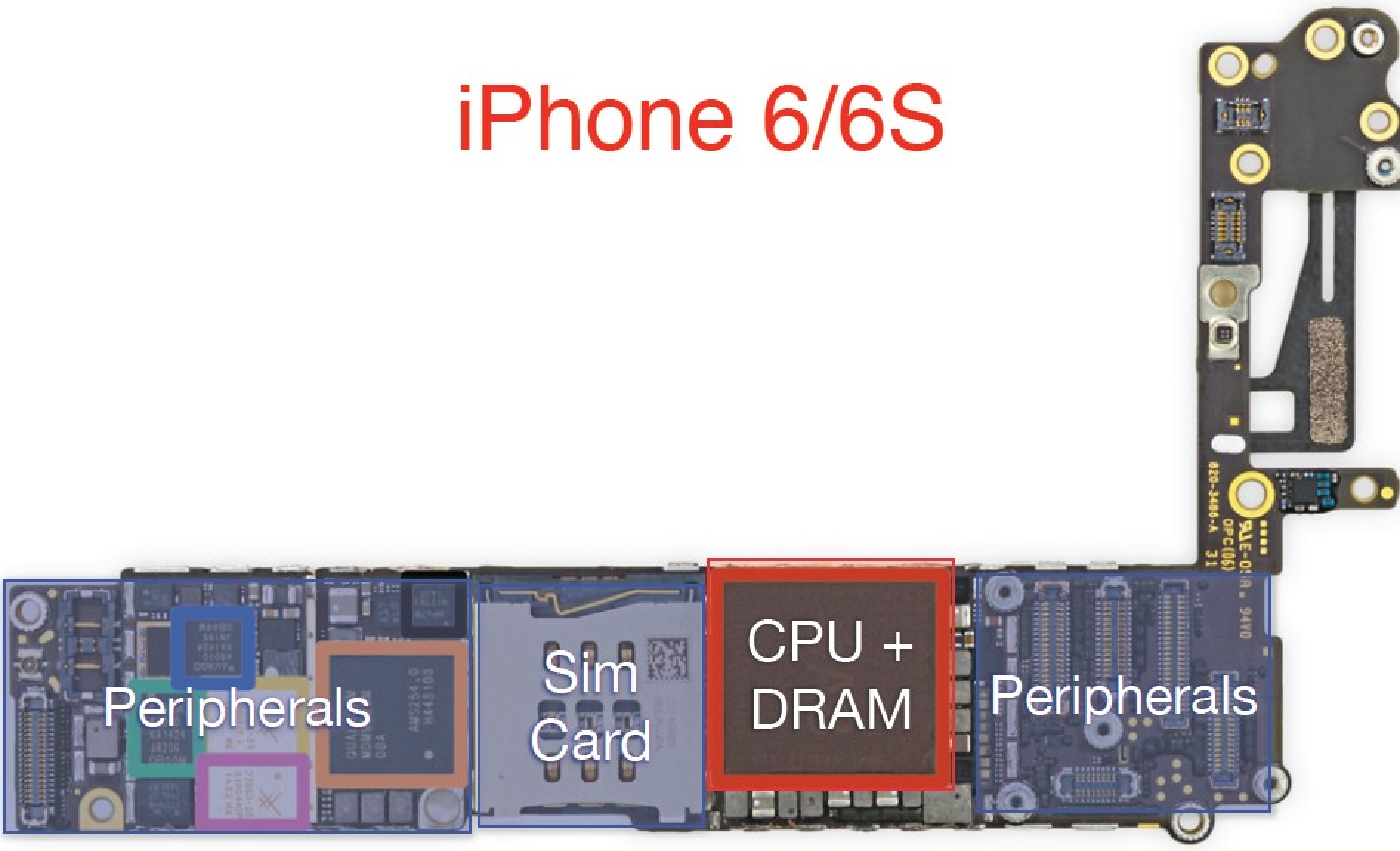
Server


PCIe
Memory
CPU Socket
Memory
CPU Socket
CPU Socket
Memory
CPU Socket
Memory

Macbook Pro w/ Retina



iPhone 6/6S



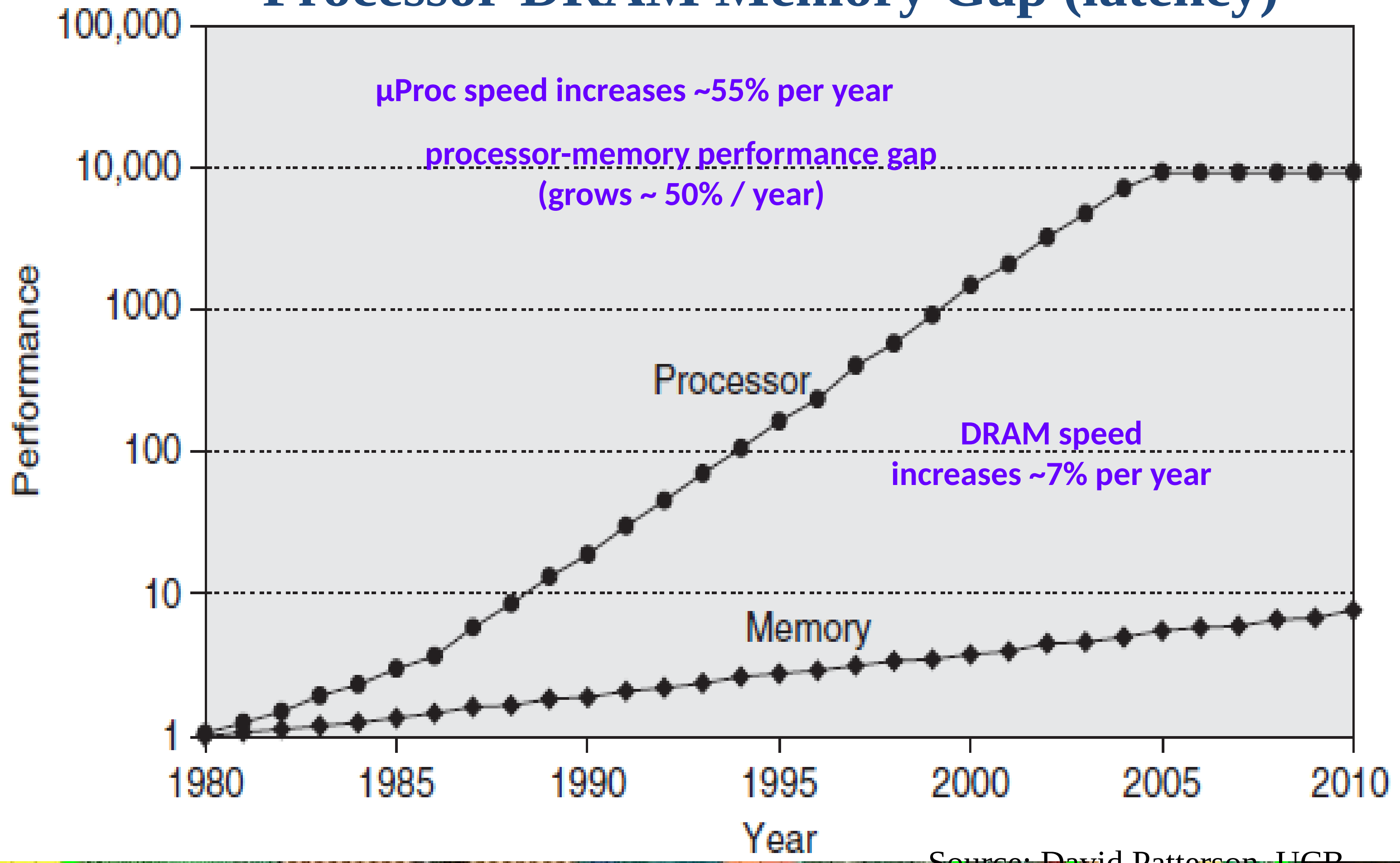
Memory in a Modern System

Overview

- **Review of Memory Technologies**
- **Overview of Memory Hierarchy**
- **Cache Design Principles**

- Learning Objectives
  - Why is that some memories slow ?
  - What is memory hierarchy?
  - Why do we need memory hierarchy?
  - What is a Cache?

Processor-DRAM Memory Gap (latency)



µProc speed increases ~55% per year

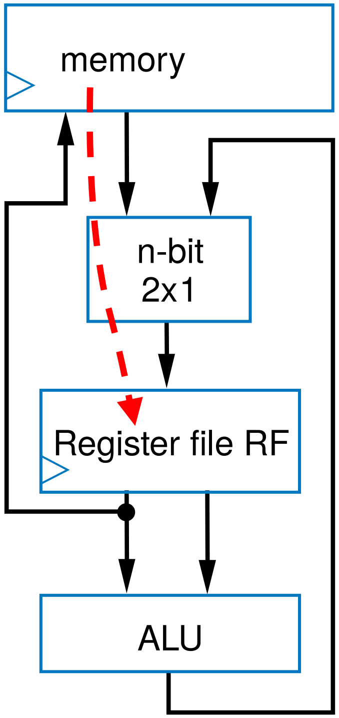
processor-memory performance gap
(grows ~ 50% / year)

DRAM speed
increases ~7% per year

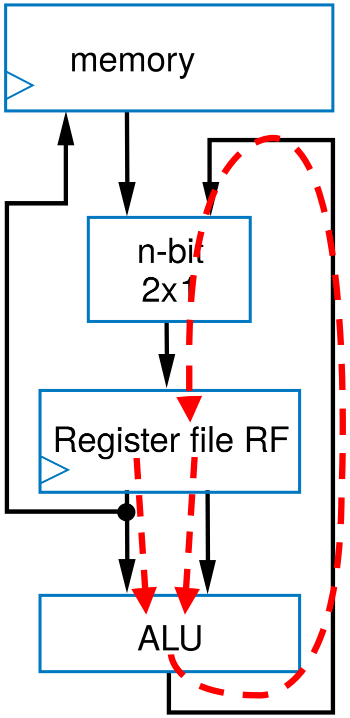
Source: David Patterson, UCB

Review: Datapath Operations

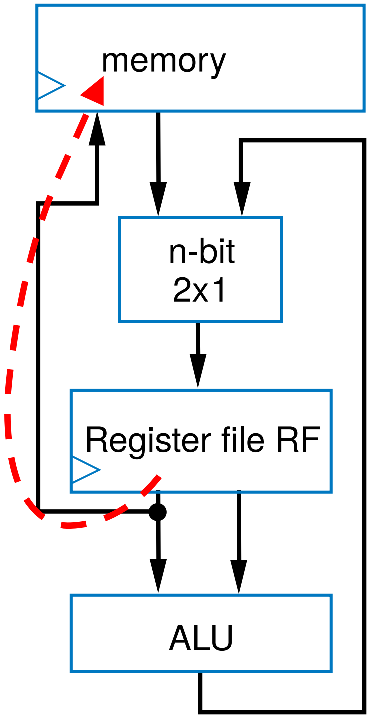
- Load operation: Load data from data memory to RF
- ALU operation: Transforms data by passing one or two RF register values through ALU, performing operation (ADD, SUB, AND, OR, etc.), and writing back into RF.
- Store operation: Stores RF register value back into data memory
- Each operation can be done in one clock cycle



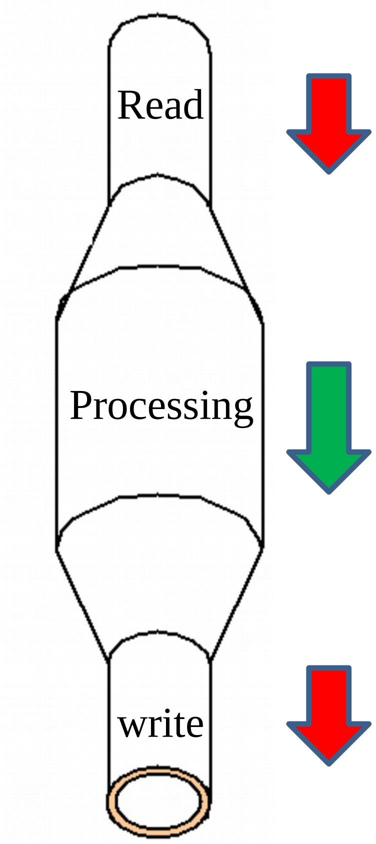
Load operation



ALU operation

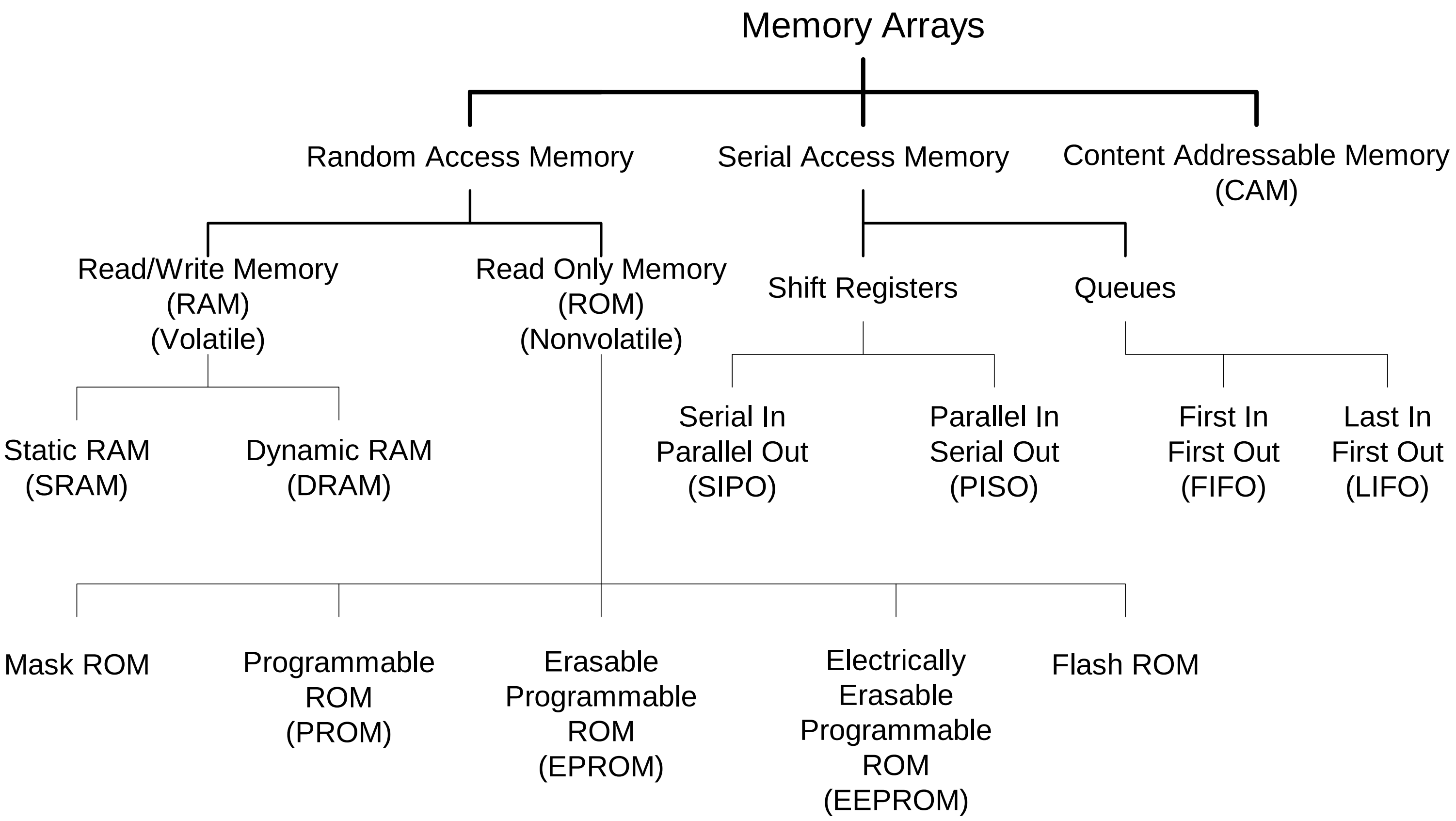


Store operation



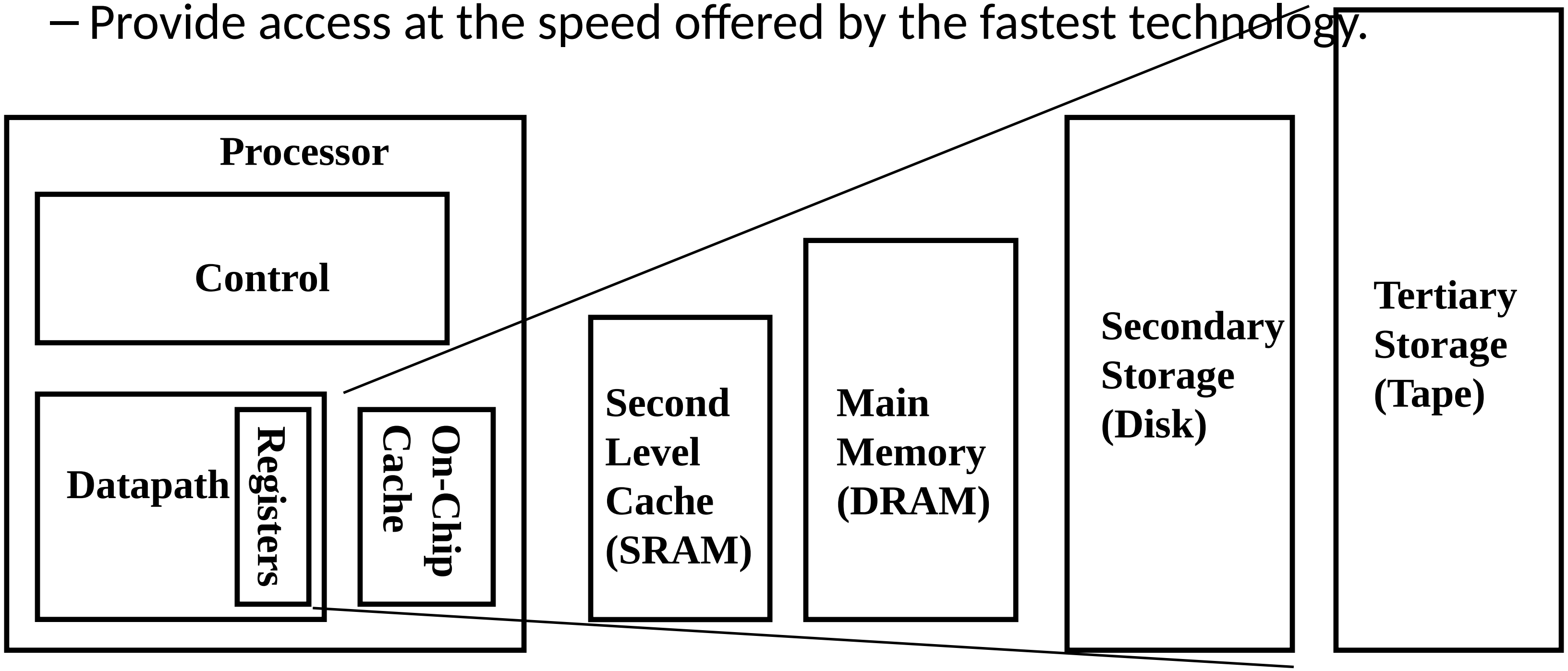
What are the Common Memory Technologies?

Memory


Memory Arrays
Random Access Memory
Serial Access Memory
Content Addressable Memory (CAM)
Read/Write Memory (RAM) (Volatile)
Read Only Memory (ROM) (Nonvolatile)
Shift Registers
Queues
Static RAM (SRAM)
Dynamic RAM (DRAM)
Serial In Parallel Out (SIPO)
Parallel In Serial Out (PISO)
First In First Out (FIFO)
Last In First Out (LIFO)
Mask ROM
Programmable ROM (PROM)
Erasable Programmable ROM (EPROM)
Electrically Erasable Programmable ROM (EEPROM)
Flash ROM

Memory Hierarchy of a Modern Computer System

- By taking advantage of the principle of locality:
  - – Present the user with as much memory as is available in the cheapest technology.
  - – Provide access at the speed offered by the fastest technology.



Processor

Control

Datapath

Registers

On-Chip Cache

Second Level Cache (SRAM)

Main Memory (DRAM)

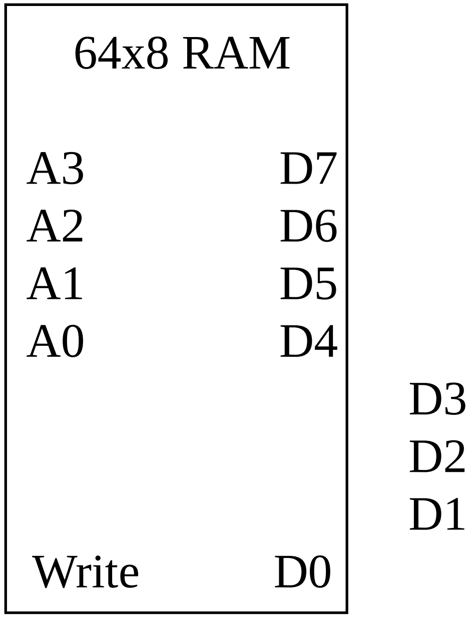
Secondary Storage (Disk)

Tertiary Storage (Tape)

Speed (ns):	1s	10s	100s	10,000,000s (10s ms)	10,000,000,000s (10s sec)
Size (bytes):	100s	Ks	Ms	Gs	Ts

Memory

Address	Data
000000	00111110
000001	01101011
000010	01011101
000011	01100011
000100	00111110
000101	00000000
000110	11111111
000111	01010101
001000	10101010
001001	00100001
001010	11011010

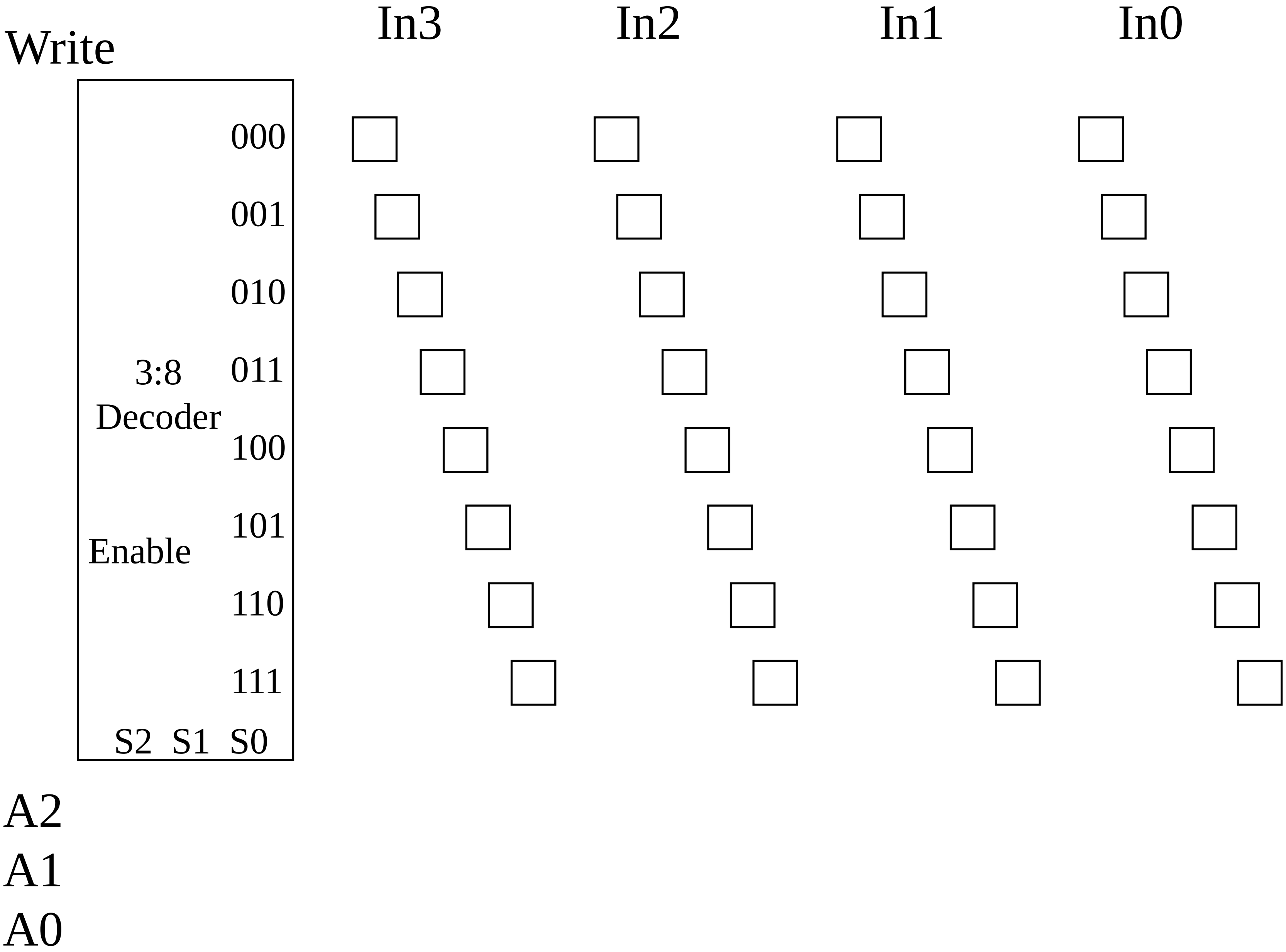


8x4 RAM

Address	Data			
000	□	□	□	□
001	□	□	□	□
010	□	□	□	□
011	□	□	□	□
100	□	□	□	□
101	□	□	□	□
110	□	□	□	□
111	□	□	□	□

A2
A1
A0

8x4 RAM


Write
In3
In2
In1
In0
000
001
010
011
3:8
Decoder
100
101
Enable
110
111
S2 S1 S0
A2
A1
A0

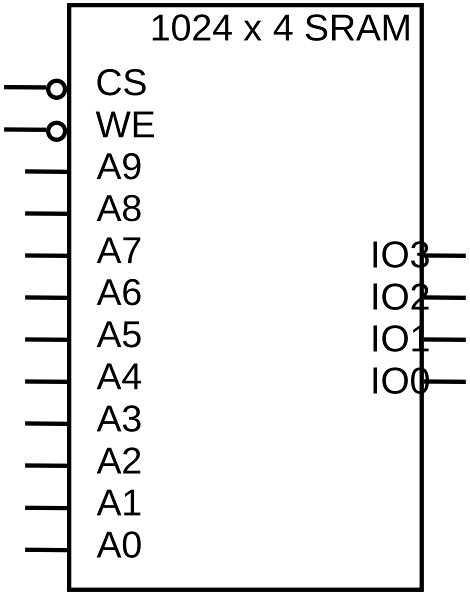
Static RAM Organization

Chip Select Line (active lo)

Write Enable Line (active lo)

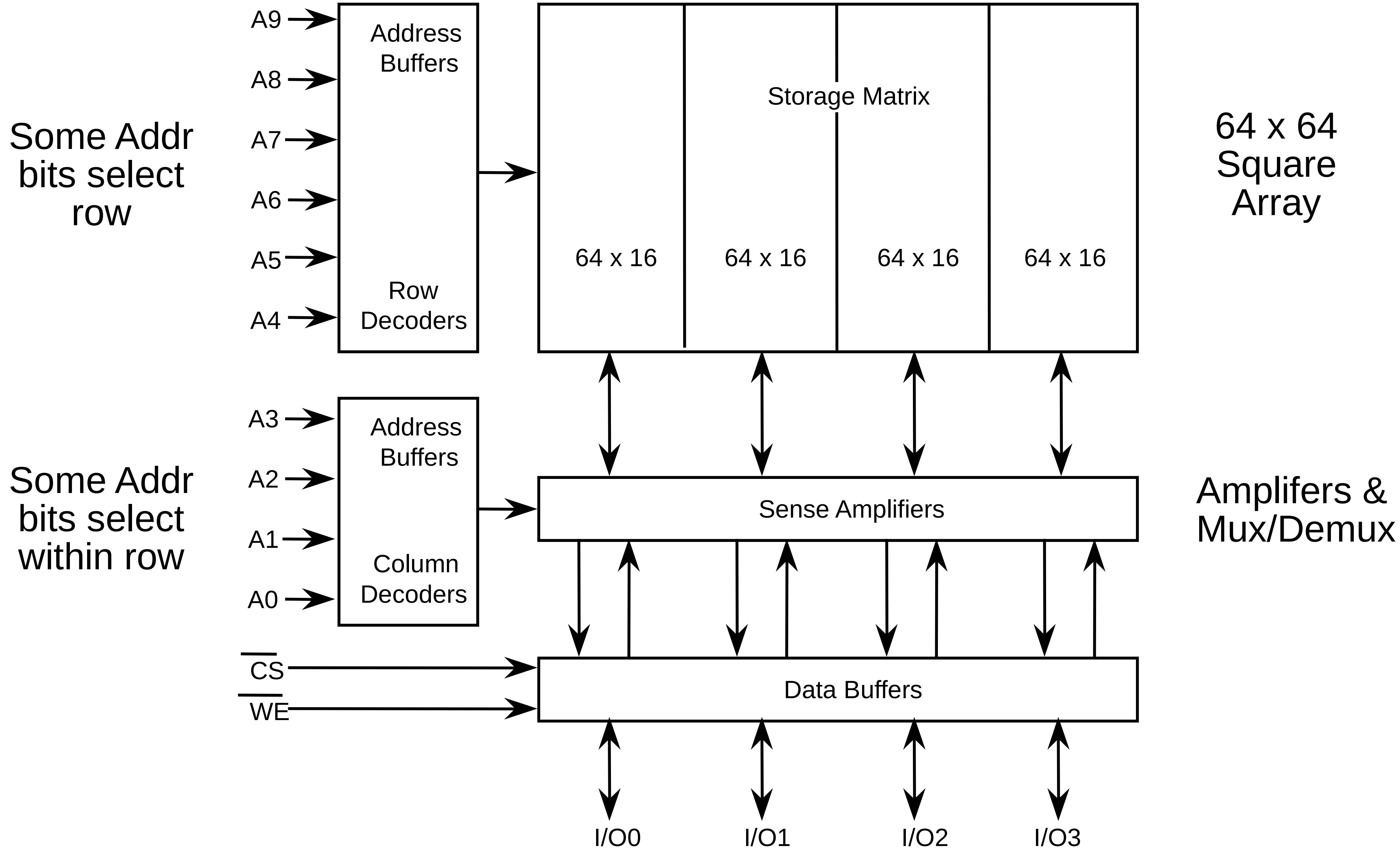
10 Address Lines

4 Bidirectional Data Lines

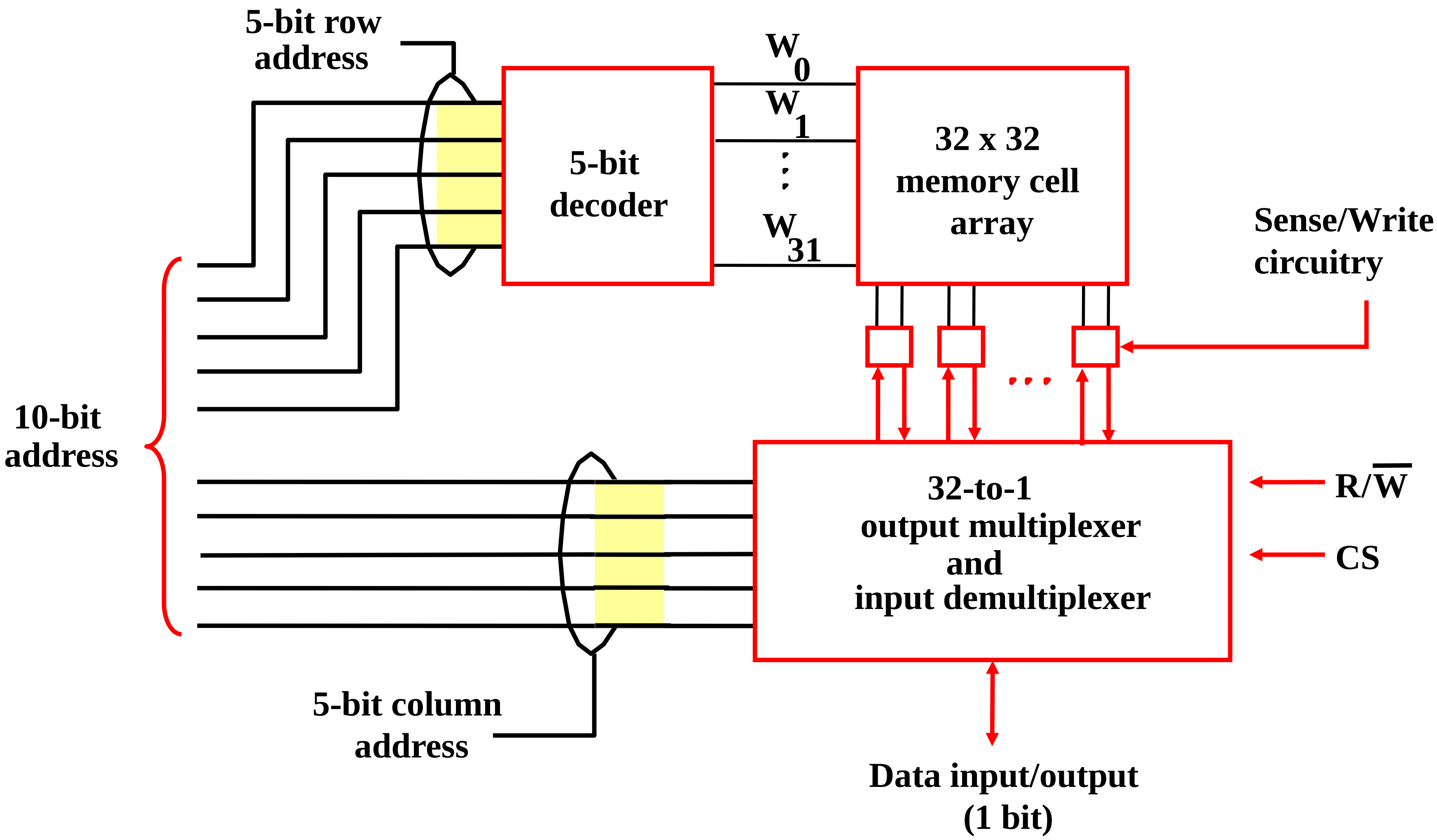


RAM Organization

Long thin layouts are not the best organization for a RAM



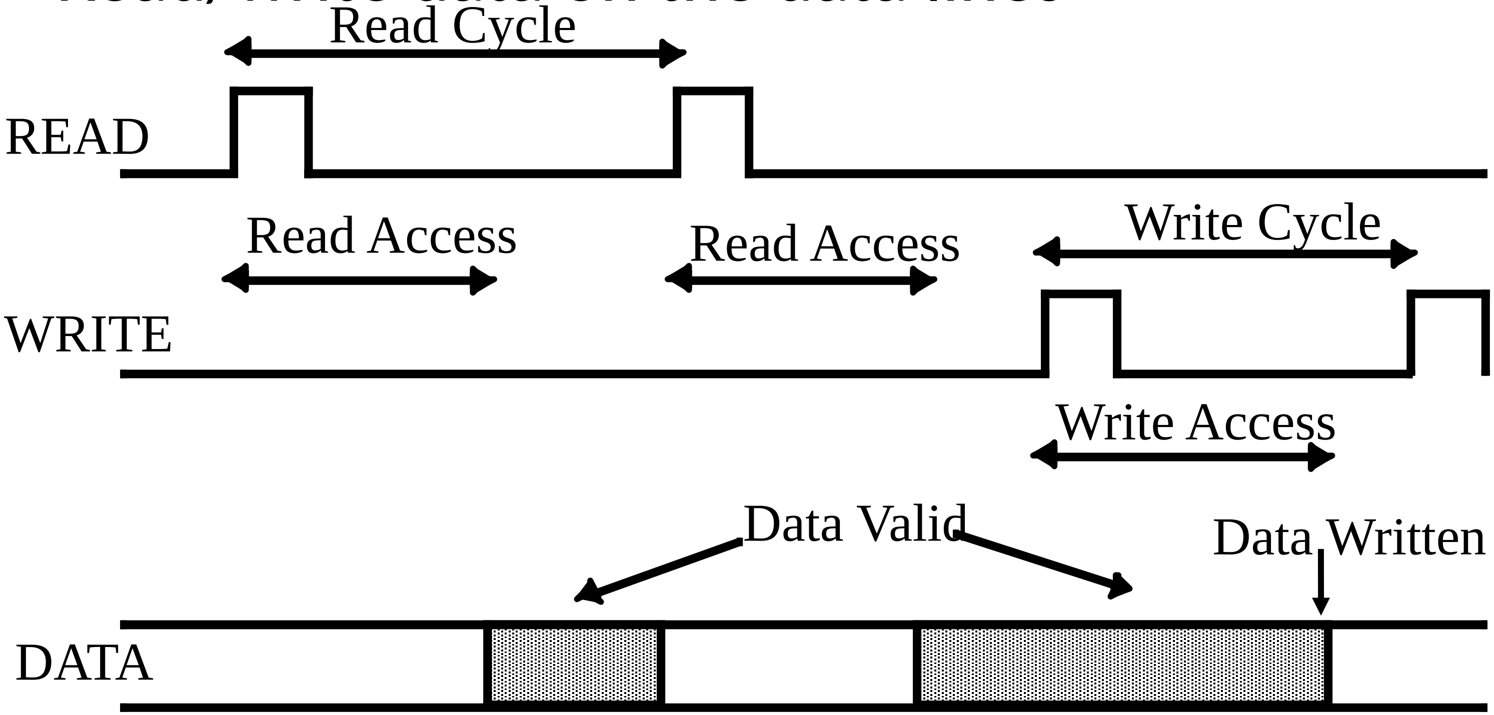
1Kx 1 bit RAM Organization



Organization of a 1K × 1 memory chip.

Memory Access Timing: the Big Picture

- Timing:
  - Send address on the address lines, wait for the word line to become stable
  - Read/write data on the data lines

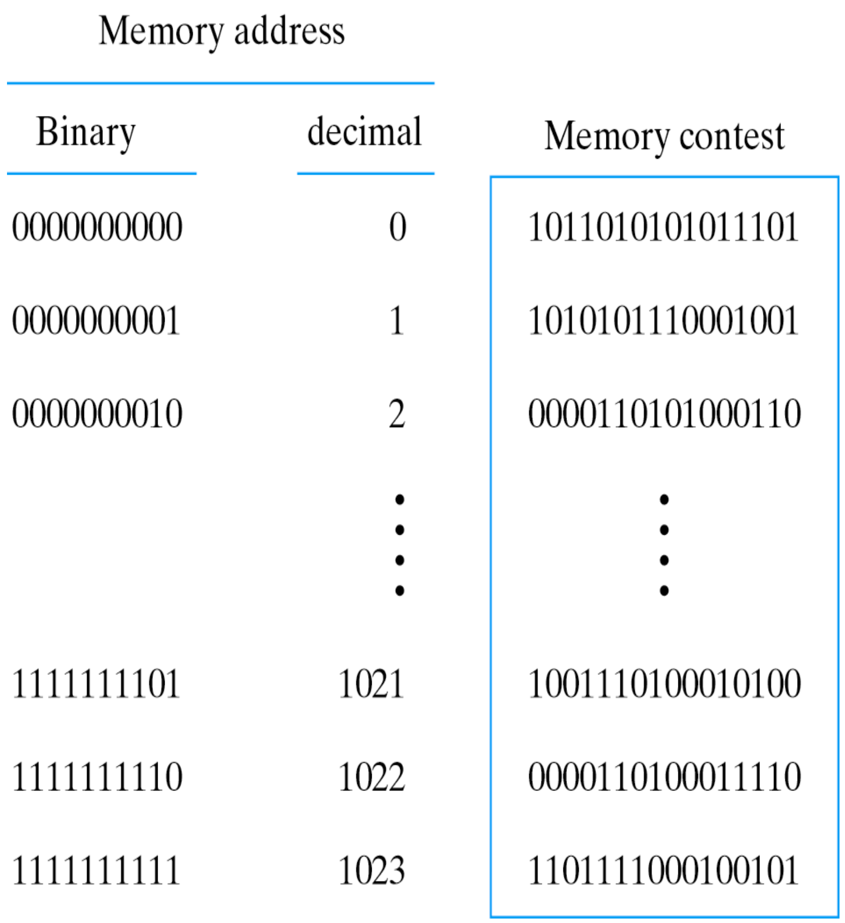


Content of a memory

- Each word in memory is assigned an identification number, called an address, starting from 0 up to 2^k-1, where k is the number of address lines.
- The number of words in a memory with one of the letters K=2^{10}, M=2^{20}, or G=2^{30}.

64K = 2^{16} 2M = 2^{21}

4G = 2^{32}

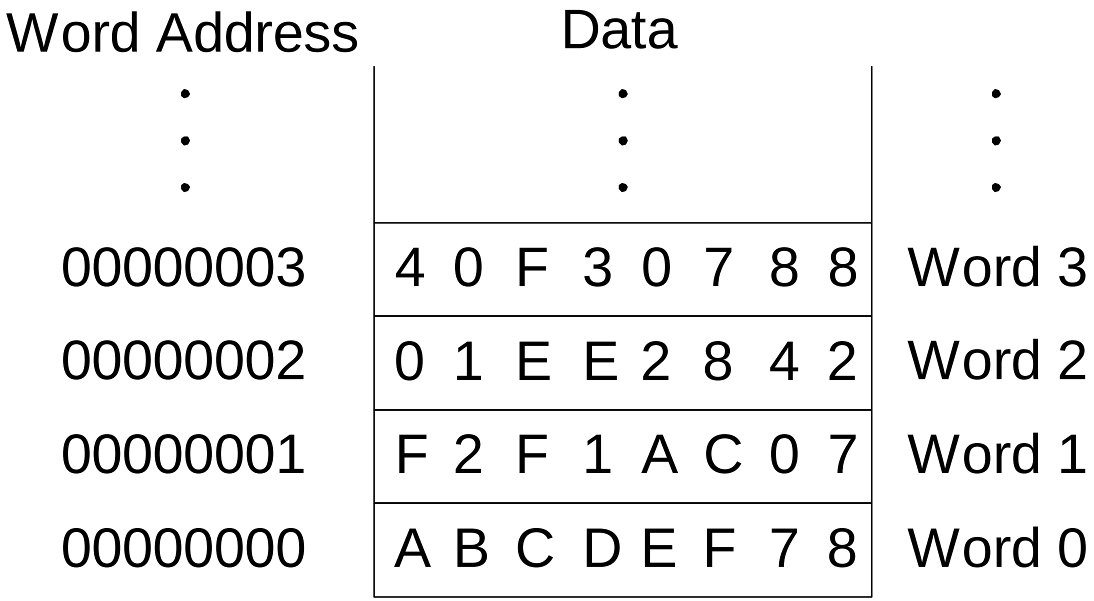


Memory address		
Binary	decimal	Memory contest
0000000000	0	1011010101011101
0000000001	1	1010101110001001
0000000010	2	0000110101000110
	⋮	⋮
1111111101	1021	1001110100010100
1111111110	1022	0000110100011110
1111111111	1023	1101111000100101

1024x16 Memory Module

Word-Addressable Memory

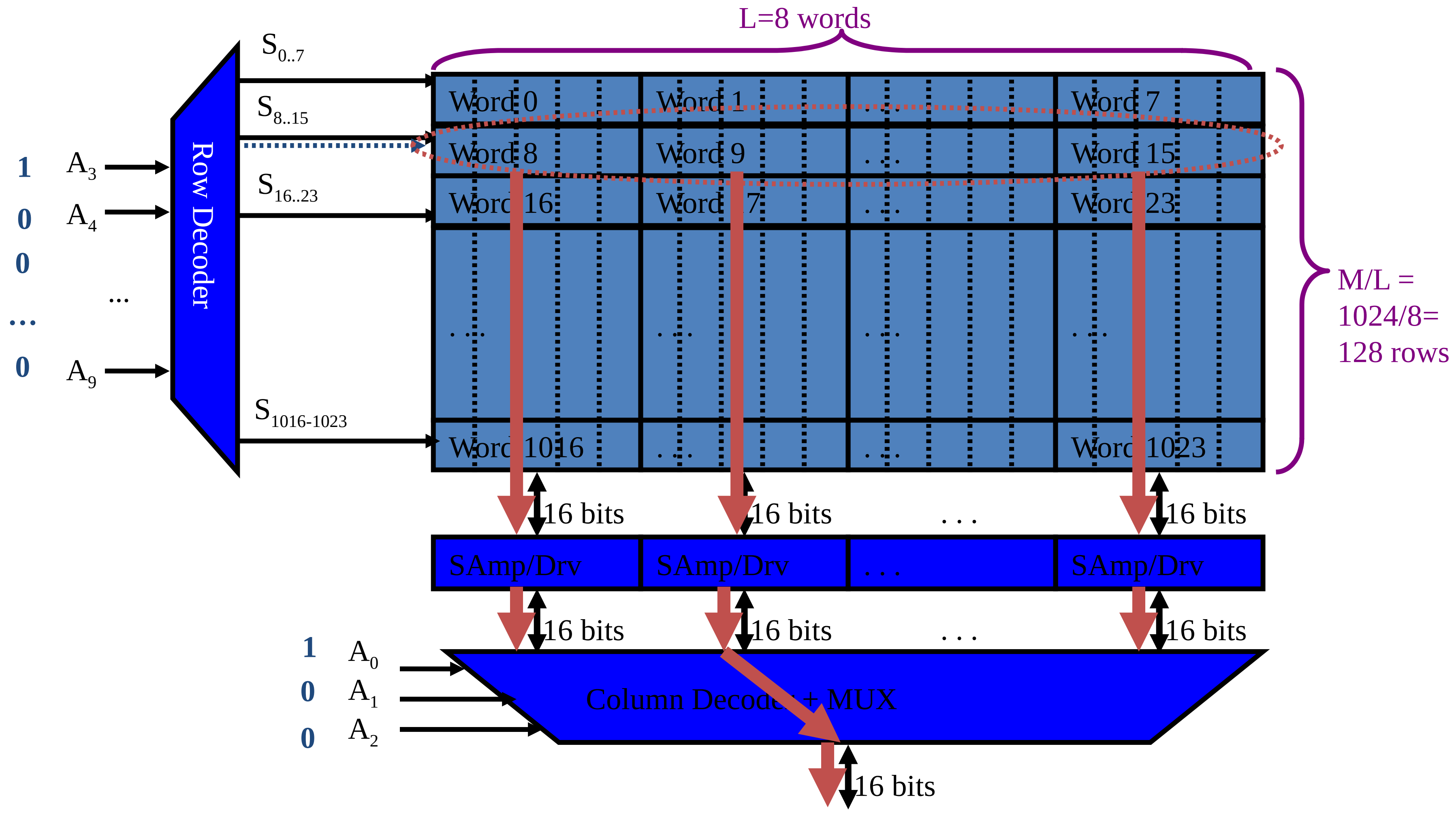
- Each 32-bit data word has a unique address



Word Address	Data	
⋮	⋮	⋮
00000003	40F30788	Word 3
00000002	01EE2842	Word 2
00000001	F2F1AC07	Word 1
00000000	ABCDEF78	Word 0

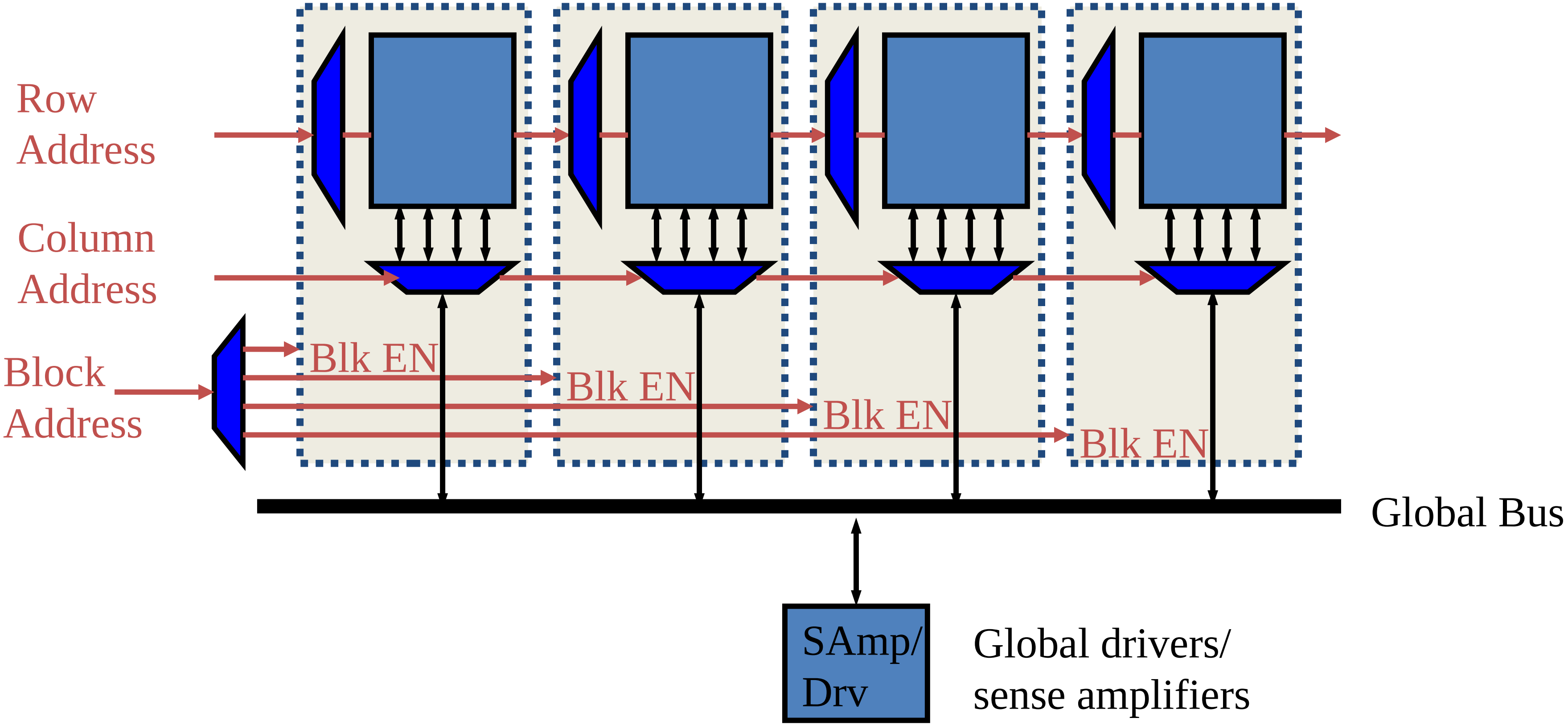
Memory Cell Array Access Example

- word=16-bit wide(N), row=8 words(L), address=10 bits (k)
- Accessing word 9= 0000001001_2



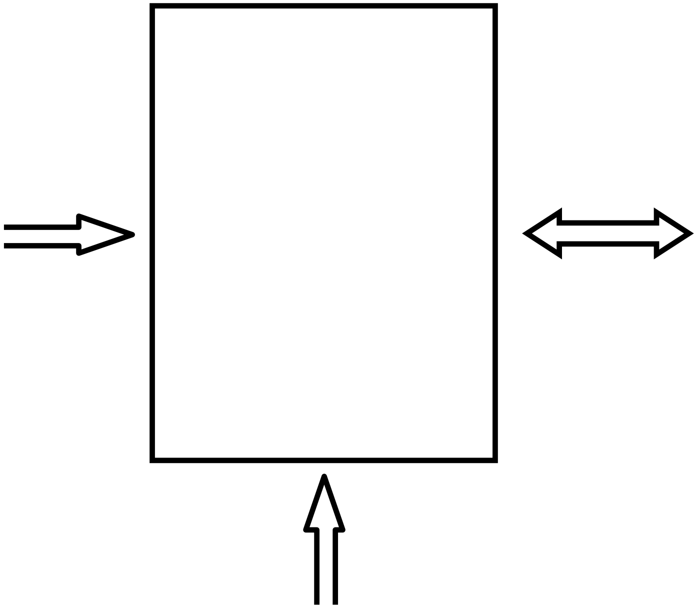
Memory Structures

- Taking the idea one step further
  - Shorter wires within each block
  - Enable only one block addr decoder ㅍ power savings



Larger Memories Using Multiple Chips

512K x 8 memory chip

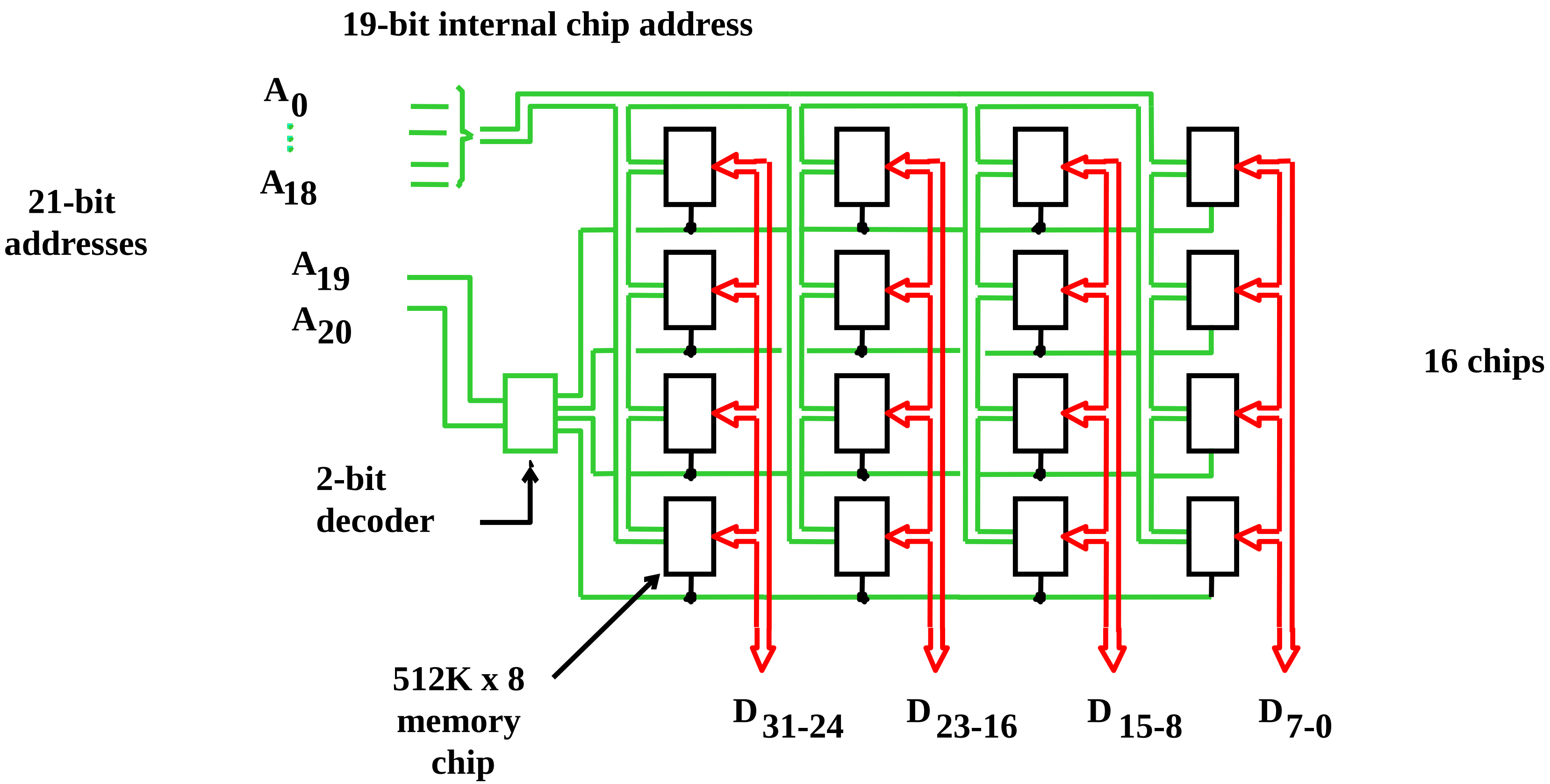


19 bit address on chip

8-bit data input/output

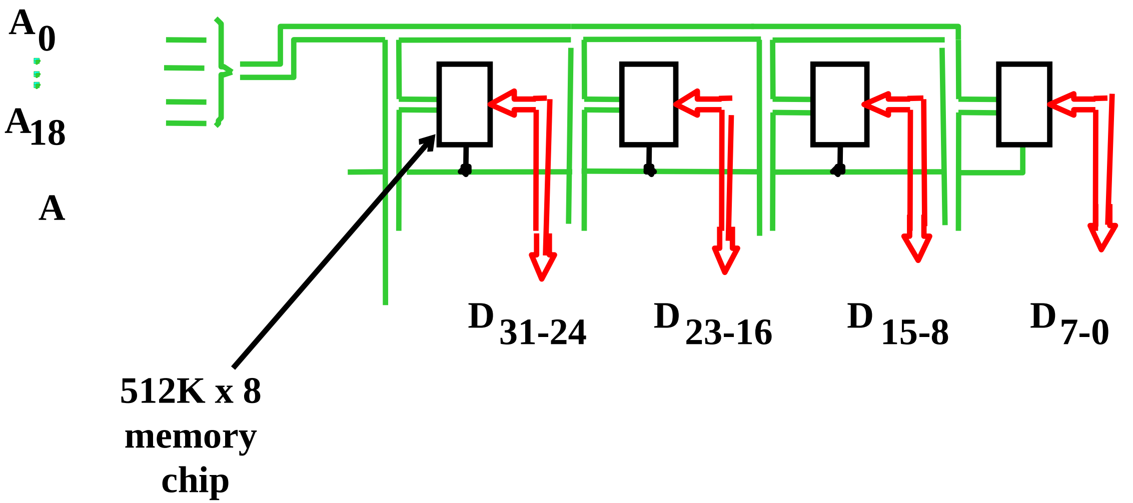
chip select (2 bits)

Question ? Design a 2Mx32 given: 512kx8



Organization of a 2M × 32 memory module using 512K × 8 static memory chips (16 chips).

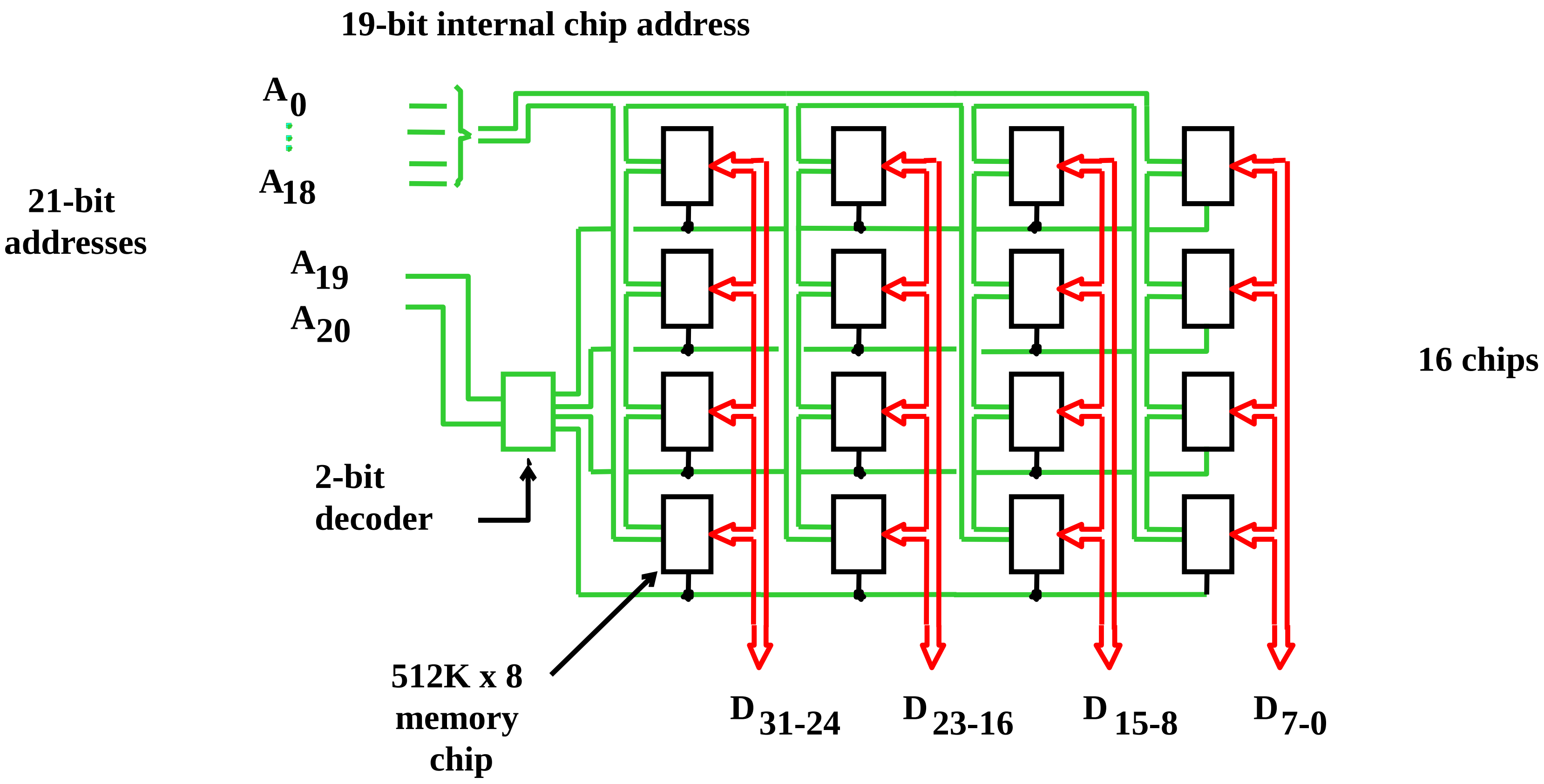
19-bit internal chip address



512K x 8
memory
chip

4 chips for each
32 bit word

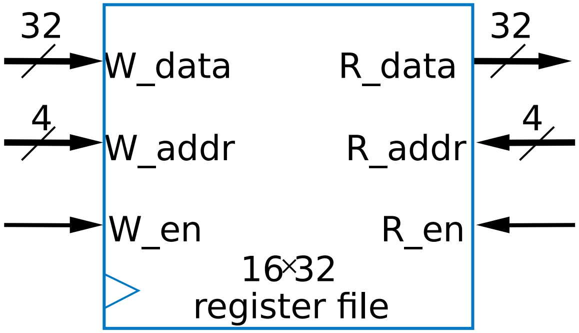
Organization of a 2M × 32 memory module using 512K × 8 static memory chips (16 chips).

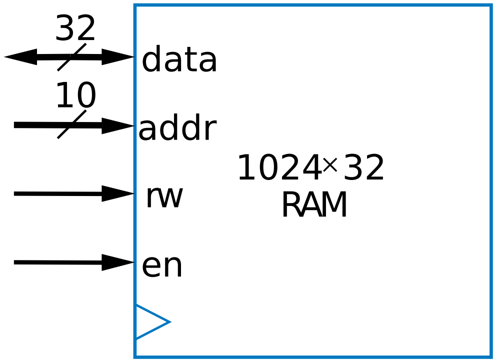


Organization of a 2M × 32 memory module using 512K × 8 static memory chips (16 chips).

Random Access Memory (RAM)

- RAM – Readable and writable memory
  - “Random access memory”
    - Strange name—Created several decades ago to contrast with sequentially-accessed storage like tape drives
  - Logically same as register file—Memory with address inputs, data inputs/outputs, and control
    - RAM usually one port; RF usually two or more
  - RAM vs. RF
    - RAM typically larger than *about* 512 or 1024 words
    - RAM typically stores bits using a bit storage approach that is more efficient than a flip-flop
    - RAM typically implemented on a chip in a square rather than rectangular shape—keeps longest wires (hence delay) short

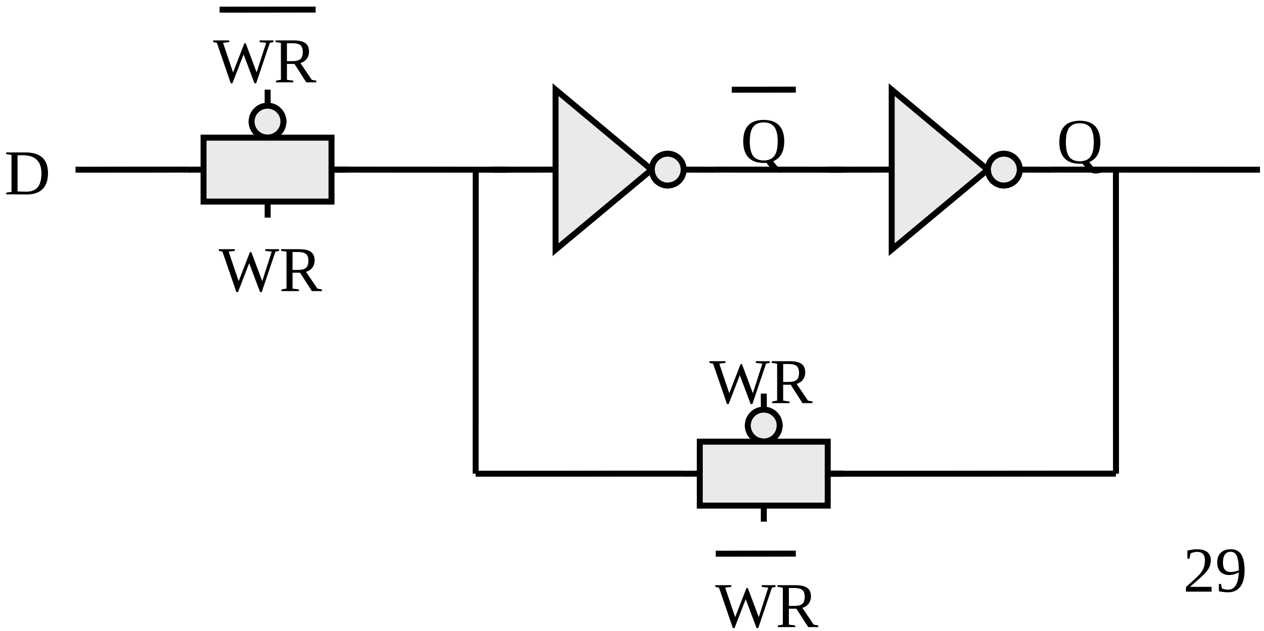




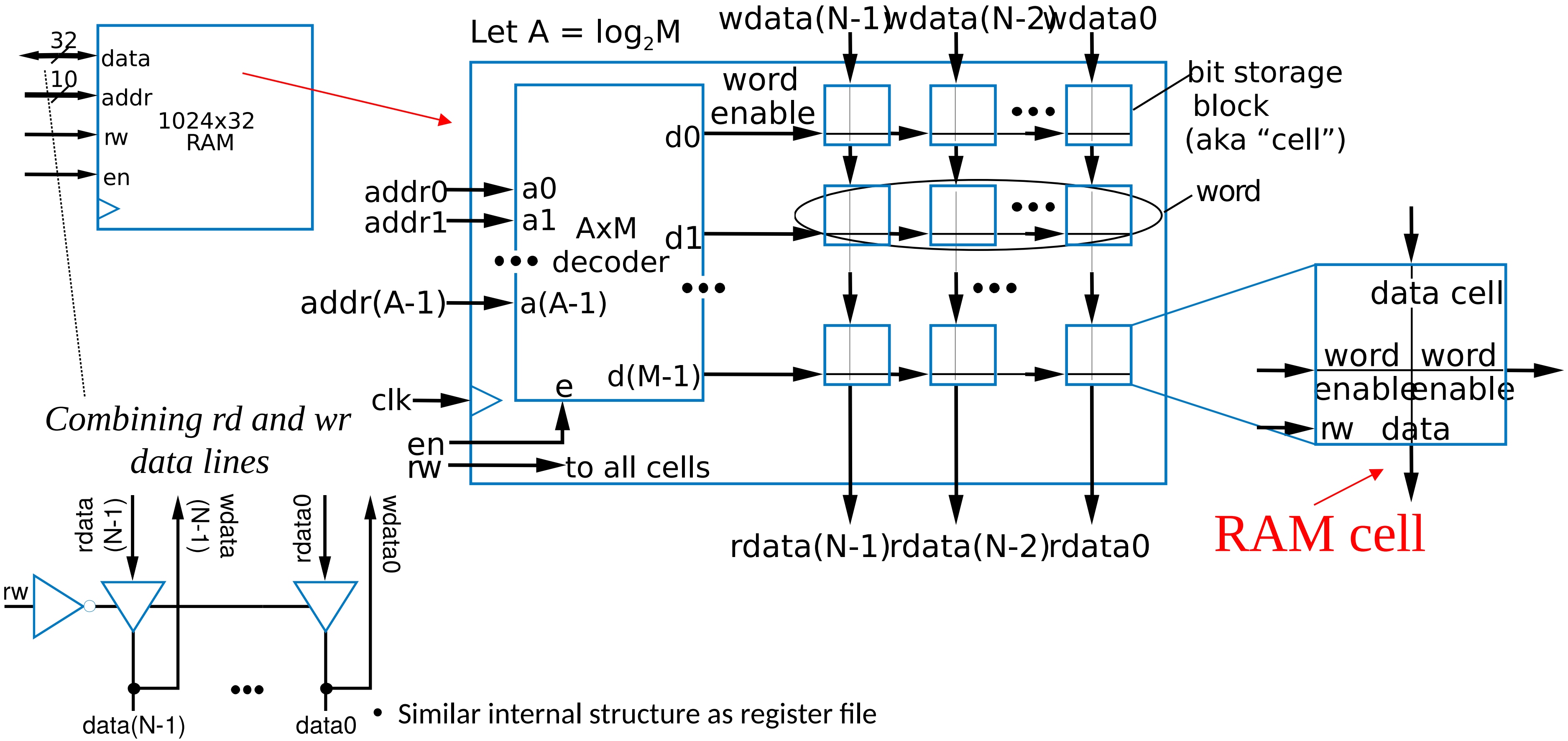
RAM block symbol

Implementing Registers in CMOS

- Uses transmission gate
  - When “WR” asserted, “write” operation will take place
  - Stack D latch structures to get n-bit register

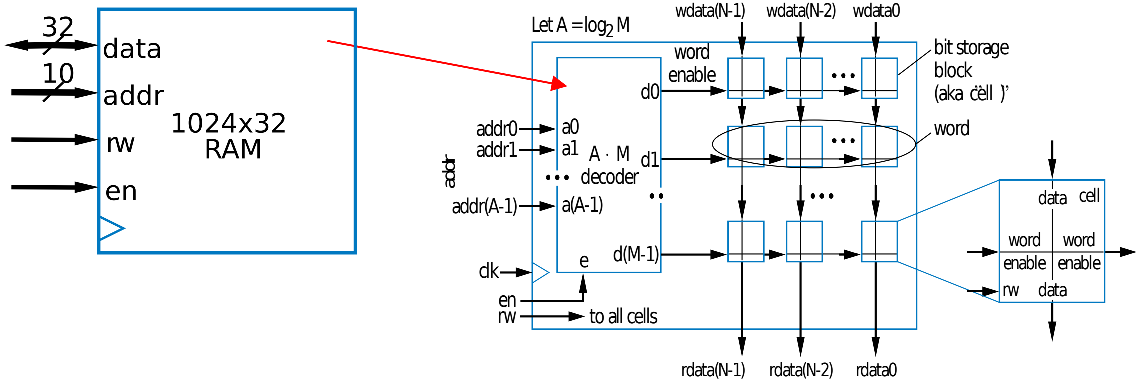


RAM Internal Structure

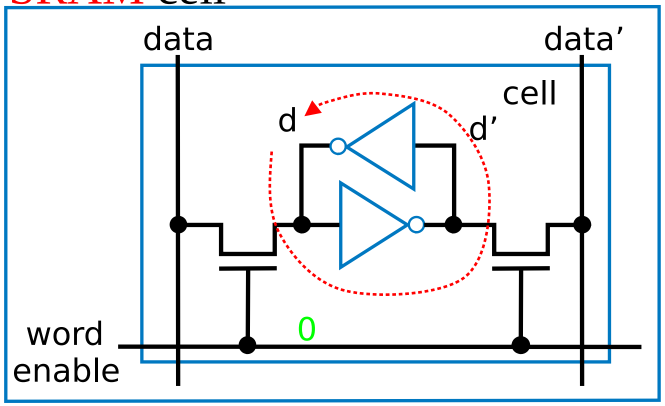


- Similar internal structure as register file
  - Decoder enables appropriate word based on address inputs
  - rw controls whether cell is written or read
  - rd and wr data lines typically combined
  - Let's see what's inside each RAM cell

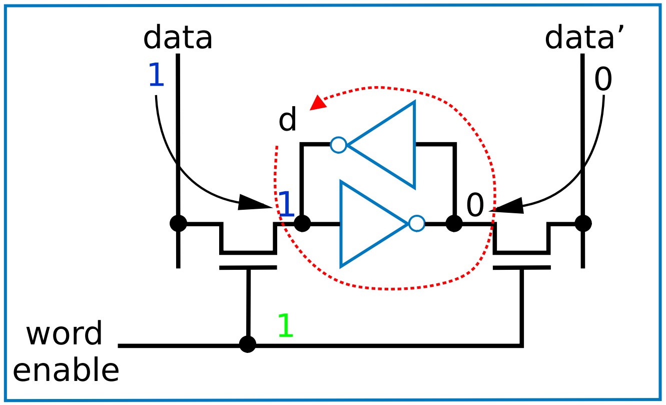
Static RAM (SRAM)

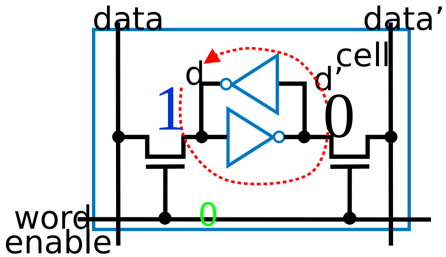


SRAM cell



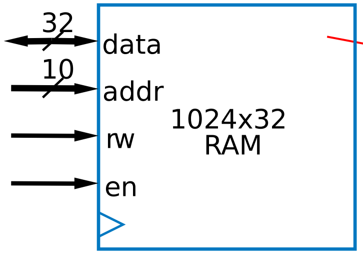
SRAM cell

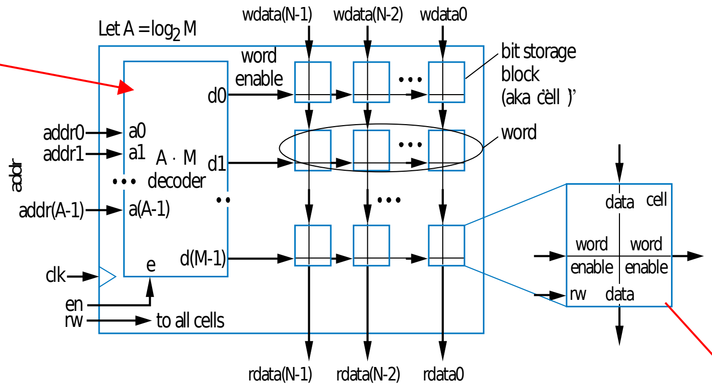




- "Static" RAM cell
  - 6 transistors (recall inverter is 2 transistors)
  - Writing this cell
    - *word enable* input comes from decoder
    - When 0, value *d* loops around inverters
      - That loop is where a bit stays stored
    - When 1, the *data* bit value enters the loop
      - *data* is the bit to be stored in this cell
      - *data'* enters on other side
      - Example shows a "1" being written into cell

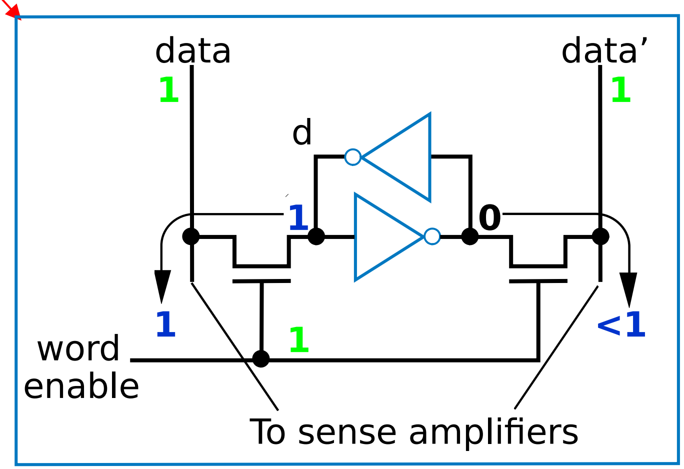
Static RAM (SRAM)





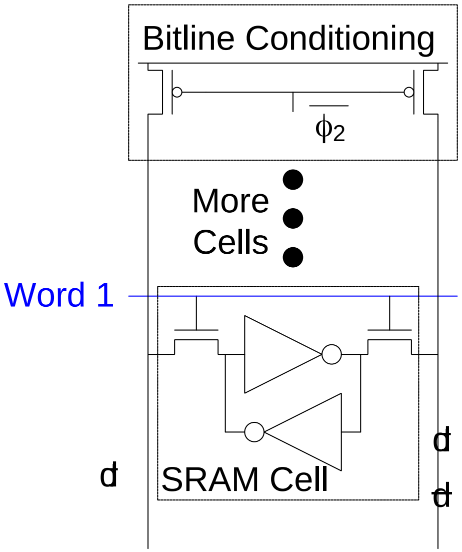
- "Static" RAM cell
  - Reading this cell
    - Somewhat trickier
    - When rw set to read, the RAM logic sets both *data* and *data'* to 1
    - The stored bit d will pull either the left line or the right bit down slightly below 1
    - "Sense amplifiers" detect which side is slightly pulled down
  - The electrical description of SRAM is really beyond our scope – just general idea here, mainly to contrast with *DRAM*...

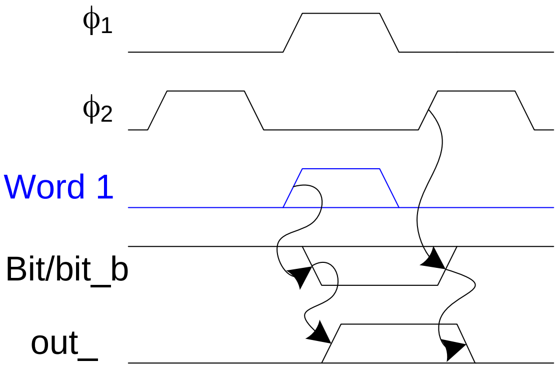
SRAM cell



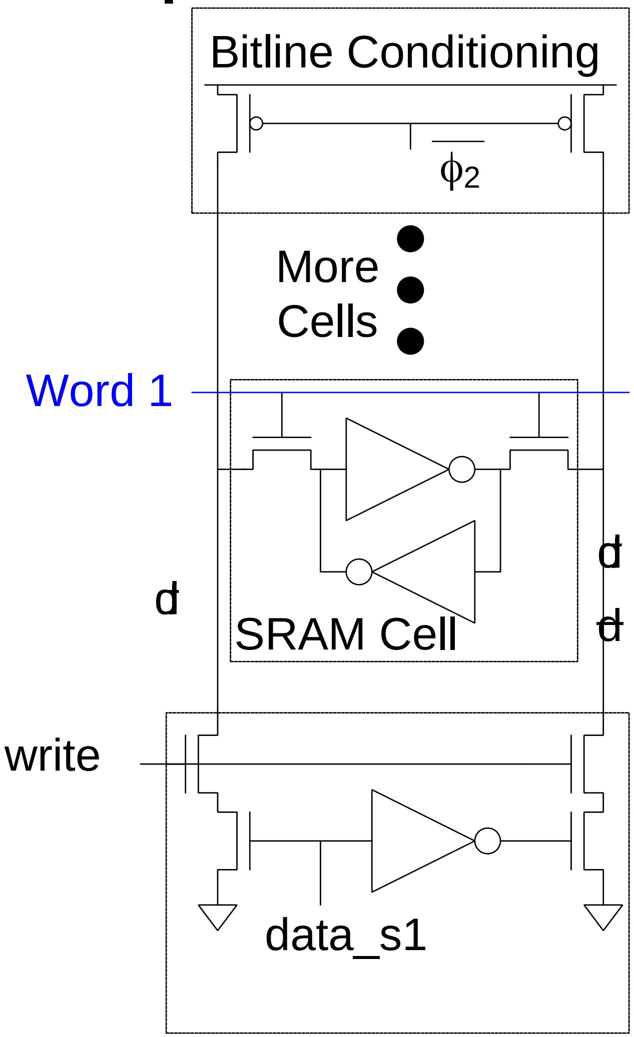
SRAM Column Example

Read

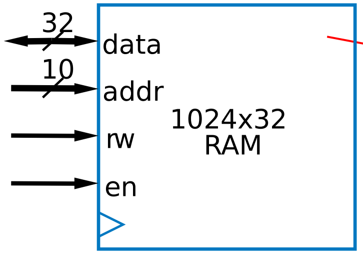


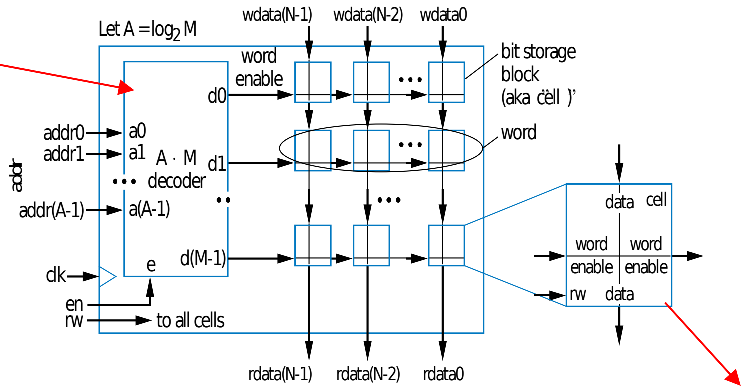


Write



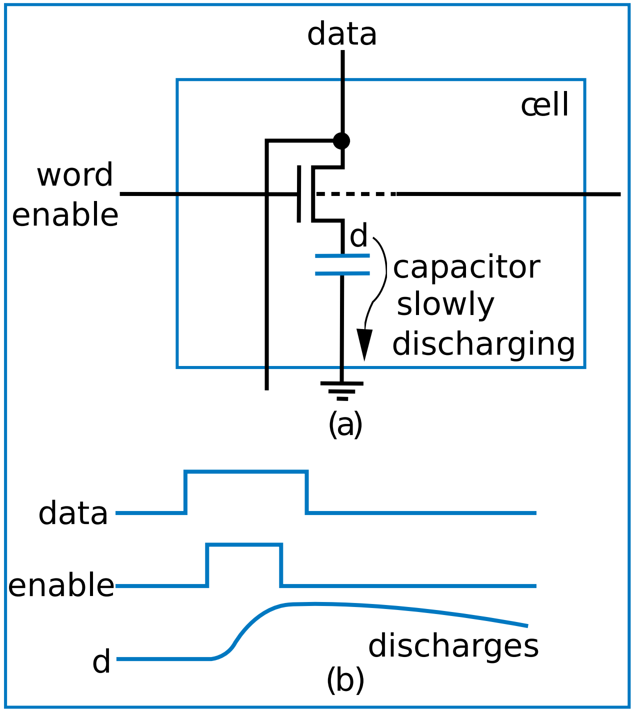
Dynamic RAM (DRAM)





- “Dynamic” RAM cell
  - 1 transistor (rather than 6)
  - Relies on *large* capacitor to store bit
    - Write: Transistor conducts, data voltage level gets stored on top plate of capacitor
    - Read: Just look at value of *d*
    - Problem: Capacitor discharges over time
      - Must “refresh” regularly, by reading *d* and then writing it right back

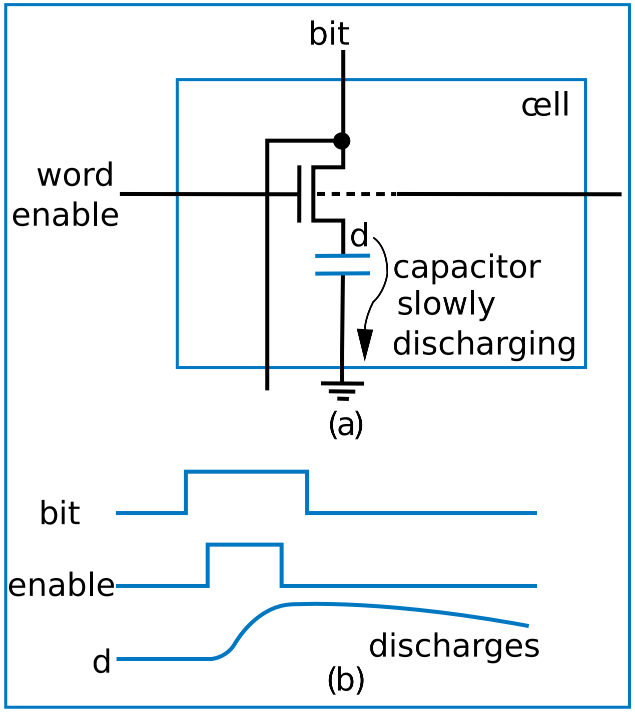
DRAM cell

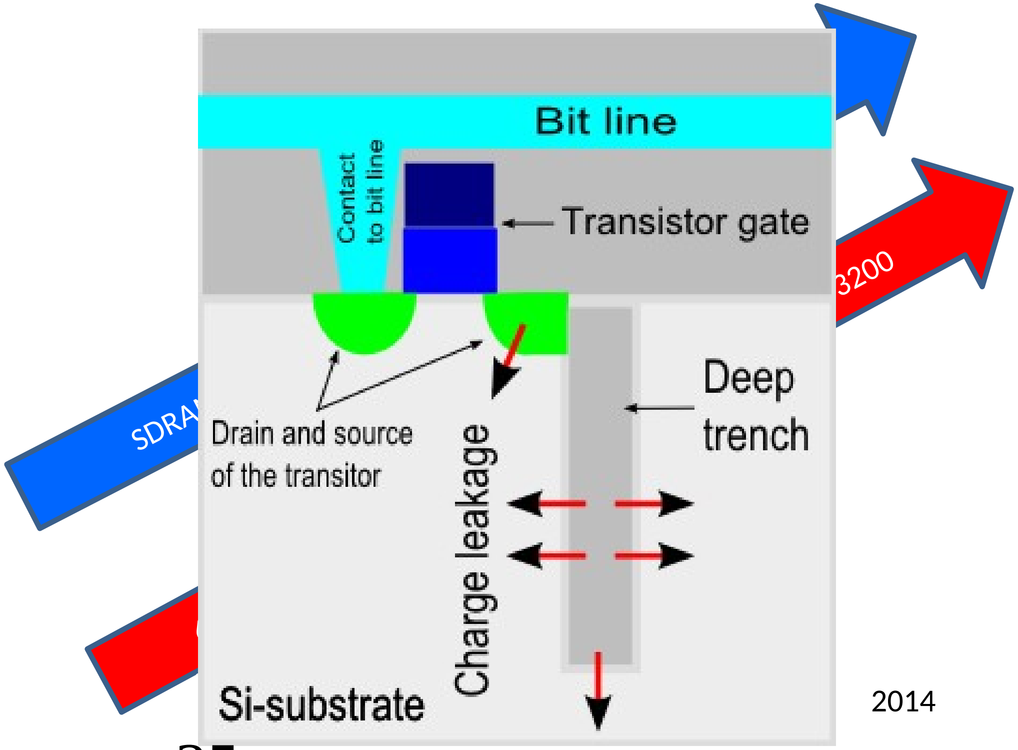


Dynamic RAM (DRAM)

- “Dynamic” RAM cell
  - 1 transistor (rather than 6)
  - Relies on capacitor to store bit

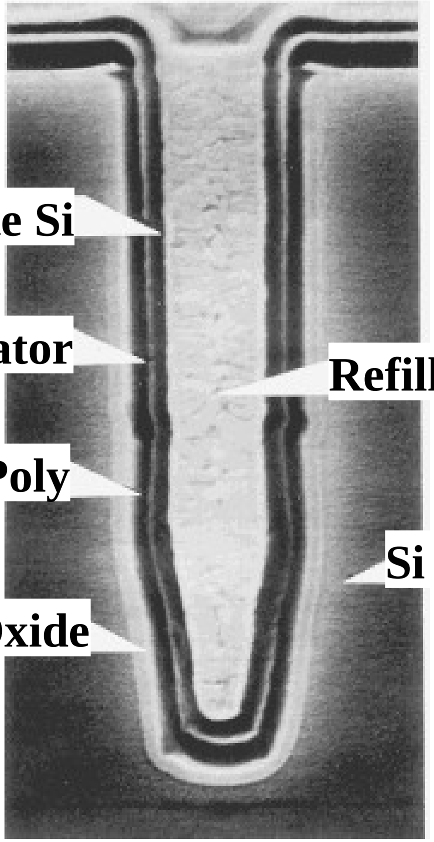
DRAM cell





2014

Dynamic RAM 1-Transistor Cell: Layout



Cell Plate Si

Capacitor Insulator

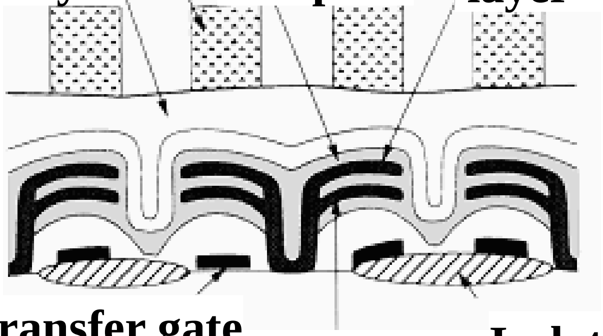
Storage Node Poly

2nd Field Oxide

Refilling Poly

Si Substrate

Trench Cell



Word line

Insulating Layer

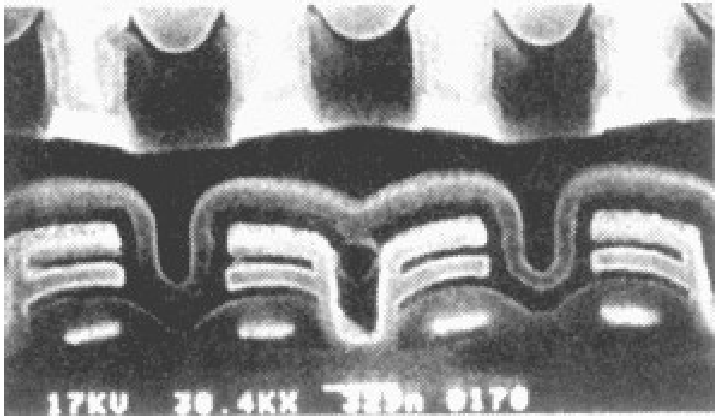
Cell plate

Capacitor Dielectric layer

Transfer gate

Isolation

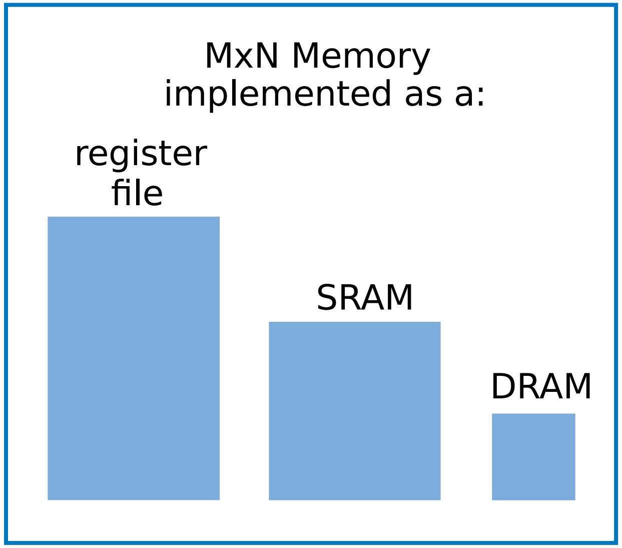
Storage electrode



Stacked-capacitor Cell

Comparing Memory

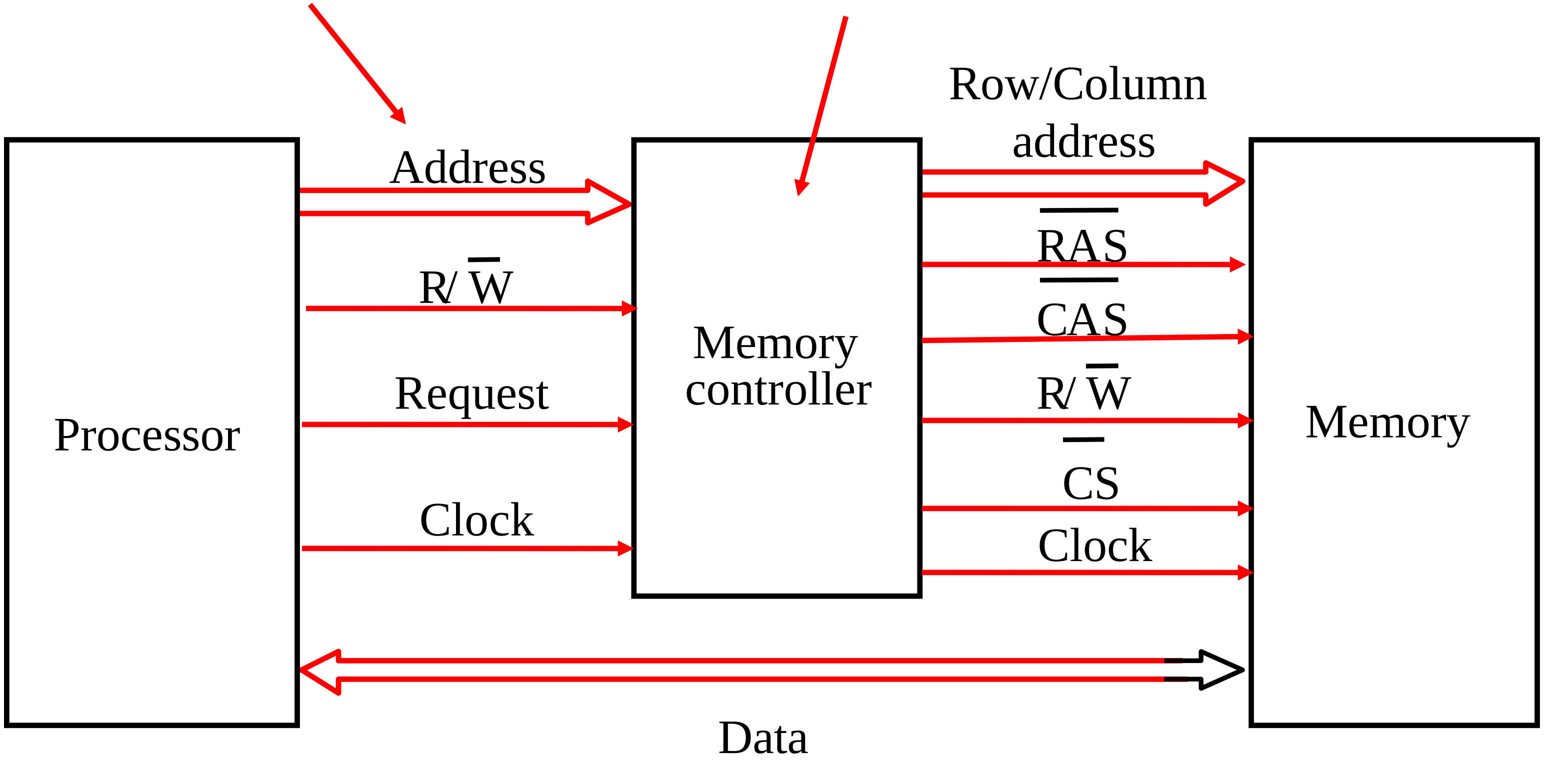
- Register file
  - Fastest
  - But biggest size
- SRAM
  - Fast
  - More compact than register file
- DRAM
  - Slowest (capacitor)
    - And refreshing takes time
  - But very compact (lower cost)
- Use register file for small items, SRAM for large items, and DRAM for huge items
  - Note: DRAM's big capacitor requires a special chip design process, so DRAM is often a separate chip



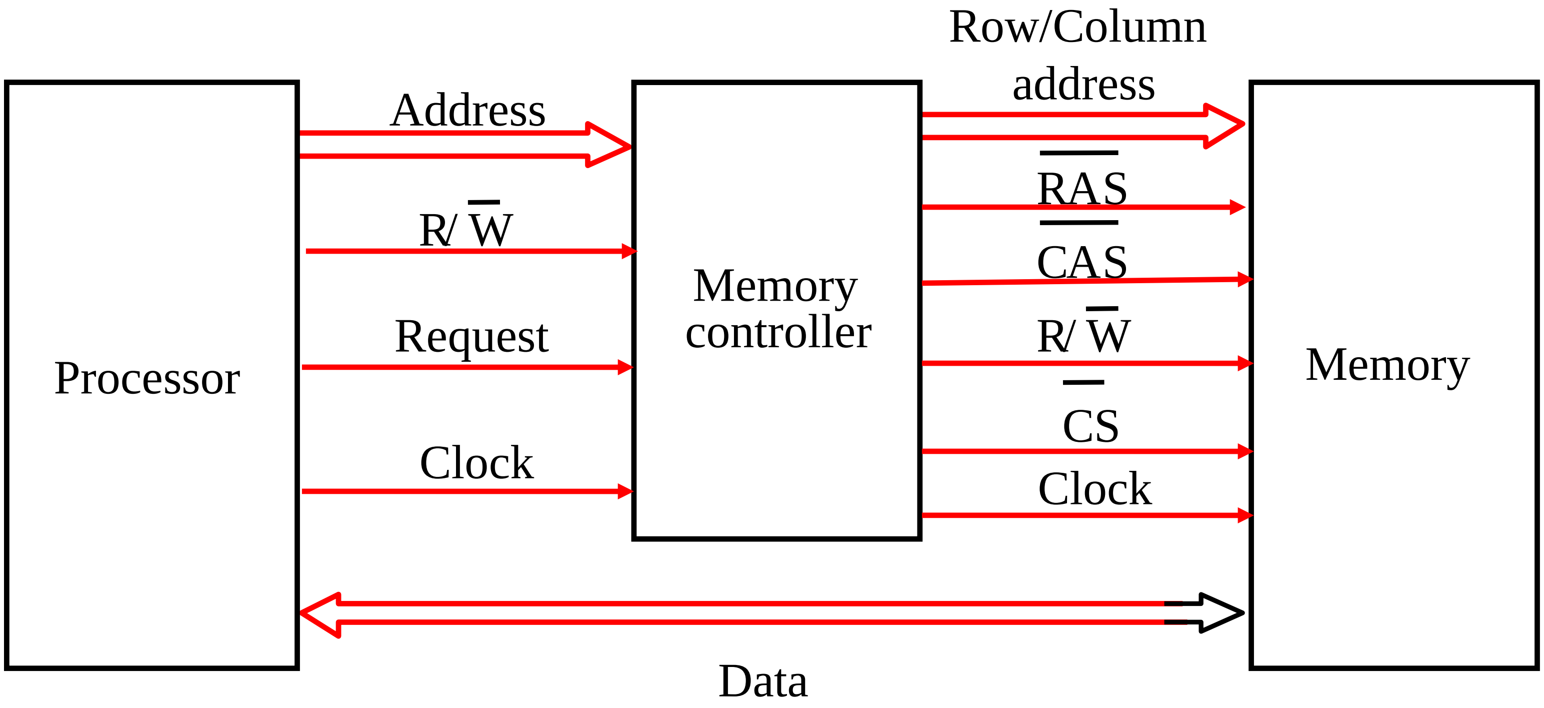
Size comparison for same number of bits (not to scale)

Processor sends
all bits of address

Memory controller does the
multiplexing of row and column
and issues strobe signals



Use of a memory controller.



Memory controller provides the refresh control if not done on the chip

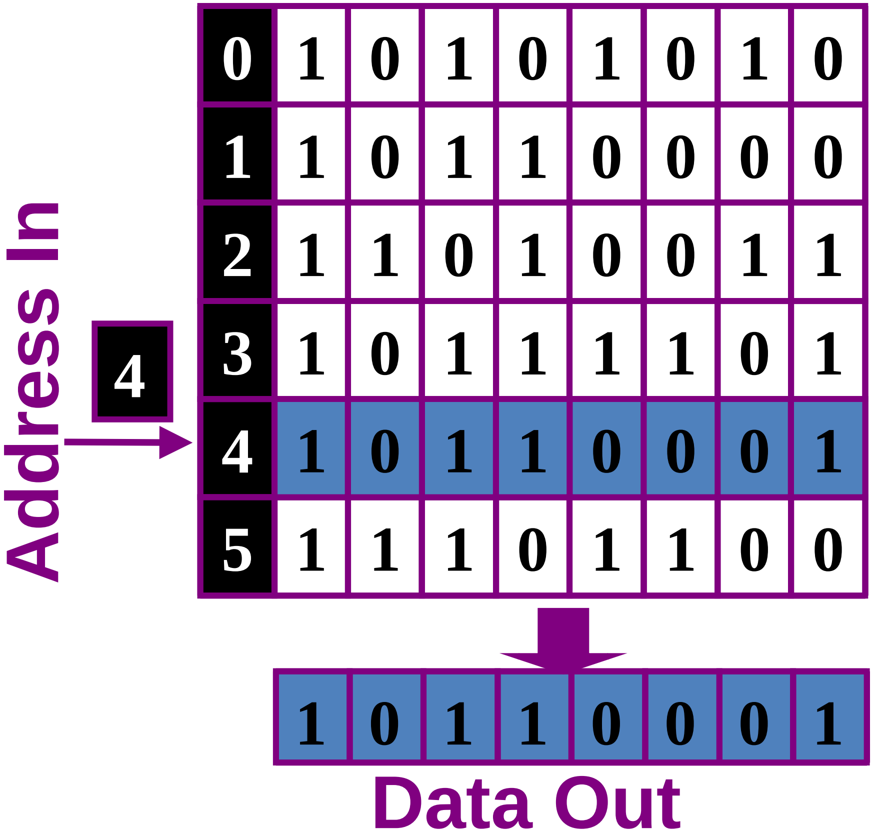
Refreshing typically once every 64 ms. At a cost of .2ms

Less than .4% overhead

Use of a memory controller.

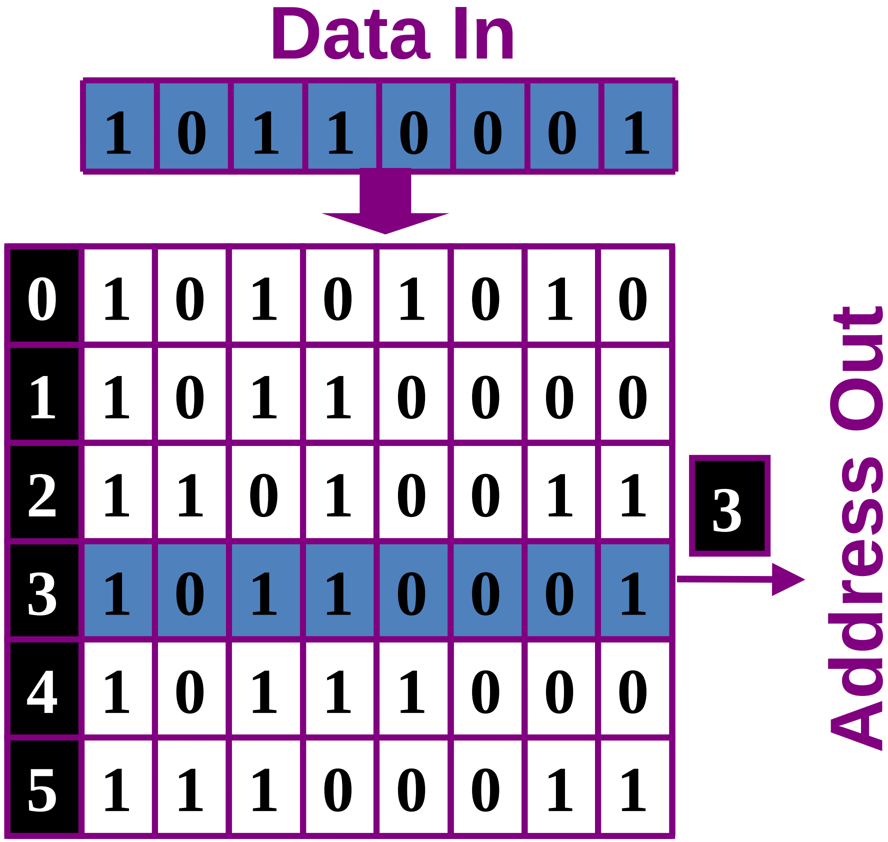
CAM

- CAM vs. RAM



0	1	0	1	0	1	0	1	0
1	1	0	1	1	0	0	0	0
2	1	1	0	1	0	0	1	1
3	1	0	1	1	1	1	0	1
4	1	0	1	1	0	0	0	1
5	1	1	1	0	1	1	0	0

1	0	1	1	0	0	0	1

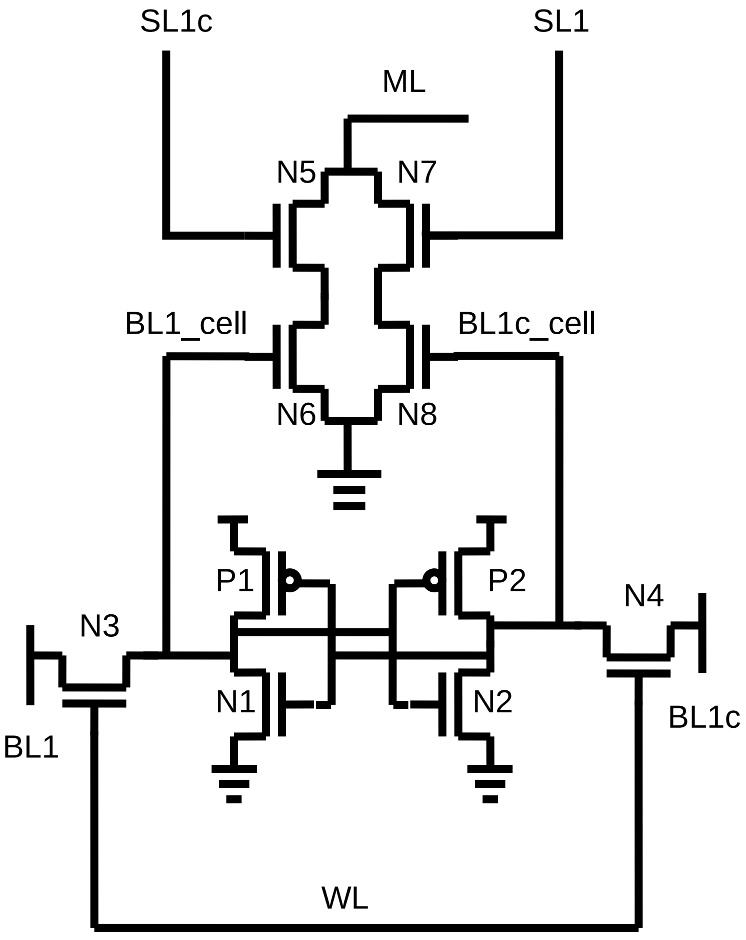


1	0	1	1	0	0	0	1

0	1	0	1	0	1	0	1	0
1	1	0	1	1	0	0	0	0
2	1	1	0	1	0	0	1	1
3	1	0	1	1	0	0	0	1
4	1	0	1	1	1	0	0	0
5	1	1	1	0	0	0	1	1

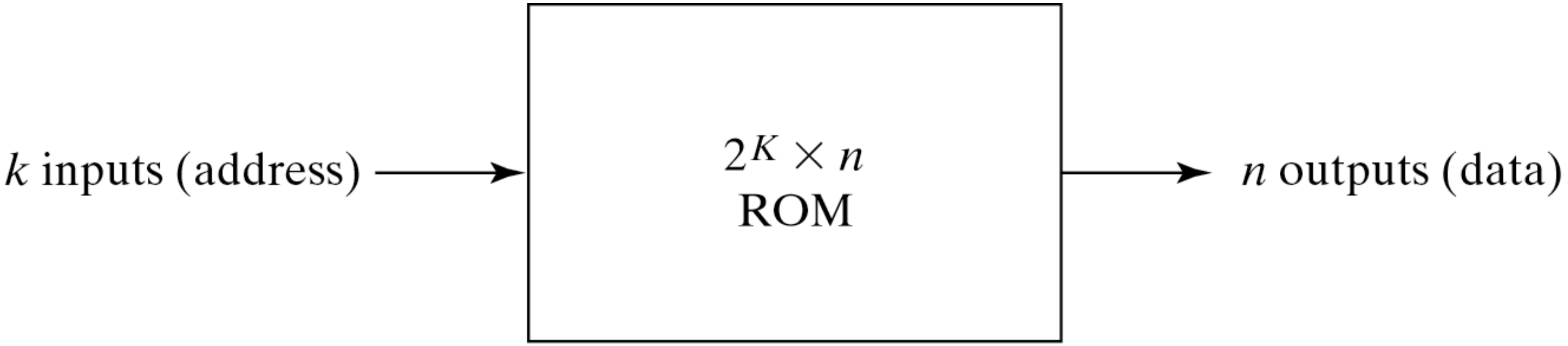
CAM

- Binary CAM Cell
  - ML pre-charged to V_{DD}
  - Match: ML remains at V_{DD}
  - Mismatch: ML discharges



Read-Only Memory

- A block diagram of a ROM is shown below. It consists of k address inputs and n data outputs.
- The number of words in a ROM is determined from the fact that k address input lines are needed to specify 2^k words.

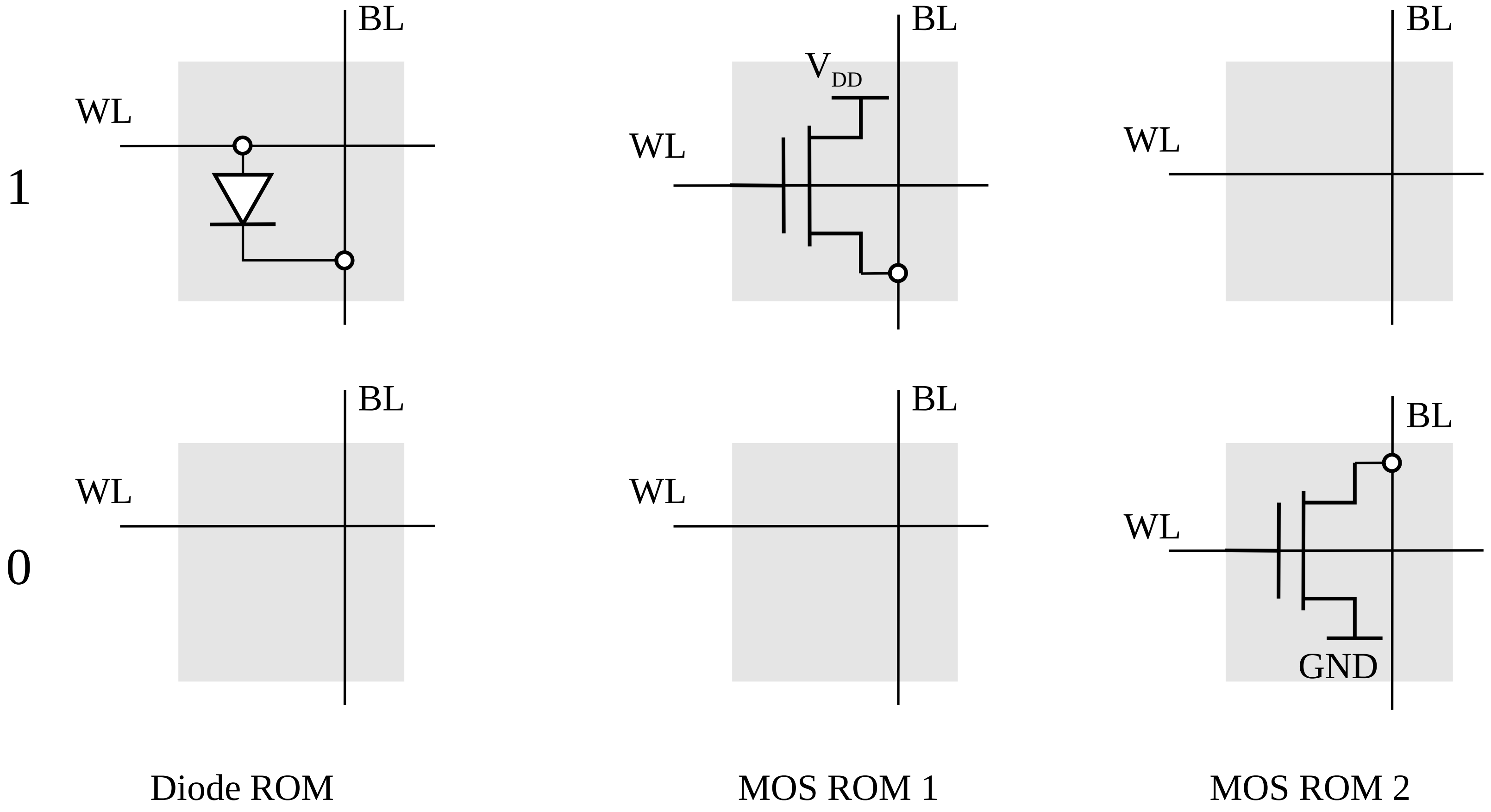


ROM Block diagram 2^k xn Module

Read-Only Memory Cells

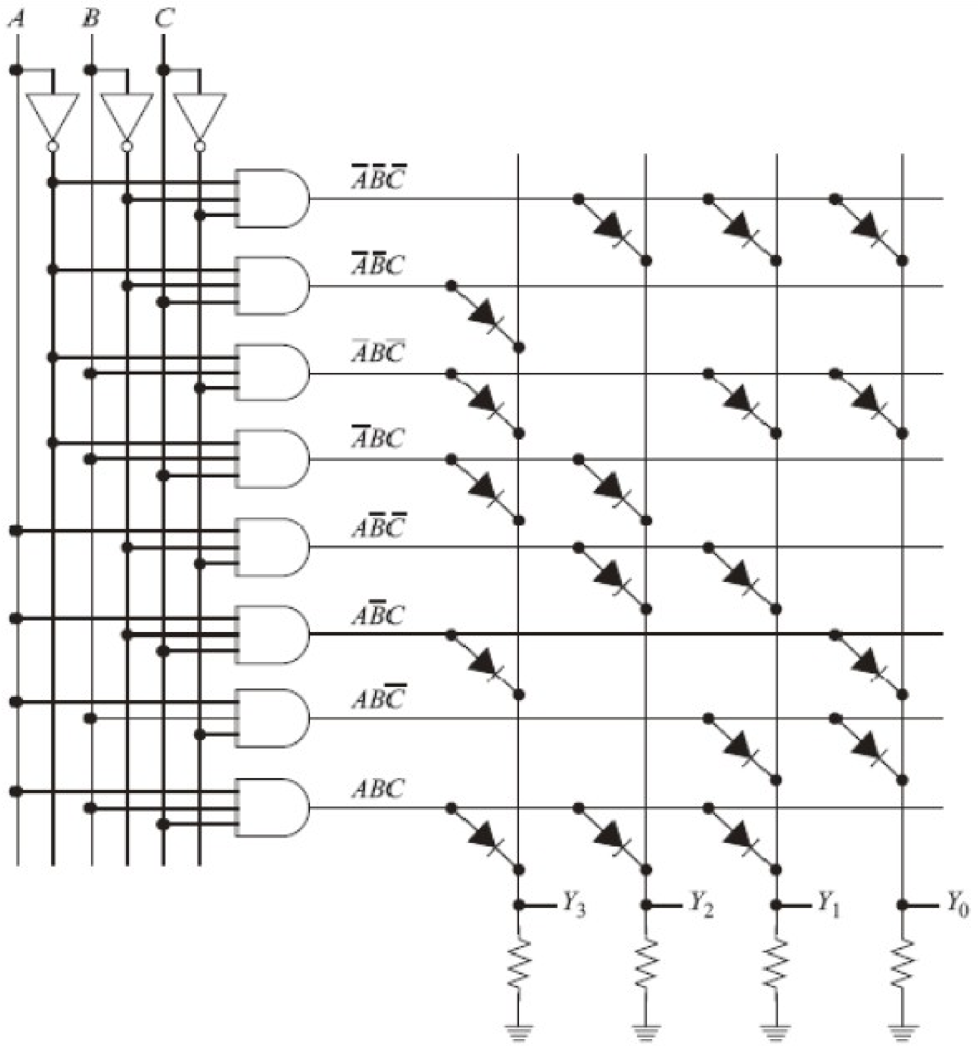
Bit line (BL) is resistively clamped to the ground, so its default value is 0
Diode disadvantage – no electrical isolation between bit and word lines

BL is resistively clamped to VDD, so its default value is 1



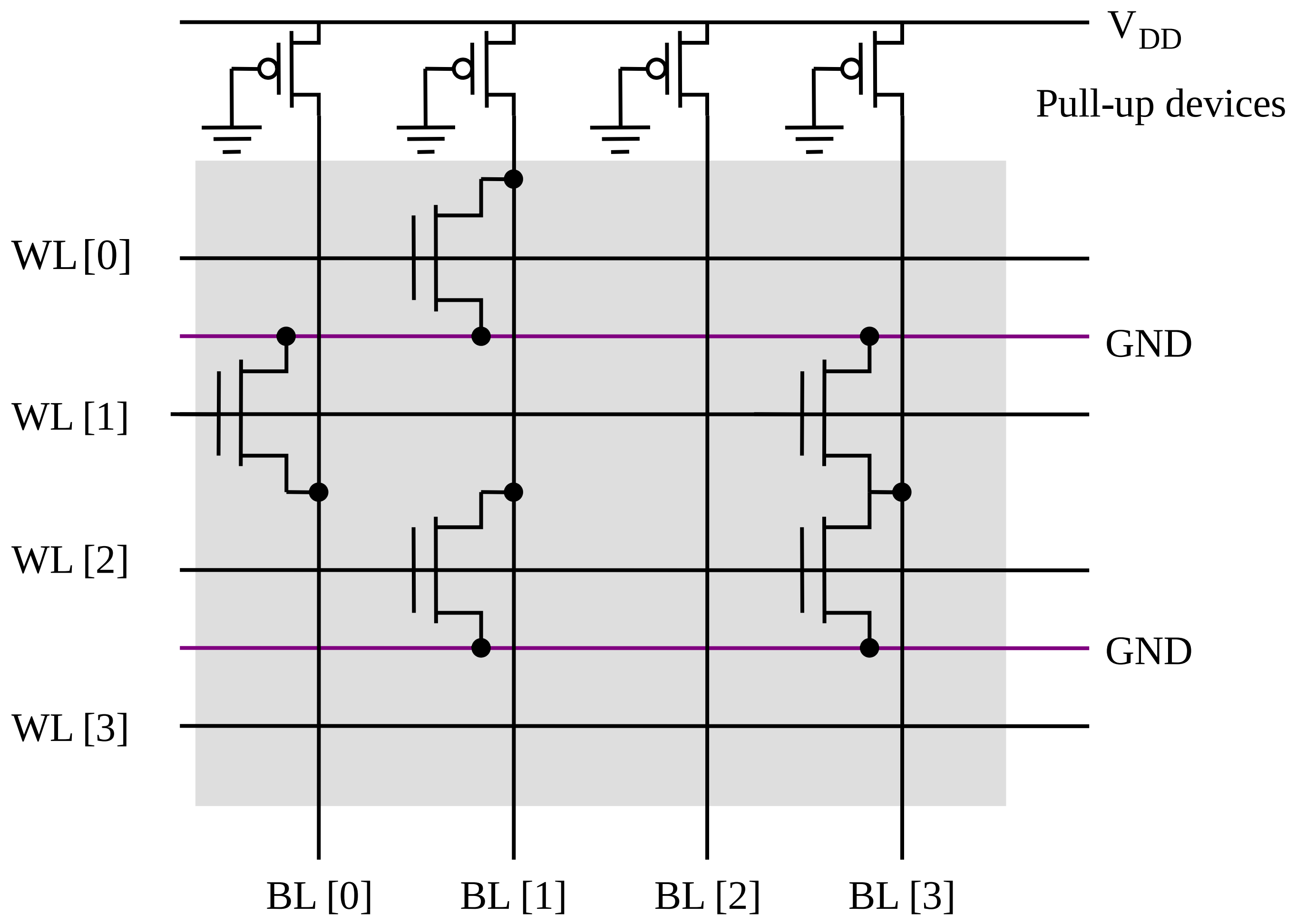
Nonvolatile Memory

ROM
PROM
EPROM

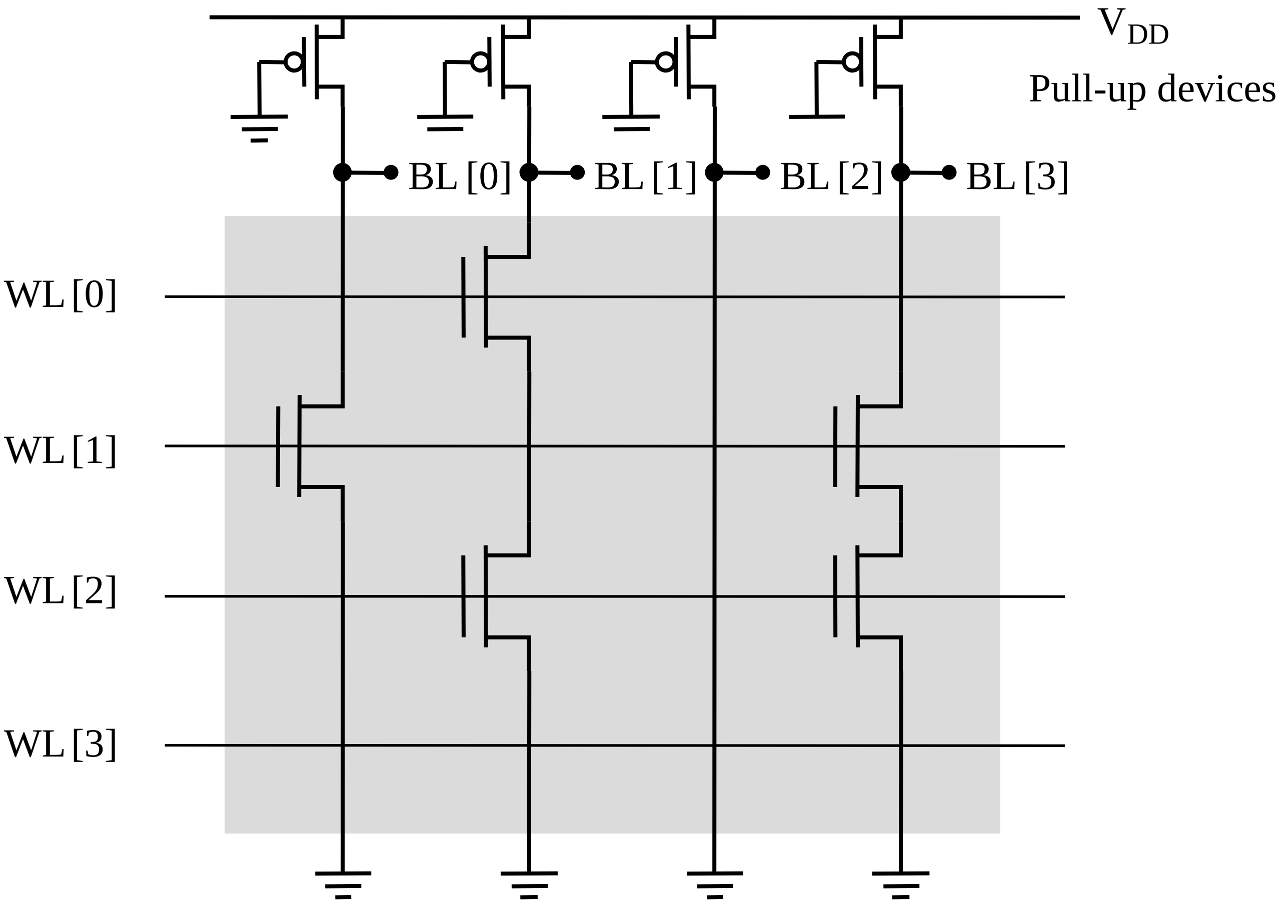

A
B
C
$\overline{A}\overline{B}\overline{C}$
$\overline{A}\overline{B}C$
$\overline{A}B\overline{C}$
$\overline{A}BC$
$A\overline{B}\overline{C}$
$A\overline{B}C$
$AB\overline{C}$
ABC
Y_3
Y_2
Y_1
Y_0

Read-only memory organization, with the fixed contents shown on the right.

MOS NOR ROM

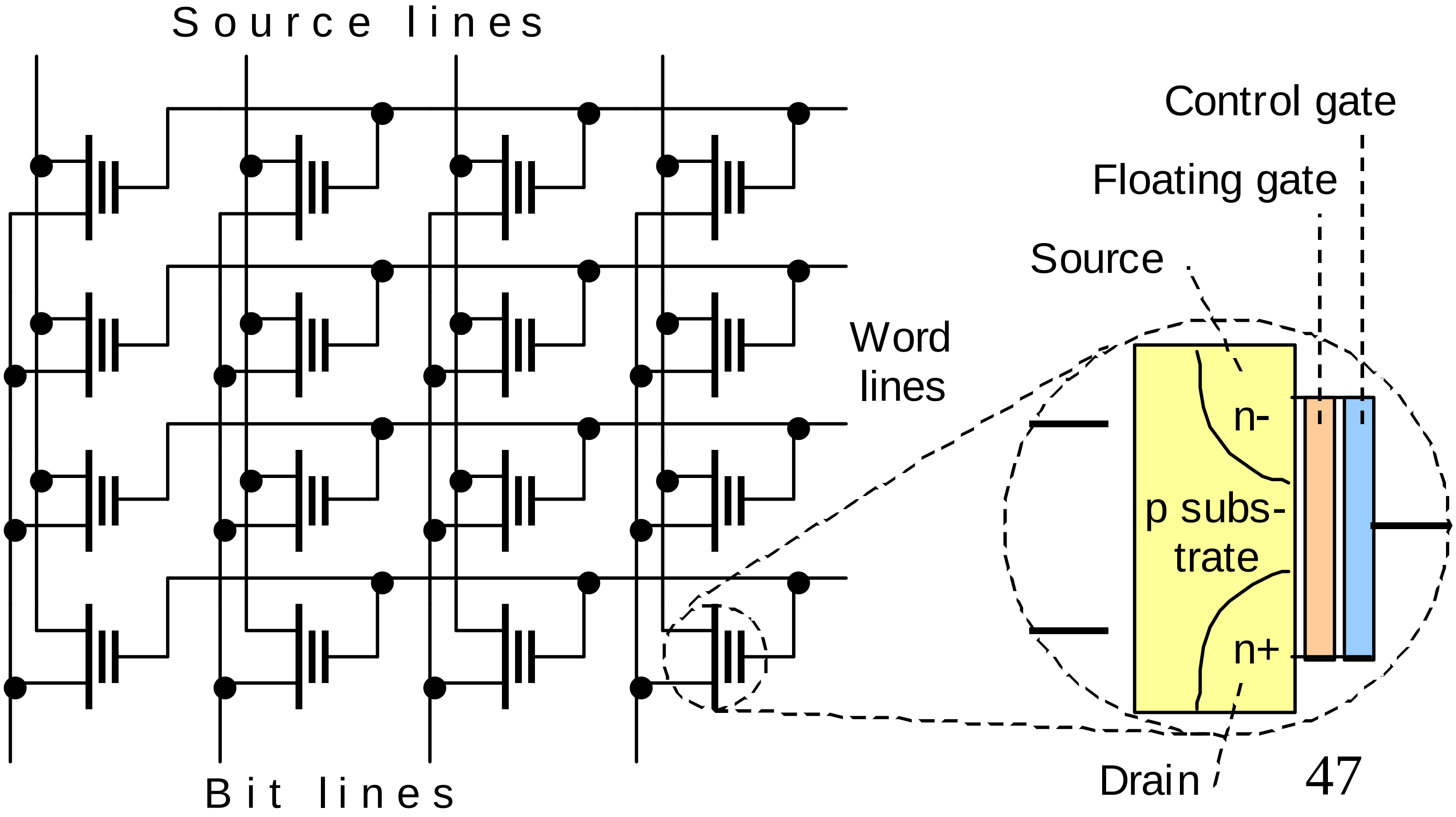


MOS NAND ROM



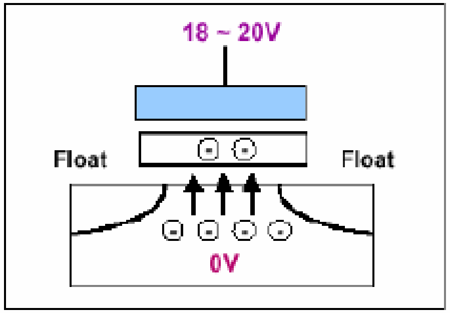
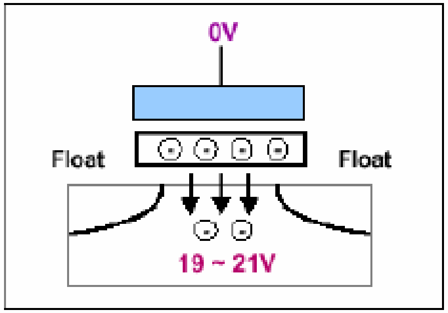
All word lines high by default with exception of selected row

Flash Memory

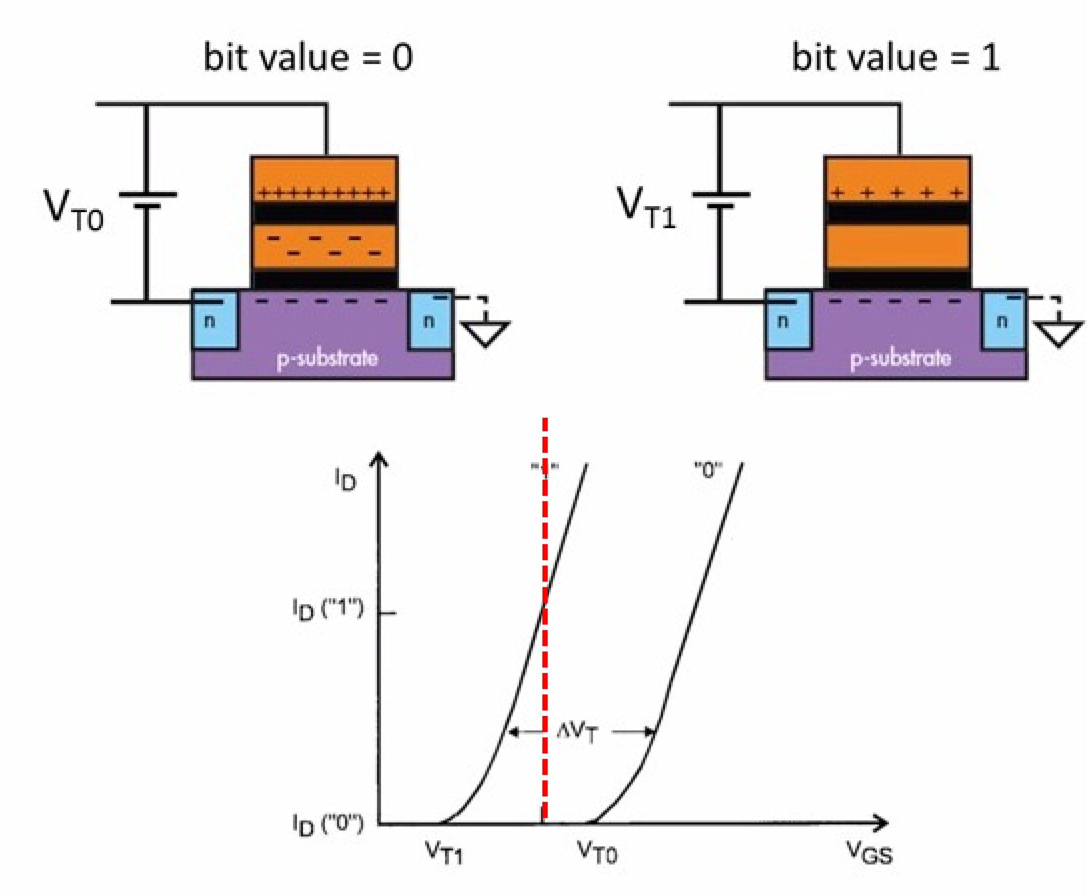


Flash memory

- Flash memory
  - Non volatile read only memory (ROM)
  - Erase Electrically or UV (EPROM)
  - Uses F-N tunneling for program & erase
  - Reads like DRAM (~ns)
  - Writes like DISK (~ms).

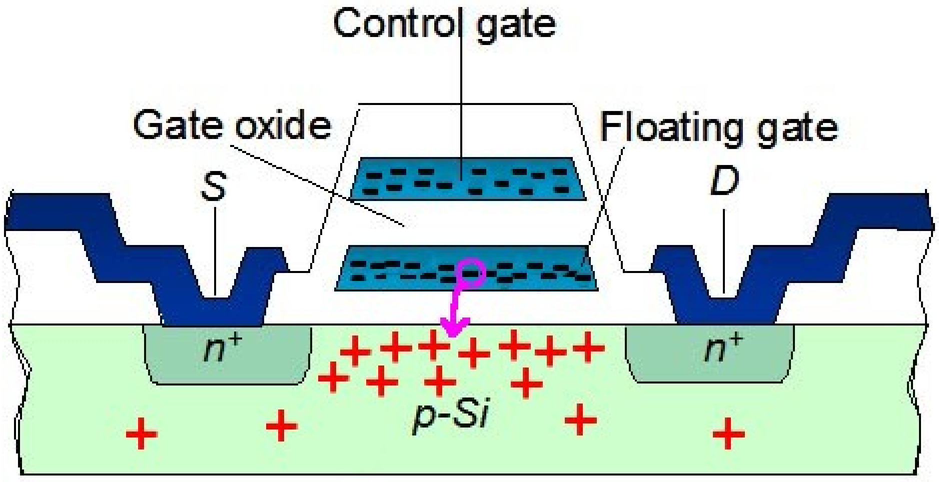
Program	Erase
	
● Use F-N Tunneling ● Channel Inversion	● Use F-N Tunneling ● Channel Accumulation

Reading Memory State



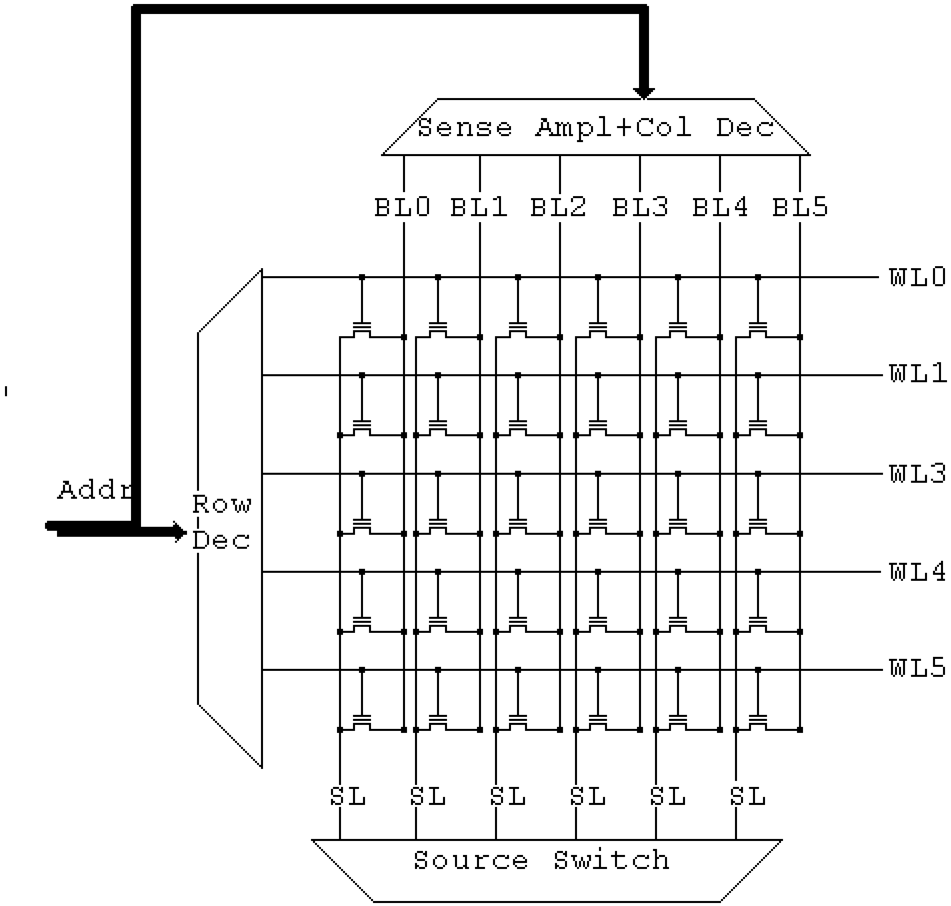
Change in Threshold Voltage due to **Screening Effect** of Floating Gate
Read mode: Apply intermediate voltage, check whether current is flowing or not

Writing Memory State



Control gate voltage determines whether electrons are injected to, or push/pulled out of floating gate.

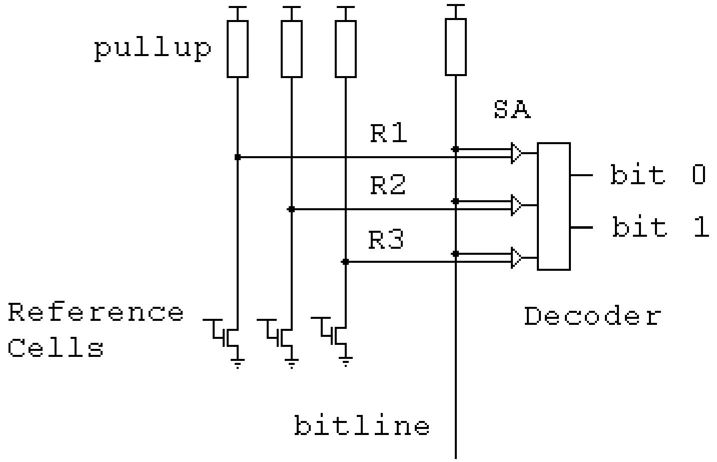
NOR Array



Reading:

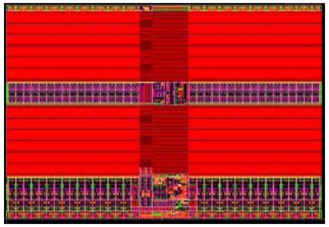
Assert a single word line. The source lines are asserted and the read of the bitline gives the contents of the cell.

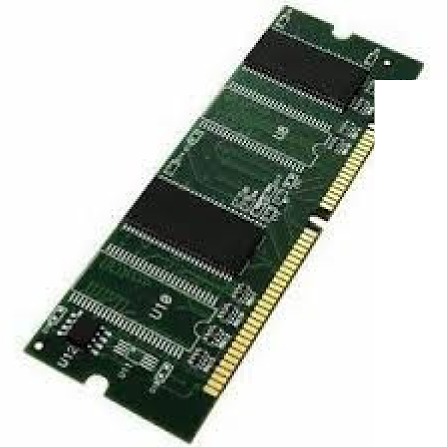
Multi-Levels



- By using reference cells set at given levels and comparing them to the value from the bitline, we can determine the value stored.

Review of memory technologies

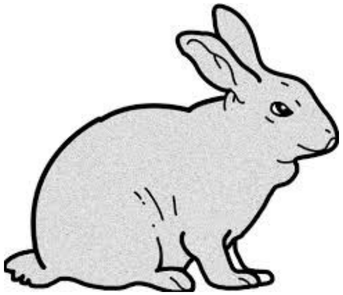




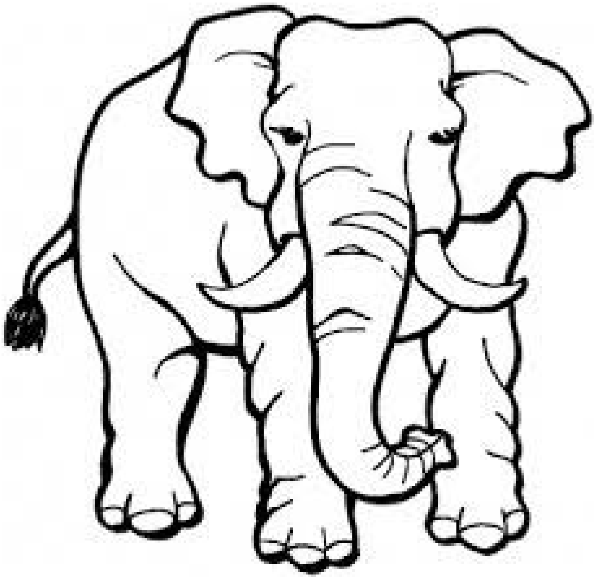
Memory type	SRAM	DRAM	Flash
Speed	Very fast	Slow	Slow
Density	Low	High	Very high
Power	High	Low	Very low
Refresh	No	Yes	No
Mechanism	Bi-stable latch	Capacitor	FN tunneling

Memory Hierarchy

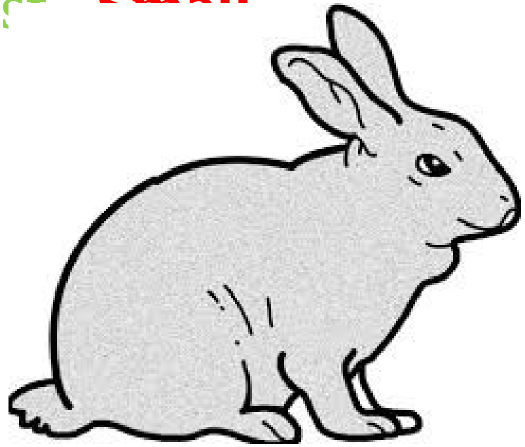
Actual Memory Systems



+



Type1: Fast, Small



Can we achieve?: Fast, Large

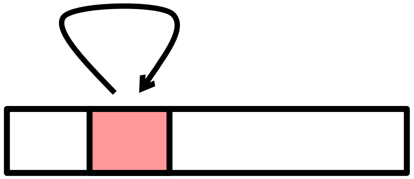
Type2:Slow, Large

(Cache Principle)

Locality

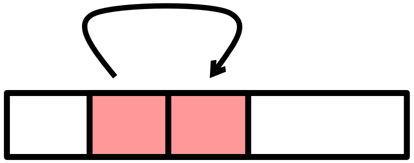
- Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

- Temporal locality:



  - Recently referenced items are likely to be referenced again in the near future

- Spatial locality:



  - Items with nearby addresses tend to be referenced close together in time

Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- Data references
  - Reference array elements in succession

    Spatial locality
  - Reference variable `sum` each iteration.

    Temporal locality
- Instruction references
  - Reference instructions in sequence.

    Spatial locality
  - Cycle through loop repeatedly.

    Temporal locality

Qualitative Estimates of Locality

- Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.
- Question: Does this function have good locality with respect to array a?

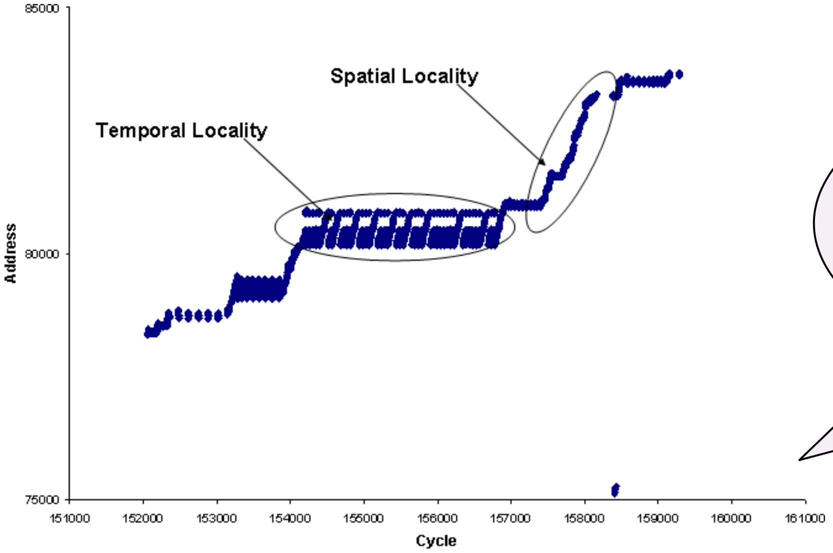
```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Locality Example

- Question: Does this function have good locality with respect to array a?

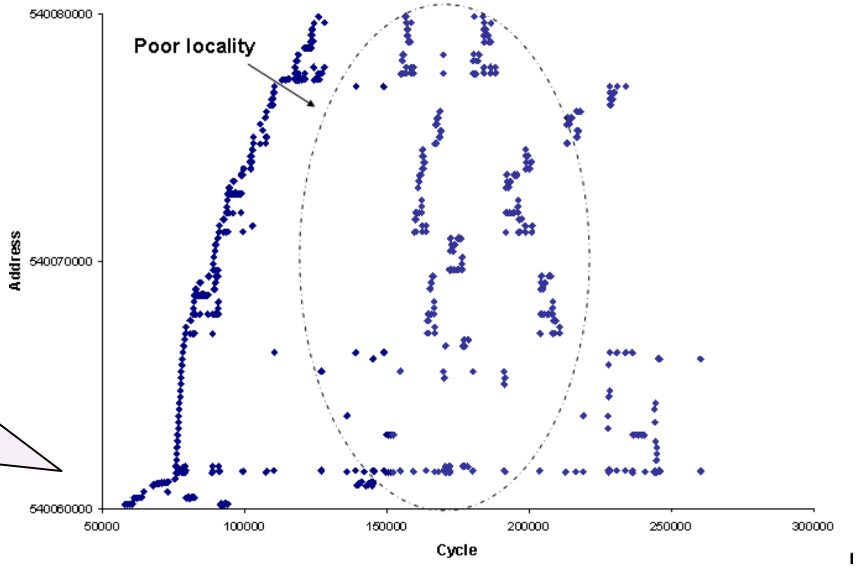
```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Locality reference example



Good locality example:
The X axis corresponds to time (in cycles)
and the Y axis corresponds to memory locations.
The highlighted sections show examples
of temporal and spatial locality.

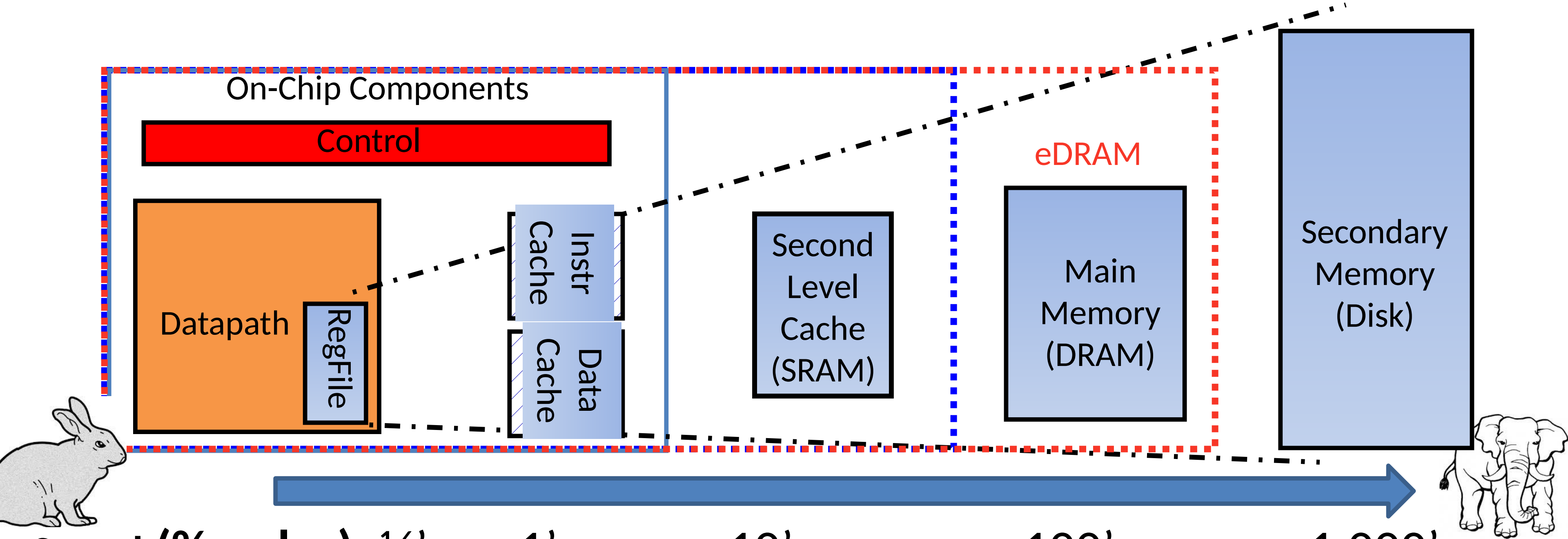
Poor locality example:
We can see that the program accesses a
large memory footprint without a specific order,
giving the pattern poor spatial locality.
We also see little temporal re-use,
which also makes this code cache unfriendly.



Source: http://www.eetimes.com/document.asp?doc_id=1275470

The Memory Hierarchy

- By taking advantage of the principle of locality
  - Can present the user with as much memory as is available in the cheapest technology
  - at the speed offered by the fastest technology



Speed (%cycles): ½'s 1's 10's 100's 1,000's

Size (bytes): 100's K's 1M's 10G's 10 G's to T's

Speed(ns): 0.5ns 2ns 6ns 100ns 10ms

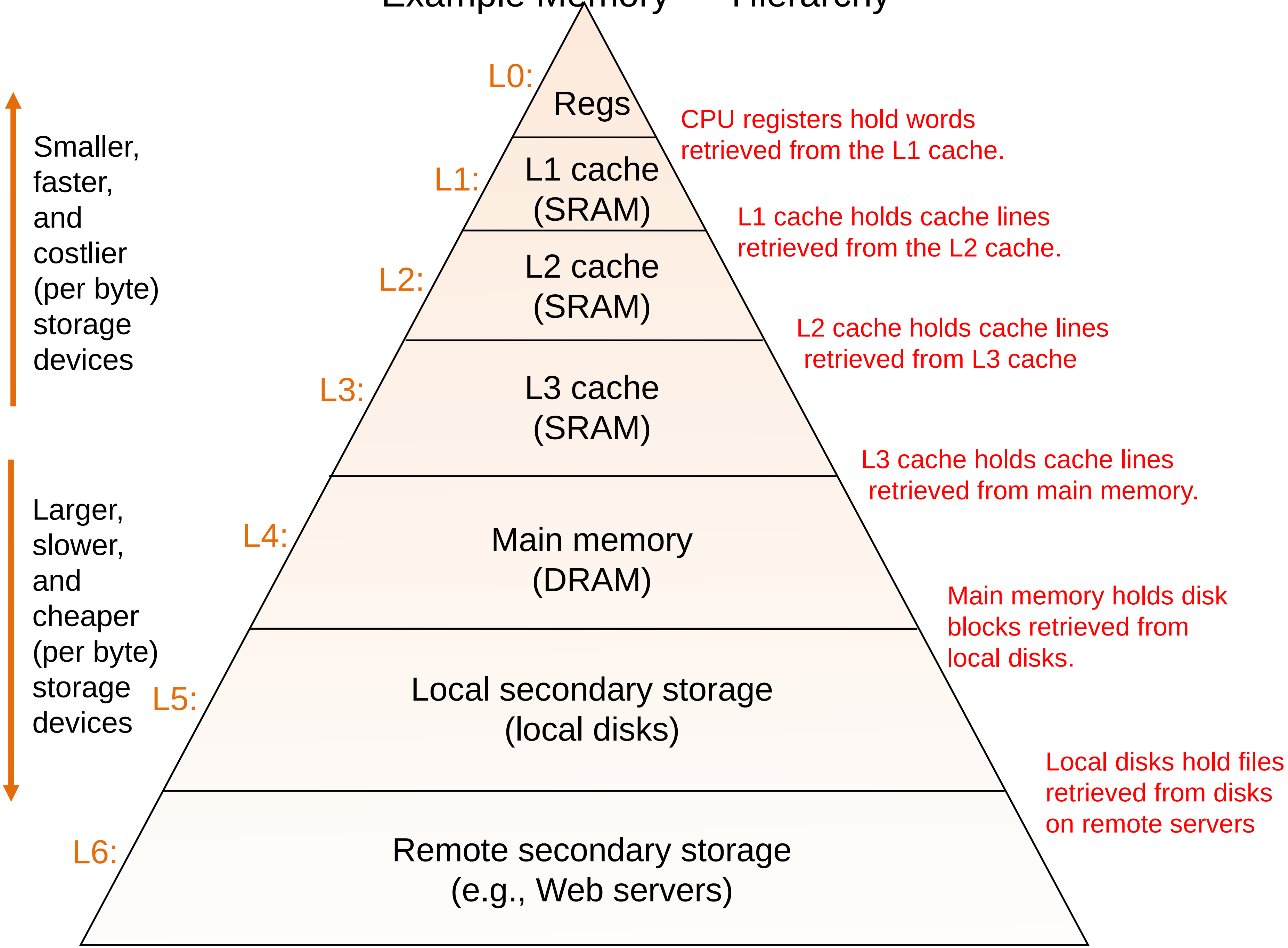
$$$$ $

Cache Design

Maurice Wilkes, “Slave Memories and Dynamic Storage Allocation,” IEEE Trans. On Electronic Computers, 1965

Awards: Turing Award,
IEEE John von Neumann Medal, Faraday Medal,
Eckert–Mauchly Award, Mountbatten Medal

Example Memory Hierarchy



Smaller, faster, and costlier (per byte) storage devices

Larger, slower, and cheaper (per byte) storage devices

L0: Regs

L1: L1 cache (SRAM)

L2: L2 cache (SRAM)

L3: L3 cache (SRAM)

L4: Main memory (DRAM)

L5: Local secondary storage (local disks)

L6: Remote secondary storage (e.g., Web servers)

CPU registers hold words retrieved from the L1 cache.

L1 cache holds cache lines retrieved from the L2 cache.

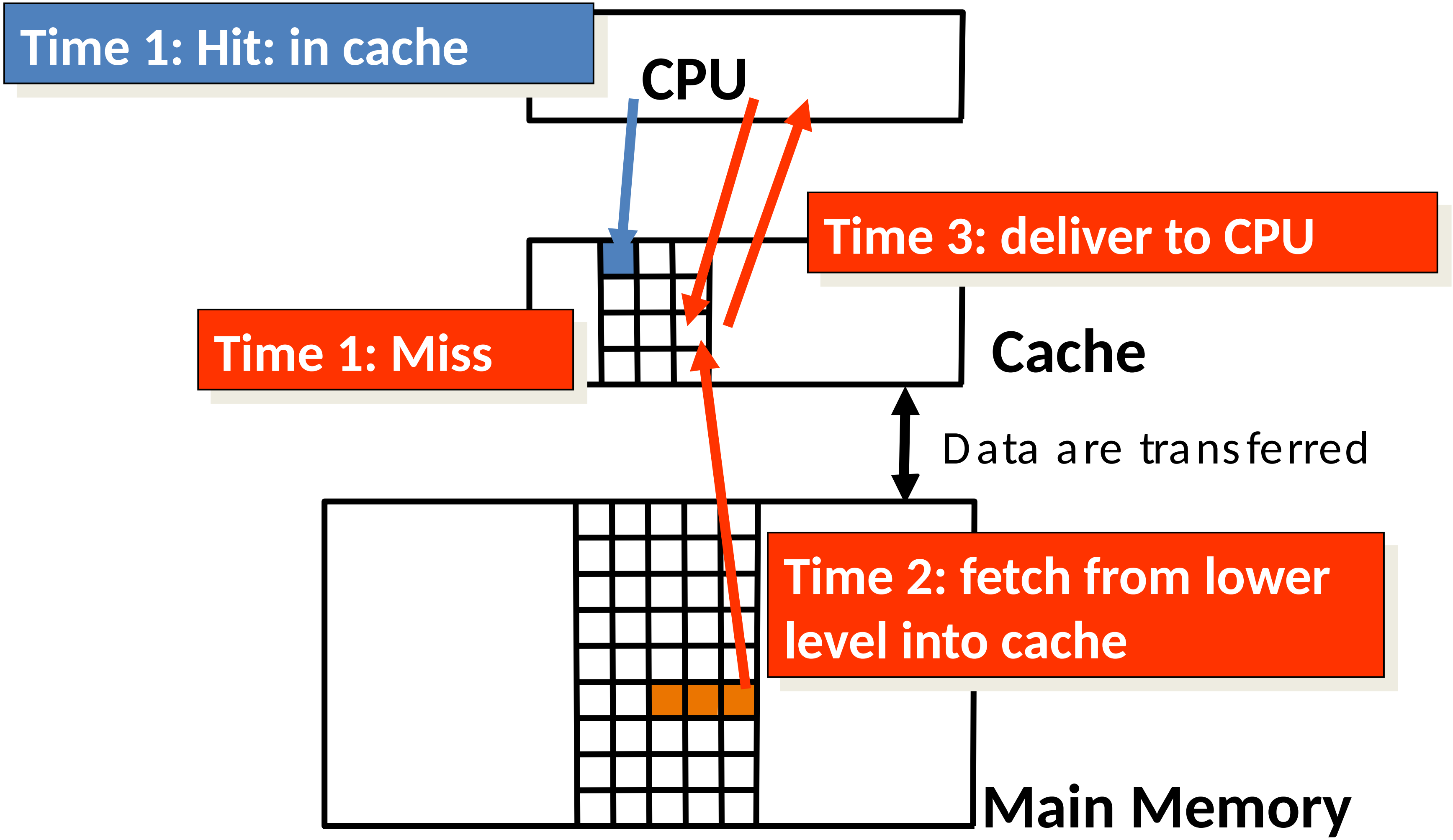
L2 cache holds cache lines retrieved from L3 cache

L3 cache holds cache lines retrieved from main memory.

Main memory holds disk blocks retrieved from local disks.

Local disks hold files retrieved from disks on remote servers

Cache Example



Time 1: Hit: in cache

CPU

Time 3: deliver to CPU

Cache

Time 1: Miss

Data are transferred

Time 2: fetch from lower level into cache

Main Memory

Average Memory Access Time = hit time (t_c) + miss rate (m) * miss penalty(t_p)

$$= t_c + m * t_p$$

Hit time = Time 1 (tc)

Miss penalty (tp)= Time 2 + Time 3

Example: Impact on Performance

- **Suppose a processor executes at**
  - Clock Rate = 1000 MHz (1 ns per cycle)
  - CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control
- Suppose that 10% of memory operations get 50 cycle miss penalty
- CPI = ideal CPI + average stalls per instruction= 1.1(cyc) +(0.30 (datamops/ins)

  x 0.10 (miss/datamop) x 50 (cycle/miss))

  = 1.1 cycle + **0.3x0.1x50** cycle =1.1cycle + 1.5 cycle = 2. 6
- 58 % of the time the processor is stalled waiting for memory!
- a 1% instruction miss rate would add an additional 0.5 cycles to the CPI!

Source: David Patterson

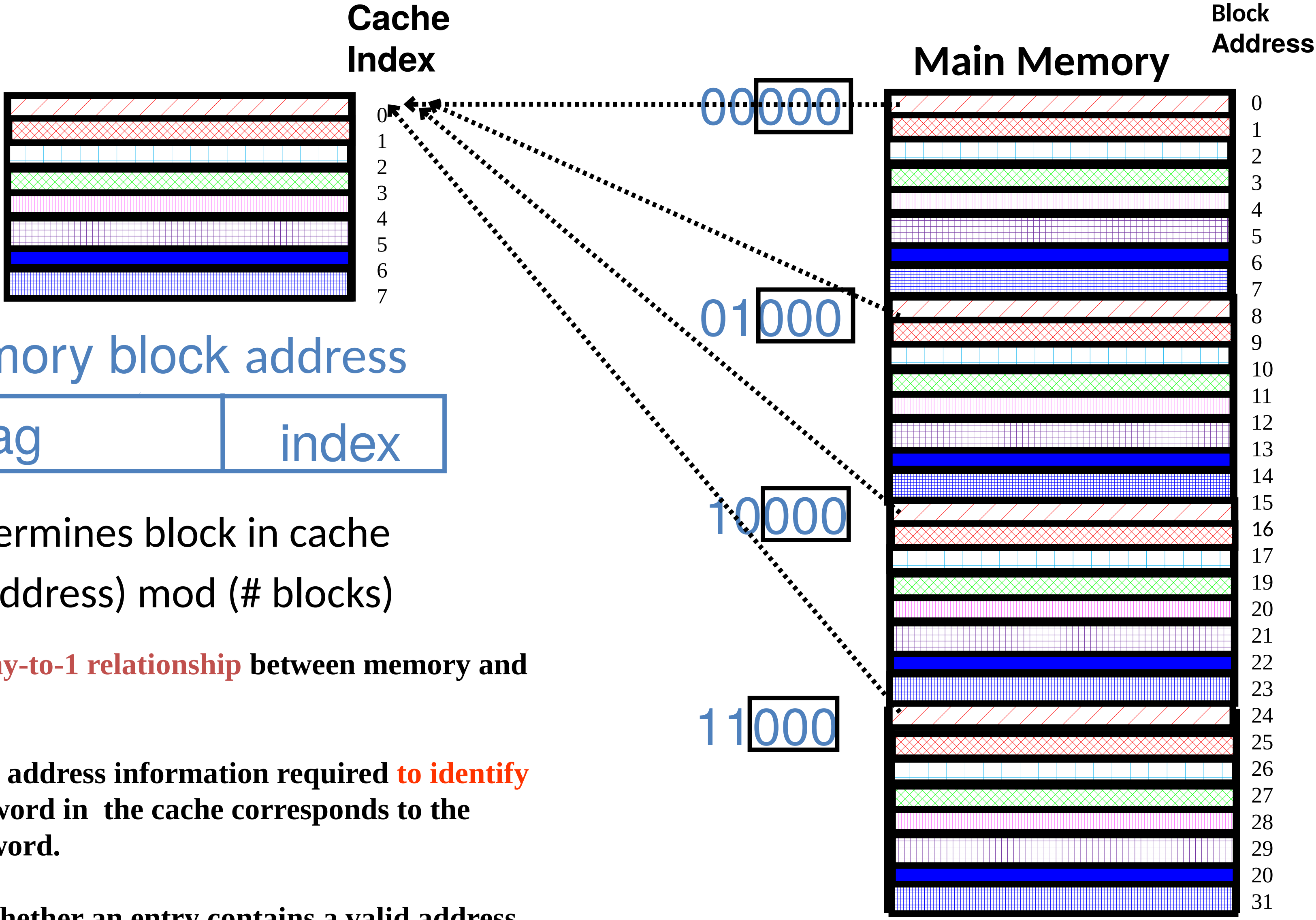
Let’s think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory

- Would you believe 99% hits is twice as good as 97%?
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - Average access time:
    97% hits: 1 cycle + 0.03 * 100 cycles = **4 cycles**
    99% hits: 1 cycle + 0.01 * 100 cycles = **2 cycles**

- This is why “miss rate” is used instead of “hit rate”

Simplest Cache: Direct Mapped



Memory block address

tag	index

- index determines block in cache
- index = (address) mod (# blocks)

There is a Many-to-1 relationship between memory and cache

tags

- **contain the address information required to identify whether a word in the cache corresponds to the requested word.**

valid bit

- **indicates whether an entry contains a valid address**

Cache Table Entry

V	tag	Data

Direct Mapped Cache: Temporal Example

lw $1,10 110 ($0)

lw $2,11 010 ($0)

lw $3,10 110 ($0)

Miss: valid

Miss: valid

Hit!

lw $1,22($0)

lw $2,26($0)

lw $3,22($0)

Index	Valid	Tag	Data
000	N		
001	N		
010	Y	11	Memory[11010]
011	N		
100	N		
101	N		
110	Y	10	Memory[10110]
111	N		

Direct Mapped Cache: Worst case, always miss!

lw $1,10 110 ($0)

lw $2,11 110 ($0)

lw $3,00 110 ($0)

Miss: valid

Miss: tag

Miss: tag

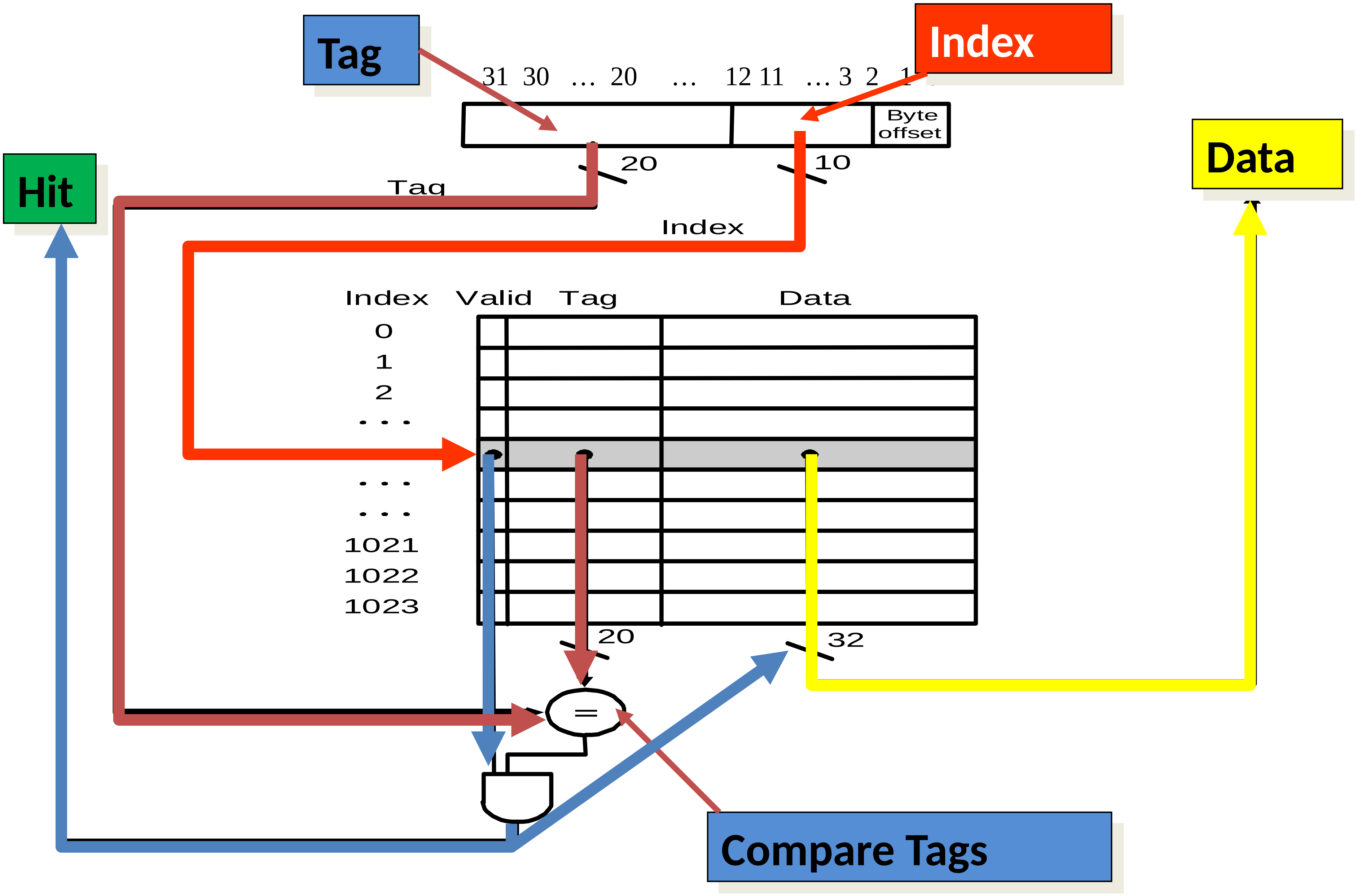
lw $1,22($0)

lw $2,30($0)

lw $3,6($0)

Index	Valid	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	00	Memory[00110]
111	N		

Direct Mapped Cache: MIPS Example



Tag

Index

Hit

Data

Compare Tags

4KB Direct Mapped Cache Example

4 Kbytes = Nominal Cache Capacity

1K = 2^{10} = 1024 Blocks

Each block = one word

(4 bytes)

Can cache up to 2^{32} bytes = 4 GB of memory

Mapping function:

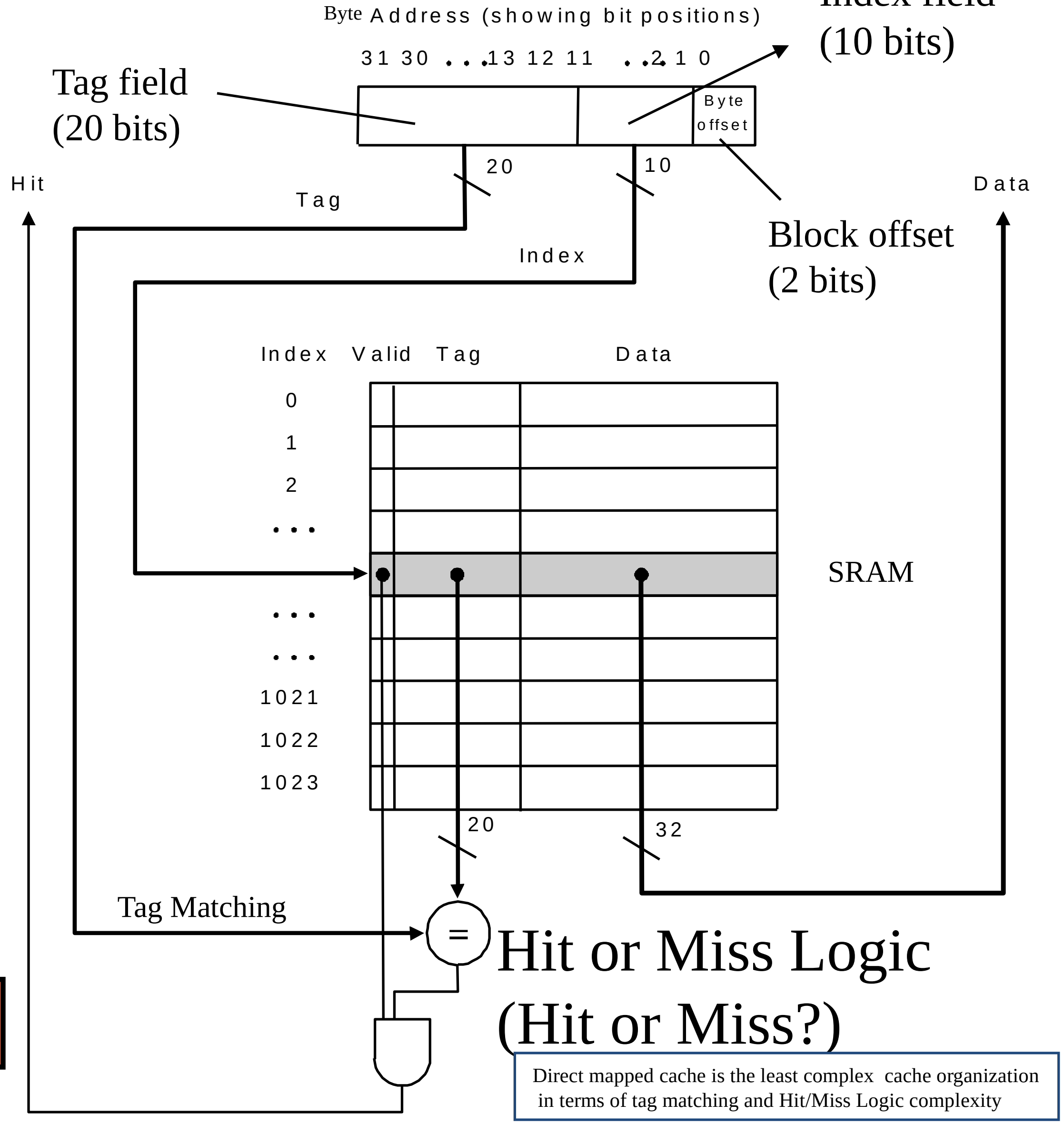
Cache Block frame number = (Block address) MOD (1024)

i.e . Index field or 10 low bits of block address

Block Address = 30 bits		Block offset = 2 bits
Tag = 20 bits	Index = 10 bits	
Tag	Index	Offset

Mapping

Address from CPU



Index field (10 bits)

Tag field (20 bits)

Block offset (2 bits)

SRAM

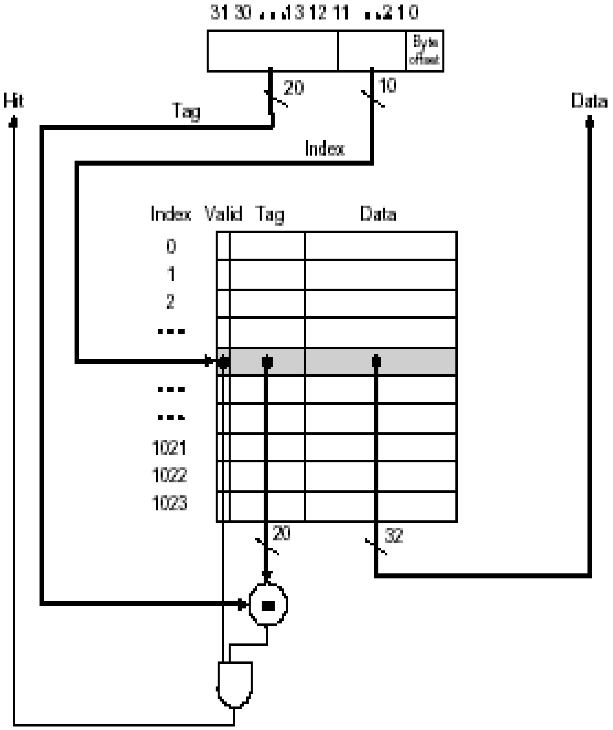
Tag Matching

Hit or Miss Logic (Hit or Miss?)

Direct mapped cache is the least complex cache organization in terms of tag matching and Hit/Miss Logic complexity

Hit Access Time = SRAM Delay + Hit/Miss Logic Delay

Direct Mapped Cache [contd…]



- What is the size of cache ?

4K

- If I read

0000 0000 0000 0000 0000 0000 1000 0001

- What is the index number checked ?

- If the number was found, what are the inputs to comparator ?

64KB Direct Mapped Cache Example

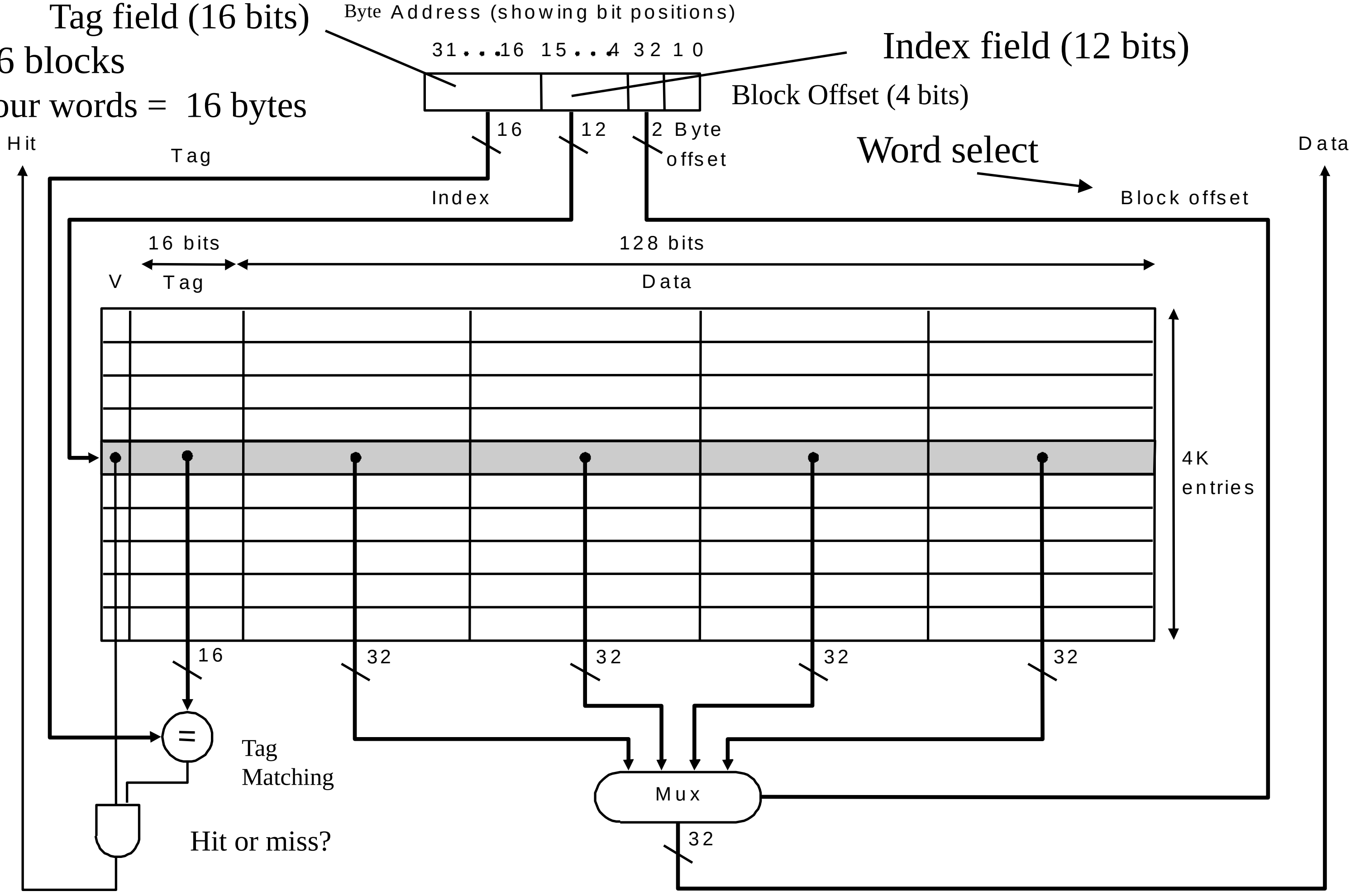
Nominal Capacity

4K= 2^{12} = 4096 blocks

Each block = four words = 16 bytes

Can cache up to 2^{32} bytes = 4 GB of memory

SRAM



Tag field (16 bits)

Index field (12 bits)

Block Offset (4 bits)

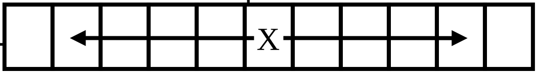
Word select

Tag Matching

Hit or miss?

Typical cache Block or line size: 32-64 bytes

Larger cache blocks take better advantage of spatial locality and thus may result in a lower miss rate



Block Address = 28 bits		Block offset = 4 bits
Tag = 16 bits	Index = 12 bits	

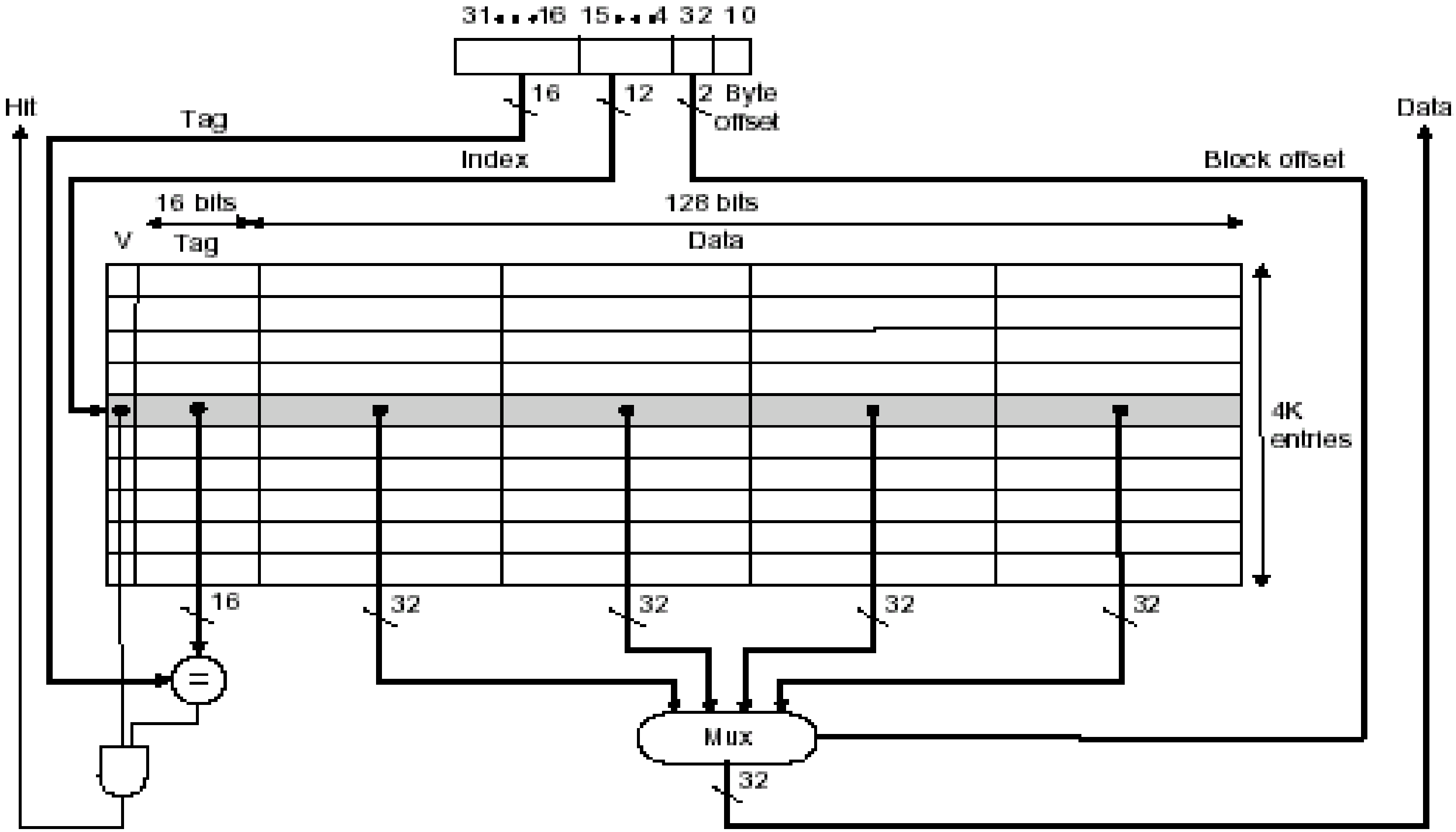
<u>Mapping Function:</u> Cache Block frame number = (Block address) MOD (4096)

i.e. index field or 12 low bit of block address

Hit Access Time = SRAM Delay + Hit/Miss Logic Delay

Direct Mapped Cache [contd…]

Taking advantage of spatial locality, we read 4 bytes at a time



Caching Terminology

- block/line : the unit of information that can be exchanged between two adjacent levels in the memory hierarchy
- cache hit: when CPU finds a requested data item in the cache
- cache miss: when CPU does not find a requested data item in the cache
- miss rate (*m*) : fraction of cache accesses that result in a miss
- hit rate (*h*) : (1- miss rate)fraction of memory access found in the faster level (cache)
- hit time (t_c) : time required to access the level 1 (cache)
- miss penalty (t_p) : time required to replace a data unit in the faster level (cache) + time involved to deliver the requested data to the processor from the cache. (hit time << miss penalty)
- average memory access time (amat)= hit time (t_c) + miss rate (*m*) * miss penalty (t_p) = $t_c + (1-h) * t_p$

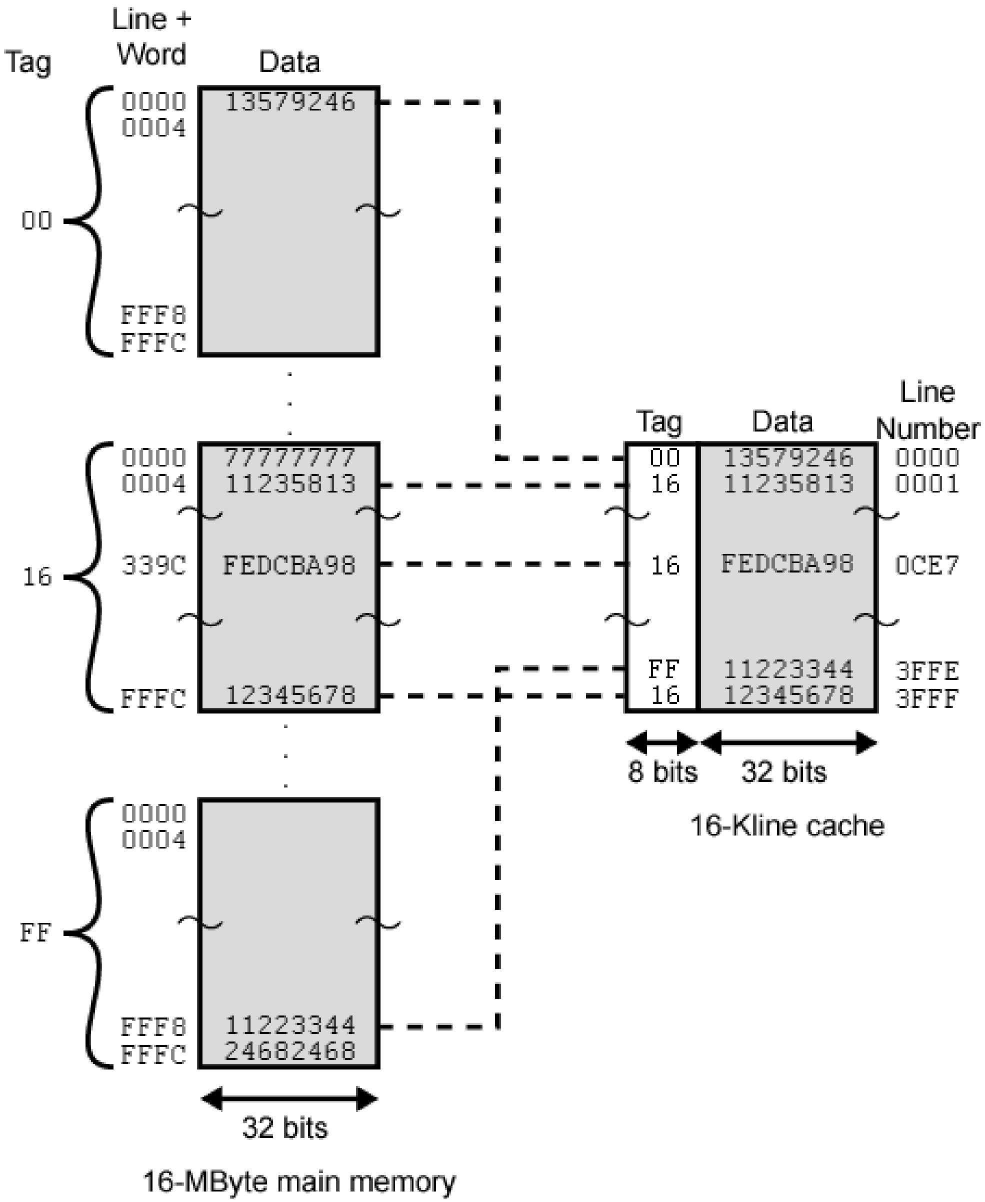
Cache Misses

- Compulsory (cold start or process migration, first reference): first access to a block
  - “Cold” fact of life: not a whole lot you can do about it
  - Note: If you are going to run “billions” of instruction, Compulsory Misses are insignificant
- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- Coherence (Invalidation): other process (e.g., I/O) updates memory

Direct Mapped Cache [contd…]

- Advantage
  - Simple
  - Fast
- Disadvantage
  - Mapping is fixed !!!

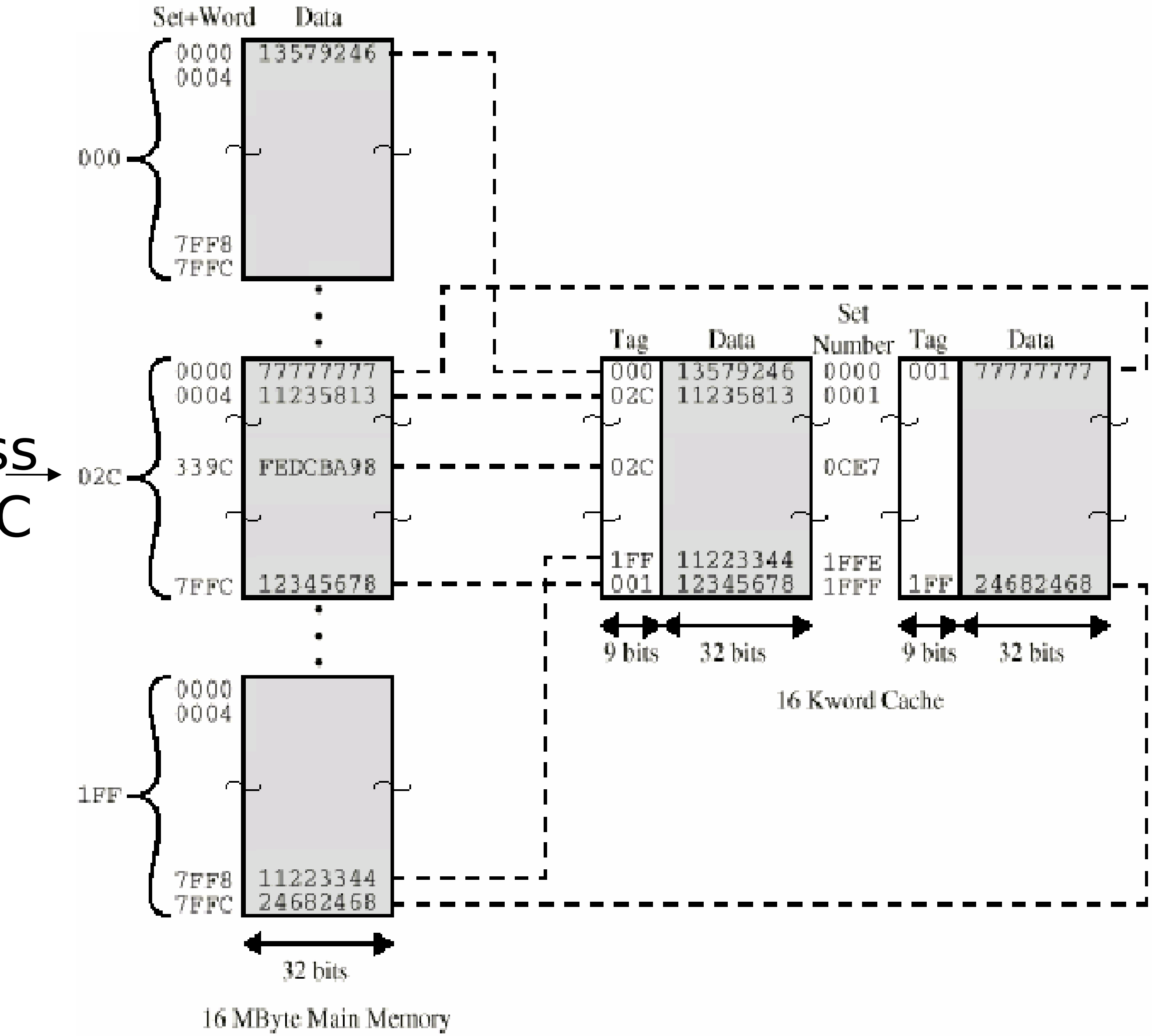
Direct Mapping Example



	Tag	Line	Word
Main memory address =	8	14	2

Two Way Set Associative Example

Address
16339C

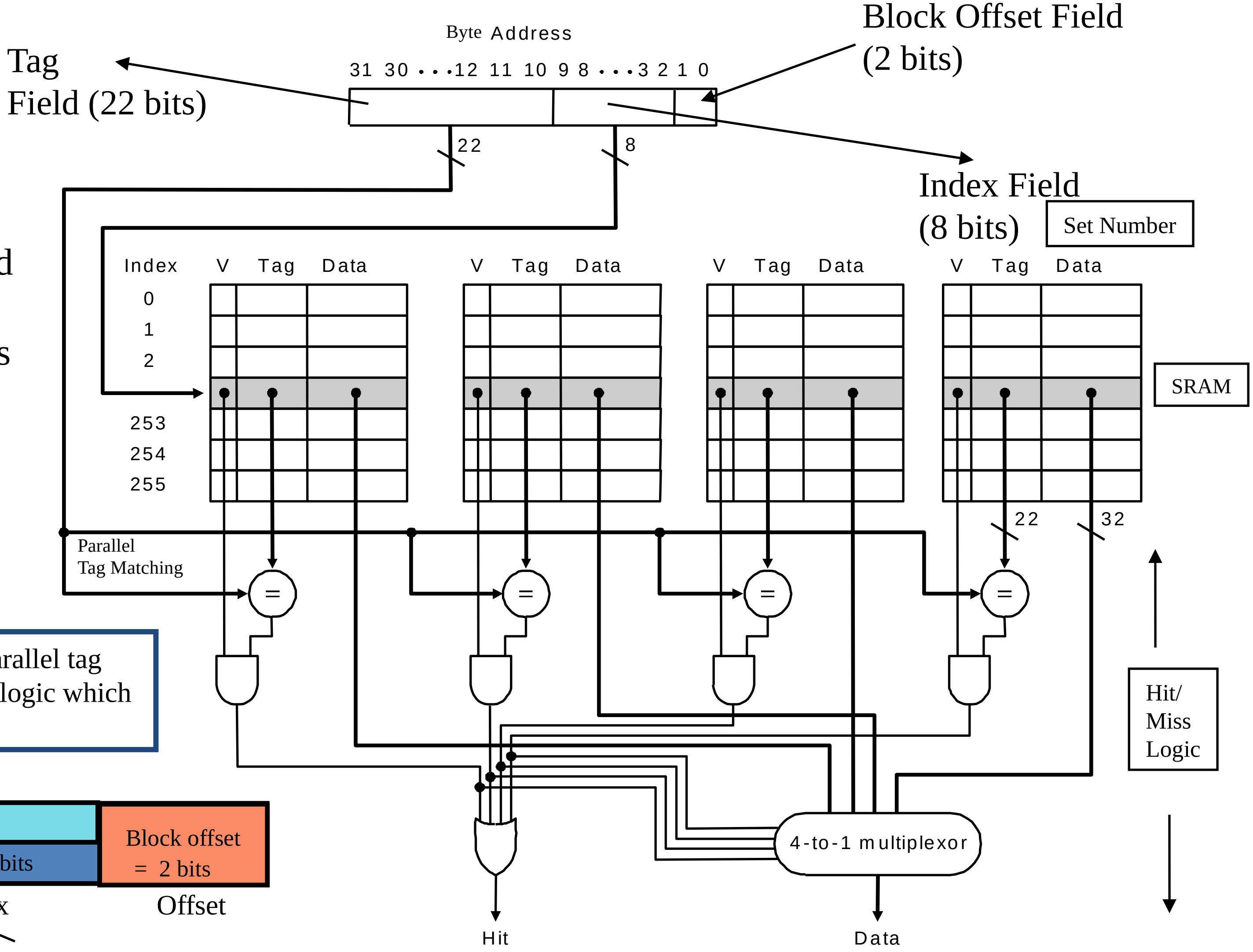


Nominal Capacity

4K Four-Way Set Associative Cache: MIPS Implementation Example

1024 block frames
Each block = one word
4-way set associative
1024 / 4= 2^8= 256 sets

Can cache up to
2^{32} bytes = 4 GB
of memory



Tag
Field (22 bits)

Block Offset Field
(2 bits)

Index Field
(8 bits)

Set associative cache requires parallel tag matching and more complex hit logic which may increase hit time

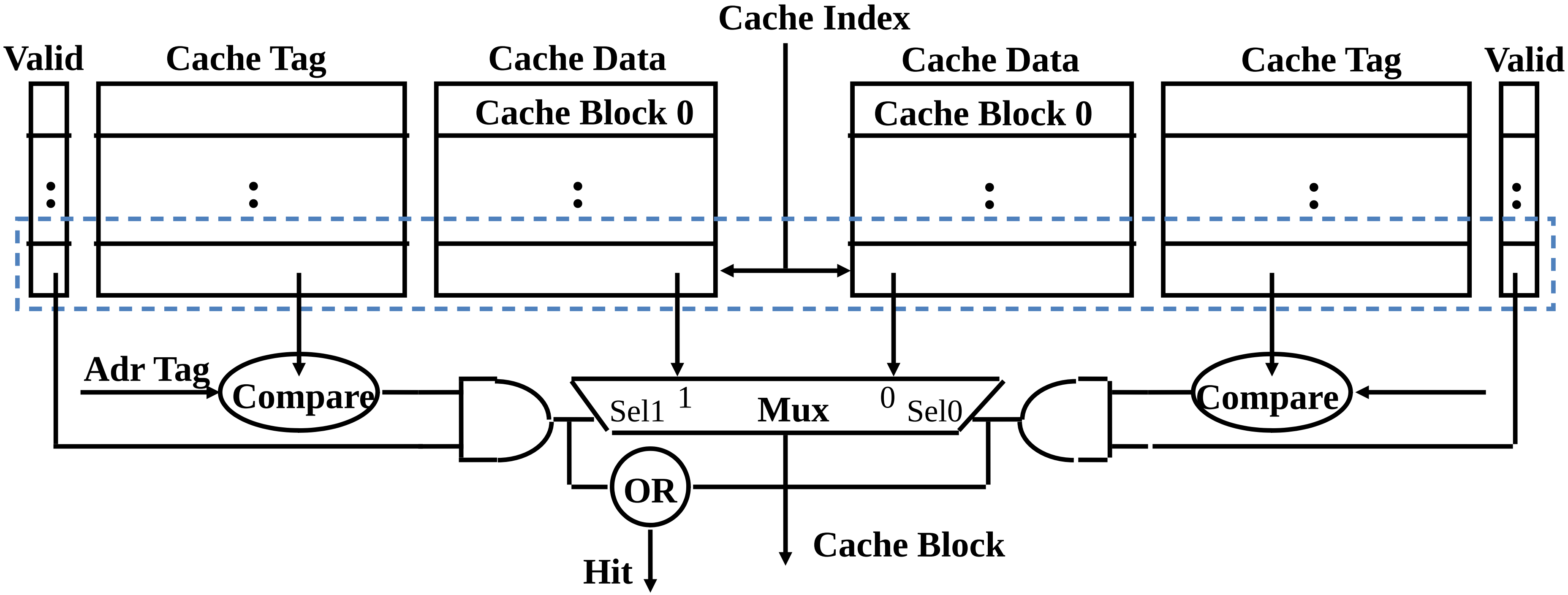
Block Address = 30 bits		Block offset = 2 bits
Tag = 22 bits	Index = 8 bits	
Tag	Index	Offset

Mapping Function: Cache Set Number = index= (Block address) MOD (256)

Hit Access Time = SRAM Delay + Hit/Miss Logic Delay

Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Cache Index selects a “set” from the cache
  - The two tags in the set are compared to the input in parallel
  - Data is selected based on the tag result



Problem 1

(a) A 64KB, direct mapped cache has 16 byte blocks. If addresses are 32 bits, how many bits are used the tag, index, and offset in this cache?

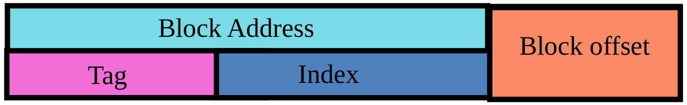
(b) How would the address be divided if the cache were 4-way set associative instead?

(c) How many bits is the index for a fully associative cache.

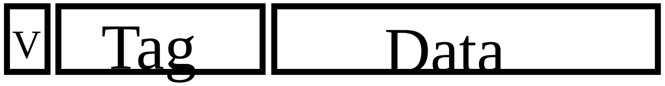
Problem 2

An ISA has 44 bit addresses with each addressable item being a byte. Design a four-way set associative cache with each of the four lines in a set containing 64 bytes. Assume that you have 256 sets in the cache.

Calculating Number of Cache Bits Needed



Address Fields



Cache Block Frame (or just cache block)

- How many total bits are needed for a direct- mapped cache with 64 KBytes of data and one word blocks, assuming a 32-bit address?
  - 64 Kbytes = 16 K words = 2^{14} words = 2^{14} blocks
  - Block size = 4 bytes => offset size = $\log_2(4)$ = 2 bits,
  - #sets = #blocks = 2^{14} => index size = 14 bits
  - Tag size = address size - index size - offset size = 32 - 14 - 2 = 16 bits
  - Bits/block = data bits + tag bits + valid bit = 32 + 16 + 1 = 49
  - Bits in cache = #blocks x bits/block = 2^{14} x 49 = 98 Kbytes

i.e nominal cache Capacity = 64 KB

Number of cache block frames

Actual number of bits in a cache block frame

- How many total bits would be needed for a 4-way set associative cache to store the same amount of data?
  - Block size and #blocks does not change.
  - #sets = #blocks/4 = $(2^{14})/4$ = 2^{12} => index size = 12 bits
  - Tag size = address size - index size - offset = 32 - 12 - 2 = 18 bits
  - Bits/block = data bits + tag bits + valid bit = 32 + 18 + 1 = 51
  - Bits in cache = #blocks x bits/block = 2^{14} x 51 = 102 Kbytes

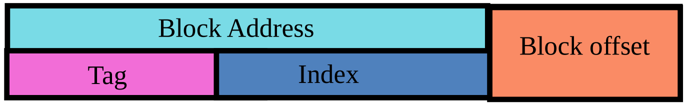
- Increase associativity => increase bits in cache

More bits in tag

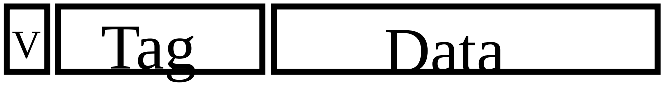
Word = 4 bytes

1 k = 1024 = 2^{10}

Calculating Cache Bits Needed



Address Fields



Cache Block Frame (or just cache block)

Nominal size

- How many total bits are needed for a direct- mapped cache with 64 KBytes of data and 8 word (32 byte) blocks, assuming a 32-bit address (it can cache 2^{32} bytes in memory)?
  - 64 Kbytes = 2^{14} words = $(2^{14})/8 = 2^{11}$ blocks

    Number of cache block frames
  - block size = 32 bytes

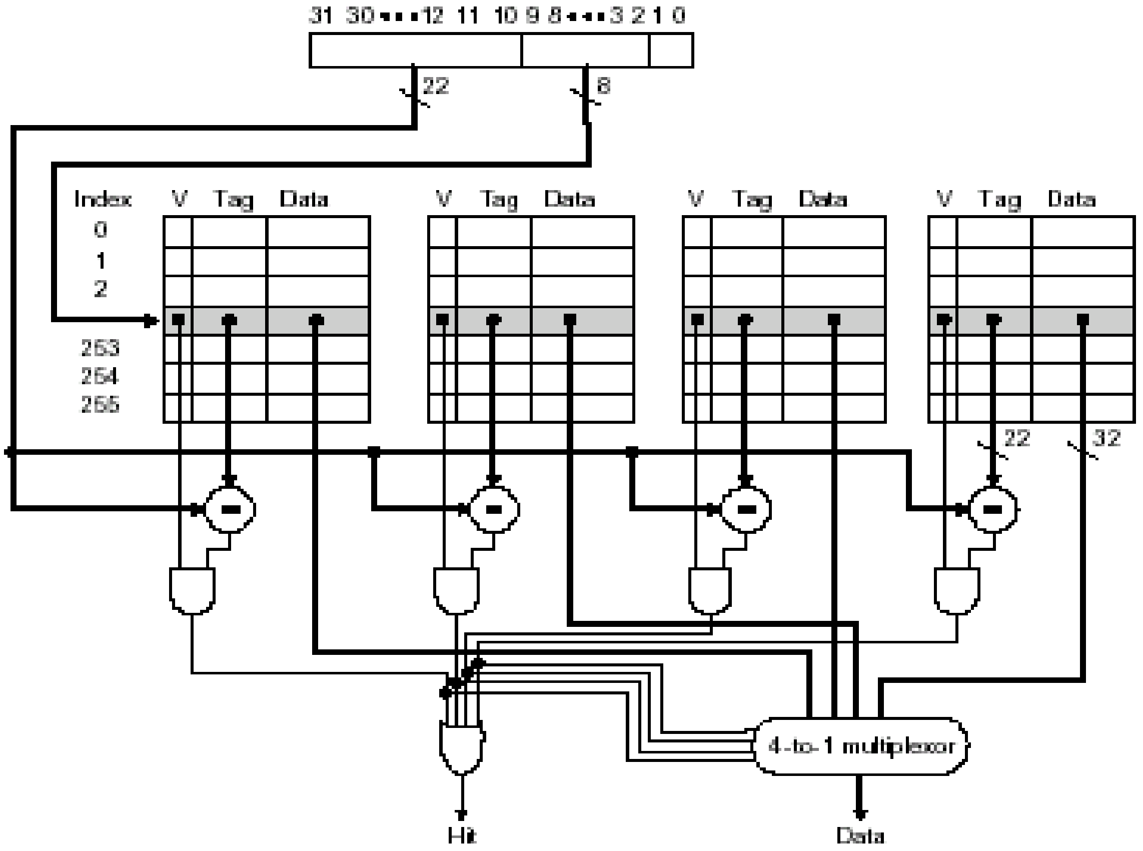
    => offset size = block offset + byte offset = $\log_2(32)$ = 5 bits,
  - #sets = #blocks = 2^{11} => index size = 11 bits
  - tag size = address size - index size - offset size = 32 - 11 - 5 = 16 bits
  - bits/block = data bits + tag bits + valid bit = 8 x 32 + 16 + 1 = 273 bits

    Actual number of bits in a cache block frame
  - bits in cache = #blocks x bits/block = 2^{11} x 273 = 68.25 Kbytes
- In e.

Fewer cache block frames thus fewer tags/valid bits

Word = 4 bytes   1 k = 1024 = 2^{10}

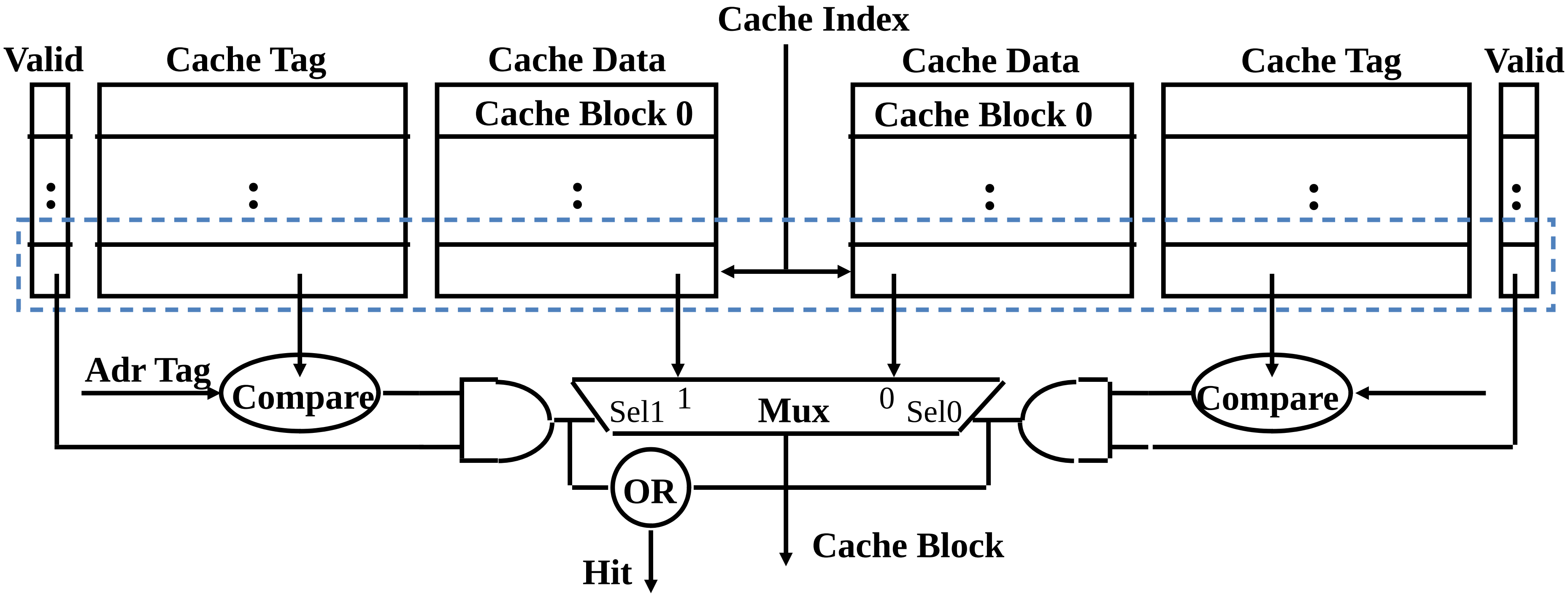
Example: 4-way set associative Cache



What is the cache size in this case ?

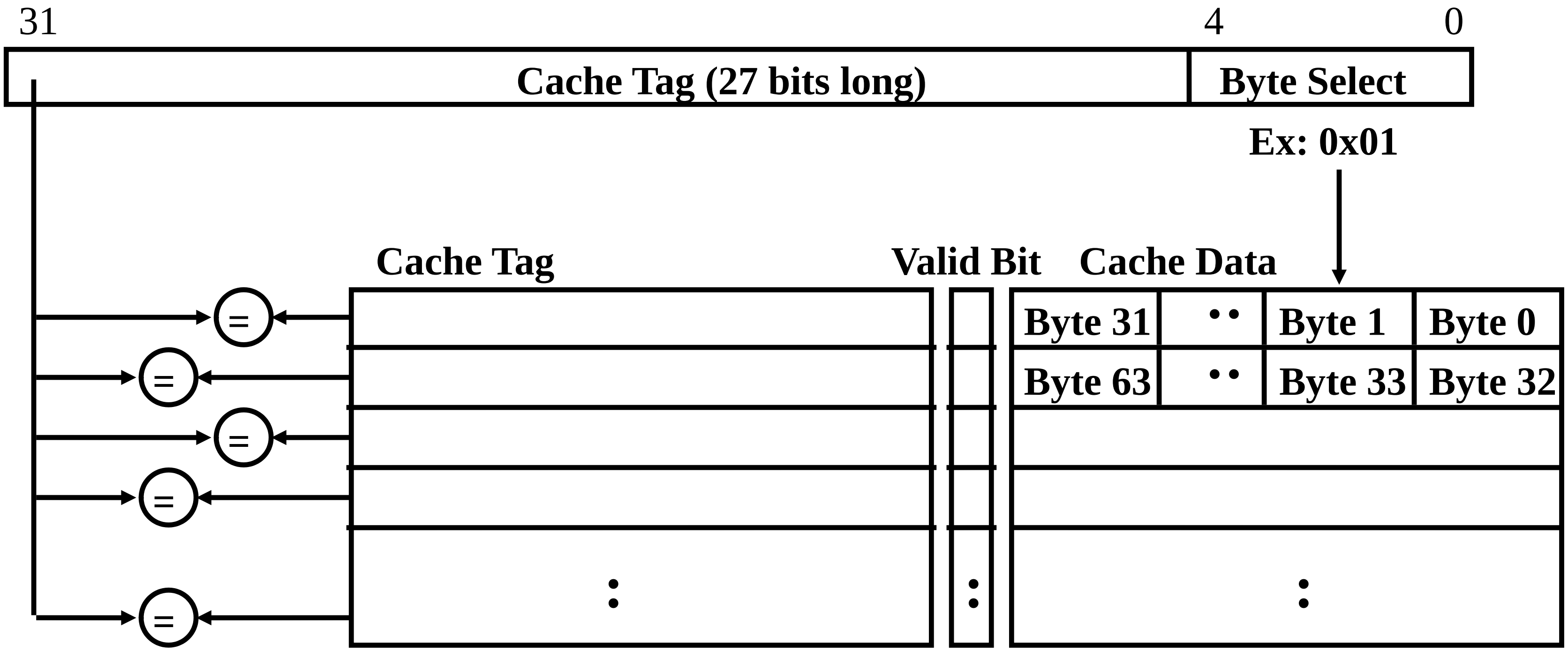
Disadvantages of Set Associative Cache

- N-way Set Associative Cache versus Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss decision and set selection
- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:

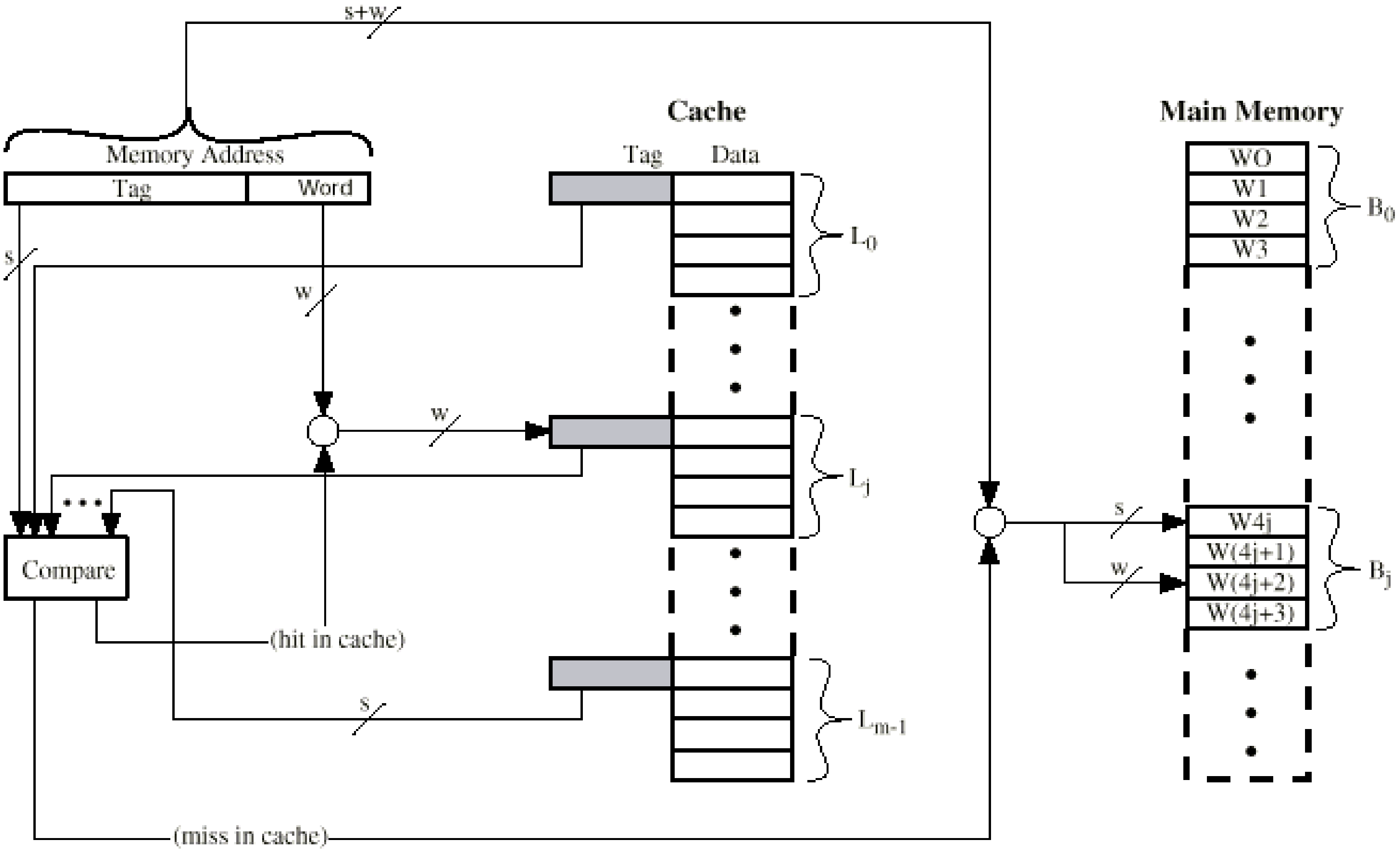


Fully Associative Cache

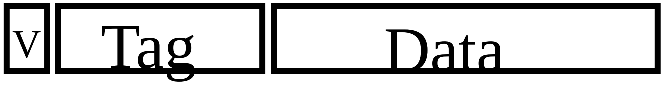
- Fully Associative Cache
  - Forget about the Cache Index
  - Compare the Cache Tags of all cache entries in parallel
  - Example: Block Size = 32 B blocks, we need N 27-bit comparators
- By definition: Conflict Miss = 0 for a fully associative cache



Fully Associative Cache Organization



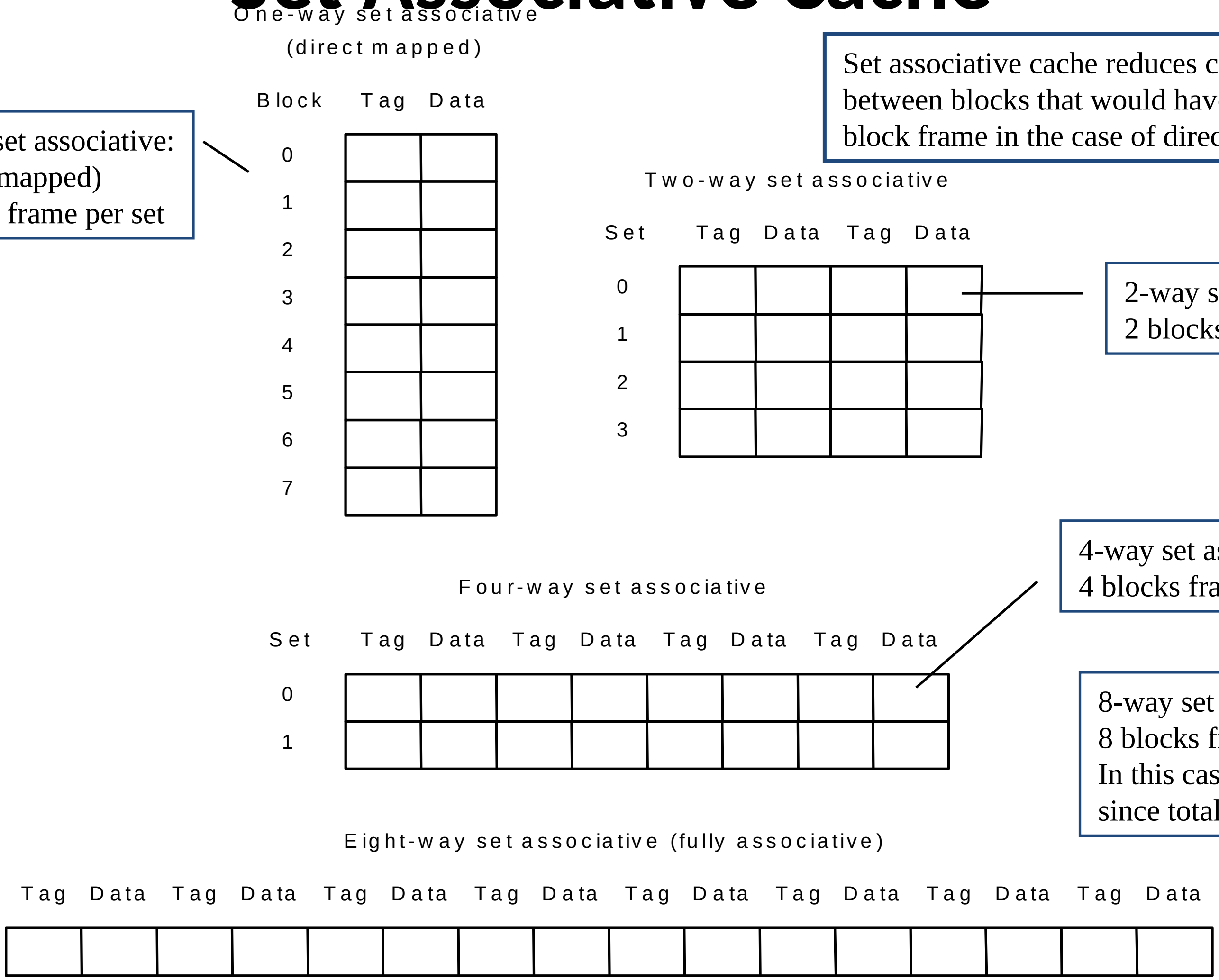
Cache Organization:
Set Associative Cache



Cache Block Frame

Why set associative?

Set associative cache reduces cache misses by <u>reducing conflicts</u> between blocks that would have been mapped to the same cache block frame in the case of direct mapped cache



1-way set associative:
(direct mapped)
1 block frame per set

2-way set associative:
2 blocks frames per set

4-way set associative:
4 blocks frames per set

8-way set associative:
8 blocks frames per set
In this case it becomes fully associative
since total number of block frames = 8

A cache with a total of 8 cache block frames shown

Design Options at Constant Cost

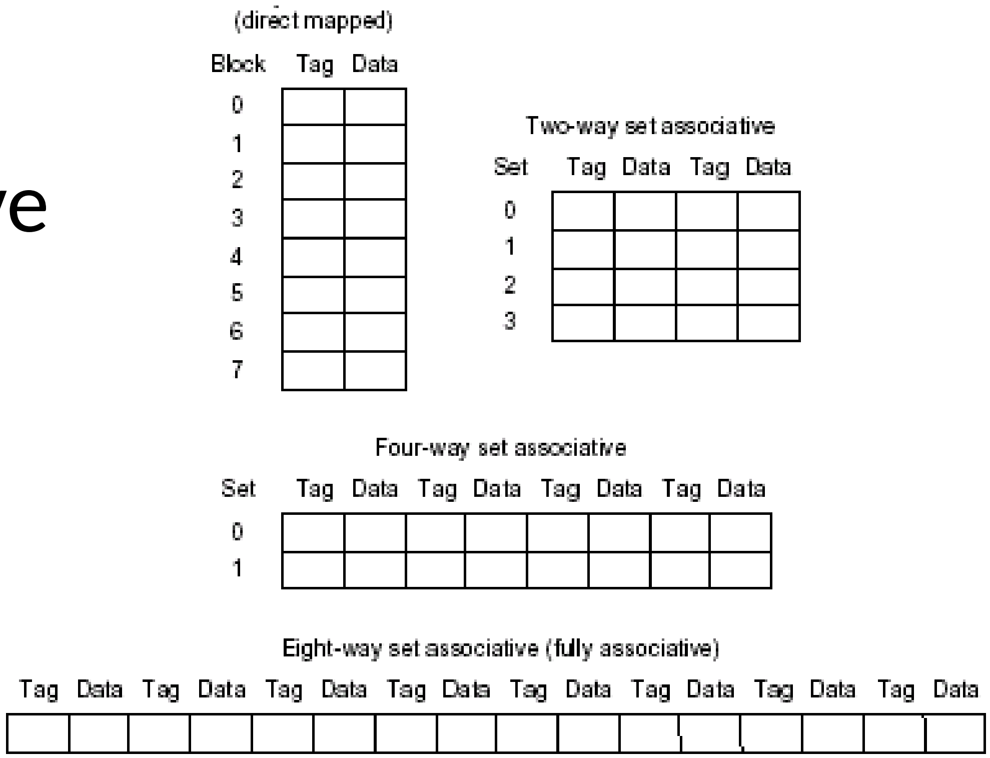
	Direct Mapped	**N-way Set Associative**	**Fully Associative**
Cache Size	Big	Medium	Small
Compulsory Miss	Same	Same	Same
Conflict Miss	High	Medium	Zero
Capacity Miss	Low	Medium	High
Coherence Miss	Same	Same	Same

Four Questions for Cache Design

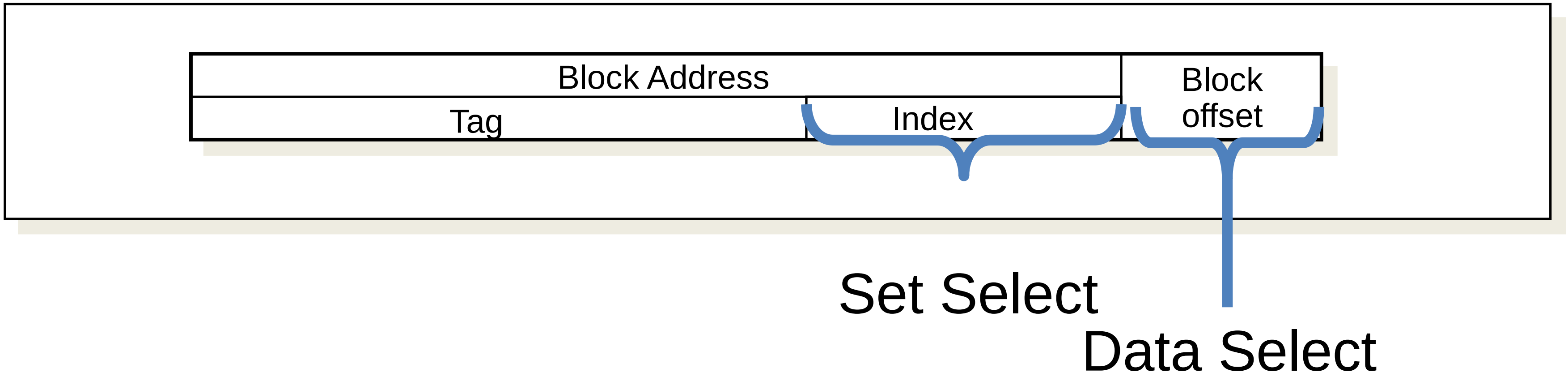
- Q1: Where can a block be placed in the upper level? *(Block placement)*
- Q2: How is a block found if it is in the upper level? *(Block identification)*
- Q3: Which block should be replaced on a miss? *(Block replacement)*
- Q4: What happens on a write? *(Write strategy)*

Where can a block be placed in the upper level?

- Direct Mapped
- Set Associative
- Fully Associative



How is a block found if it is in the upper level?



- Direct indexing (using index and block offset), tag compares, or combination
- Increasing associativity shrinks index, expands tag

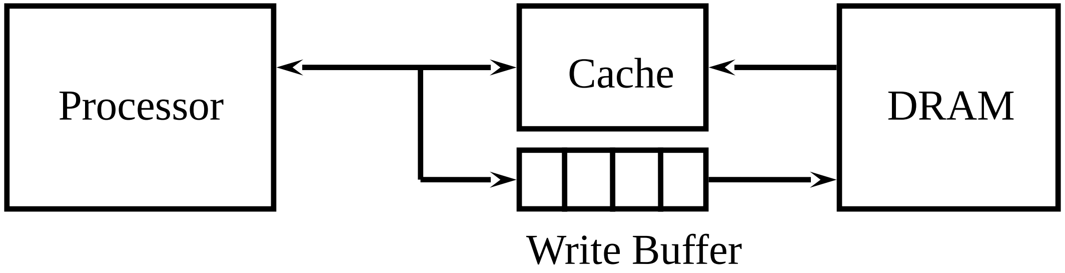
Which block should be replaced on a miss?

- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

Associativity:	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

What happens on a write?

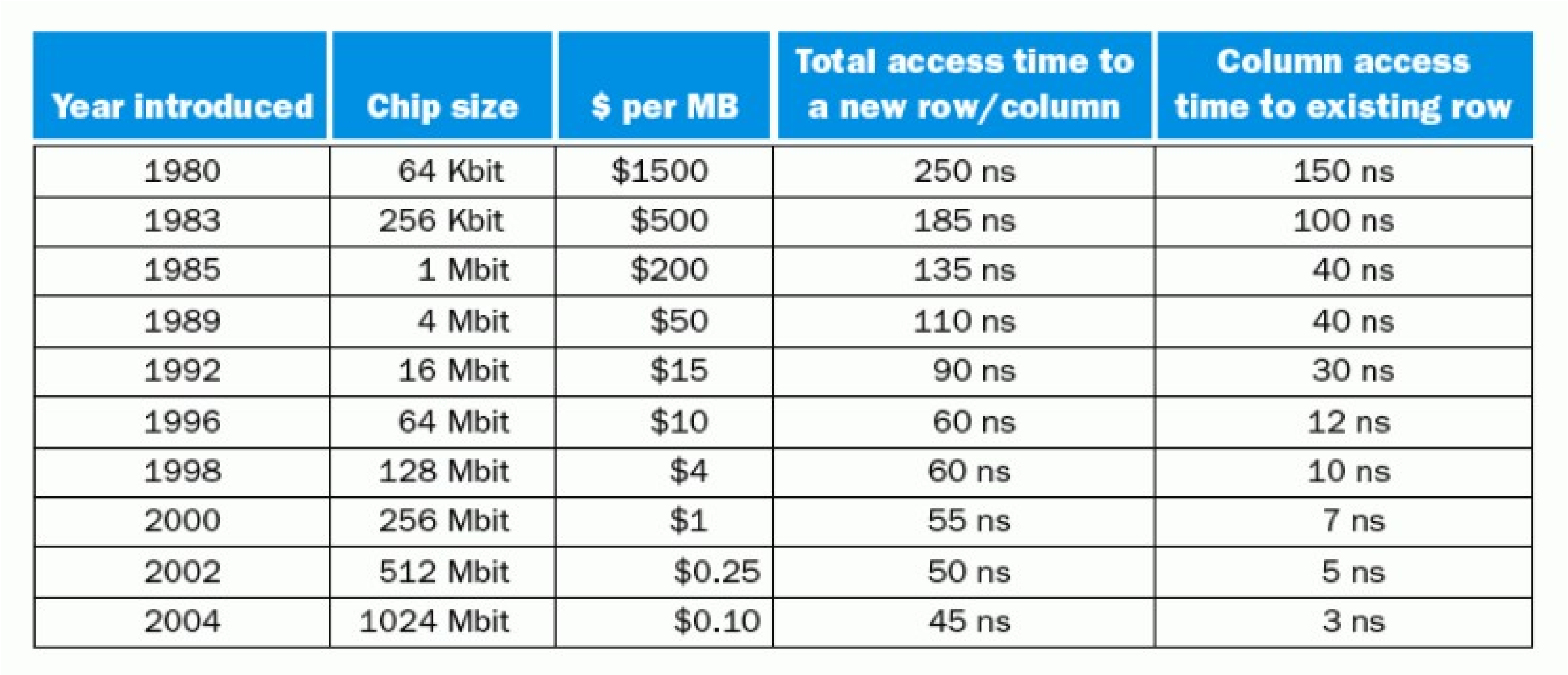
- *Write through*—The information is written to both the block in the cache and to the block in the lower-level memory.
- *Write back*—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- WT always combined with write buffers so that don't wait for lower level memory



Intel Cache Evolution

Problem	**Solution**	**Processor on which feature first appears**
External memory slower than the system bus.	Add external cache using faster memory technology.	386
Increased processor speed results in external bus becoming a bottleneck for cache access.	Move external cache on-chip, operating at the same speed as the processor.	486
Internal cache is rather small, due to limited space on chip	Add external L2 cache using faster technology than main memory	486
Contention occurs when both the Instruction Prefetcher and the Execution Unit simultaneously require access to the cache. In that case, the Prefetcher is stalled while the Execution Unit’s data access takes place.	Create separate data and instruction caches.	Pentium
Increased processor speed results in external bus becoming a bottleneck for L2 cache access.	Create separate back-side bus that runs at higher speed than the main (front-side) external bus. The BSB is dedicated to the L2 cache.	Pentium Pro
	Move L2 cache on to the processor chip.	Pentium II
Some applications deal with massive databases and must have rapid access to large amounts of data. The on-chip caches are too small.	Add external L3 cache.	Pentium III
	Move L3 cache on-chip.	Pentium 4

Main memory Evolution



Year introduced	Chip size	$ per MB	Total access time to a new row/column	Column access time to existing row
1980	64 Kbit	$1500	250 ns	150 ns
1983	256 Kbit	$500	185 ns	100 ns
1985	1 Mbit	$200	135 ns	40 ns
1989	4 Mbit	$50	110 ns	40 ns
1992	16 Mbit	$15	90 ns	30 ns
1996	64 Mbit	$10	60 ns	12 ns
1998	128 Mbit	$4	60 ns	10 ns
2000	256 Mbit	$1	55 ns	7 ns
2002	512 Mbit	$0.25	50 ns	5 ns
2004	1024 Mbit	$0.10	45 ns	3 ns

Comparison of Cache Sizes

Processor	Type	Year of Introduction	L1 cache[a]	L2 cache	L3 cache
IBM 360/85	Mainframe	1968	16 to 32 KB	—	—
PDP-11/70	Minicomputer	1975	1 KB	—	—
VAX 11/780	Minicomputer	1978	16 KB	—	—
IBM 3033	Mainframe	1978	64 KB	—	—
IBM 3090	Mainframe	1985	128 to 256 KB	—	—
Intel 80486	PC	1989	8 KB	—	—
Pentium	PC	1993	8 KB/8 KB	256 to 512 KB	—
PowerPC 601	PC	1993	32 KB	—	—
PowerPC 620	PC	1996	32 KB/32 KB	—	—
PowerPC G4	PC/server	1999	32 KB/32 KB	256 KB to 1 MB	2 MB
IBM S/390 G4	Mainframe	1997	32 KB	256 KB	2 MB
IBM S/390 G6	Mainframe	1999	256 KB	8 MB	—
Pentium 4	PC/server	2000	8 KB/8 KB	256 KB	—
IBM SP	High-end server/ supercomputer	2000	64 KB/32 KB	8 MB	—
CRAY MTA[b]	Supercomputer	2000	8 KB	2 MB	—
Itanium	PC/server	2001	16 KB/16 KB	96 KB	4 MB
SGI Origin 2001	High-end server	2001	32 KB/32 KB	4 MB	—
Itanium 2	PC/server	2002	32 KB	256 KB	6 MB
IBM POWER5	High-end server	2003	64 KB	1.9 MB	36 MB
CRAY XD-1	Supercomputer	2004	64 KB/64 KB	1MB	—

Memory Hierarchy in Power 4/5

Cache	Capacity	Associativity	Line size	Write policy	Repl. alg	Comments
L1 I-cache	64 KB	Direct/2-way	128 B		LRU	Sector cache (4 sectors)
L1 D-cache	32 KB	2-way/4-way	128 B	Write-through	LRU	
L2 Unified	1.5 MB/2 MB	8-way/10-way	128 B	Write-back	Pseudo-LRU	
L3	32 MB/36MB	8-way/12-way	512 B	Write-back	?	Sector cache (4 sectors)

Latency	P4 (1.7 GHz)	P5 (1.9 GHz)
L1 (I and D)	1	1
L2	12	13
L3	123	87
Main Memory	351	220

Virtual Memory

Provides *illusion* of very large memory
– sum of the memory of many jobs greater than physical memory
– address space of each job larger than physical memory

Allows available (fast and expensive) physical memory to be very well utilized

Simplifies memory management

Exploits memory hierarchy to keep average access time low.

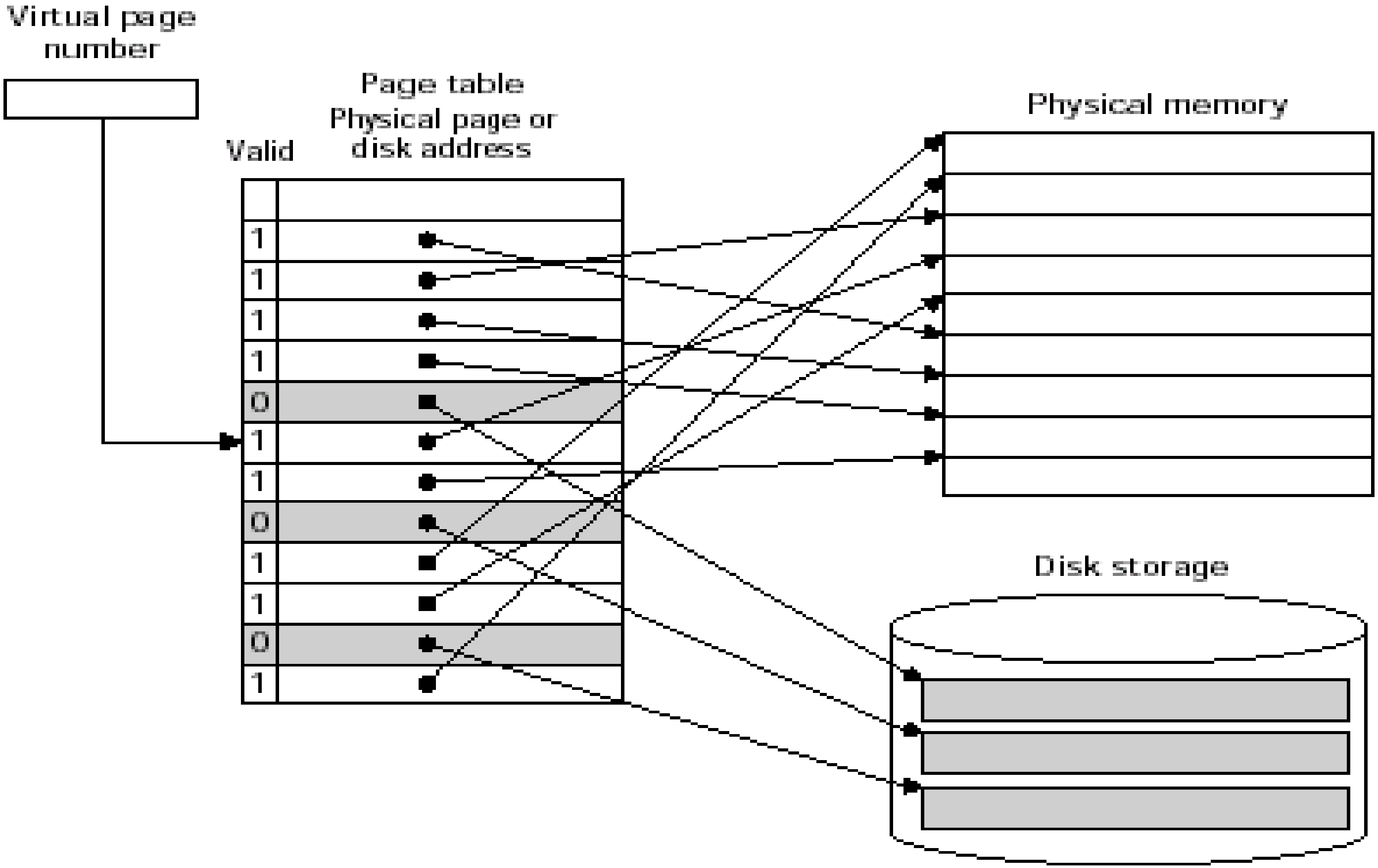
Involves at least two storage levels: *main* and *secondary*

Virtual Address -- address used by the programmer

Virtual Address Space -- collection of such addresses

Memory Address -- address of word in physical memory
also known as “physical address” or “real address”

Page Table address Mapping



Implementing paging

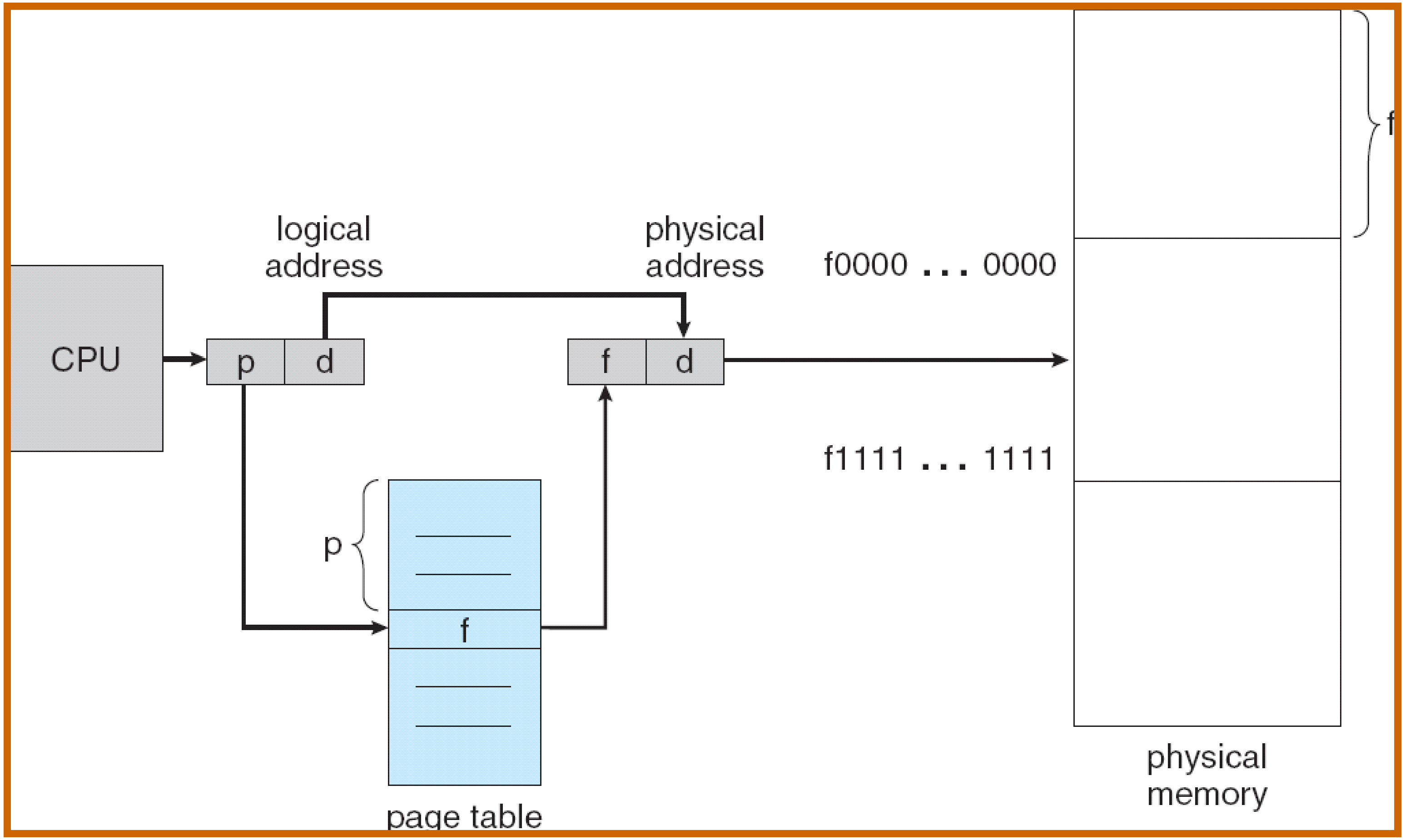
- Keep track of all free physical page frames

- To run a program of size n pages, need to find n free frames and load program
  - Pages do not need to be contiguous!

- Set up mapping from virtual to physical in page table

- At memory reference time, translate virtual address to physical address using page table

Paging Hardware



Paging Examples

Assume a page size of 1K and a 15-bit logical address space.

How many pages are in the system?

- Answer: 2^5 = 32.
- Assuming a 15-bit address space with 8 logical pages. How large are the pages?

- Answer: 2^5 = 32.
- Assuming a 15-bit address space with 8 logical pages. How large are the pages?
- Answer: 2^12 = 4K. It takes 3 bits to reference 8 logical pages (2^3 = 8). This leaves 12 bits for the page size thus pages are 2^12.

Consider logical address 1025 and the following

page table for some process P0. Assume a 15-bit address space with a page size of 1K. What is the physical address to which logical address 1025 will be mapped?

8
0
2

Consider logical address 1025 and the following

page table for some process P0. Assume a 15-bit address space with a page size of 1K. What is the physical address to which logical address 1025 maps?

8
0
2

Step 1. Convert to binary:

000010000000001

Consider logical address 1025 and the following

page table for some process P0. Assume a 15-bit address space with a page size of 1K. What is the physical address to which logical address 1025 maps?

8
0
2

Step2. Determine the logical page number:

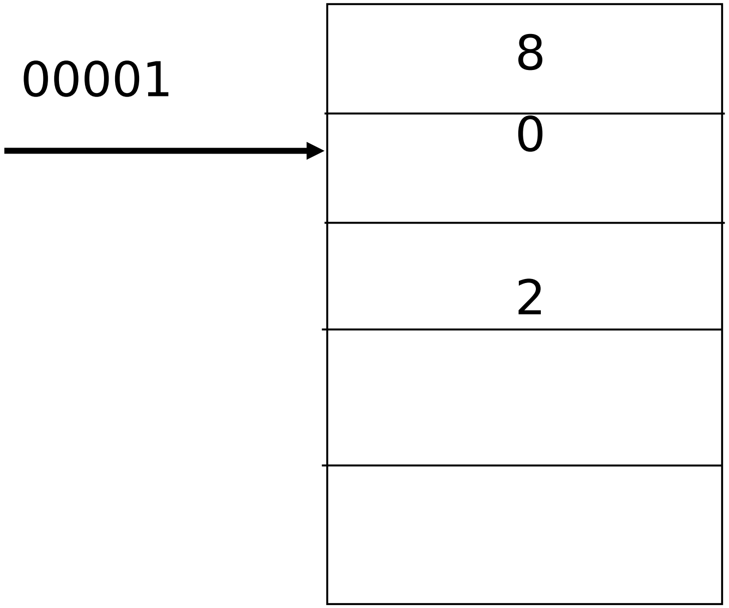
Since there are 5-bits allocated to the logical page, the address is broken up as follows:

00001 0000000001

Logical page number offset within
page

Consider logical address 1025 and the following

page table for some process P0. What is the physical address?

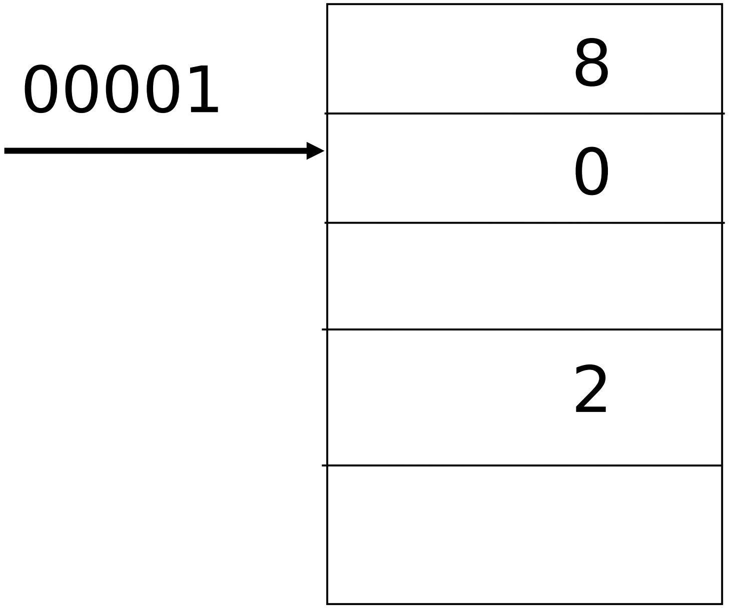


Step 3. Use logical page number as an index into the page table.

00001 0000000001

Consider logical address 1025 and the following

page table for some process P0. What is the physical address?



00001

8

0

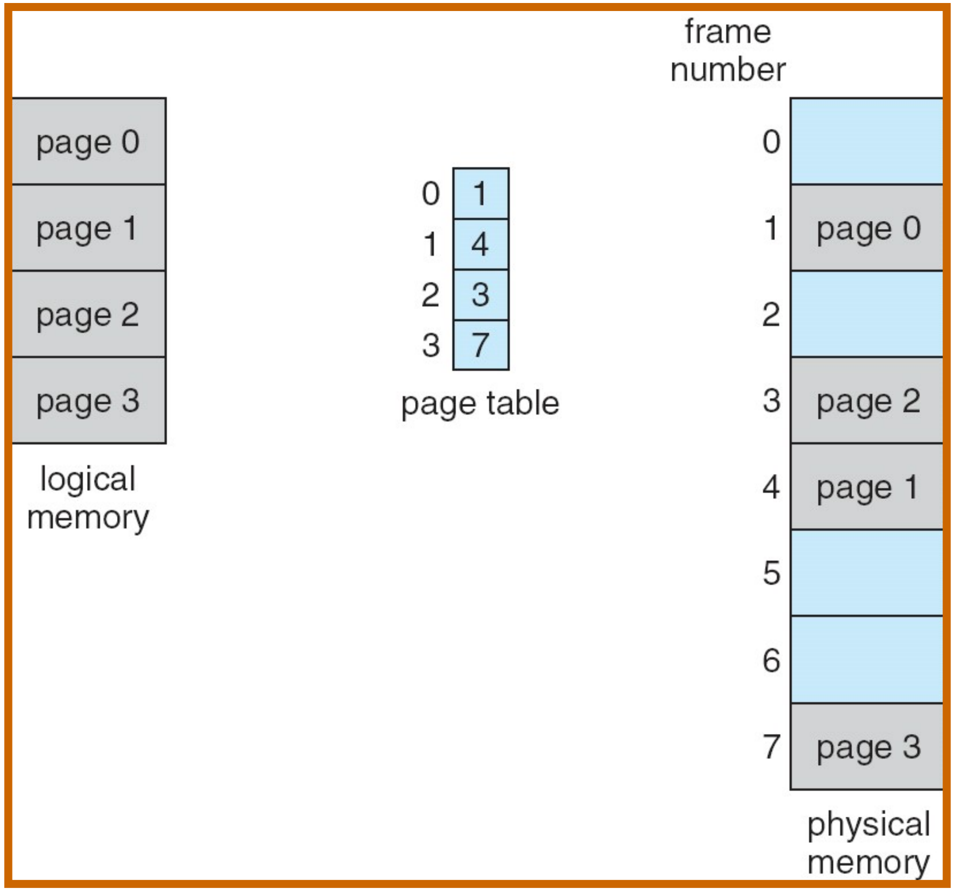
2

Take the physical page number from the page table and concatenate the offset.

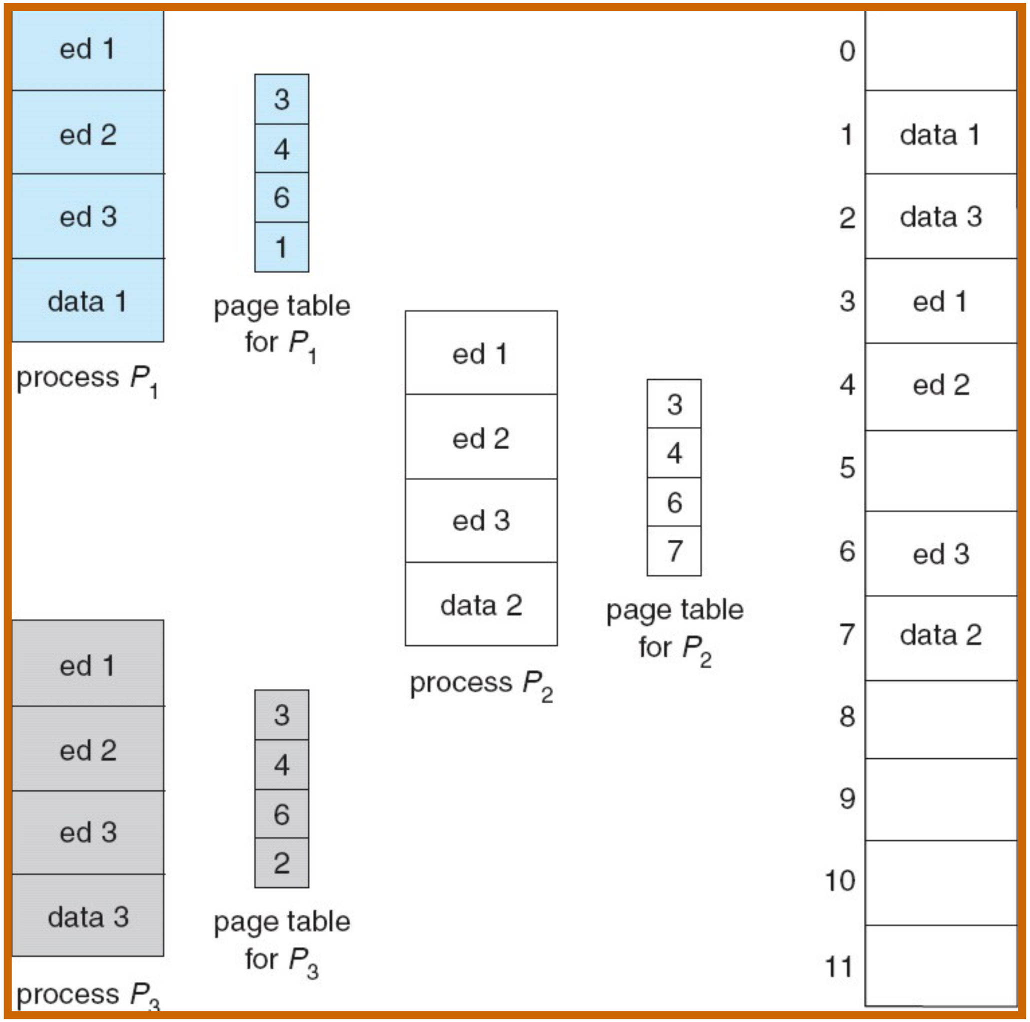
The physical address is byte 1.

000000000000001

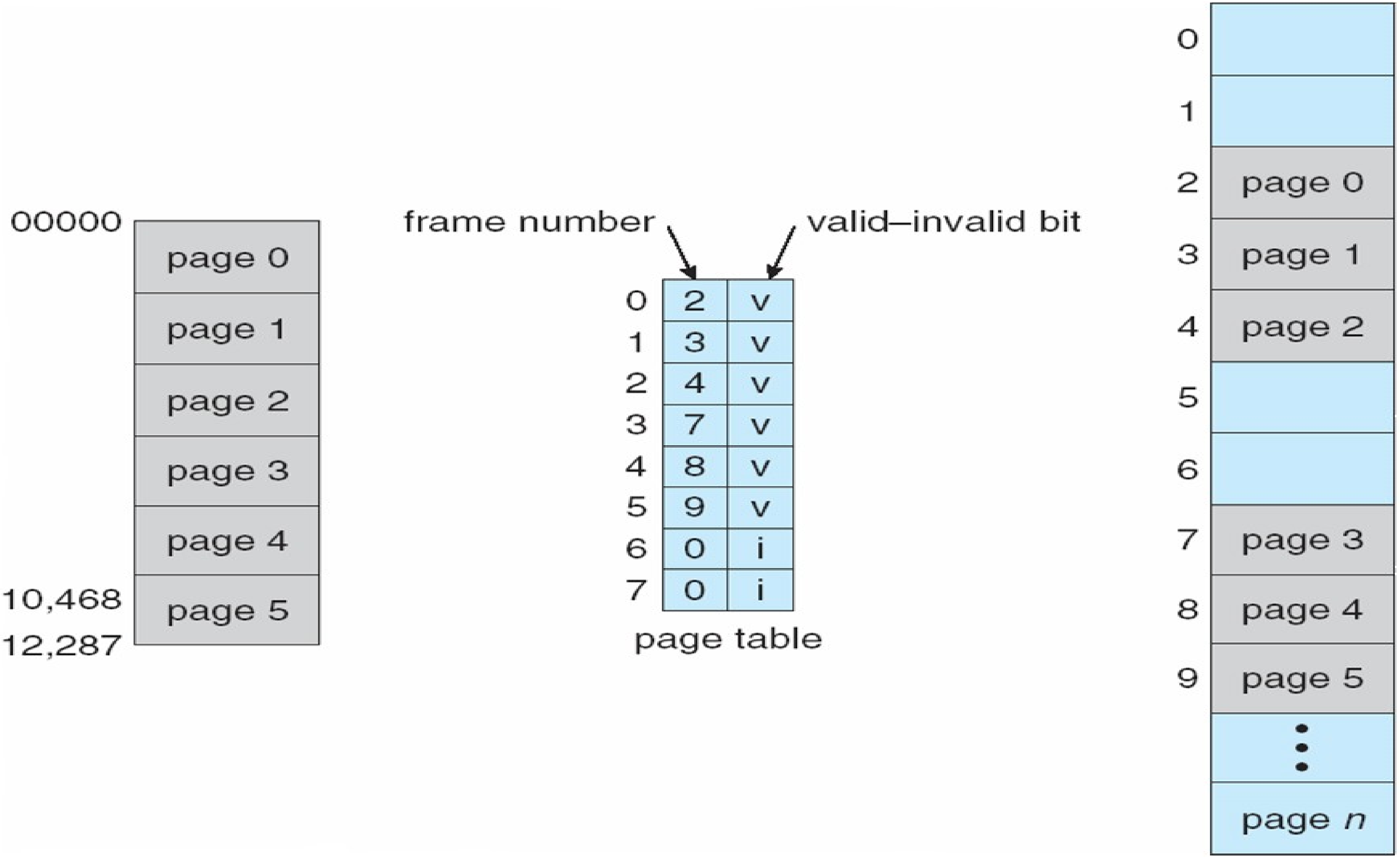
Paging Model of Logical and Physical Memory



Shared Pages Example



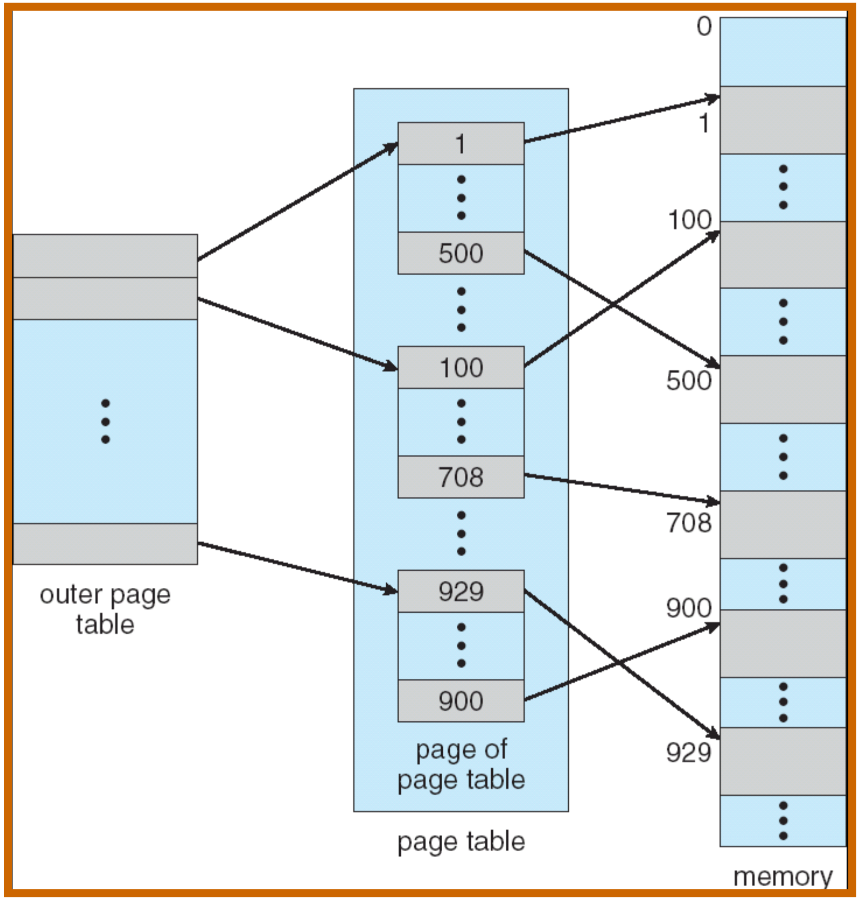
Valid (v) or Invalid (i) Bit in a Page Table



Hierarchical Page Tables

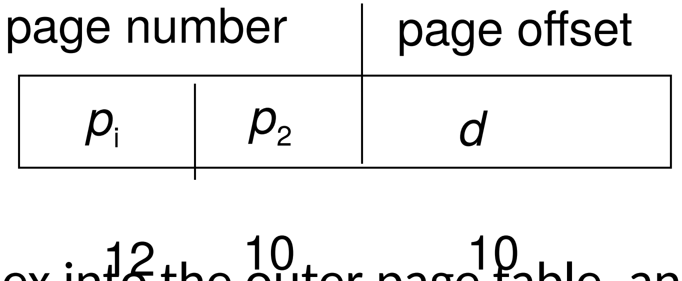
- Break up the logical address space into multiple page tables

- A simple technique is a two-level page table



Two-Level Paging Example

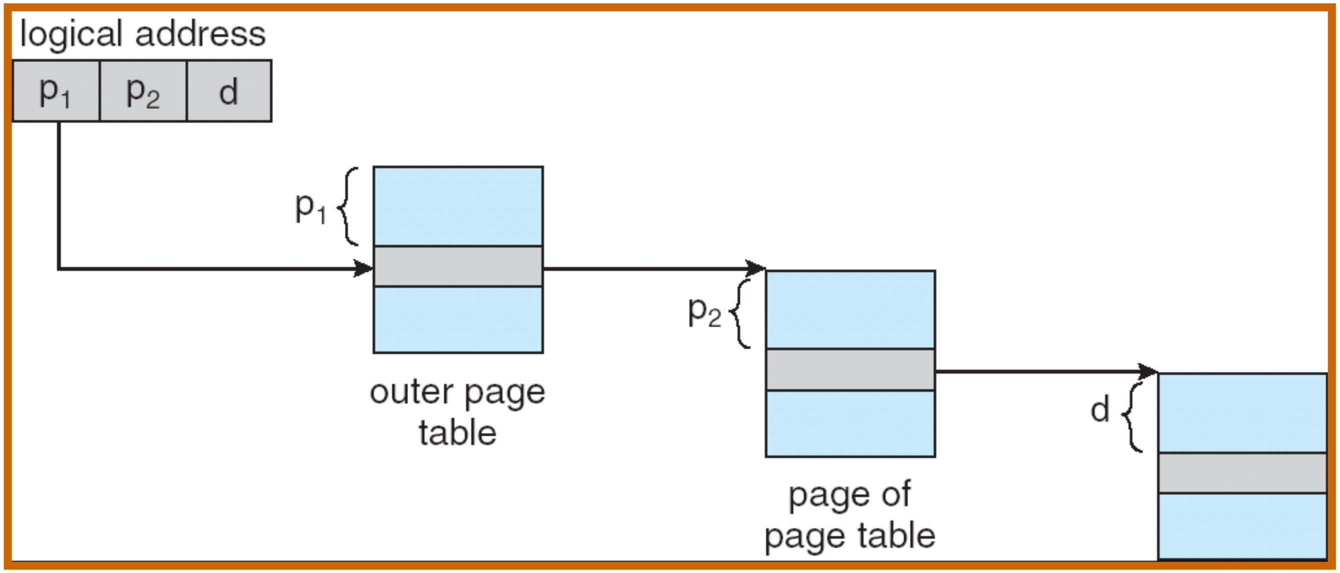
- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



page number		page offset
p_i	p_2	d
12	10	10

where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

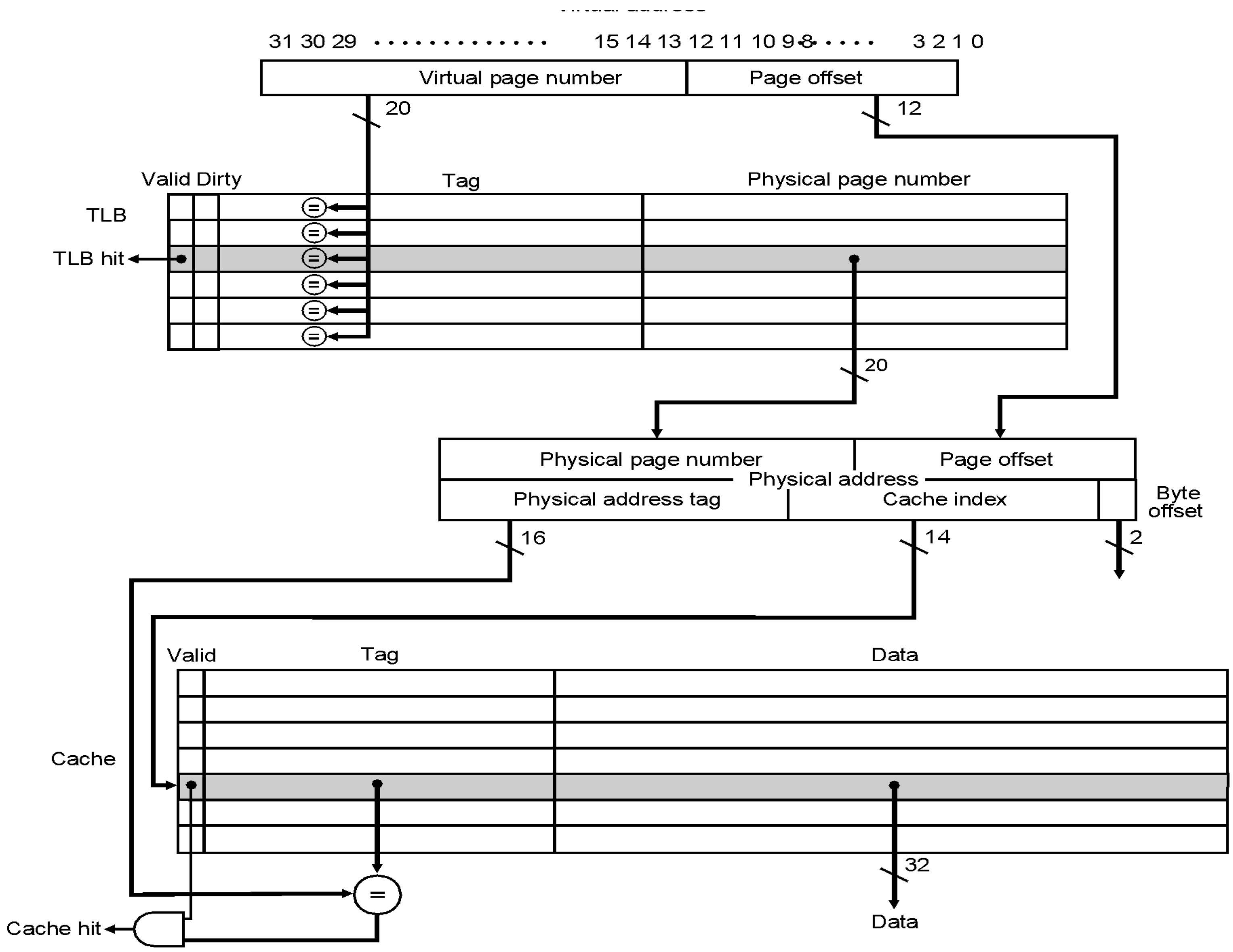
Address-Translation Scheme



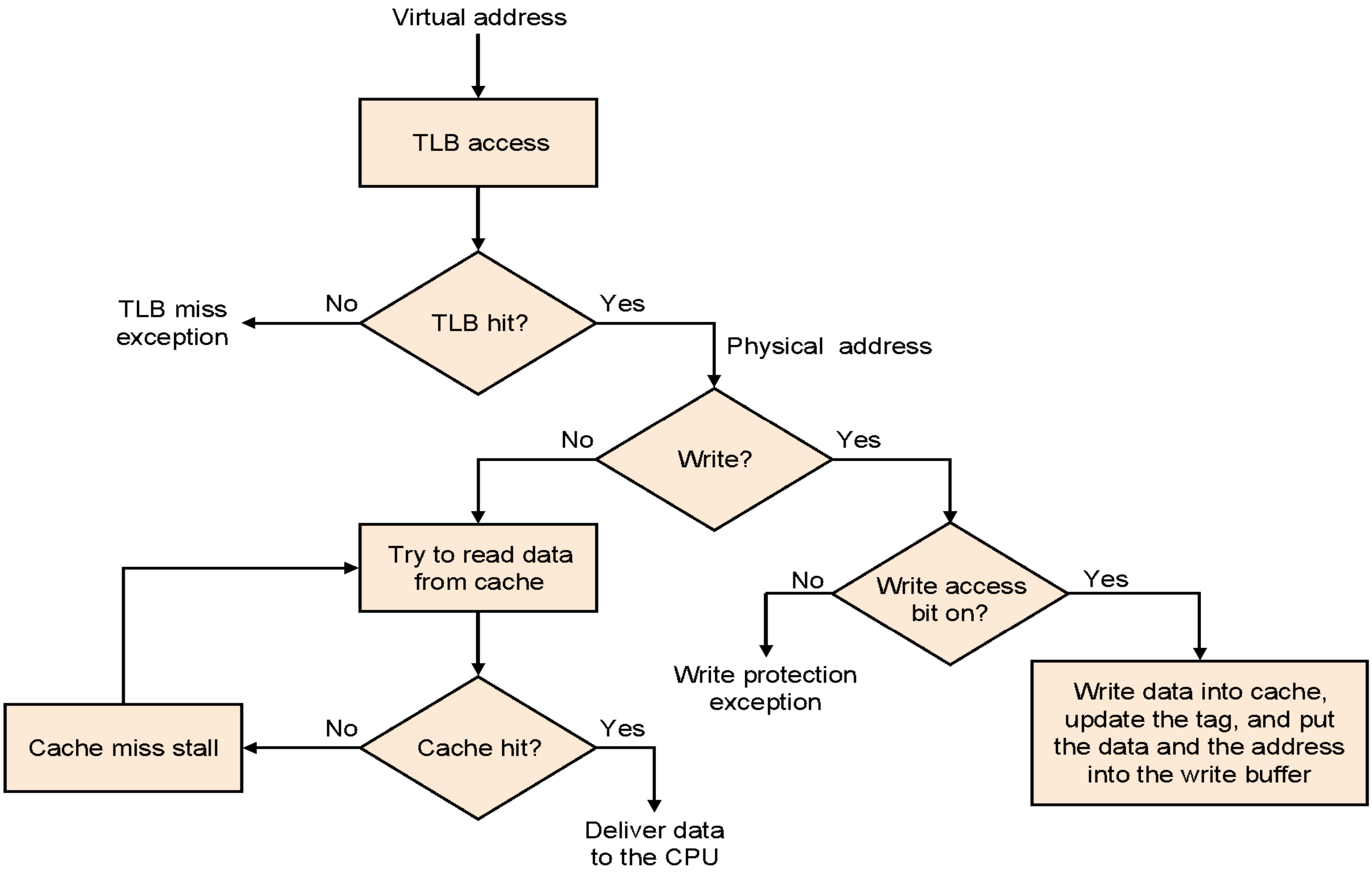
Paging in 64 bit Linux

Platform	Page Size	Address Bits Used	Paging Levels	Address Splitting
Alpha	8 KB	43	3	10+10+10+13
IA64	4 KB	39	3	9+9+9+12
PPC64	4 KB	41	3	10+10+9+12
sh64	4 KB	41	3	10+10+9+12
X86_64	4 KB	48	4	9+9+9+9+12

TLB look up and cache access



Cache, sequence of events for



The Arm Cortex-A8 Memory subsystem

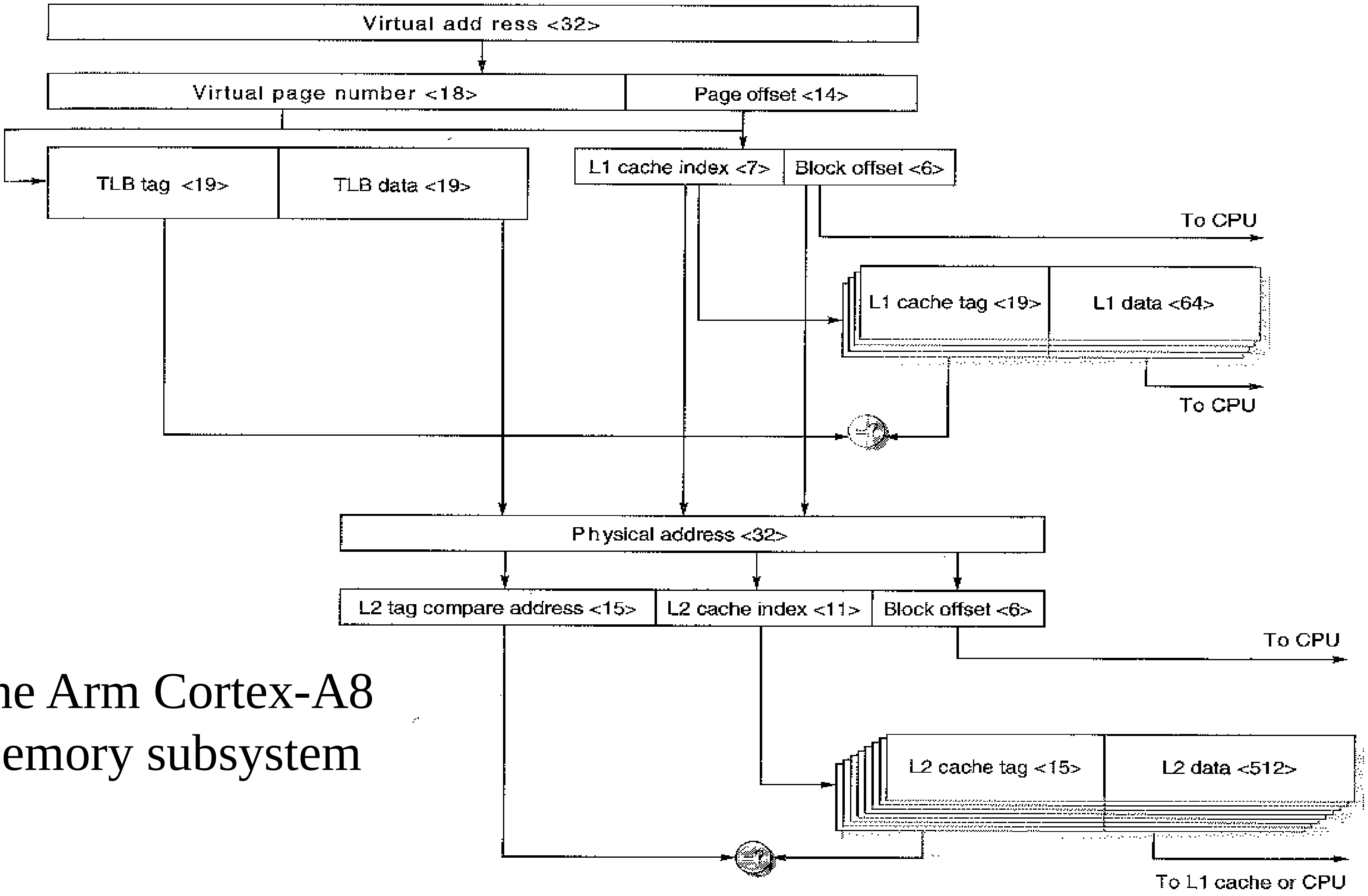


Figure 2.16 The virtual address, physical address, indexes, tags, and data blocks for the ARM Cortex-A8 data caches and data TLB. Since the instruction and data hierarchies are symmetric, we show only one. The TLB (instruction or data) is fully associative with 32 entries. The L1 cache is four-way set associative with 64-byte blocks and 32 KB capacity. The L2 cache is eight-way set associative with 64-byte blocks and 1 MB capacity. This figure doesn't show the valid bits and protection bits for the caches and TLB, nor the use of the way prediction bits that would dictate the predicted bank of the L1 cache.

Summary

- **SRAM** is fast but expensive and not very dense:
  - Good choice for providing the user FAST access time.
- **DRAM** is slow but cheap and dense:
  - Good choice for presenting the user with a BIG memory system
- **Flash Memory : Read** is fast but write is slow

- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon.
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon.
- **Cache**

  By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.
- **Direct Mapped , Set associative, full associative**
- **Virtual Memory**