

# Chapter 3

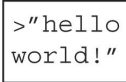


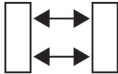
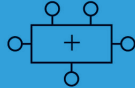

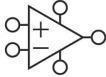


## ***Digital Design and Computer Architecture, 2<sup>nd</sup> Edition***

---

David Money Harris and Sarah L. Harris

# Chapter 3 :: Topics

- Introduction
- Latches and Flip-Flops
- Synchronous Logic Design
- Finite State Machines
- Timing of Sequential Logic
- Parallelism

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

# Introduction

- Outputs of sequential logic depend on current *and* prior input values – it has **memory**.
- Some definitions:
  - **State**: all the information about a circuit necessary to explain its future behavior
  - **Latches and flip-flops**: state elements that store one bit of state
  - **Synchronous sequential circuits**: combinational logic followed by a bank of flip-flops

# Sequential Circuits

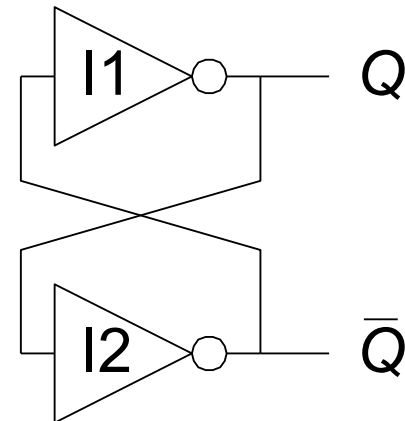
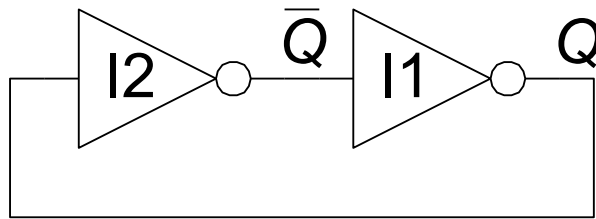
- Give sequence to events
- Have memory (short-term)
- Use feedback from output to input to store information

# State Elements

- The state of a circuit influences its future behavior
- State elements store state
  - Bistable circuit
  - SR Latch
  - D Latch
  - D Flip-flop

# Bistable Circuit

- Fundamental building block of other state elements
- Two outputs:  $Q$ ,  $\bar{Q}$
- No inputs



# Bistable Circuit Analysis

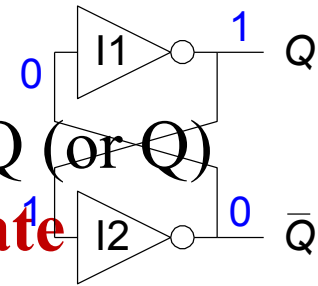
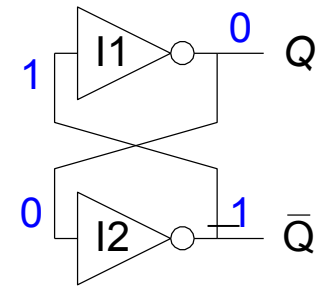
- Consider the two possible cases:

-  $Q = 0$ :

then  $Q = 1$ ,  $Q = 0$  (consistent)

-  $Q = 1$ :

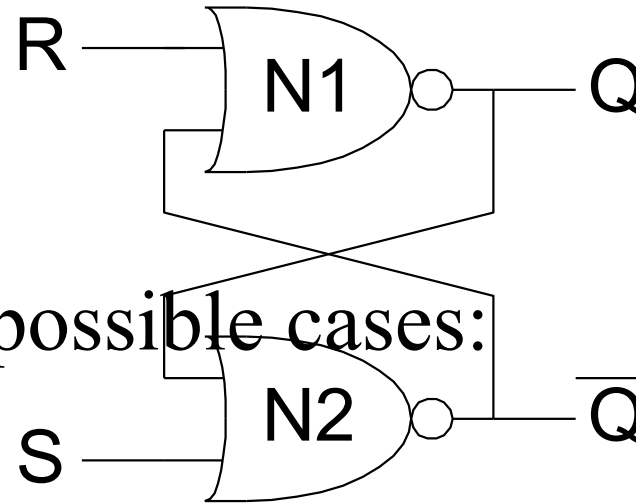
then  $Q = 0$ ,  $Q = 1$  (consistent)



- Stores 1 bit of state in the state variable,  $Q$  (or  $\bar{Q}$ )
- But there are **no inputs to control the state**

# SR (Set/Reset) Latch

- SR Latch



- Consider the four possible cases:

- $S = 1, R = 0$

- $S = 0, R = 1$

- $S = 0, R = 0$

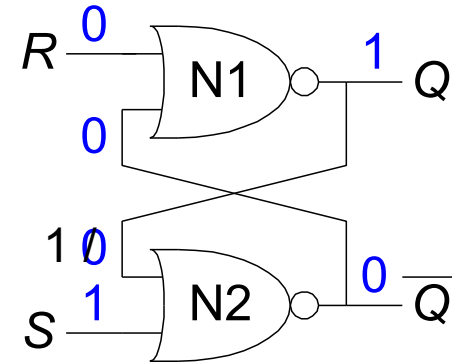
- $S = 1, R = 1$



# SR Latch Analysis

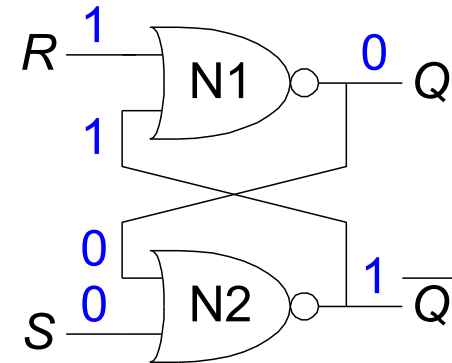
**-S = 1, R = 0:**

then  $Q = 1$  and  $\overline{Q} = 0$



**-S = 0, R = 1:**

then  $Q = 0$  and  $\overline{Q} = 1$



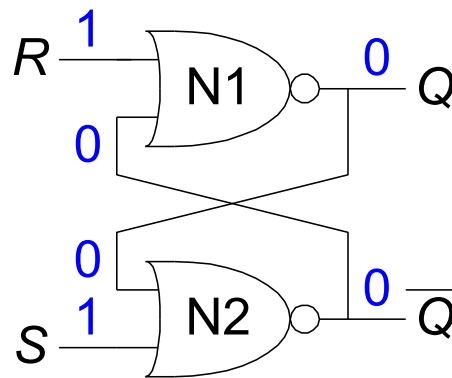
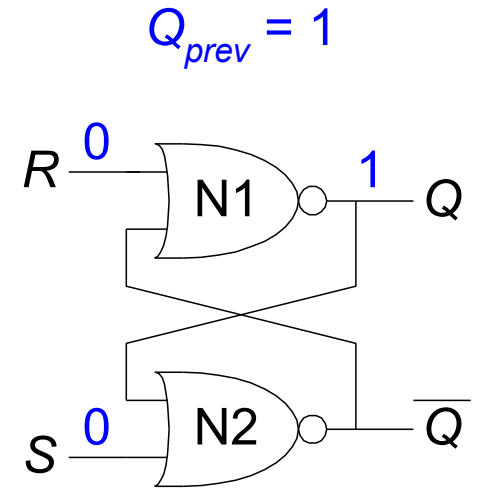
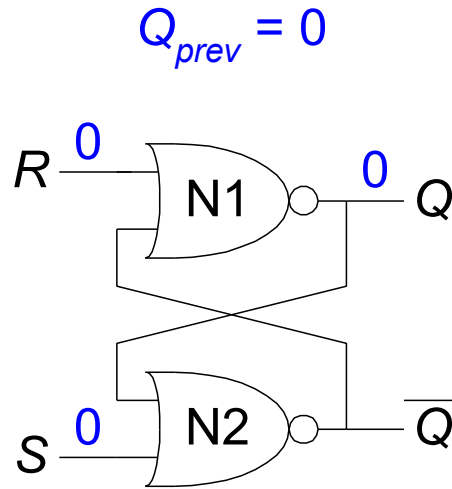
# SR Latch Analysis

$-S = 0, R = 0:$

then  $Q = Q_{prev}$

$-S = 1, R = 1:$

then  $Q = 0, \bar{Q} = 0$



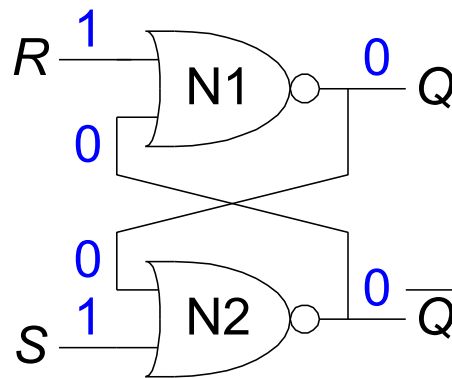
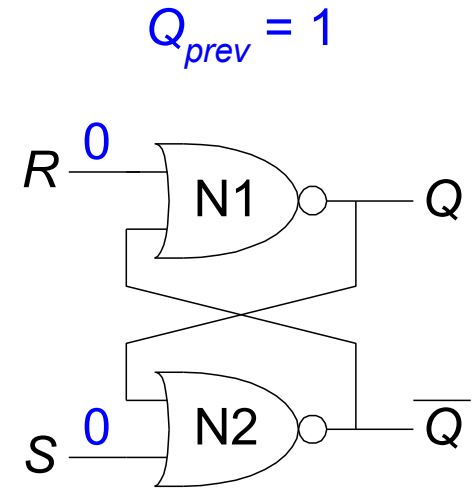
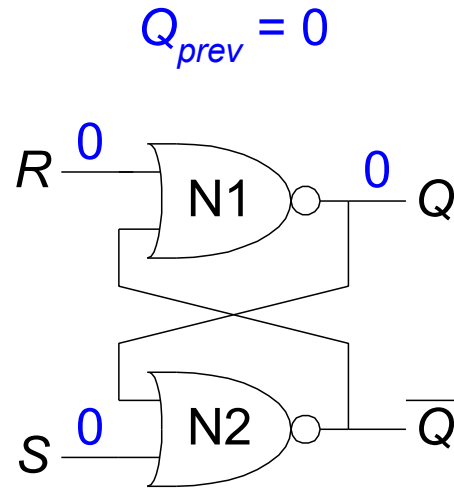
# SR Latch Analysis

$-S = 0, R = 0:$

then  $Q = Q_{prev}$   
**-Memory!**

$-S = 1, R = 1:$

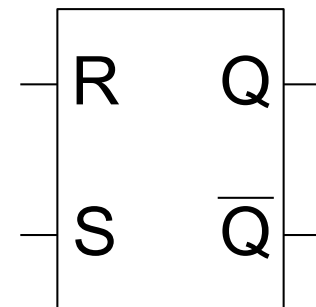
then  $Q = 0, \bar{Q} = 0$   
**-Invalid State**  
 $Q \neq \text{NOT } Q$



# SR Latch Symbol

- SR stands for Set/Reset Latch
  - Stores one bit of state ( $Q$ )
- Control what value is being stored with  $S$ ,  $R$  inputs
  - Set:** Make the output 1  
( $S = 1, R = 0, Q = \mathbf{1}$ )
  - Reset:** Make the output 0  
( $S = 0, R = 1, Q = \mathbf{0}$ )

SR Latch  
Symbol

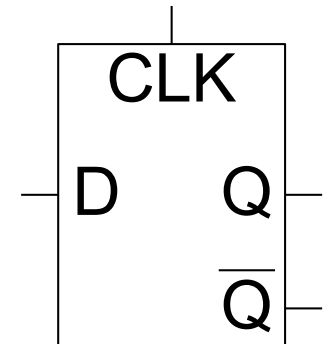


- **Must do something to avoid invalid state (when  $S = R = 1$ )**

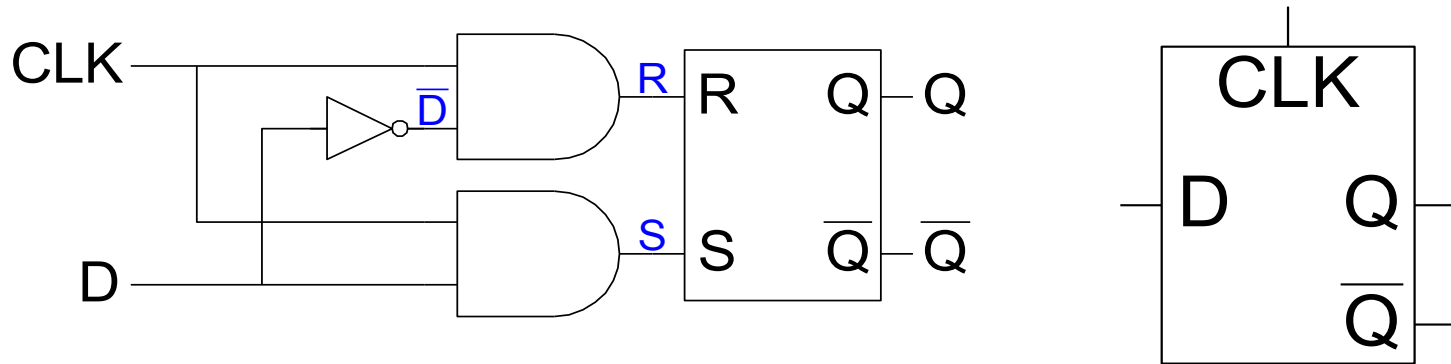
# D Latch

- Two inputs:  $CLK$ ,  $D$ 
  - $CLK$ : controls *when* the output changes
  - $D$  (the data input): controls *what* the output changes to
- Function
  - When  $CLK = 1$ ,  
 $D$  passes through to  $Q$  (*transparent*)
  - When  $CLK = 0$ ,  
 $Q$  holds its previous value (*opaque*)
- Avoids invalid case when  
 $Q \neq \text{NOT } Q$

D Latch  
Symbol

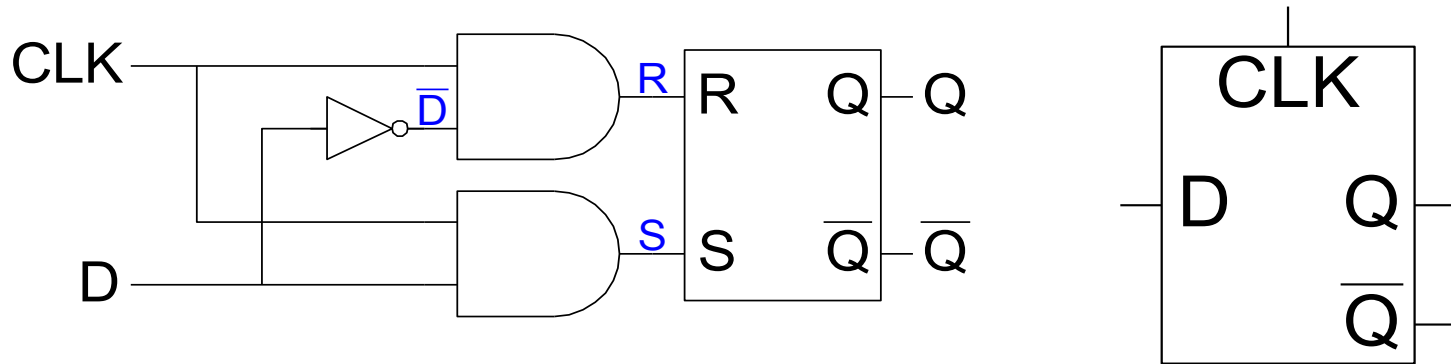


# D Latch Internal Circuit



$CLK$	$D$	$\overline{D}$	$S$	$R$	$Q$	$\overline{Q}$
0	X					
1	0					
1	1					

# D Latch Internal Circuit

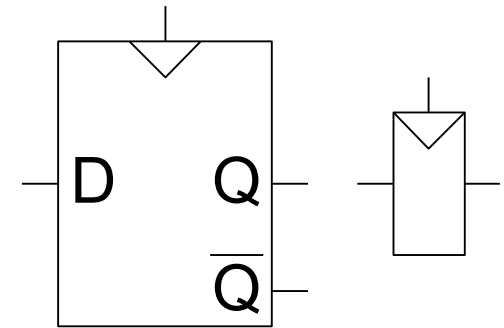


$CLK$	$D$	$\overline{D}$	$S$	$R$	$Q$	$\overline{Q}$
0	X	$\overline{X}$	0	0	$Q_{prev}$	$\overline{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

# D Flip-Flop

- **Inputs:**  $CLK$ ,  $D$
- **Function**
  - Samples  $D$  on rising edge of  $CLK$ 
    - When  $CLK$  rises from 0 to 1,  $D$  passes through to  $Q$
    - Otherwise,  $Q$  holds its previous value
  - $Q$  changes only on rising edge of  $CLK$
- Called *edge-triggered*
- Activated on the clock edge

D Flip-Flop Symbols





# D Flip-Flop Internal Circuit

- Two back-to-back latches (L1 and L2) controlled by complementary clocks

- When  $CLK = 0$

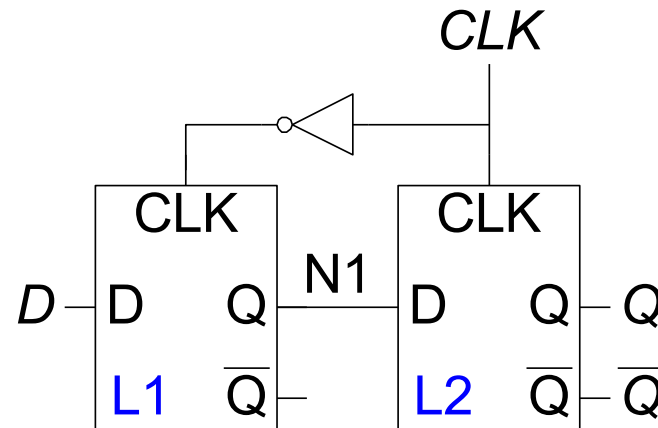
- L1 is transparent
- L2 is opaque
- $D$  passes through to N1

- When  $CLK = 1$

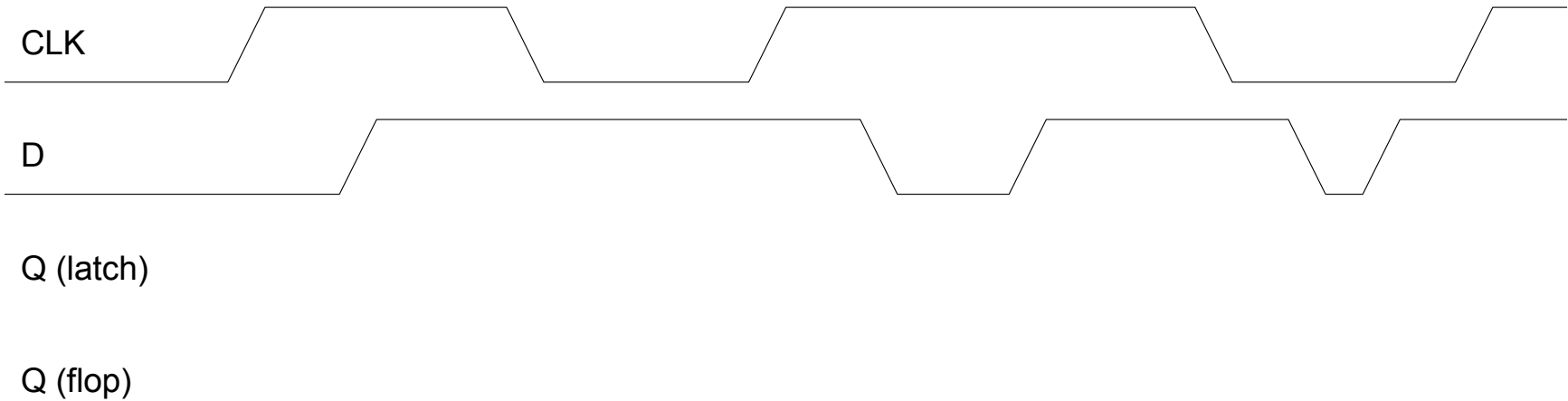
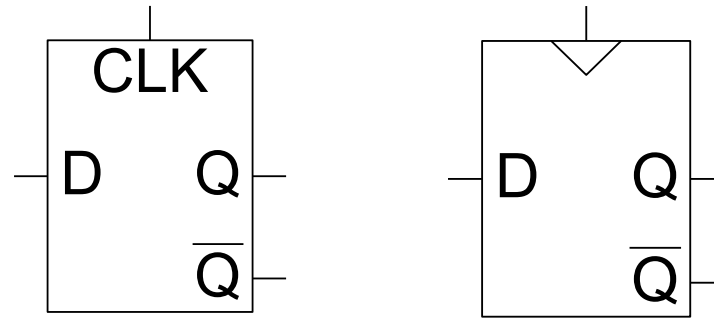
- L2 is transparent
- L1 is opaque
- N1 passes through to  $Q$

- Thus, on the edge of the clock (when  $CLK$  rises from 0 to 1)

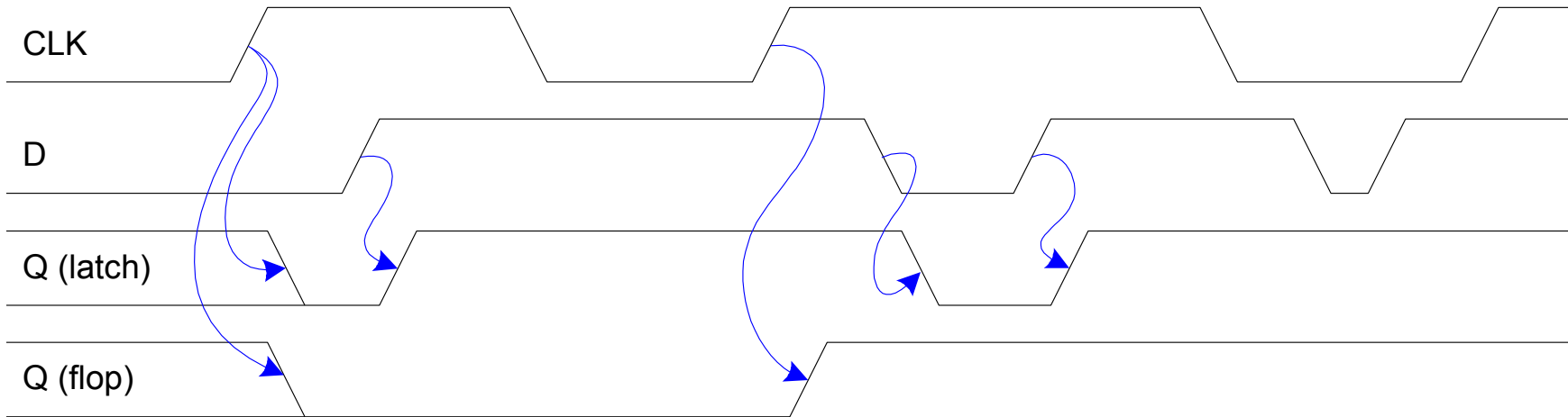
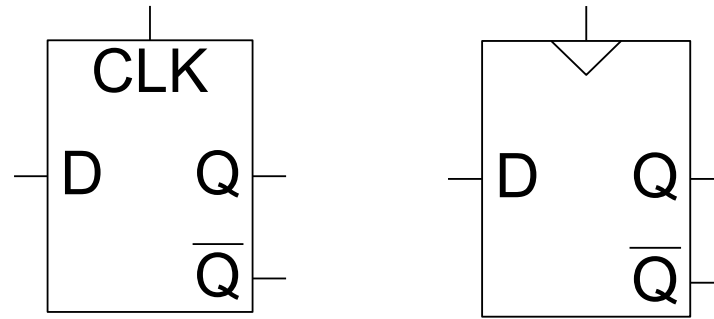
- $D$  passes through to  $Q$



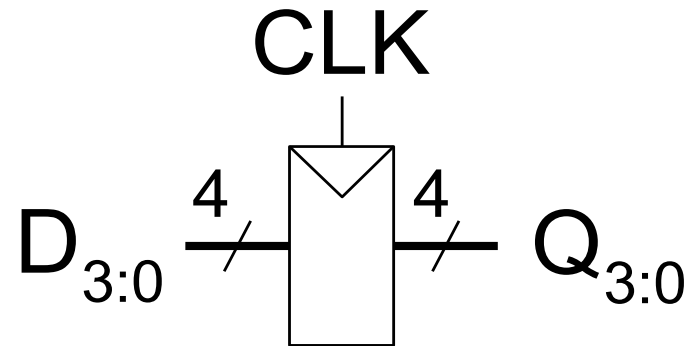
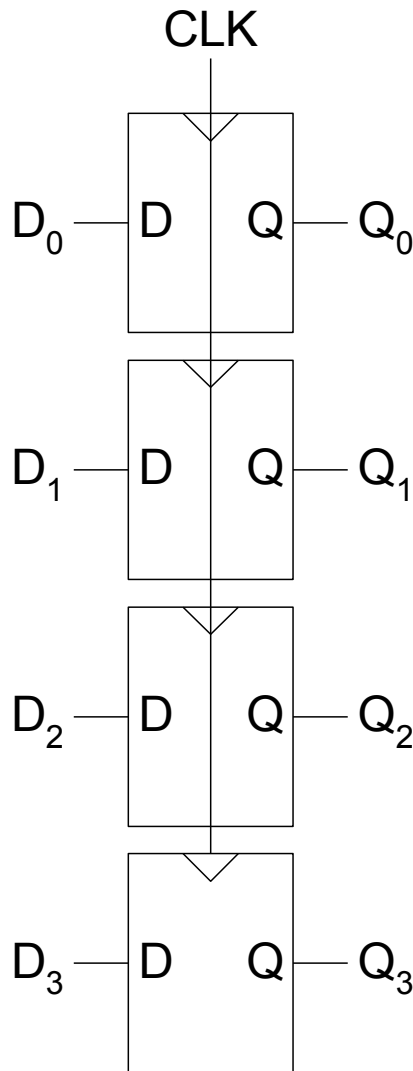
# D Latch vs. D Flip-Flop



# D Latch vs. D Flip-Flop

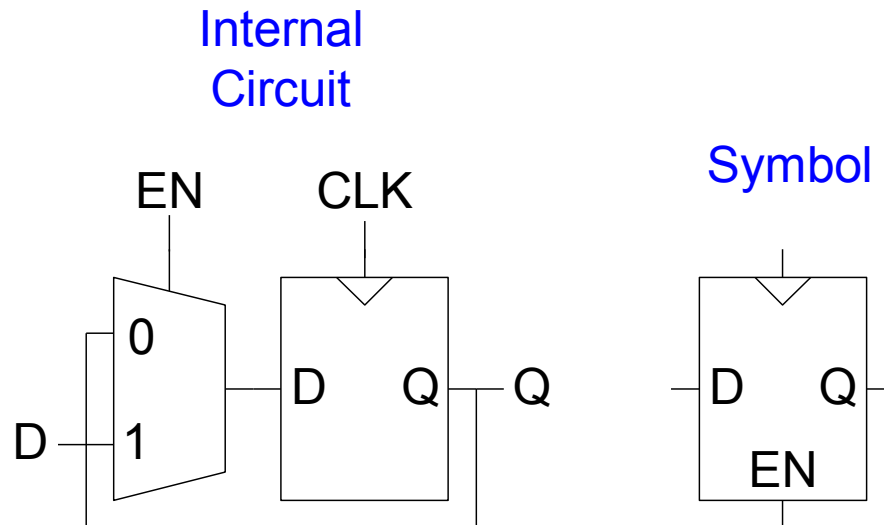


# Registers



# Enabled Flip-Flops

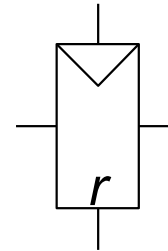
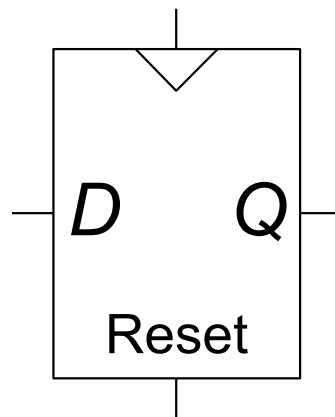
- **Inputs:**  $CLK$ ,  $D$ ,  $EN$ 
  - The enable input ( $EN$ ) controls when new data ( $D$ ) is stored
- **Function**
  - $EN = 1$ :  $D$  passes through to  $Q$  on the clock edge
  - $EN = 0$ : the flip-flop retains its previous state



# Resettable Flip-Flops

- **Inputs:**  $CLK$ ,  $D$ ,  $Reset$
- **Function:**
  - $Reset = 1$ :  $Q$  is forced to 0
  - $Reset = 0$ : flip-flop behaves as ordinary D flip-flop

## Symbols

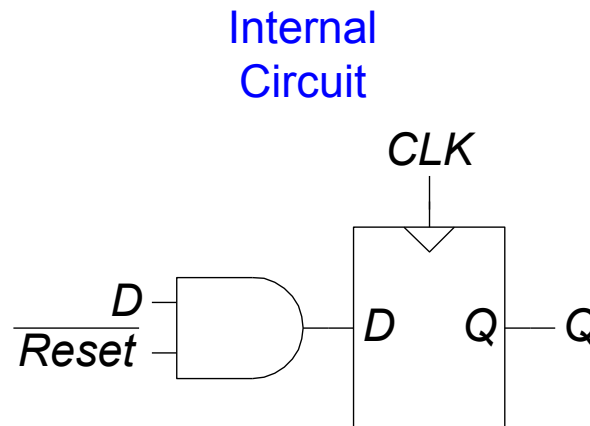


# Resettable Flip-Flops

- Two types:
  - **Synchronous:** resets at the clock edge only
  - **Asynchronous:** resets immediately when  $Reset = 1$
- Asynchronously resettable flip-flop requires changing the internal circuitry of the flip-flop
- Synchronously resettable flip-flop?

# Resettable Flip-Flops

- Two types:
  - **Synchronous:** resets at the clock edge only
  - **Asynchronous:** resets immediately when  $Reset = 1$
- Asynchronously resettable flip-flop requires changing the internal circuitry of the flip-flop
- Synchronously resettable flip-flop?

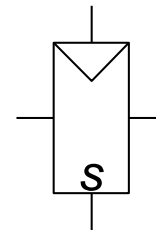
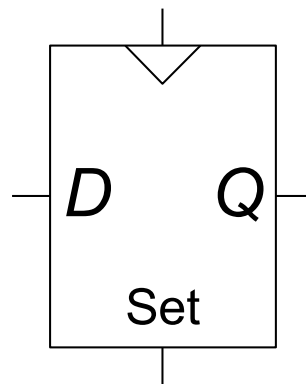




# Settable Flip-Flops



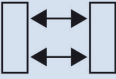
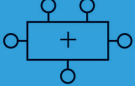
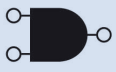
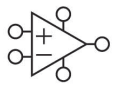


- **Inputs:**  $CLK$ ,  $D$ ,  $Set$
- **Function:**
  - $Set = 1$ :  $Q$  is set to 1
  - $Set = 0$ : the flip-flop behaves as ordinary D flip-flop

## Symbols



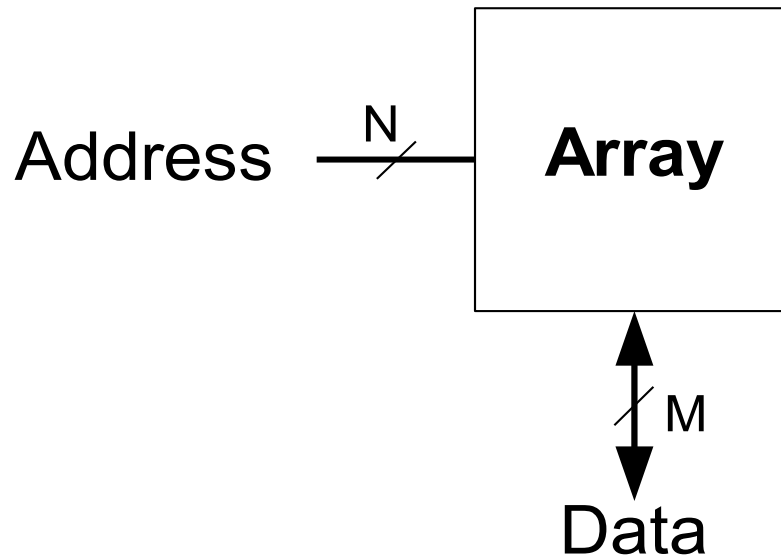
# Chapter 5 :: Topics

- Introduction (done)
- Arithmetic Circuits (done)
- Number Systems (done)
- Sequential Building Blocks (done)
- Memory Arrays (now)
- Logic Arrays (now)

Application Software	<code>&gt;"hello world!"</code>
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

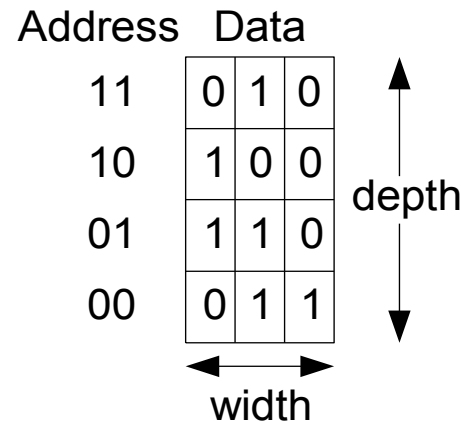
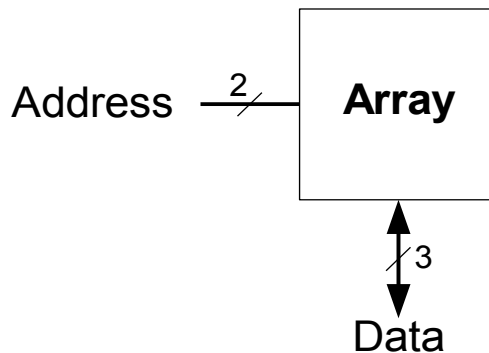
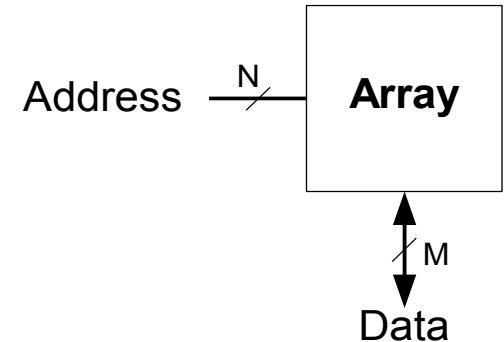
# Memory Arrays

- Efficiently store large amounts of data
- 3 common types:
  - Dynamic random access memory (DRAM)
  - Static random access memory (SRAM)
  - Read only memory (ROM)
- $M$ -bit data value read/ written at each unique  $N$ -bit address



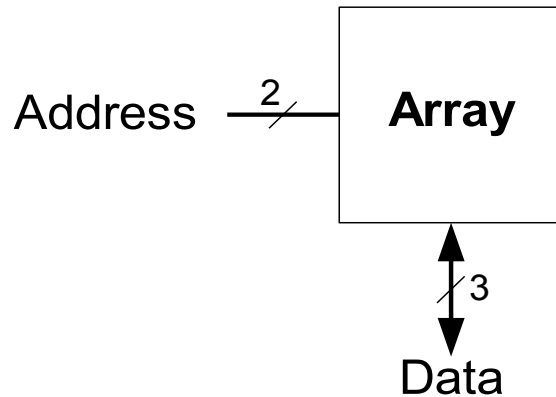
# Memory Arrays

- 2-dimensional array of bit cells
- Each bit cell stores one bit
- $N$  address bits and  $M$  data bits:
  - $2^N$  rows and  $M$  columns
  - **Depth:** number of rows (number of words)
  - **Width:** number of columns (size of word)
  - **Array size:** depth  $\times$  width =  $2^N \times M$



# Memory Array Example

- $2^2 \times 3$ -bit array
- Number of words: 4
- Word size: 3-bits
- For example, the 3-bit word stored at address 10 is 100

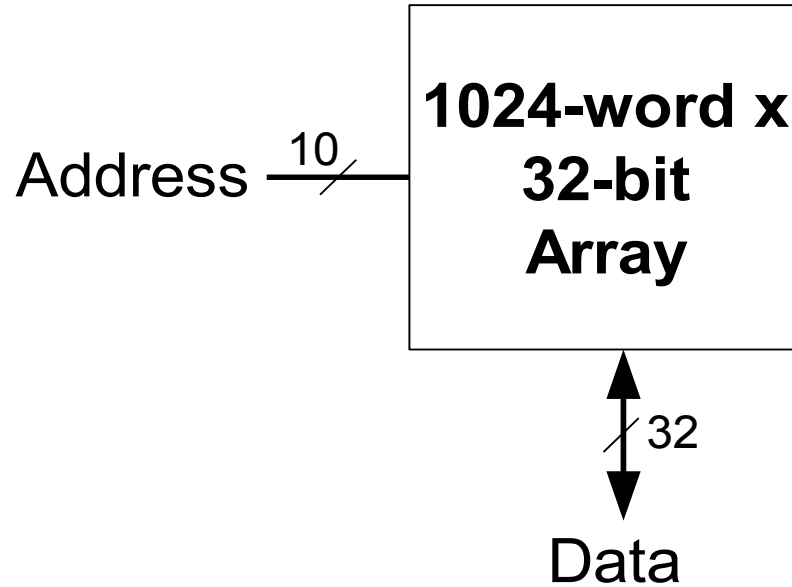


Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

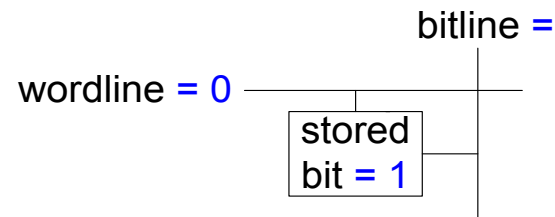
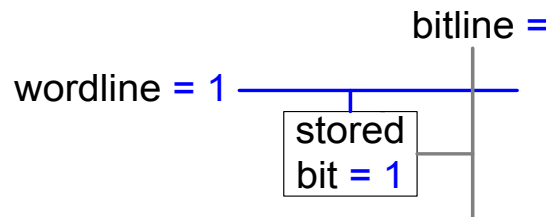
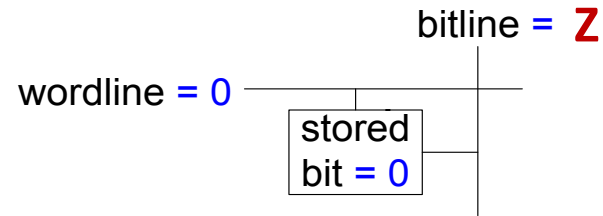
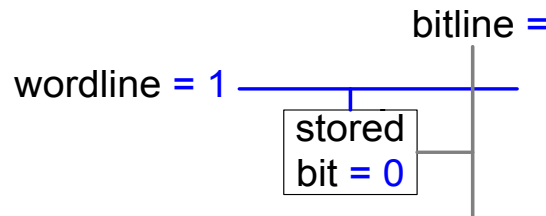
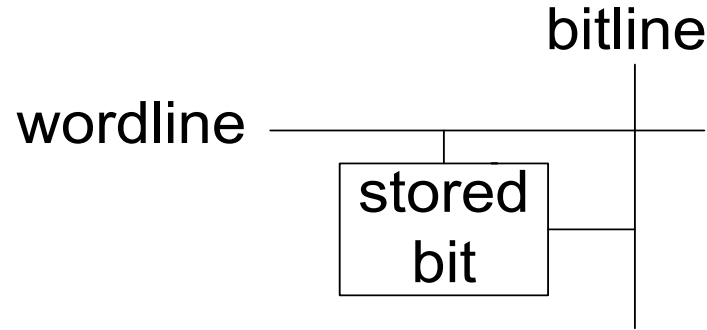
width

depth

# Memory Arrays



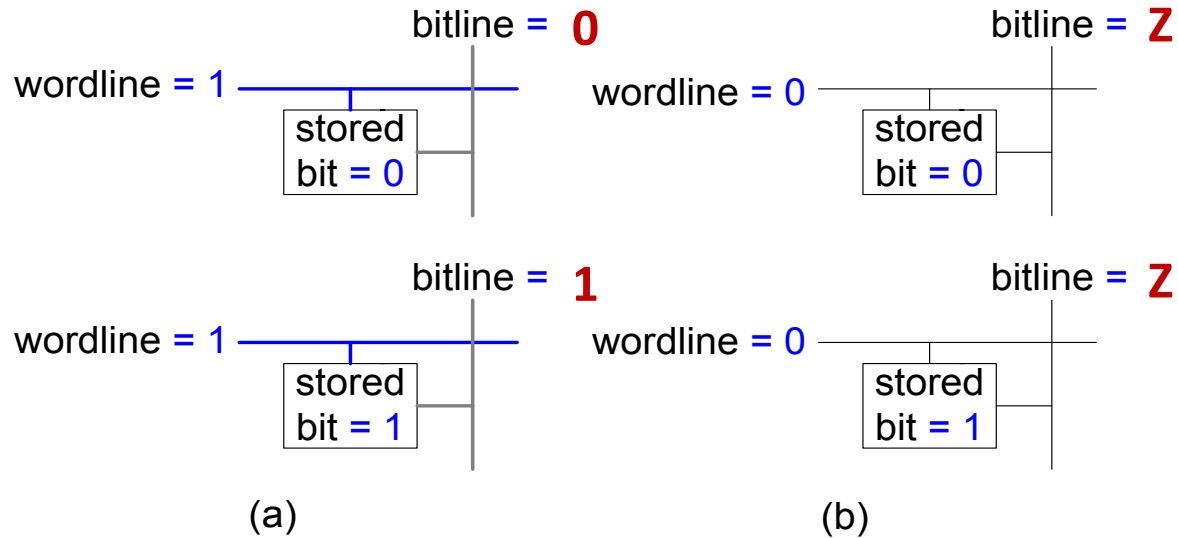
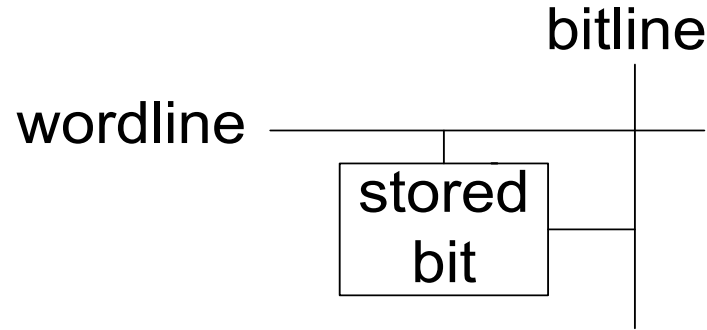
# Memory Array Bit Cells



(a)

(b)

# Memory Array Bit Cells

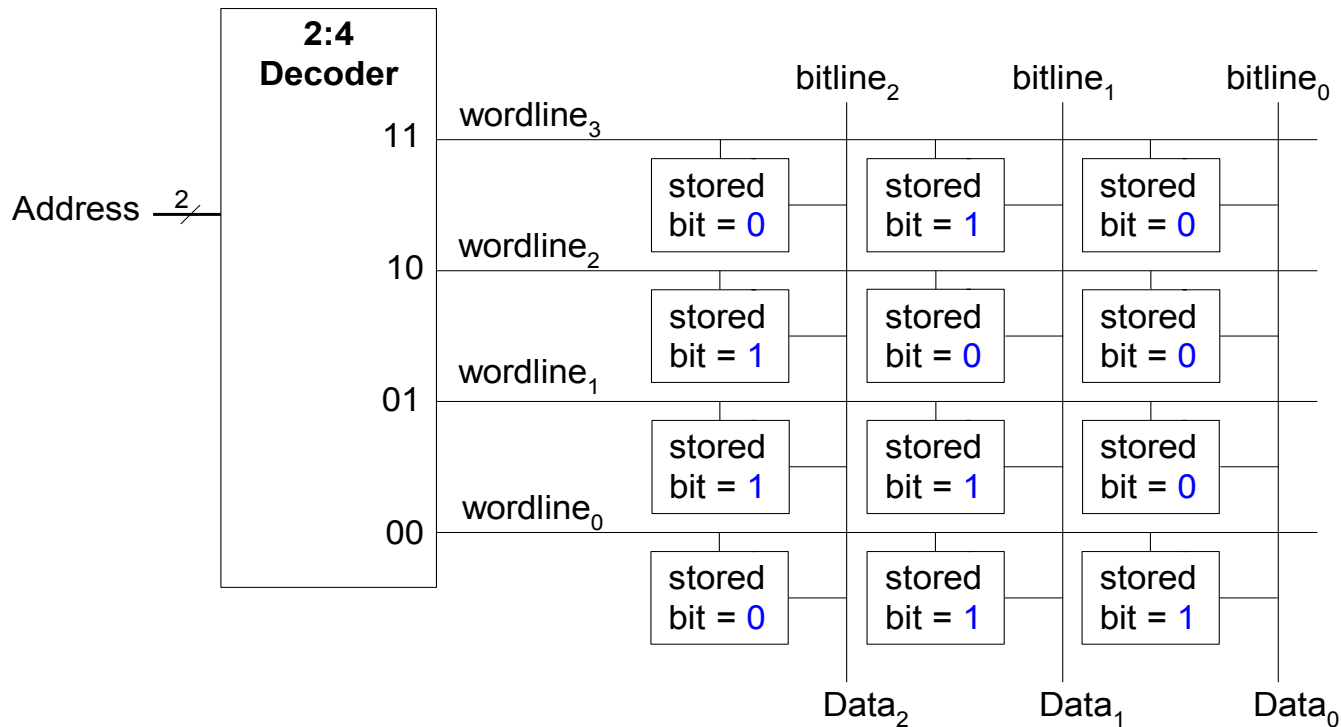




# Memory Array

- **Wordline:**

- like an enable
- single row in memory array read/written
- corresponds to unique address
- only one wordline HIGH at once



# Types of Memory

- Random access memory (RAM): **volatile**
- Read only memory (ROM): **nonvolatile**

# RAM: Random Access Memory

- **Volatile:** loses its data when power off
- Read and written quickly
- Main memory in your computer is RAM (DRAM)

Historically called *random* access memory because any data word accessed as easily as any other (in contrast to sequential access memories such as a tape recorder)

# ROM: Read Only Memory

- **Nonvolatile:** retains data when power off
- Read quickly, but writing is impossible or slow
- Flash memory in cameras, thumb drives, and digital cameras are all ROMs

Historically called *read only* memory because ROMs were written at manufacturing time or by burning fuses. Once ROM was configured, it could not be written again. This is no longer the case for Flash memory and other types of ROMs.

# Types of RAM

- **DRAM** (Dynamic random access memory)
- **SRAM** (Static random access memory)
- Differ in how they store data:
  - DRAM uses a capacitor
  - SRAM uses cross-coupled inverters

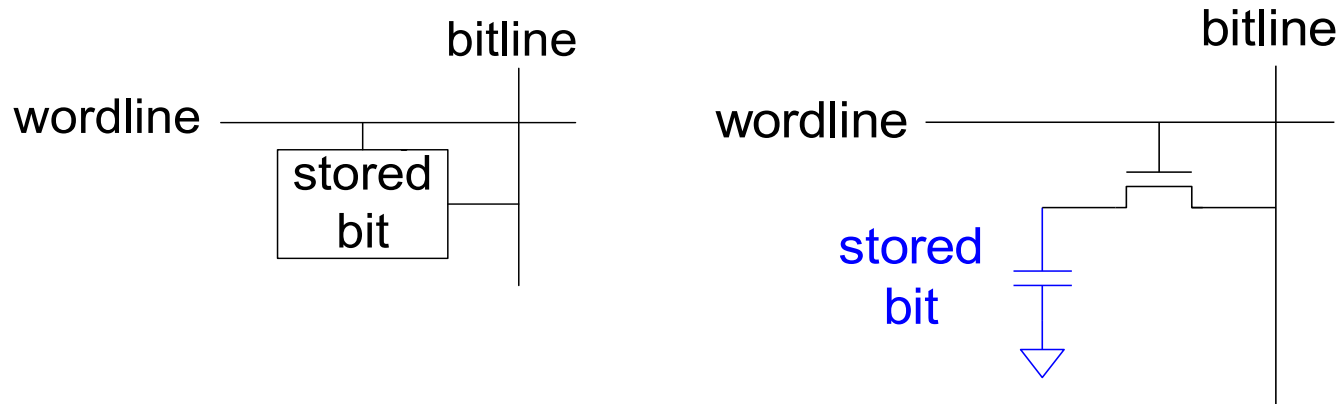
# Robert Dennard, 1932 -

- Invented DRAM in 1966 at IBM
- Others were skeptical that the idea would work
- By the mid-1970's DRAM in virtually all computers

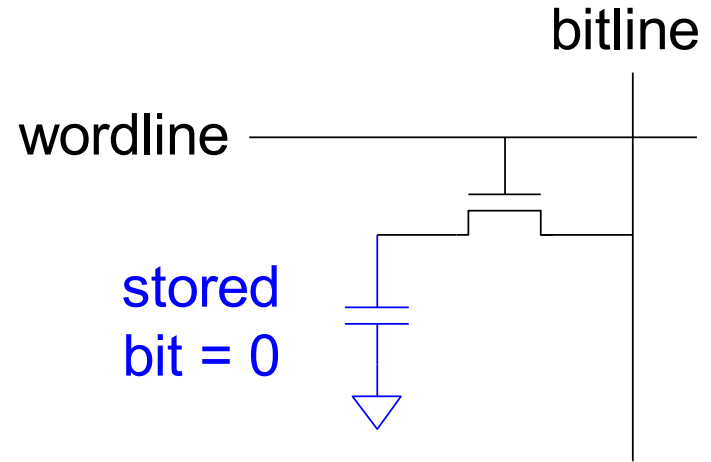
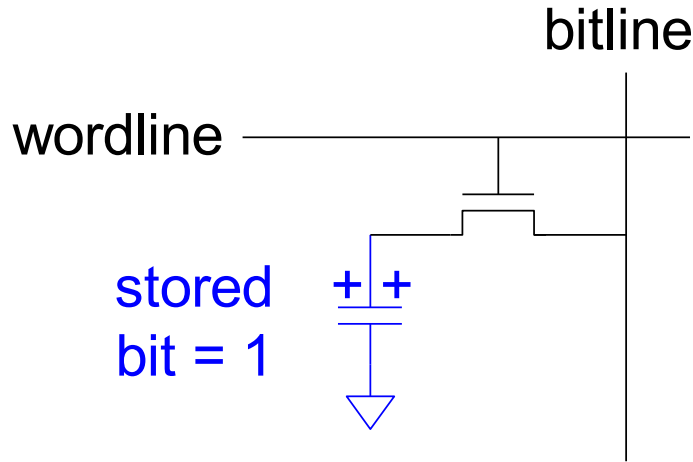


# DRAM

- Data bits stored on capacitor
- *Dynamic* because the value needs to be refreshed (rewritten) periodically and after read:
  - Charge leakage from the capacitor degrades the value
  - Reading destroys the stored value

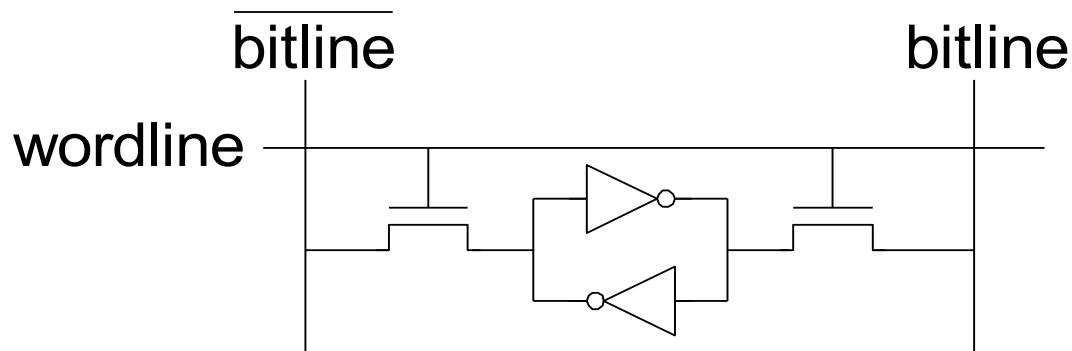
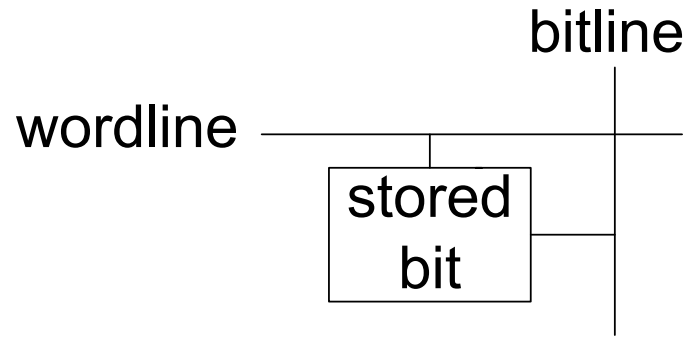


# DRAM

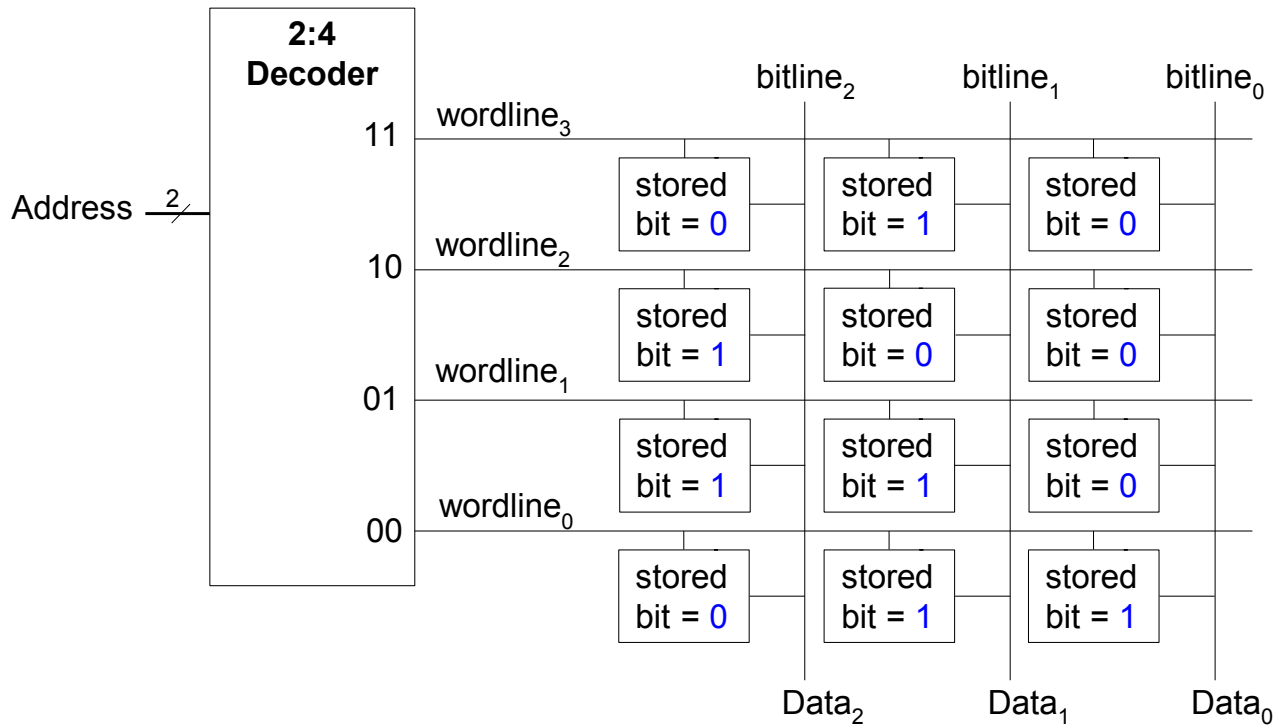




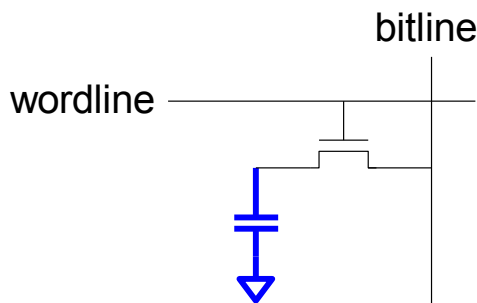
# SRAM



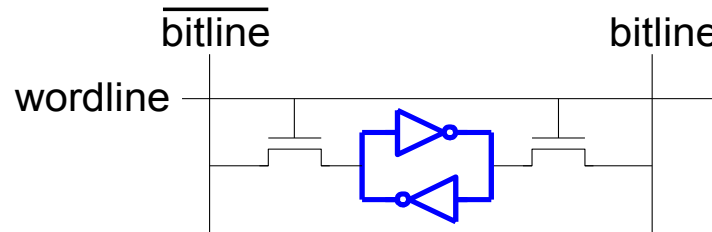
# Memory Arrays Review



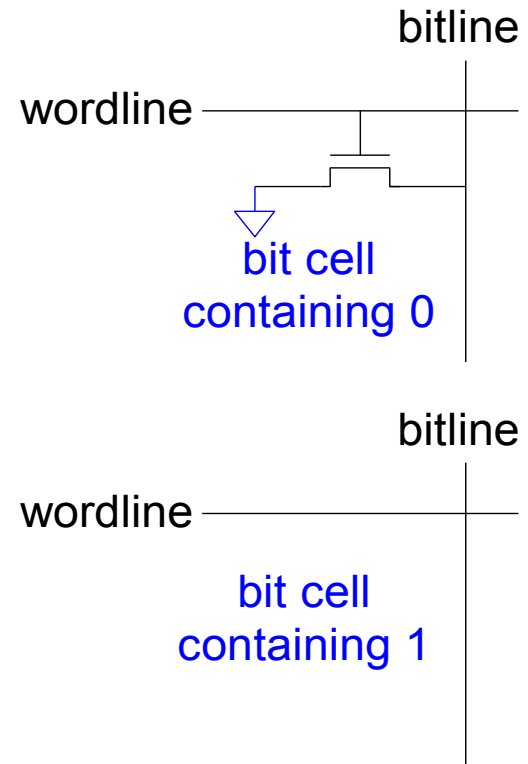
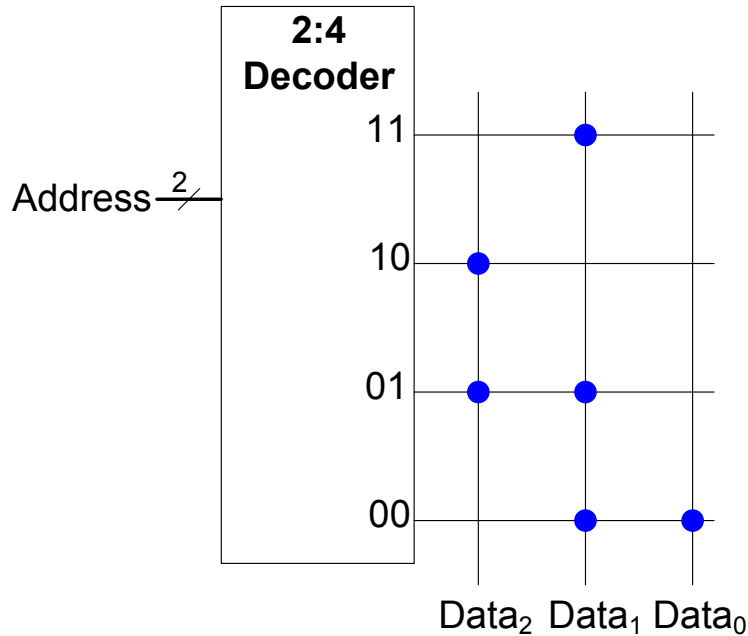
DRAM bit cell:



SRAM bit cell:



# ROM: Dot Notation

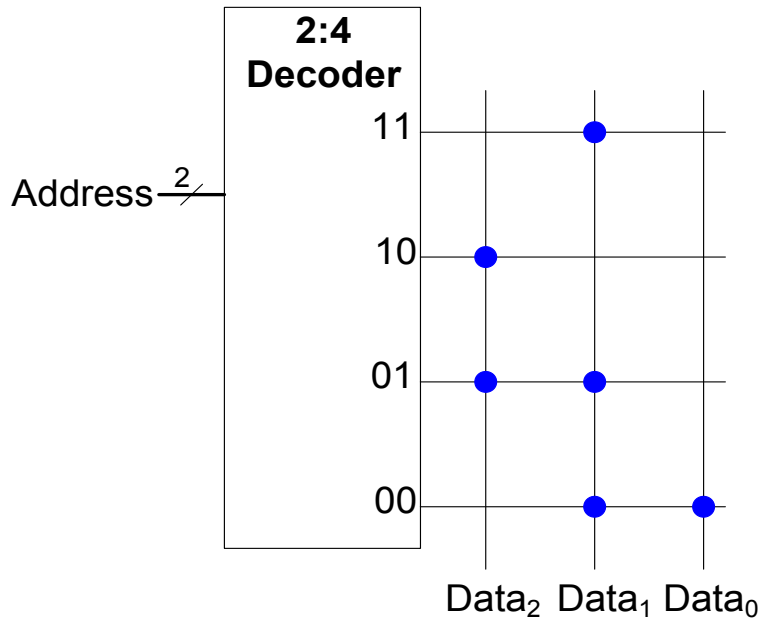


# Fujio Masuoka, 1944 -

- Developed memories and high speed circuits at Toshiba, 1971-1994
- Invented Flash memory as an unauthorized project pursued during nights and weekends in the late 1970's
- The process of erasing the memory reminded him of the flash of a camera
- Toshiba slow to commercialize the idea; Intel was first to market in 1988
- Flash has grown into a \$25 billion per year market



# ROM Storage

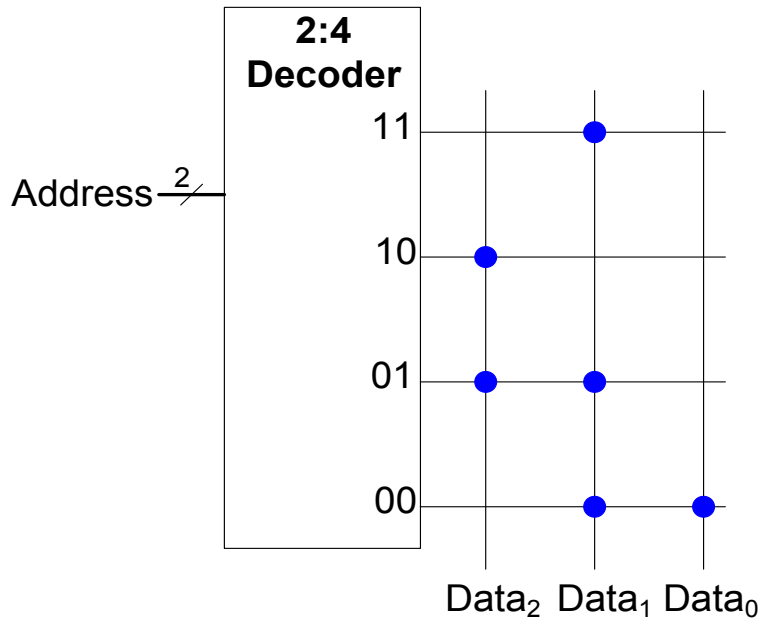


Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

width

depth

# ROM Logic



$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = A_1 + A_0$$

$$Data_0 = \underline{A_1 A_0}$$



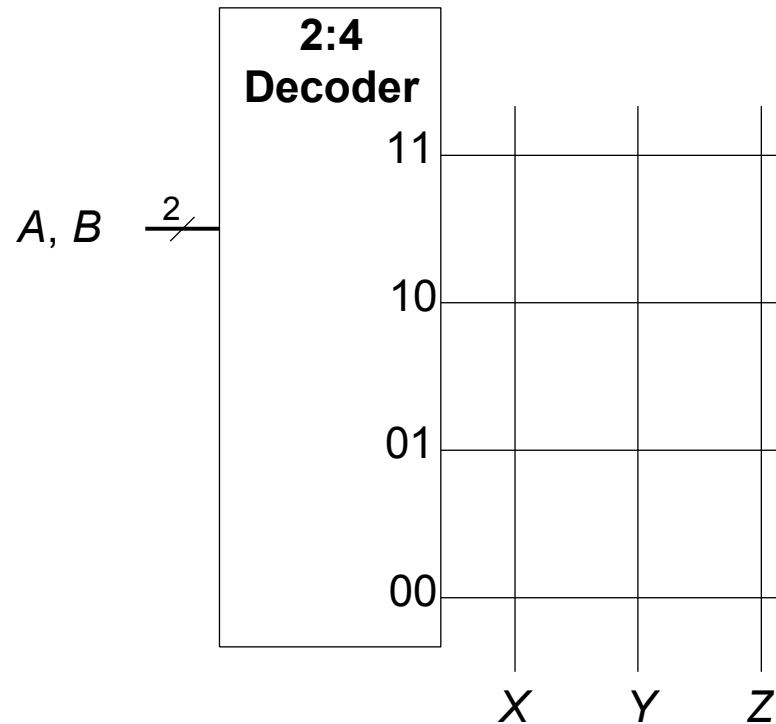
# Example: Logic with ROMs

Implement the following logic functions using a  $2^2 \times 3$ -bit ROM:

$$-X = AB$$

$$-Y = A + B$$

$$-Z = \overline{A} B$$



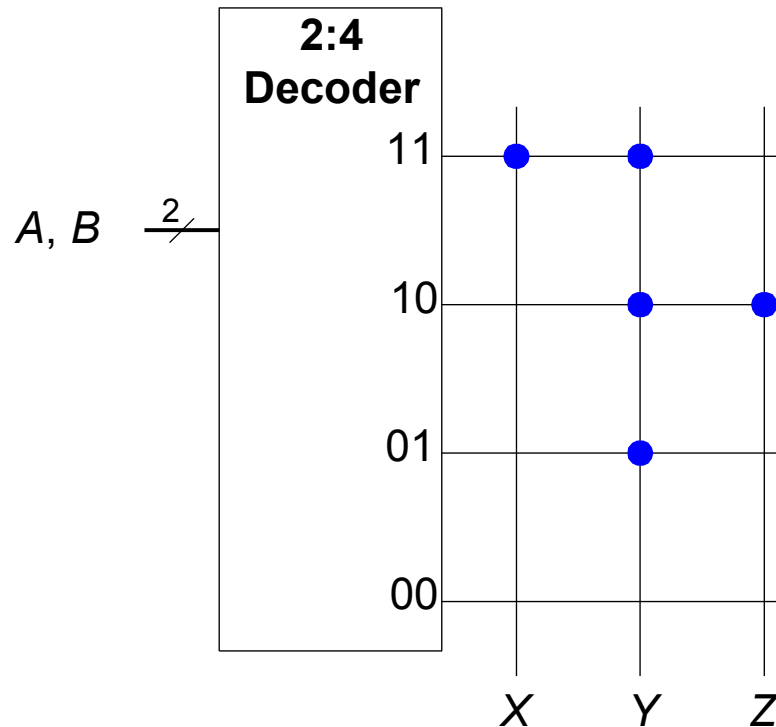
# Example: Logic with ROMs

Implement the following logic functions using a  $2^2 \times 3$ -bit ROM:

$$-X = AB$$

$$-Y = A + B$$

$$-Z = \overline{A} B$$







# Logic with Memory Arrays

Implement the following logic functions using a  $2^2 \times 3$ -bit memory array:

$$-X = AB$$

$$-Y = A + B$$

$$-Z = \overline{A}B$$

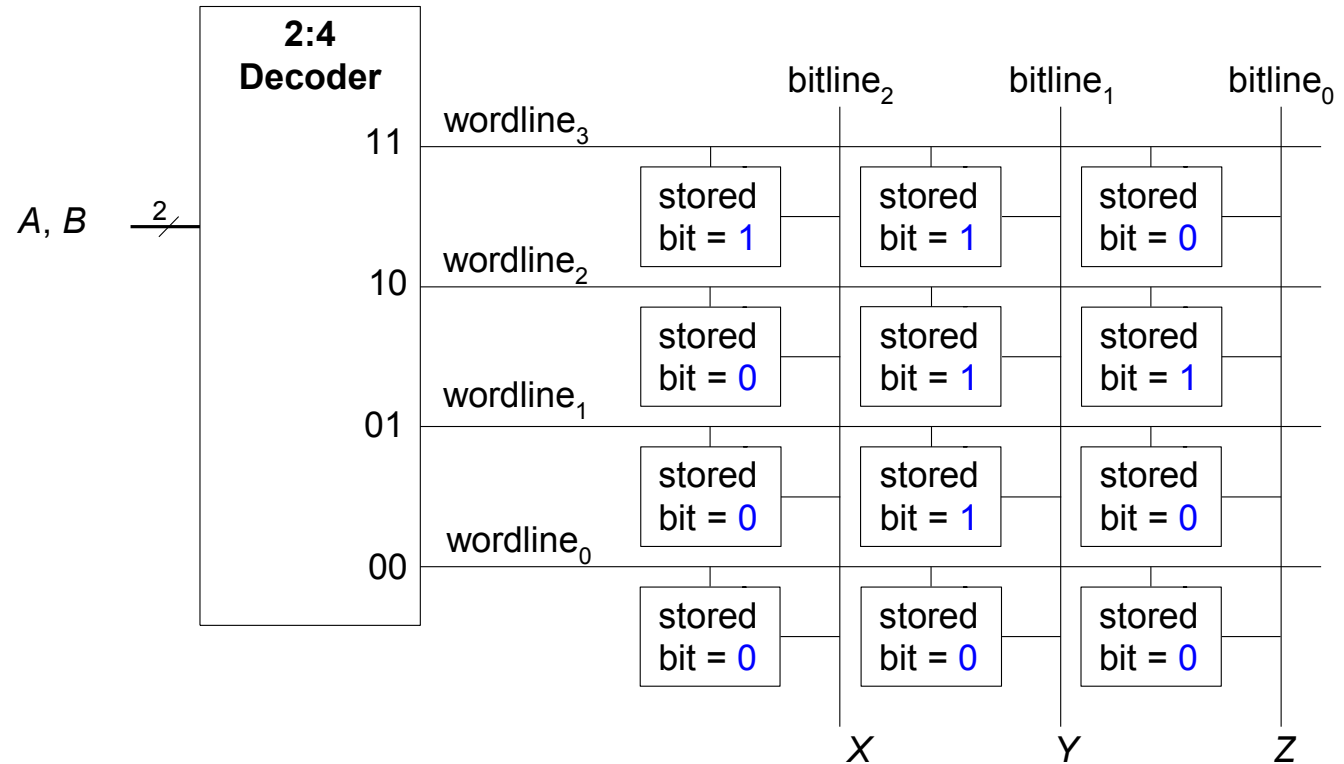
# Logic with Memory Arrays

Implement the following logic functions using a  $2^2 \times 3$ -bit memory array:

$$-X = AB$$

$$-Y = A + B$$

$$-Z = \overline{A} B$$



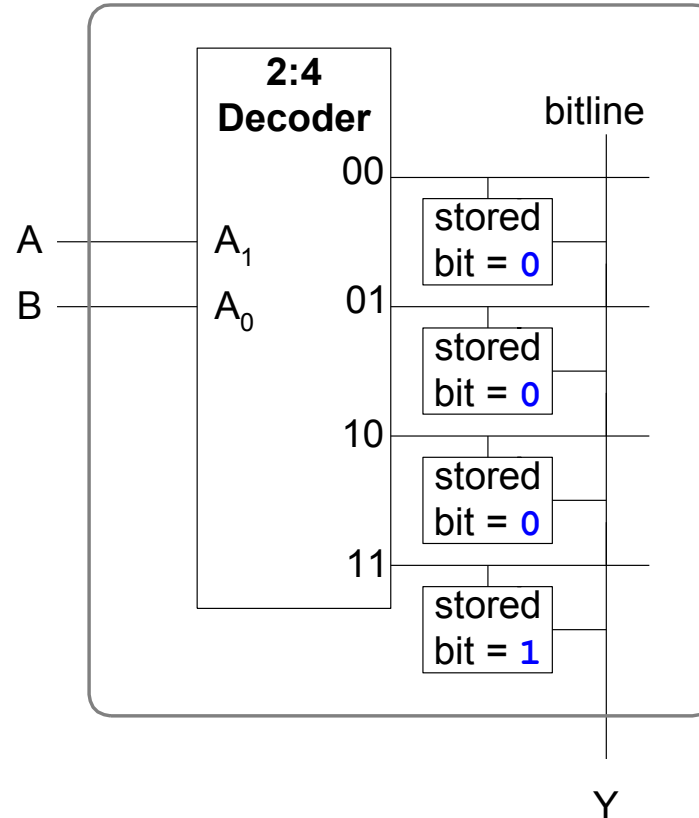
# Logic with Memory Arrays

Called *lookup tables* (LUTs): look up output at each input combination (address)

**Truth Table**

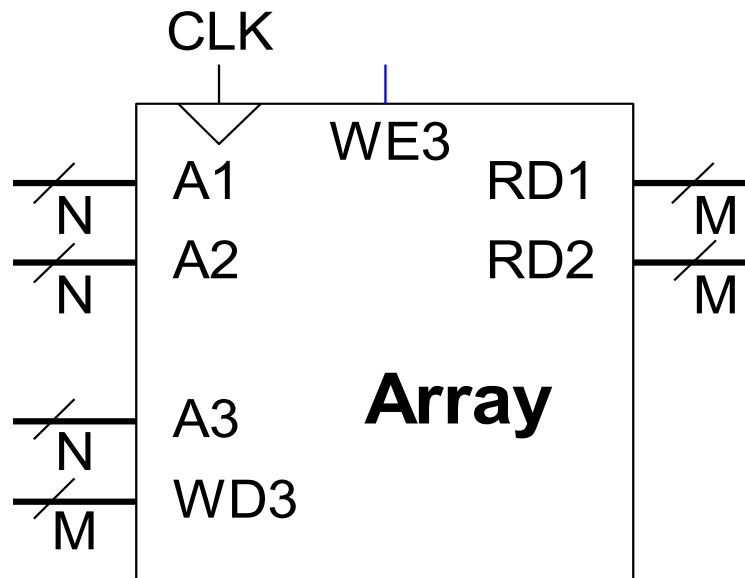
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

4-word x 1-bit Array



# Multi-ported Memories

- **Port:** address/data pair
- 3-ported memory
  - 2 read ports (A1/RD1, A2/RD2)
  - 1 write port (A3/WD3, WE3 enables writing)
- **Register file:** small multi-ported memory



# SystemVerilog Memory Arrays

```
// 256 x 3 memory module with one read/write port
module dmem( input  logic      clk, we,
             input  logic[7:0]  a
             input  logic [2:0] wd,
             output logic [2:0] rd);

    logic [2:0] RAM[255:0];

    assign rd = RAM[a];

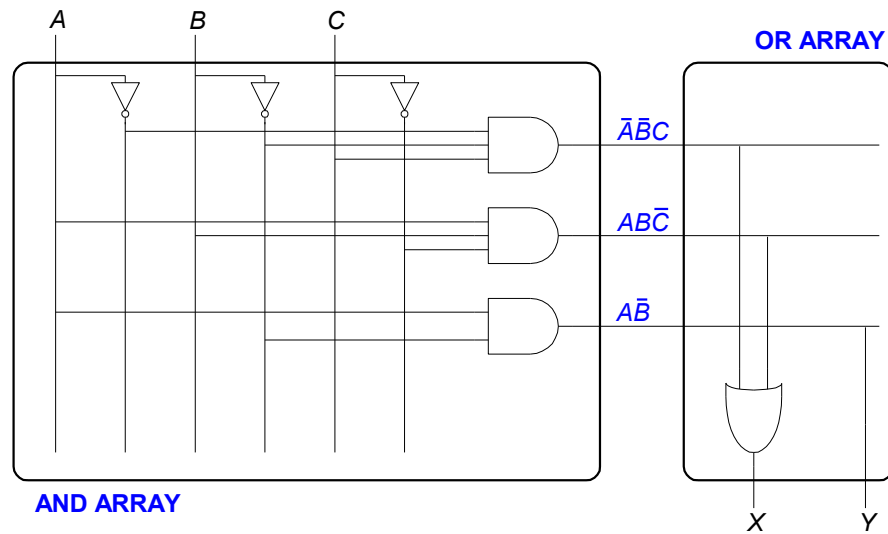
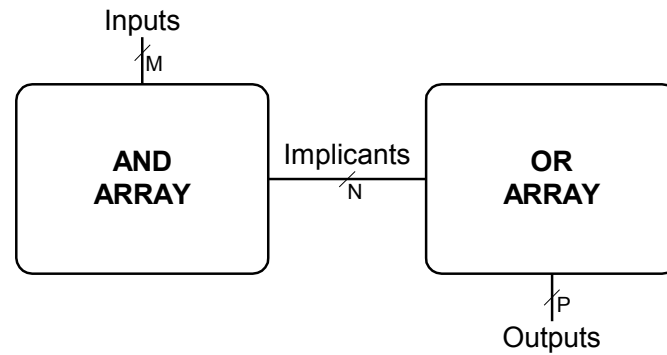
    always @(posedge clk)
        if (we)
            RAM[a] <= wd;
endmodule
```

# Logic Arrays

- **PLAs** (Programmable logic arrays)
  - AND array followed by OR array
  - Combinational logic only
  - Fixed internal connections
- **FPGAs** (Field programmable gate arrays)
  - Array of Logic Elements (LEs)
  - Combinational and sequential logic
  - Programmable internal connections

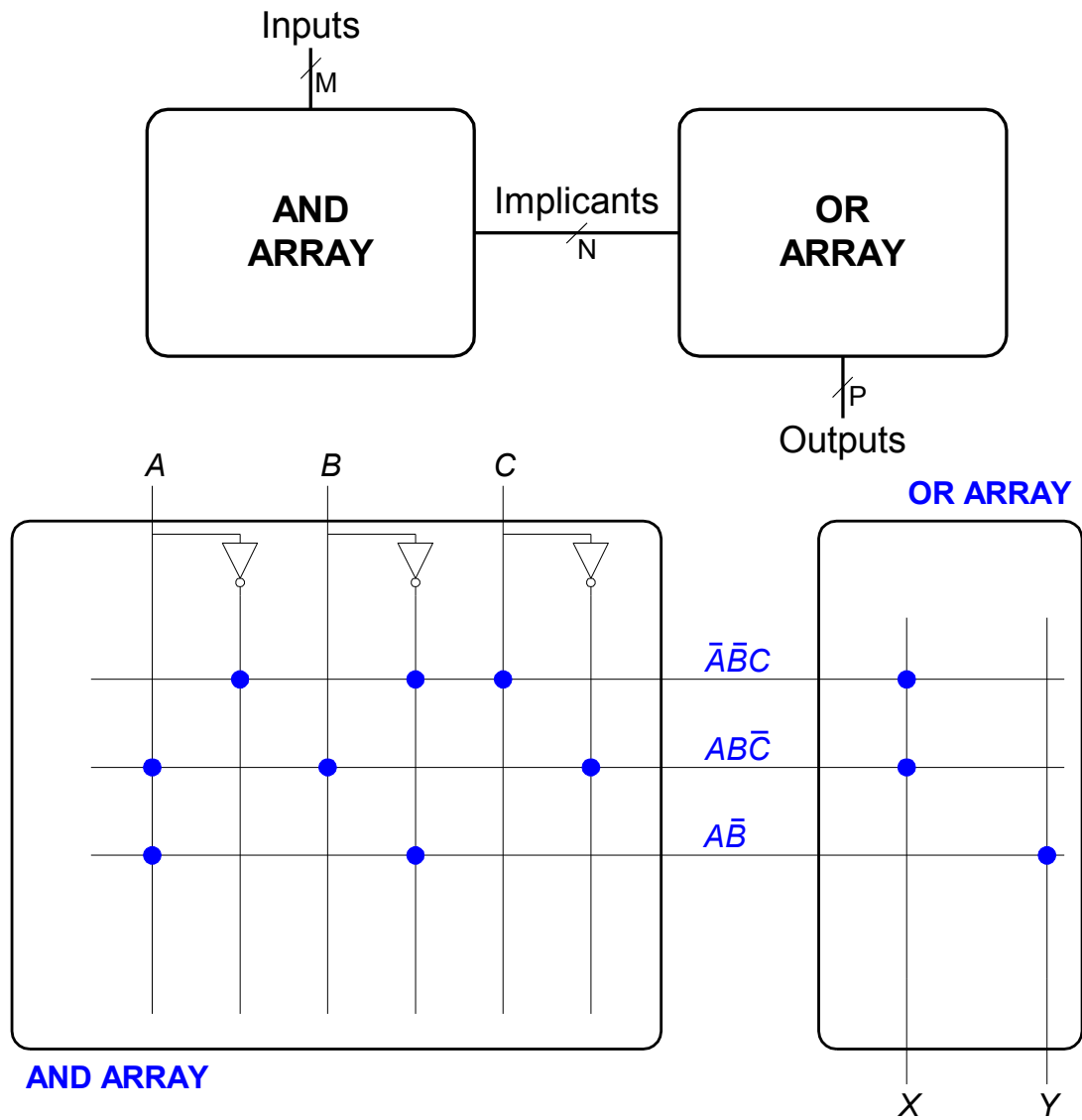
# PLAs

- $X = A\bar{B}\bar{C} + ABC\bar{C}$
- $Y = AB\bar{C}$





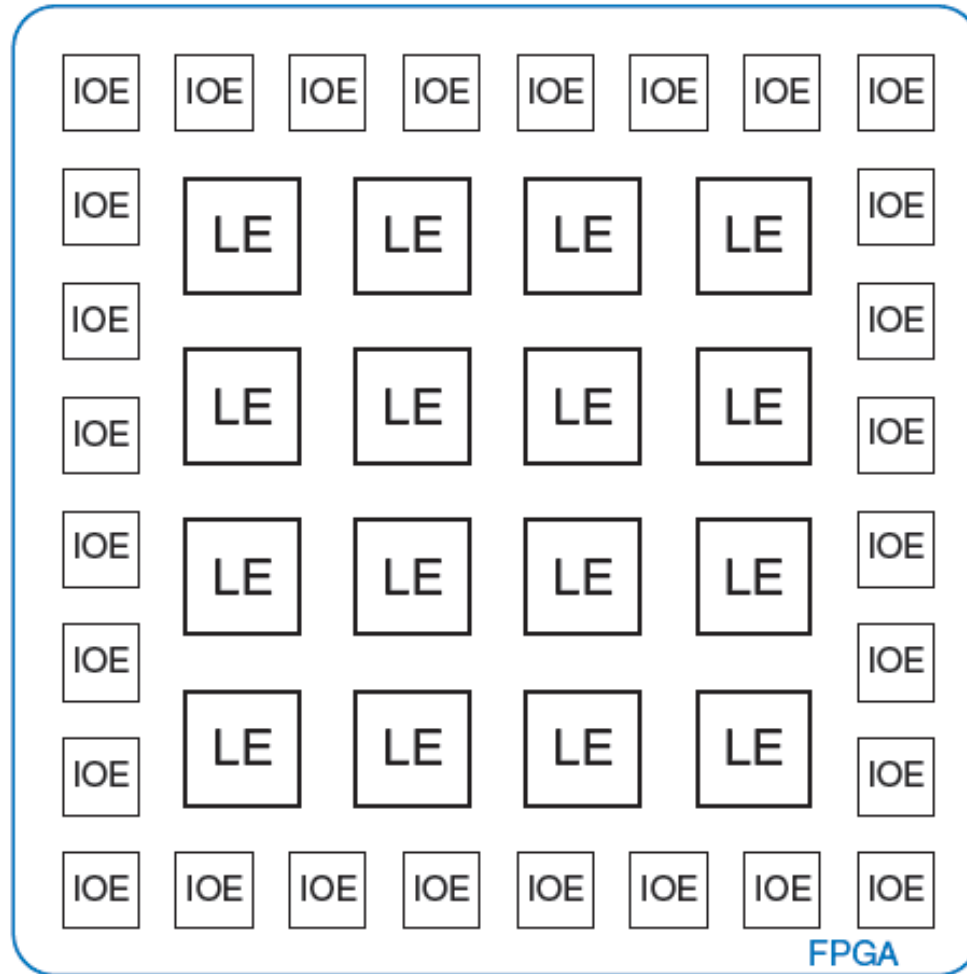
# PLAs: Dot Notation



# FPGA: Field Programmable Gate Array

- Composed of:
  - **LEs** (Logic elements): perform logic
  - **IOEs** (Input/output elements): interface with outside world
  - **Programmable interconnection**: connect LEs and IOEs
  - Some FPGAs include other building blocks such as multipliers and RAMs

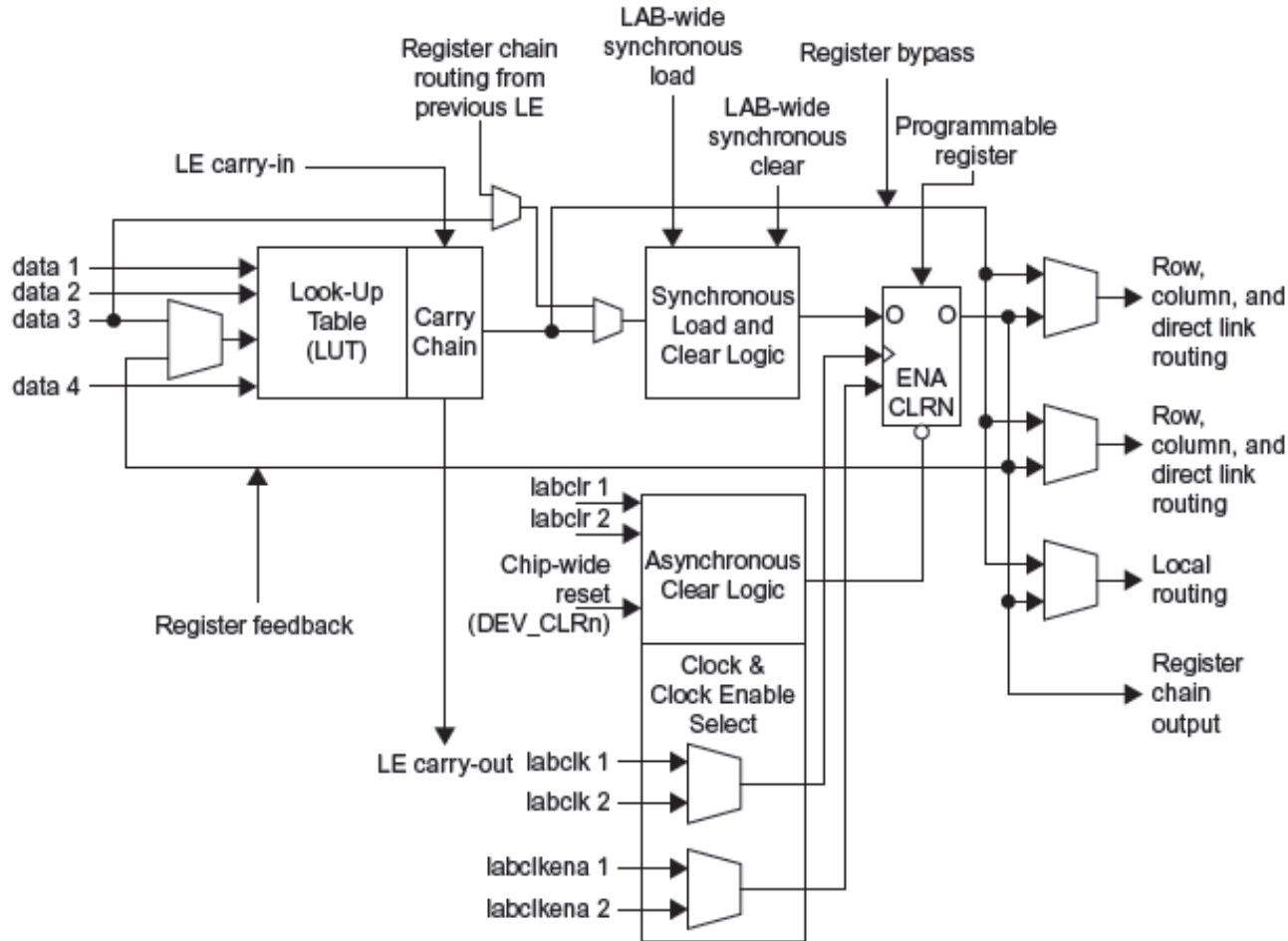
# General FPGA Layout



# LE: Logic Element

- Composed of:
  - **LUTs** (lookup tables): perform combinational logic
  - **Flip-flops**: perform sequential logic
  - **Multiplexers**: connect LUTs and flip-flops

# Altera Cyclone IV LE



# Altera Cyclone IV LE

- The Spartan CLB has:
  - 1 four-input LUT
  - 1 registered output
  - 1 combinational output

# LE Configuration Example

Show how to configure a Cyclone IV LE to perform the following functions:

$$-X = A\bar{B}\bar{C} + ABC$$

$$-Y = AB$$

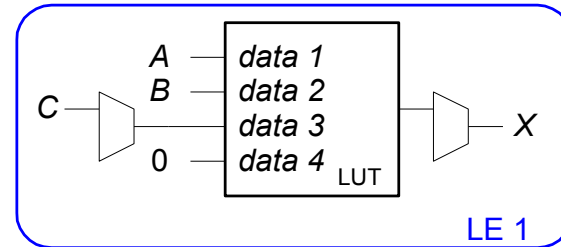
# LE Configuration Example

Show how to configure a Cyclone IV LE to perform the following functions:

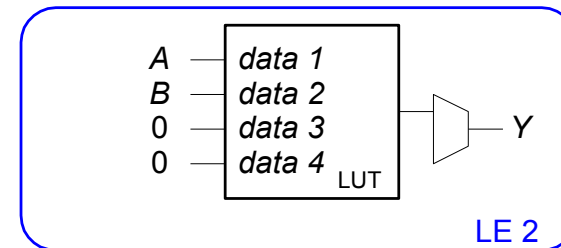
$$-X = A\bar{B}\bar{C} + ABC$$

$$-Y = AB$$

(A) data 1	(B) data 2	(C) data 3	data 4	(X) LUT output
0	0	0	X	0
0	0	1	X	1
0	1	0	X	0
0	1	1	X	0
1	0	0	X	0
1	0	1	X	0
1	1	0	X	1
1	1	1	X	0



(A) data 1	(B) data 2	data 3	data 4	(Y) LUT output
0	0	X	X	0
0	1	X	X	0
1	0	X	X	1
1	1	X	X	0





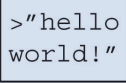


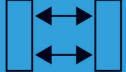
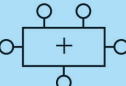

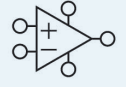


# FPGA Design Flow

Using a CAD tool (such as Altera's Quartus II)

- **Enter the design** using schematic entry or an HDL
- **Simulate** the design
- **Synthesize** design and map it onto FPGA
- **Download the configuration** onto the FPGA
- **Test** the design

# Chapter 8 :: Topics

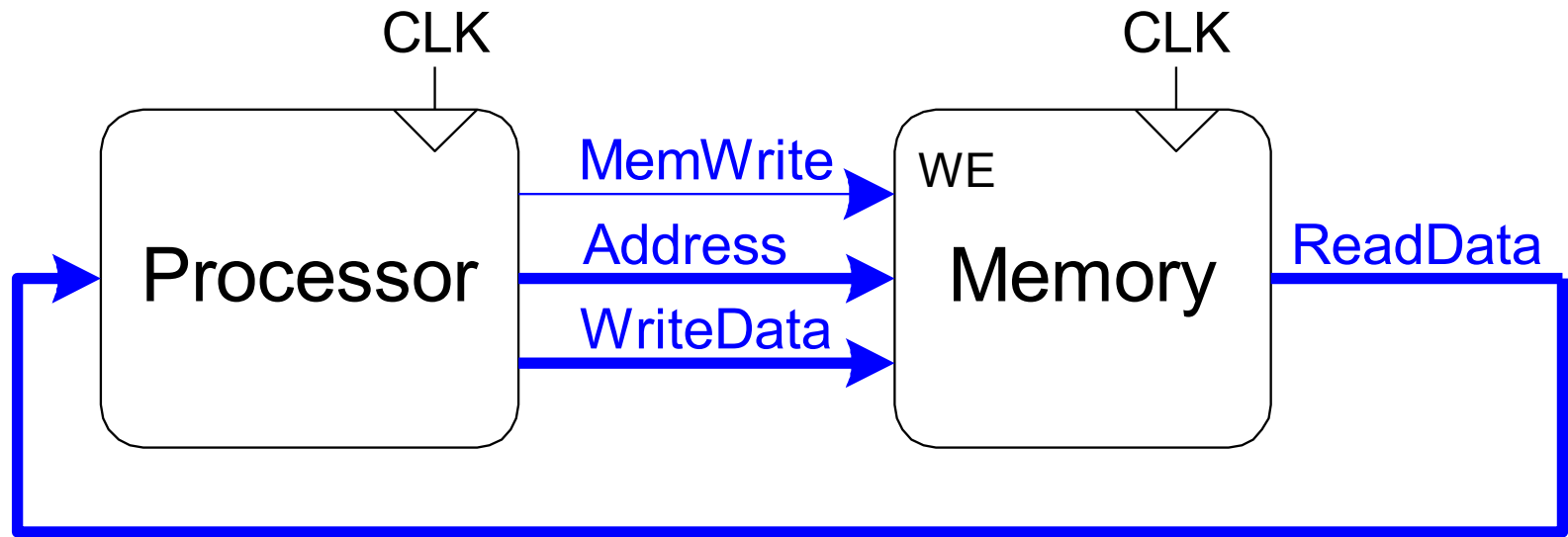
- Introduction (now)
- Memory System Performance Analysis (now)
- Caches (now)
- Virtual Memory (now)
- Memory-Mapped I/O (later)
- Summary (later)

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

# Introduction

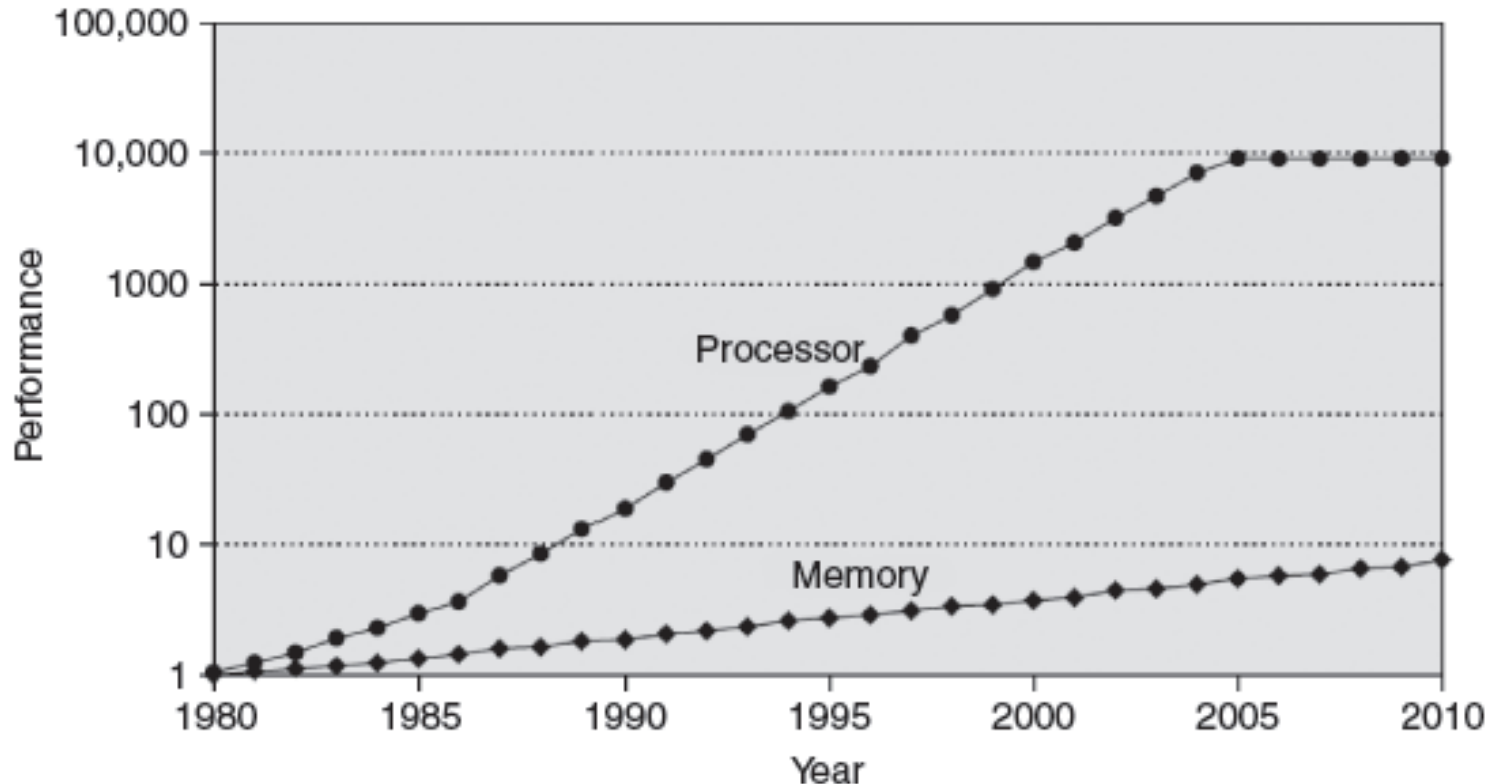
- Computer performance depends on:
  - Processor performance
  - Memory system performance

## Memory Interface



# Processor-Memory Gap

In prior chapters, assumed access memory in 1 clock cycle – but hasn't been true since the 1980's



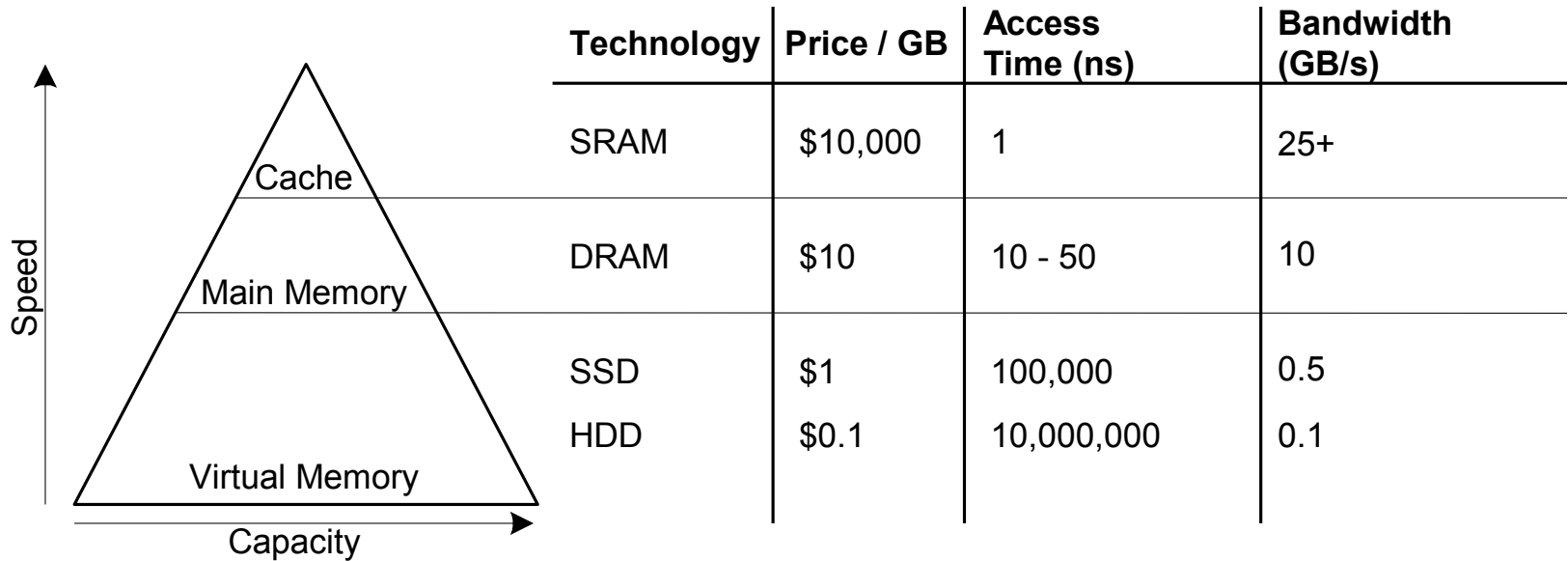
# Memory System Challenge

- Make memory system appear as fast as processor
- Use hierarchy of memories
- Ideal memory:
  - Fast
  - Cheap (inexpensive)
  - Large (capacity)

**But can only choose two!**



# Memory Hierarchy



# Locality

Exploit locality to make memory accesses fast

- **Temporal Locality:**

- Locality in time
- If data used recently, likely to use it again soon
- **How to exploit:** keep recently accessed data in higher levels of memory hierarchy

- **Spatial Locality:**

- Locality in space
- If data used recently, likely to use nearby data soon
- **How to exploit:** when access data, bring nearby data into higher levels of memory hierarchy too

# Memory Performance

- **Hit:** data found in that level of memory hierarchy
- **Miss:** data not found (must go to next level)

$$\begin{aligned}\text{Hit Rate} &= \# \text{ hits} / \# \text{ memory accesses} \\ &= 1 - \text{Miss Rate}\end{aligned}$$

$$\begin{aligned}\text{Miss Rate} &= \# \text{ misses} / \# \text{ memory accesses} \\ &= 1 - \text{Hit Rate}\end{aligned}$$

- **Average memory access time (AMAT):** average time for processor to access data

$$\text{AMAT} = t_{\text{cache}} + MR_{\text{cache}}[t_{MM} + MR_{MM}(t_{VM})]$$



# Memory Performance Example 1

- A program has 2,000 loads and stores
- 1,250 of these data values in cache
- Rest supplied by other levels of memory hierarchy
- **What are the hit and miss rates for the cache?**

# Memory Performance Example 1

- A program has 2,000 loads and stores
- 1,250 of these data values in cache
- Rest supplied by other levels of memory hierarchy
- **What are the hit and miss rates for the cache?**

$$\text{Hit Rate} = 1250/2000 = \mathbf{0.625}$$

$$\text{Miss Rate} = 750/2000 = \mathbf{0.375} = 1 - \text{Hit Rate}$$

# Memory Performance Example 2

- Suppose processor has 2 levels of hierarchy: cache and main memory
- $t_{\text{cache}} = 1$  cycle,  $t_{MM} = 100$  cycles
- **What is the AMAT of the program from Example 1?**

# Memory Performance Example 2

- Suppose processor has 2 levels of hierarchy: cache and main memory
- $t_{\text{cache}} = 1$  cycle,  $t_{MM} = 100$  cycles
- **What is the AMAT of the program from Example 1?**

$$\begin{aligned}\text{AMAT} &= t_{\text{cache}} + MR_{\text{cache}}(t_{MM}) \\ &= [1 + 0.375(100)] \text{ cycles} \\ &= \mathbf{38.5 \text{ cycles}}\end{aligned}$$

# Gene Amdahl, 1922-

- **Amdahl's Law:** the effort spent increasing the performance of a subsystem is wasted unless the subsystem affects a large percentage of overall performance
- Co-founded 3 companies, including one called Amdahl Corporation in 1970



# Cache

- Highest level in memory hierarchy
- Fast (typically  $\sim 1$  cycle access time)
- Ideally supplies most data to processor
- Usually holds most recently accessed data

# Cache Design Questions

- What data is held in the cache?
- How is data found?
- What data is replaced?

Focus on data loads, but stores follow same principles

# What data is held in the cache?

- Ideally, cache anticipates needed data and puts it in cache
- But impossible to predict future
- Use past to predict future – temporal and spatial locality:
  - **Temporal locality:** copy newly accessed data into cache
  - **Spatial locality:** copy neighboring data into cache too



# Cache Terminology

- **Capacity ( $C$ ):**
  - number of data bytes in cache
- **Block size ( $b$ ):**
  - bytes of data brought into cache at once
- **Number of blocks ( $B = C/b$ ):**
  - number of blocks in cache:  $B = C/b$
- **Degree of associativity ( $N$ ):**
  - number of blocks in a set
- **Number of sets ( $S = B/N$ ):**
  - each memory address maps to exactly one cache set

# How is data found?

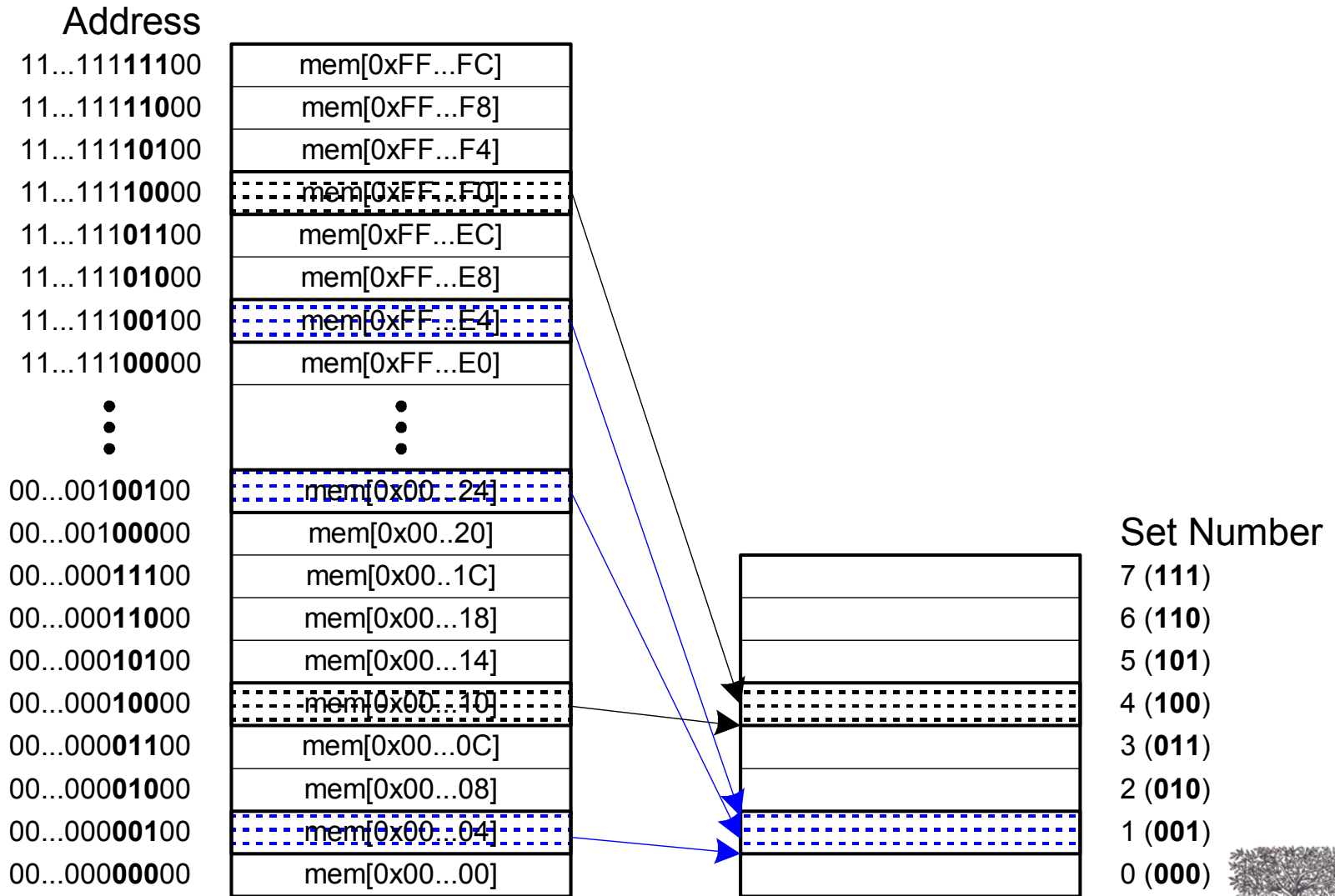
- Cache organized into  $S$  sets
- Each memory address maps to exactly one set
- Caches categorized by # of blocks in a set:
  - **Direct mapped:** 1 block per set
  - **$N$ -way set associative:**  $N$  blocks per set
  - **Fully associative:** all cache blocks in 1 set
- Examine each organization for a cache with:
  - Capacity ( $C = 8$  words)
  - Block size ( $b = 1$  word)
  - So, number of blocks ( $B = 8$ )

# Example Cache Parameters

- **$C = 8$**  words (capacity)
- **$b = 1$**  word (block size)
- So,  **$B = 8$**  (# of blocks)

Ridiculously small, but will illustrate organizations

# Direct Mapped Cache

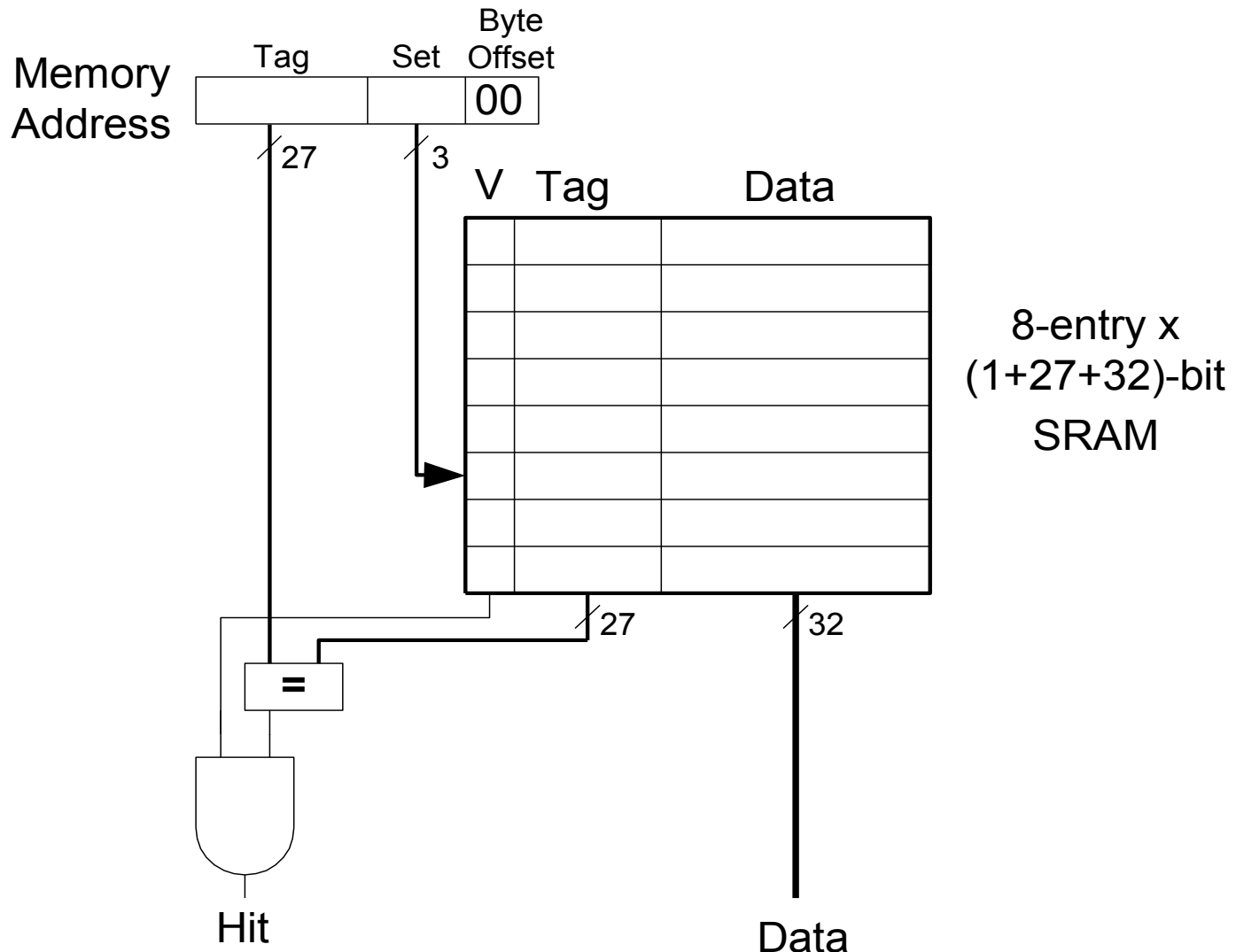


2<sup>30</sup> Word Main Memory

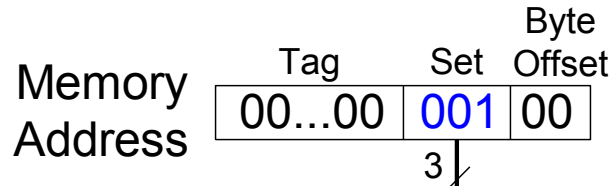
2<sup>3</sup> Word Cache



# Direct Mapped Cache Hardware



# Direct Mapped Cache Performance



# MIPS assembly code

```
    addi $t0, $0, 5
loop: beq  $t0, $0, done
      lw   $t1, 0x4($0)
      lw   $t2, 0xC($0)
      lw   $t3, 0x8($0)
      addi $t0, $t0, -1
      j    loop
done:
```

V	Tag	Data	
0			Set 7 (111)
0			Set 6 (110)
0			Set 5 (101)
0			Set 4 (100)
1	00...00	mem[0x00...0C]	Set 3 (011)
1	00...00	mem[0x00...08]	Set 2 (010)
1	00...00	mem[0x00...04]	Set 1 (001)
0			Set 0 (000)

Miss Rate = ?



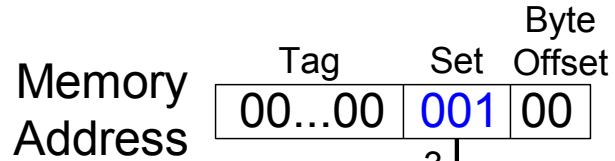
# Direct Mapped Cache Performance

## # MIPS assembly code

```

    addi $t0, $0, 5
loop: beq  $t0, $0, done
      lw   $t1, 0x4($0)
      lw   $t2, 0xC($0)
      lw   $t3, 0x8($0)
      addi $t0, $t0, -1
      j    loop
done:

```



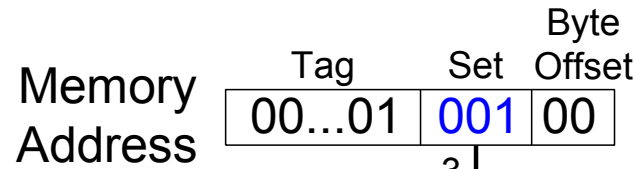
3

V	Tag	Data	
0			Set 7 (111)
0			Set 6 (110)
0			Set 5 (101)
0			Set 4 (100)
1	00...00	mem[0x00...0C]	Set 3 (011)
1	00...00	mem[0x00...08]	Set 2 (010)
1	00...00	mem[0x00...04]	Set 1 (001)
0			Set 0 (000)

**Miss Rate = 3/15  
= 20%**

**Temporal Locality  
Compulsory Misses**

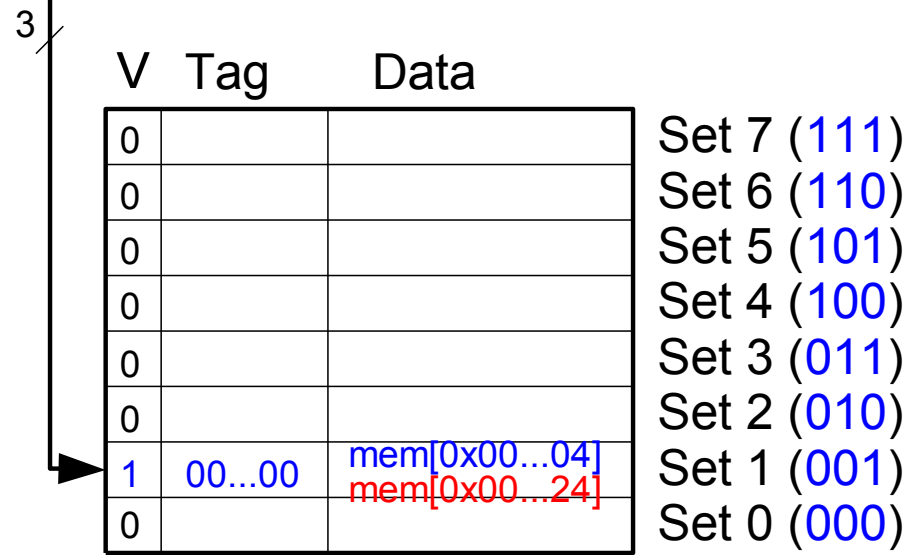
# Direct Mapped Cache: Conflict



# MIPS assembly code

```

addi $t0, $0, 5
loop: beq $t0, $0, done
      lw  $t1, 0x4($0)
      lw  $t2, 0x24($0)
      addi $t0, $t0, -1
      j   loop
done:
    
```

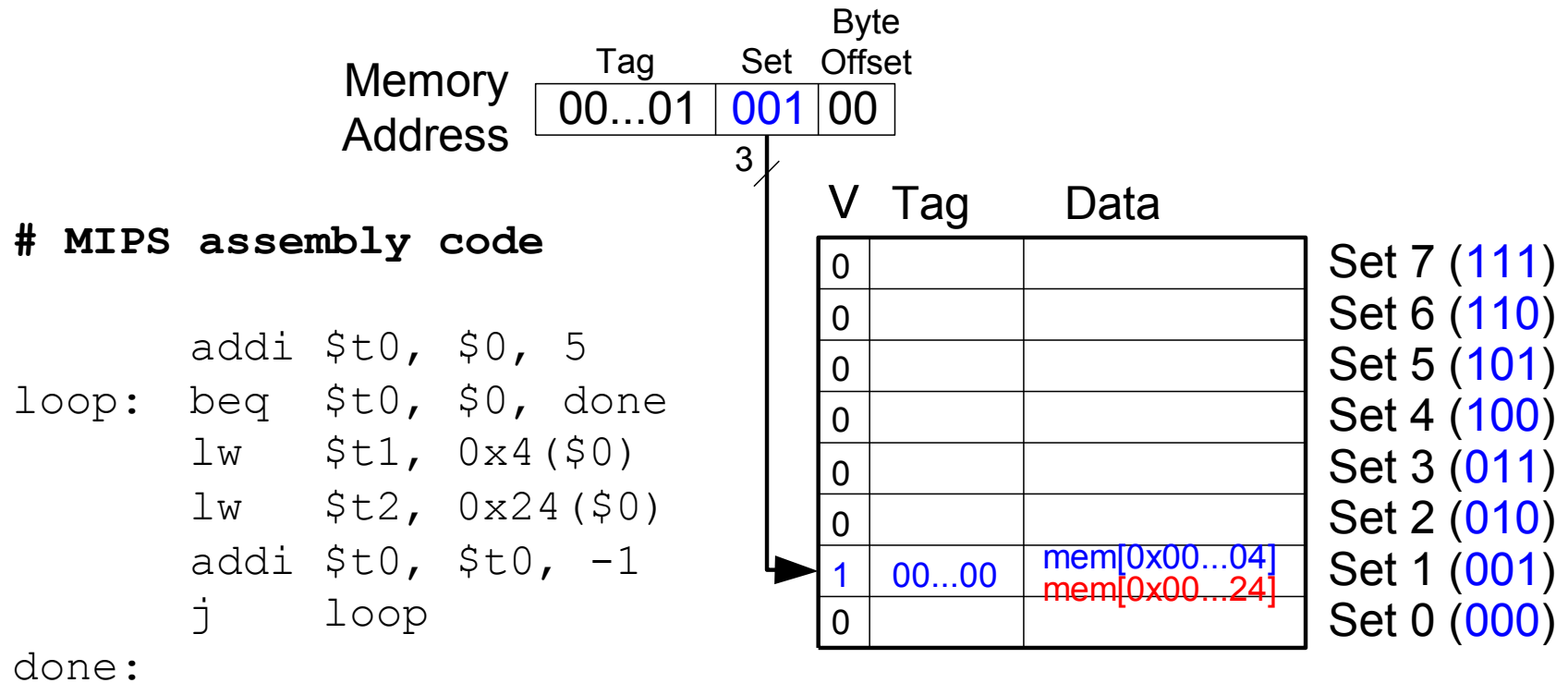


**Miss Rate = ?**





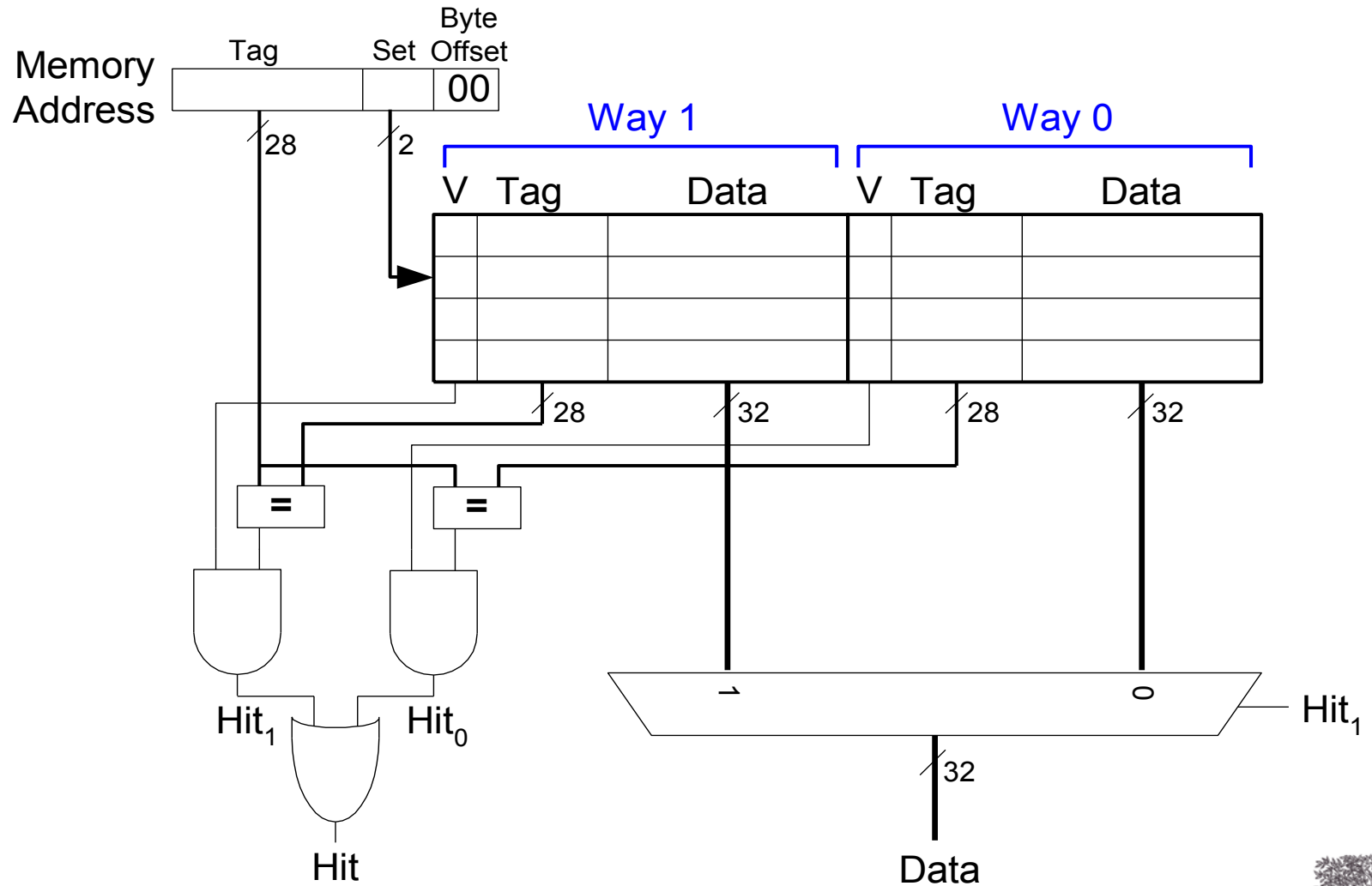
# Direct Mapped Cache: Conflict



**Miss Rate = 10/10  
= 100%**

**Conflict Misses**

# N-Way Set Associative Cache



# N-Way Set Associative

# MIPS assembly code

```
    addi $t0, $0, 5
loop: beq  $t0, $0, done
      lw   $t1, 0x4($0)
      lw   $t2, 0x24($0)
      addi $t0, $t0, -1
      j    loop
done:
```

Miss Rate = ?

Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
0			0			Set 1
0			0			Set 0

# N-Way Set Associative

# MIPS assembly code

```
    addi $t0, $0, 5
loop: beq  $t0, $0, done
      lw   $t1, 0x4($0)
      lw   $t2, 0x24($0)
      addi $t0, $t0, -1
      j    loop
done:
```

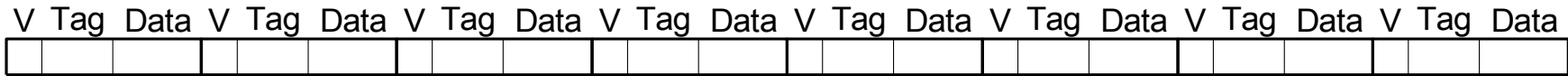
**Miss Rate = 2/10  
= 20%**

**Associativity reduces  
conflict misses**

Way 1			Way 0		
V	Tag	Data	V	Tag	Data
0			0		
0			0		
1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]
0			0		

Set 3  
Set 2  
Set 1  
Set 0

# Fully Associative Cache

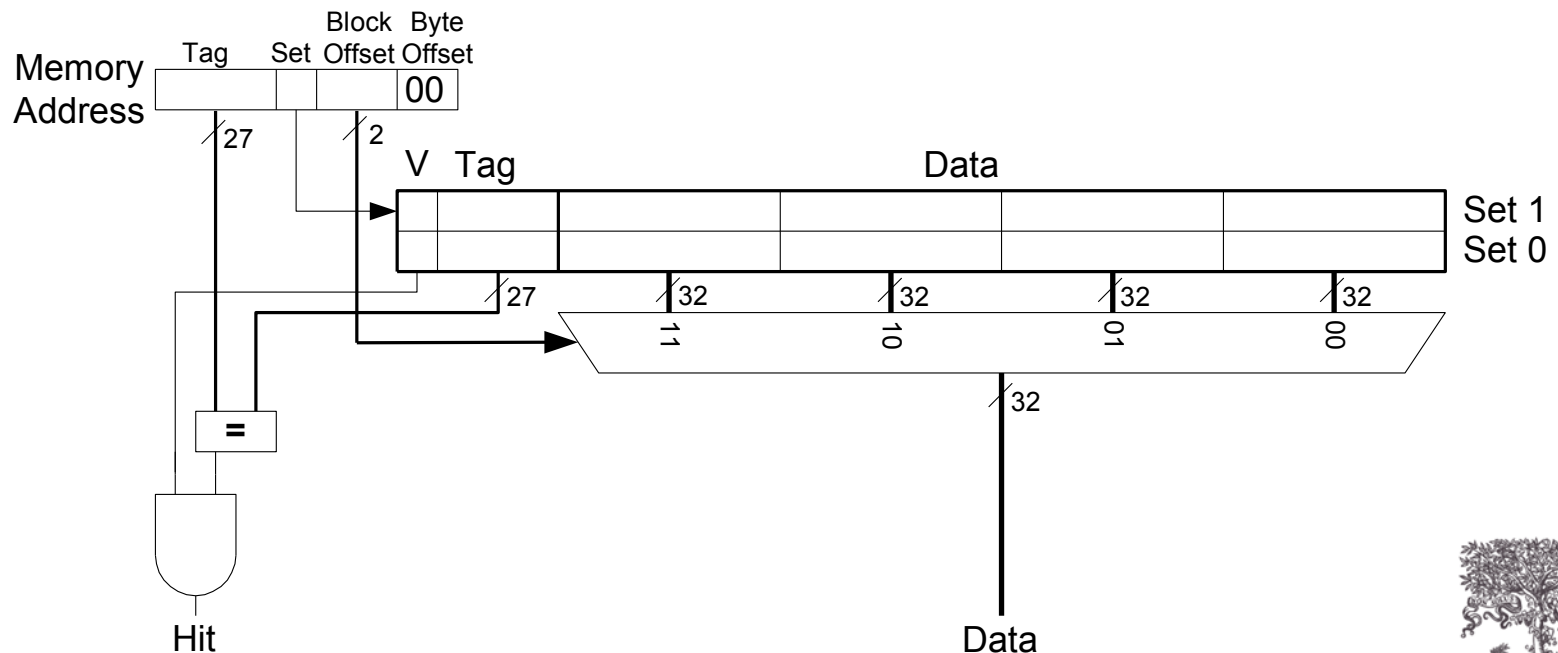


**Reduces conflict misses**  
**Expensive to build**

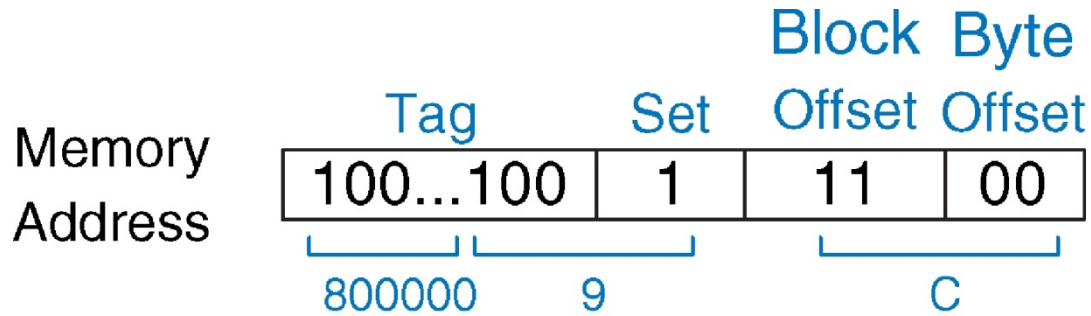


# Spatial Locality?

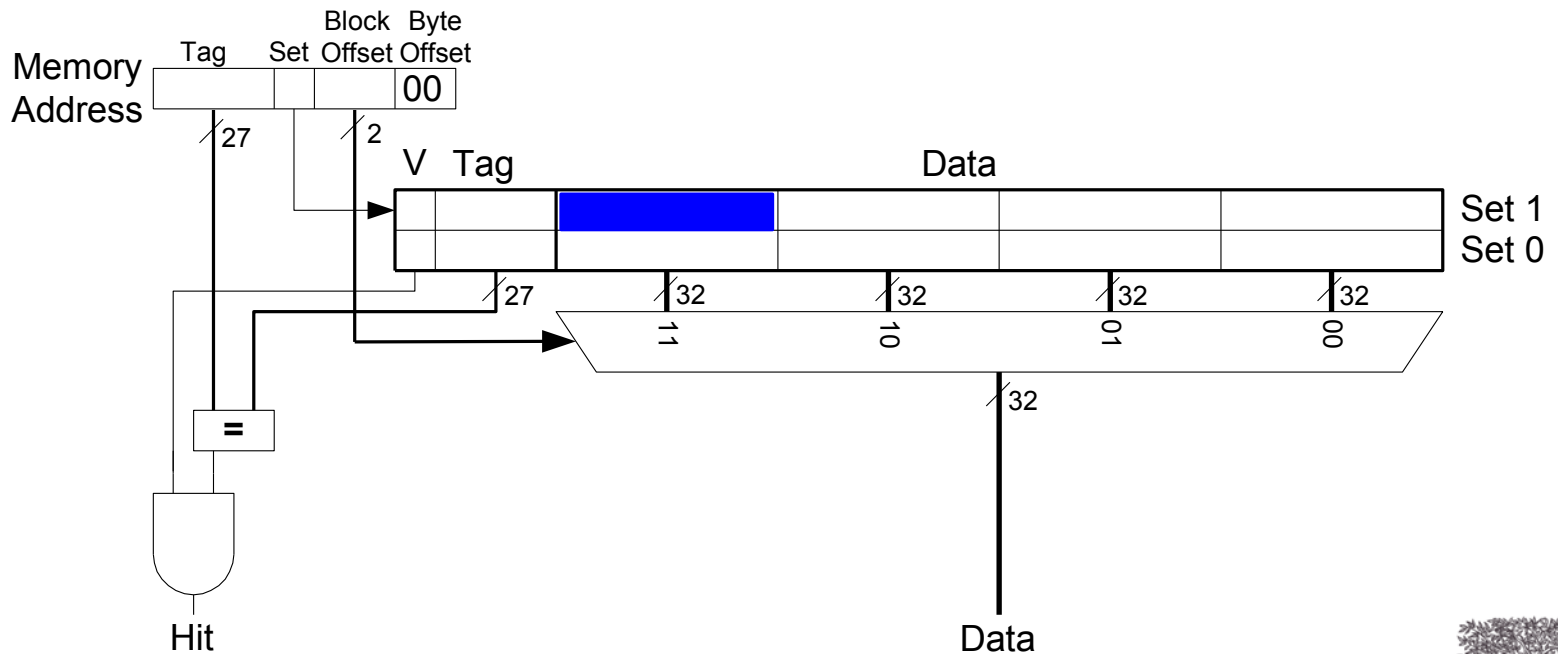
- Increase block size:
  - Block size,  $b = 4$  words
  - $C = 8$  words
  - Direct mapped (1 block per set)
  - Number of blocks,  $B = 2$  ( $C/b = 8/4 = 2$ )



# Cache with Larger Block Size



© 2007 Elsevier, Inc. All rights reserved



# Direct Mapped Cache Performance

```
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

**Miss Rate = ?**

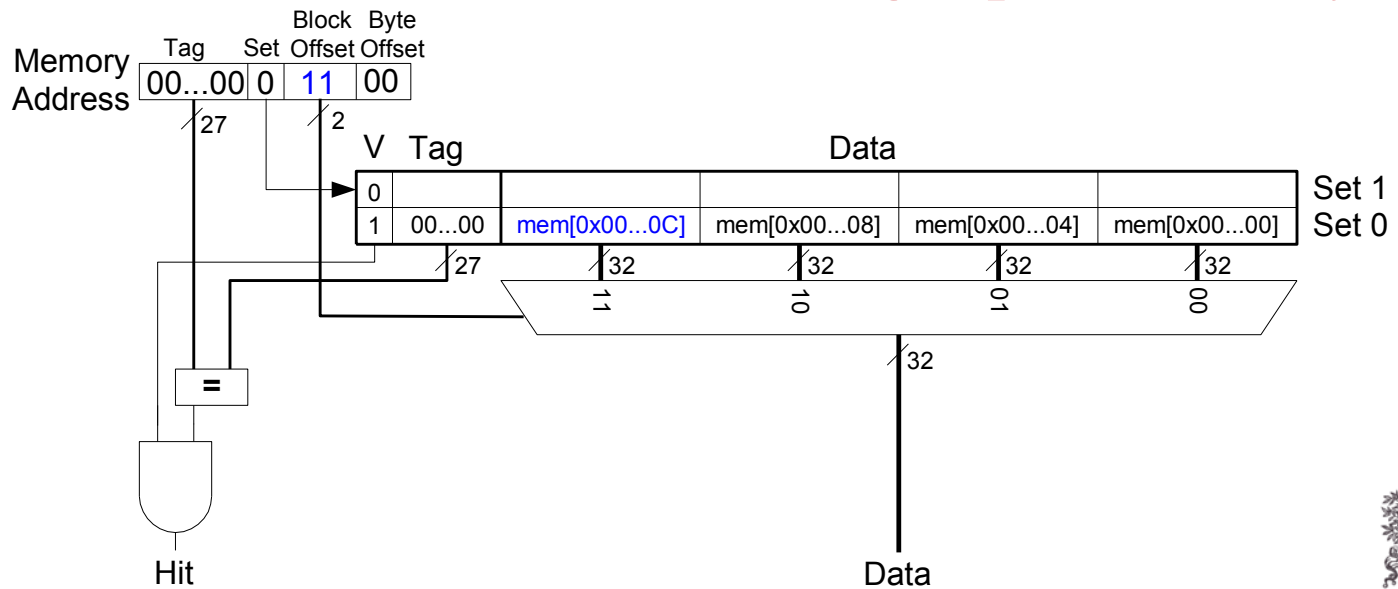


# Direct Mapped Cache Performance

```
    addi $t0, $0, 5
loop: beq  $t0, $0, done
      lw   $t1, 0x4($0)
      lw   $t2, 0xC($0)
      lw   $t3, 0x8($0)
      addi $t0, $t0, -1
      j    loop
done:
```

**Miss Rate = 1/15**  
**= 6.67%**

**Larger blocks**  
**reduce compulsory misses**  
**through spatial locality**



# Cache Organization Recap

- Capacity:  $C$
- Block size:  $b$
- Number of blocks in cache:  $B = C/b$
- Number of blocks in a set:  $N$
- Number of sets:  $S = B/N$

<b>Organization</b>	<b>Number of Ways (<math>N</math>)</b>	<b>Number of Sets (<math>S = B/N</math>)</b>
<b>Direct Mapped</b>	1	$B$
<b>N-Way Set Associative</b>	$1 < N < B$	$B / N$
<b>Fully Associative</b>	$B$	1

# Capacity Misses

- Cache is too small to hold all data of interest at once
- If cache full: program accesses data X & evicts data Y
- **Capacity miss** when access Y again
- How to choose Y to minimize chance of needing it again?
- **Least recently used (LRU) replacement**: the least recently used block in a set evicted

# Types of Misses

- **Compulsory:** first time data accessed
  - **Capacity:** cache too small to hold all data of interest
  - **Conflict:** data of interest maps to same location in cache
- Miss penalty:** time it takes to retrieve a block from lower level of hierarchy

# LRU Replacement

# MIPS assembly

```
lw $t0, 0x04 ($0)
```

```
lw $t1, 0x24 ($0)
```

```
lw $t2, 0x54 ($0)
```

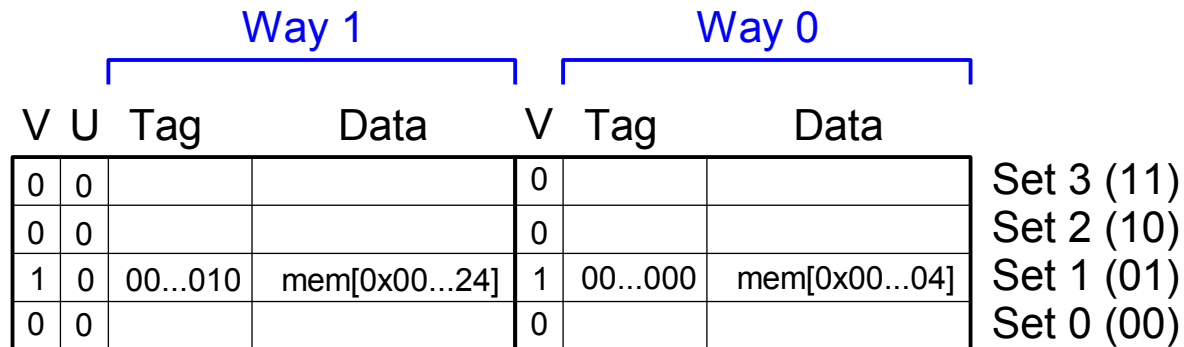
Way 1				Way 0		
V	U	Tag	Data	V	Tag	Data
0	0			0		
0	0			0		
0	0			0		
0	0			0		

Set 3 (11)  
Set 2 (10)  
Set 1 (01)  
Set 0 (00)

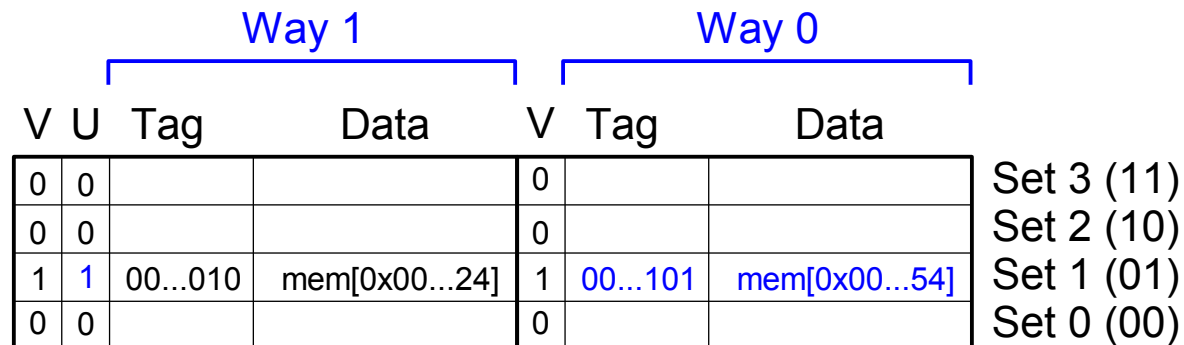
# LRU Replacement

# MIPS assembly

```
lw $t0, 0x04 ($0)
lw $t1, 0x24 ($0)
lw $t2, 0x54 ($0)
```



(a)



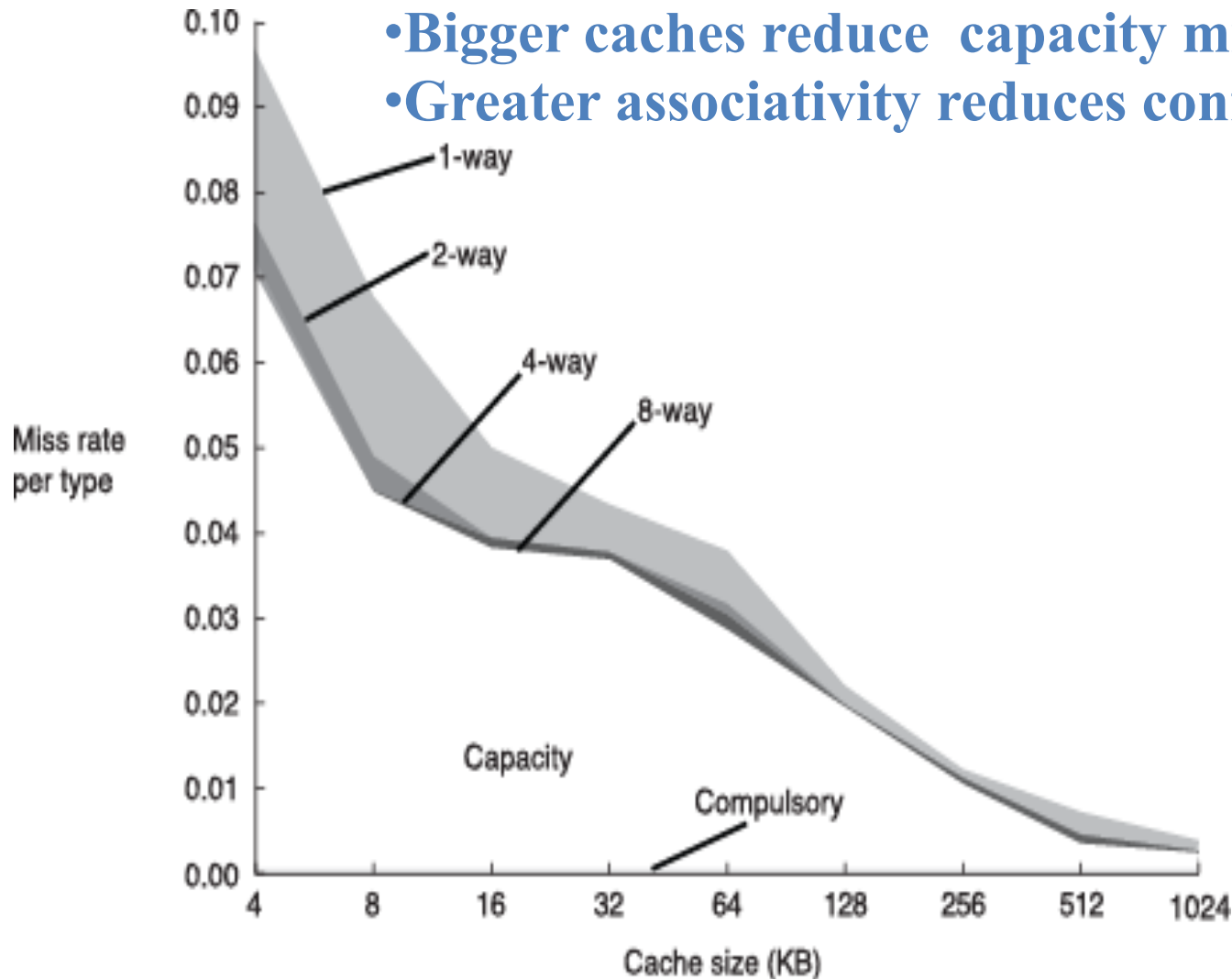
(b)

# Cache Summary

- **What data is held in the cache?**
  - Recently used data (temporal locality)
  - Nearby data (spatial locality)
- **How is data found?**
  - Set is determined by address of data
  - Word within block also determined by address
  - In associative caches, data could be in one of several ways
- **What data is replaced?**
  - Least-recently used way in the set

# Miss Rate Trends

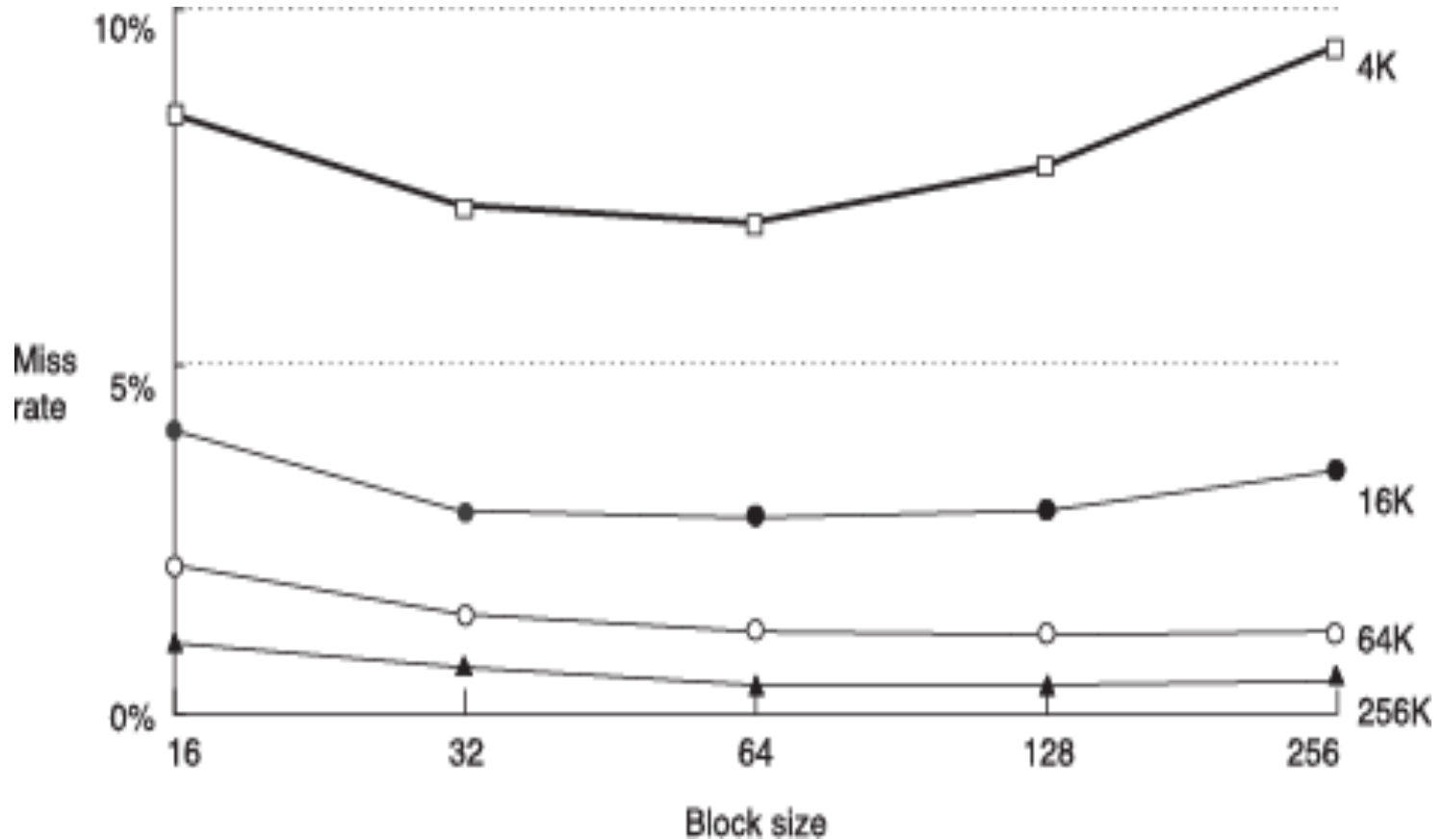
- Bigger caches reduce capacity misses
- Greater associativity reduces conflict misses



Adapted from Patterson & Hennessy, *Computer Architecture: A Quantitative Approach*, 2011



# Miss Rate Trends

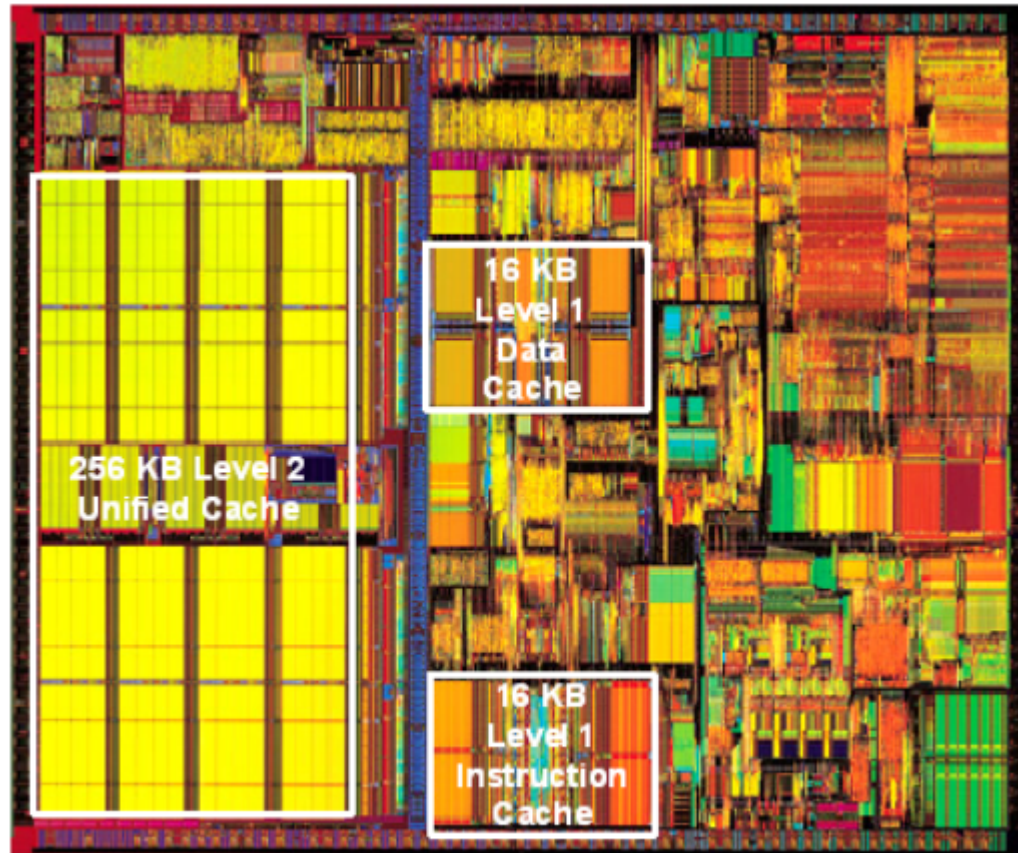


- Bigger blocks reduce compulsory misses
- Bigger blocks increase conflict misses

# Multilevel Caches

- Larger caches have lower miss rates, longer access times
- Expand memory hierarchy to multiple levels of caches
- Level 1: small and fast (e.g. 16 KB, 1 cycle)
- Level 2: larger and slower (e.g. 256 KB, 2-6 cycles)
- Most modern PCs have L1, L2, and L3 cache

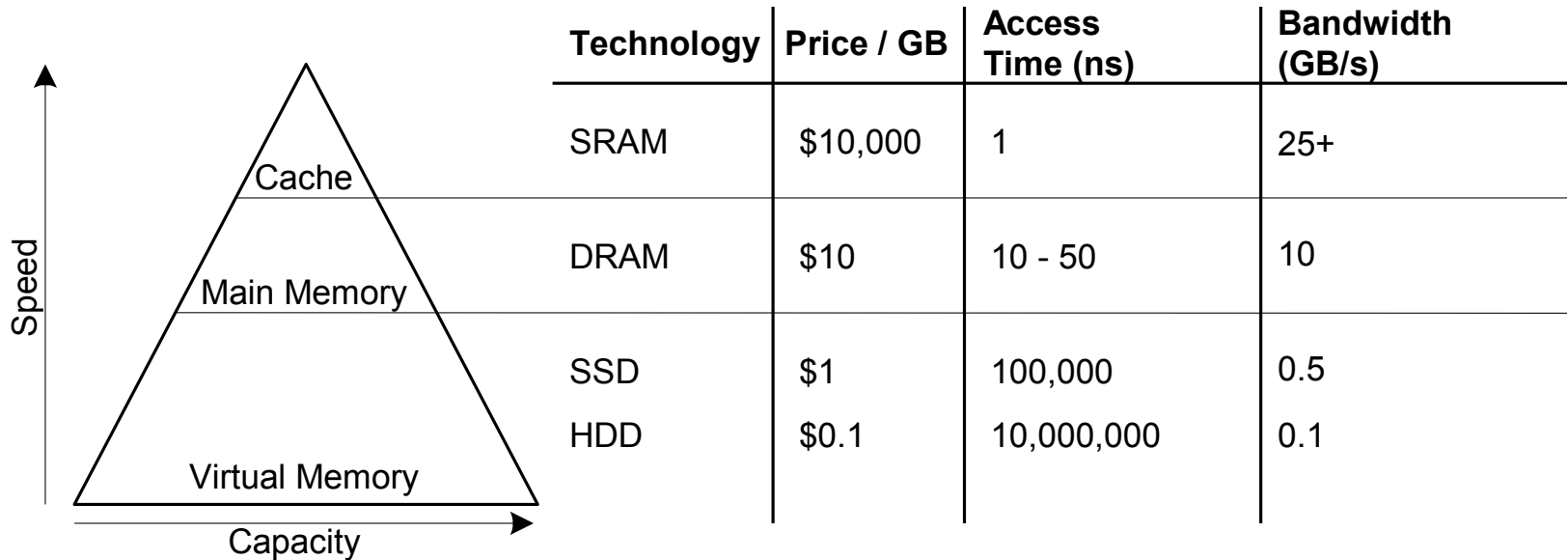
# Intel Pentium III Die



# Virtual Memory

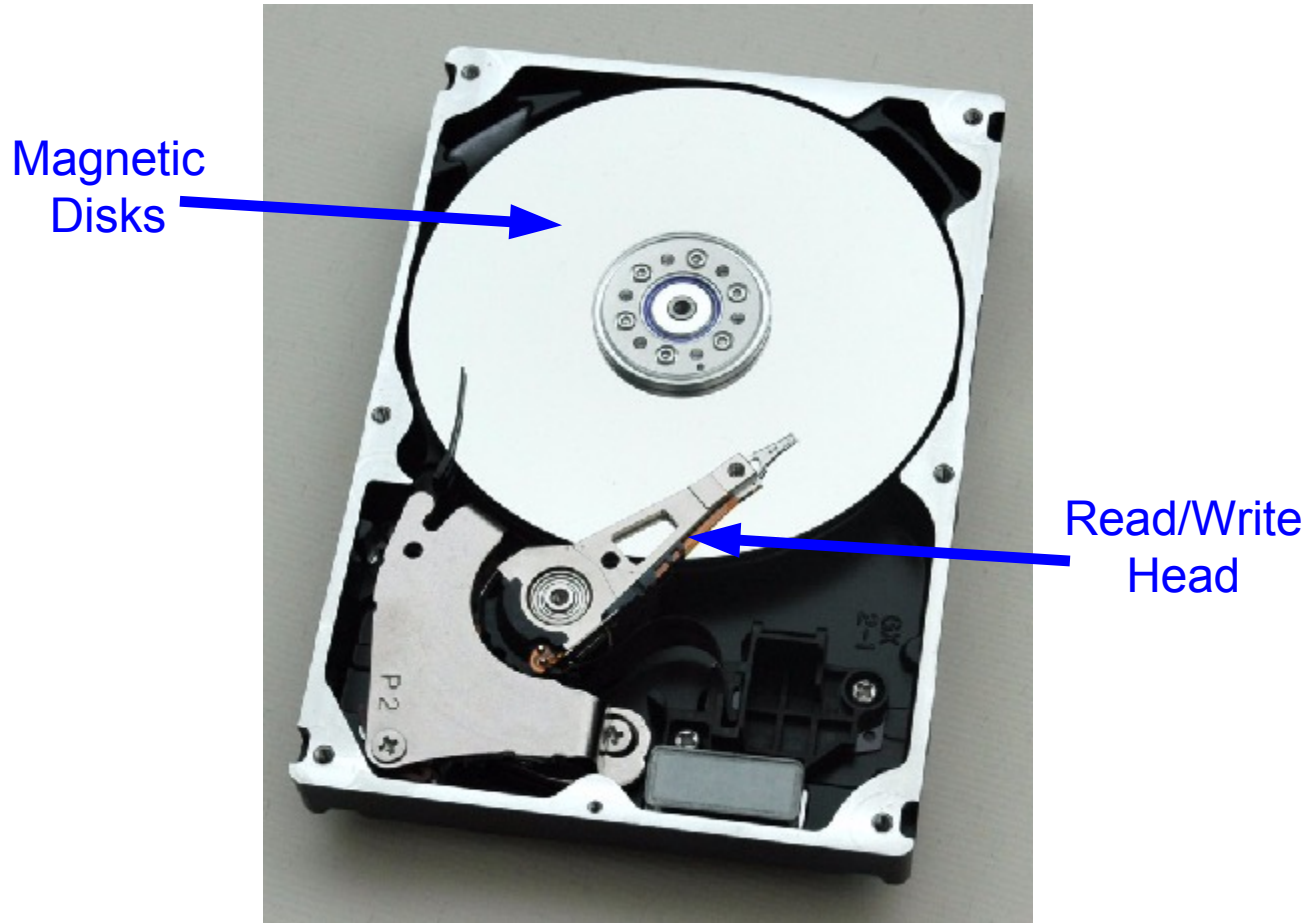
- Gives the illusion of bigger memory
- Main memory (DRAM) acts as cache for hard disk

# Memory Hierarchy



- **Physical Memory:** DRAM (Main Memory)
- **Virtual Memory:** Hard drive
  - Slow, Large, Cheap

# Hard Disk



**Takes milliseconds to *seek* correct location on disk**

# Virtual Memory

## •Virtual addresses

- Programs use virtual addresses
- Entire virtual address space stored on a hard drive
- Subset of virtual address data in DRAM
- CPU translates virtual addresses into ***physical addresses*** (DRAM addresses)
- Data not in DRAM fetched from hard drive

## •Memory Protection

- Each program has own virtual to physical mapping
- Two programs can use same virtual address for different data
- Programs don't need to be aware others are running
- One program (or virus) can't corrupt memory used by another

# Cache/Virtual Memory Analogues

<b>Cache</b>	<b>Virtual Memory</b>
Block	Page
Block Size	Page Size
Block Offset	Page Offset
Miss	Page Fault
Tag	Virtual Page Number

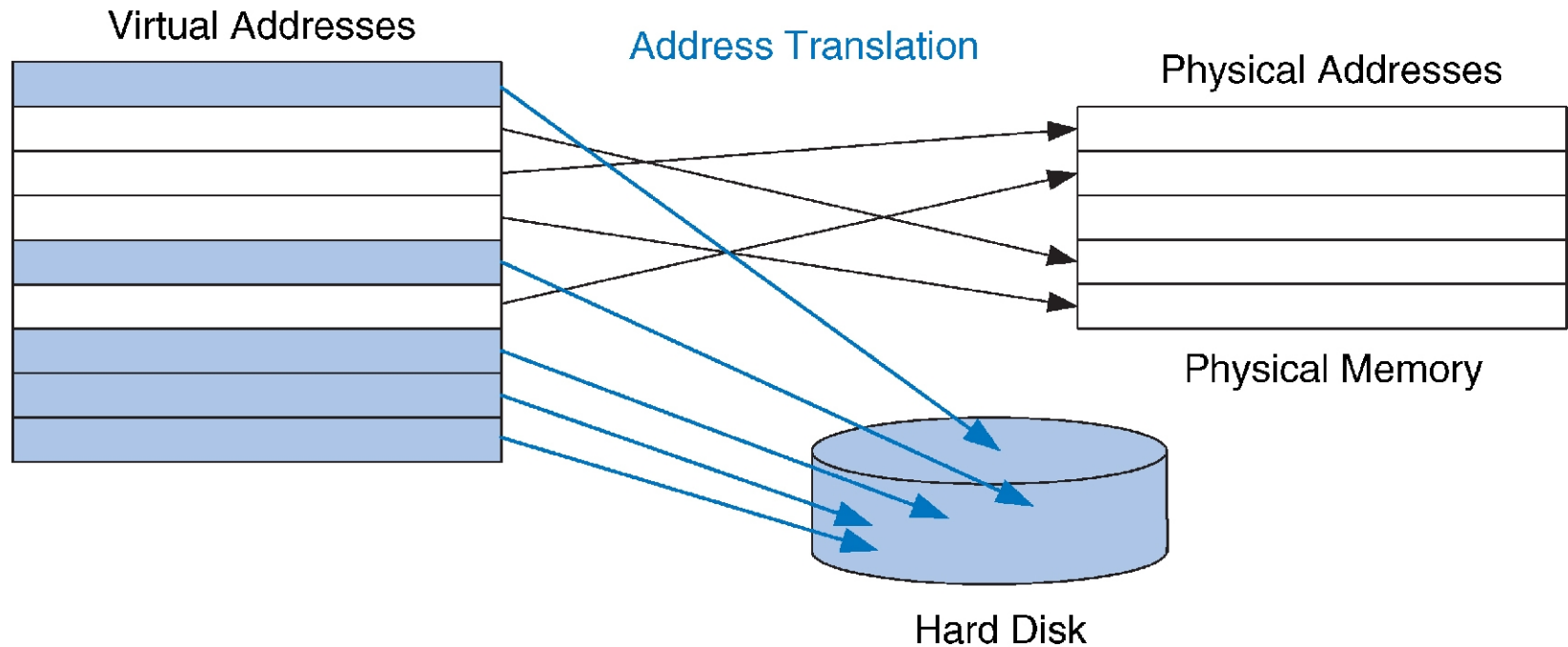
**Physical memory acts as cache for virtual memory**



# Virtual Memory Definitions

- **Page size:** amount of memory transferred from hard disk to DRAM at once
- **Address translation:** determining physical address from virtual address
- **Page table:** lookup table used to translate virtual addresses to physical addresses

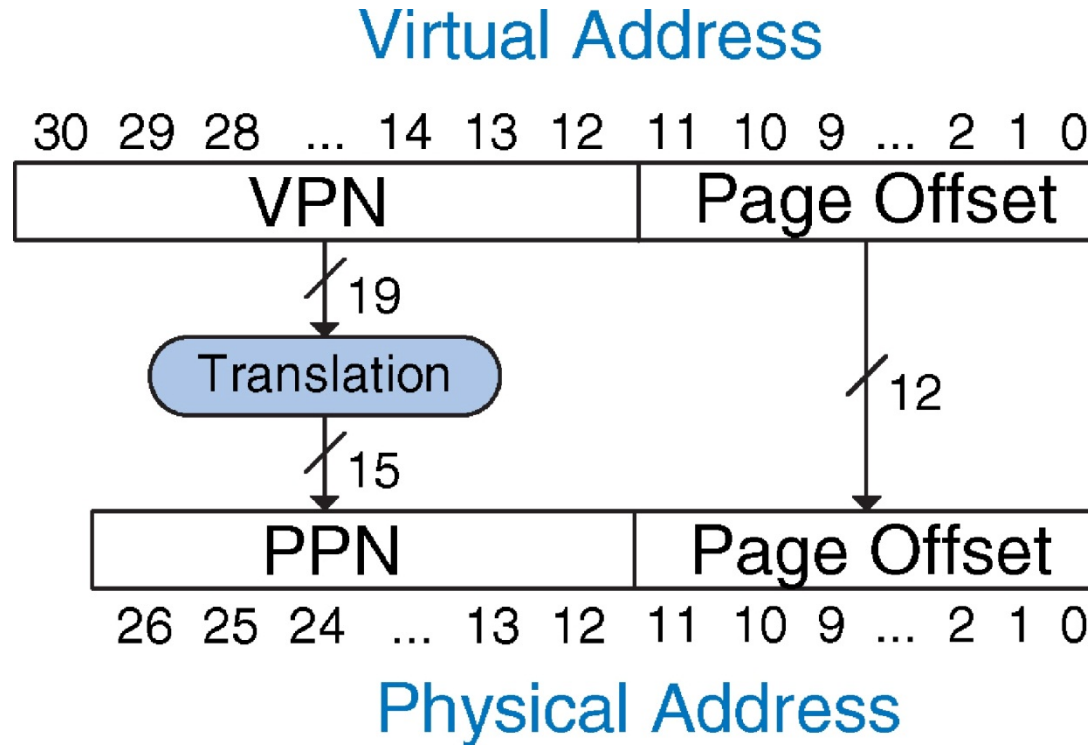
# Virtual & Physical Addresses



© 2007 Elsevier, Inc. All rights reserved

**Most accesses hit in physical memory**  
**But programs have the large capacity of virtual memory**

# Address Translation



© 2007 Elsevier, Inc. All rights reserved

# Virtual Memory Example

- **System:**

- Virtual memory size: 2 GB =  $2^{31}$  bytes
- Physical memory size: 128 MB =  $2^{27}$  bytes
- Page size: 4 KB =  $2^{12}$  bytes

# Virtual Memory Example

## •System:

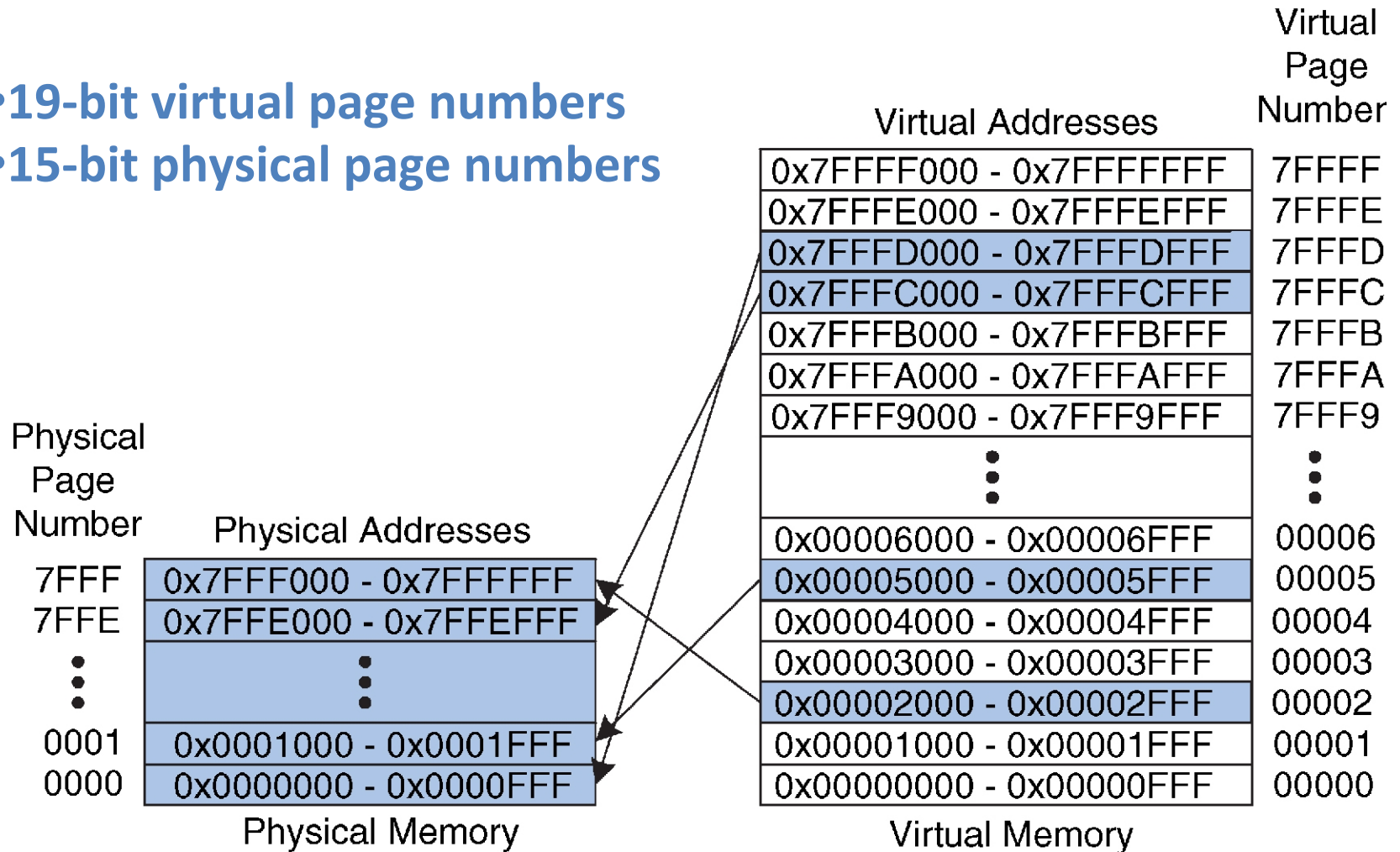
- Virtual memory size: 2 GB =  $2^{31}$  bytes
- Physical memory size: 128 MB =  $2^{27}$  bytes
- Page size: 4 KB =  $2^{12}$  bytes

## •Organization:

- Virtual address: **31** bits
- Physical address: **27** bits
- Page offset: **12** bits
- # Virtual pages =  $2^{31}/2^{12} = 2^{19}$  (VPN = 19 bits)
- # Physical pages =  $2^{27}/2^{12} = 2^{15}$  (PPN = 15 bits)

# Virtual Memory Example

- 19-bit virtual page numbers
- 15-bit physical page numbers

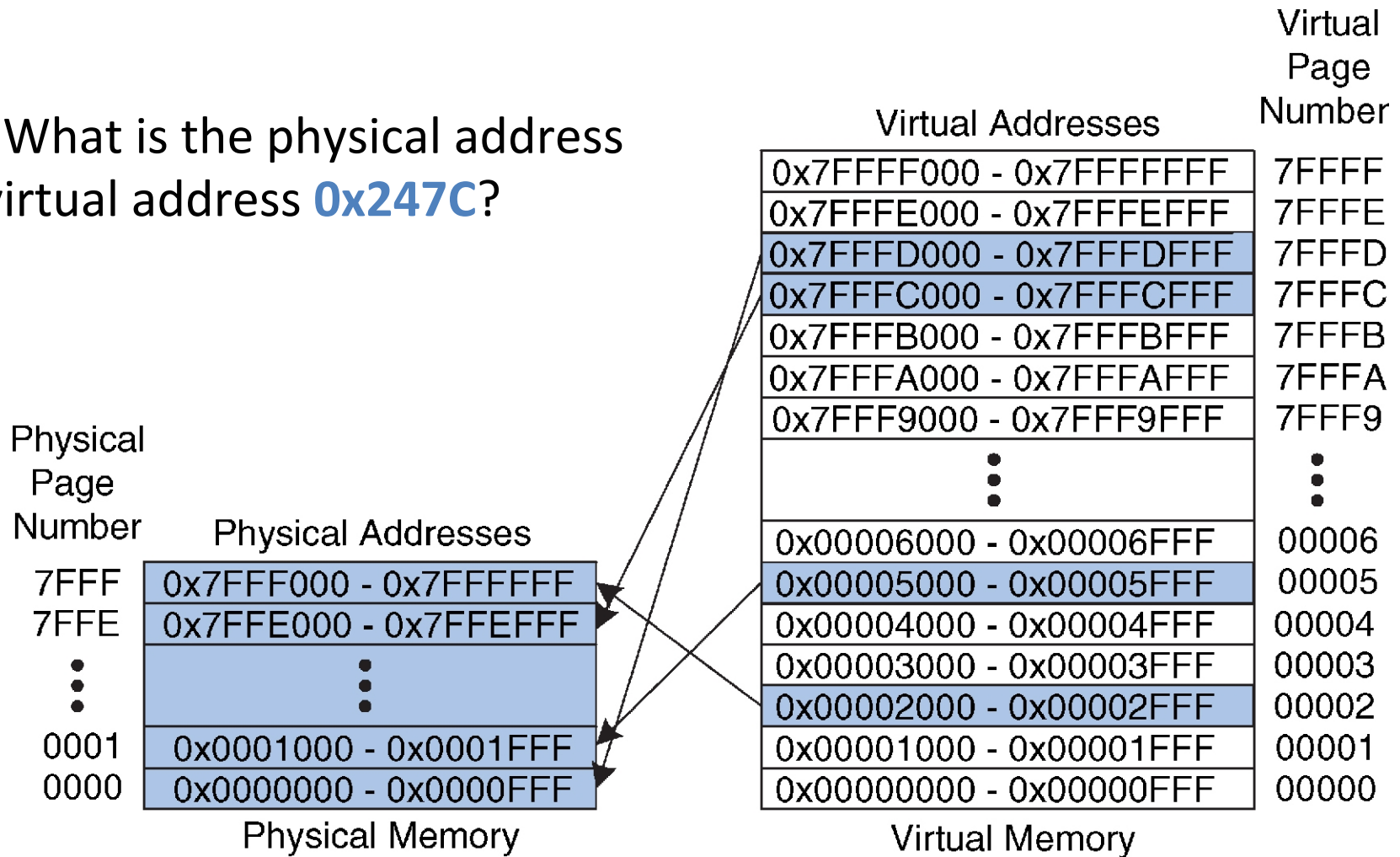


© 2007 Elsevier, Inc. All rights reserved



# Virtual Memory Example

What is the physical address of virtual address **0x247C**?



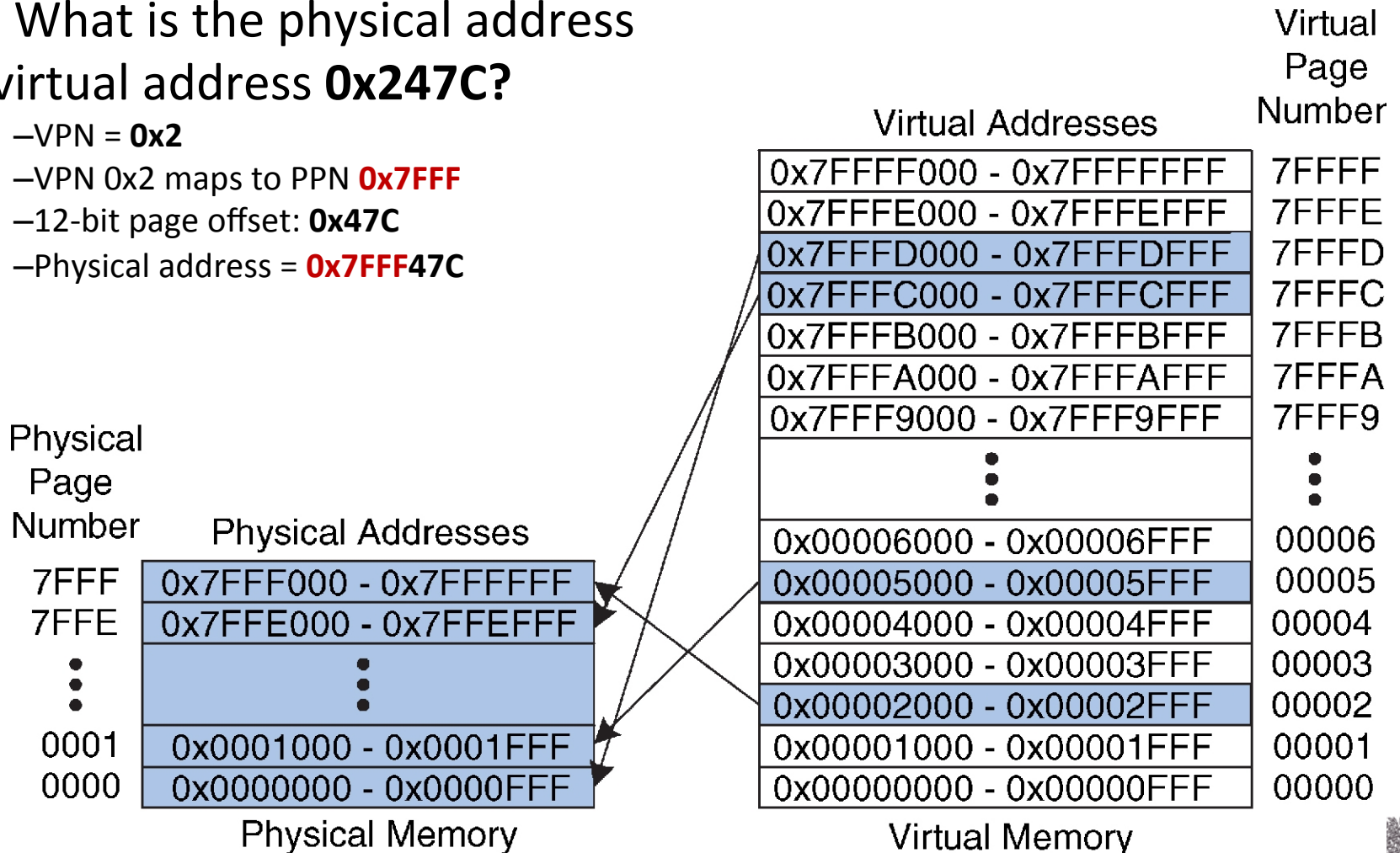
© 2007 Elsevier, Inc. All rights reserved



# Virtual Memory Example

What is the physical address of virtual address **0x247C**?

- VPN = **0x2**
- VPN 0x2 maps to PPN **0x7FFF**
- 12-bit page offset: **0x47C**
- Physical address = **0x7FFF47C**





# How to perform translation?

- **Page table**

- Entry for each virtual page

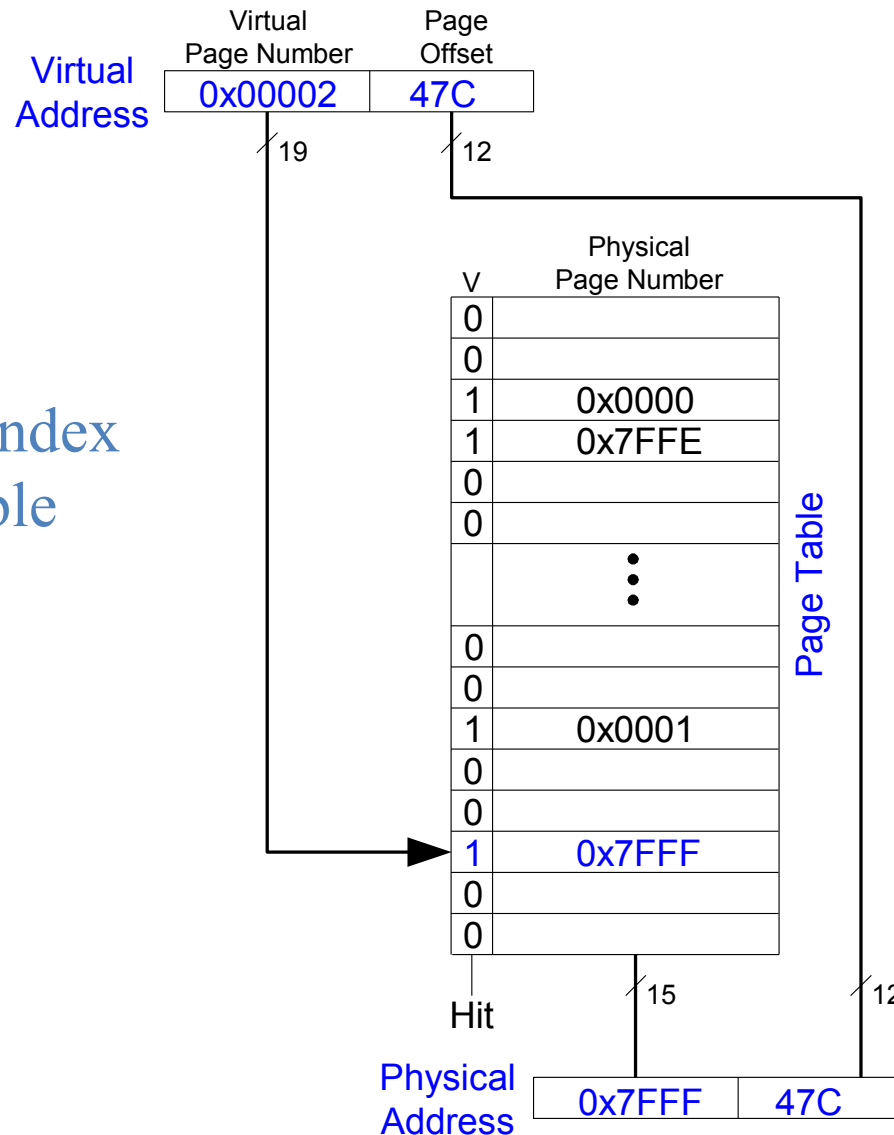
- Entry fields:

- **Valid bit:** 1 if page in physical memory

- **Physical page number:** where the page is located

# Page Table Example

VPN is index  
into page table



# Page Table Example 1

What is the physical address of virtual address **0x5F20**?

V	Physical Page Number
0	
0	
1	0x0000
1	0x7FFE
0	
0	
	⋮
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

Page Table



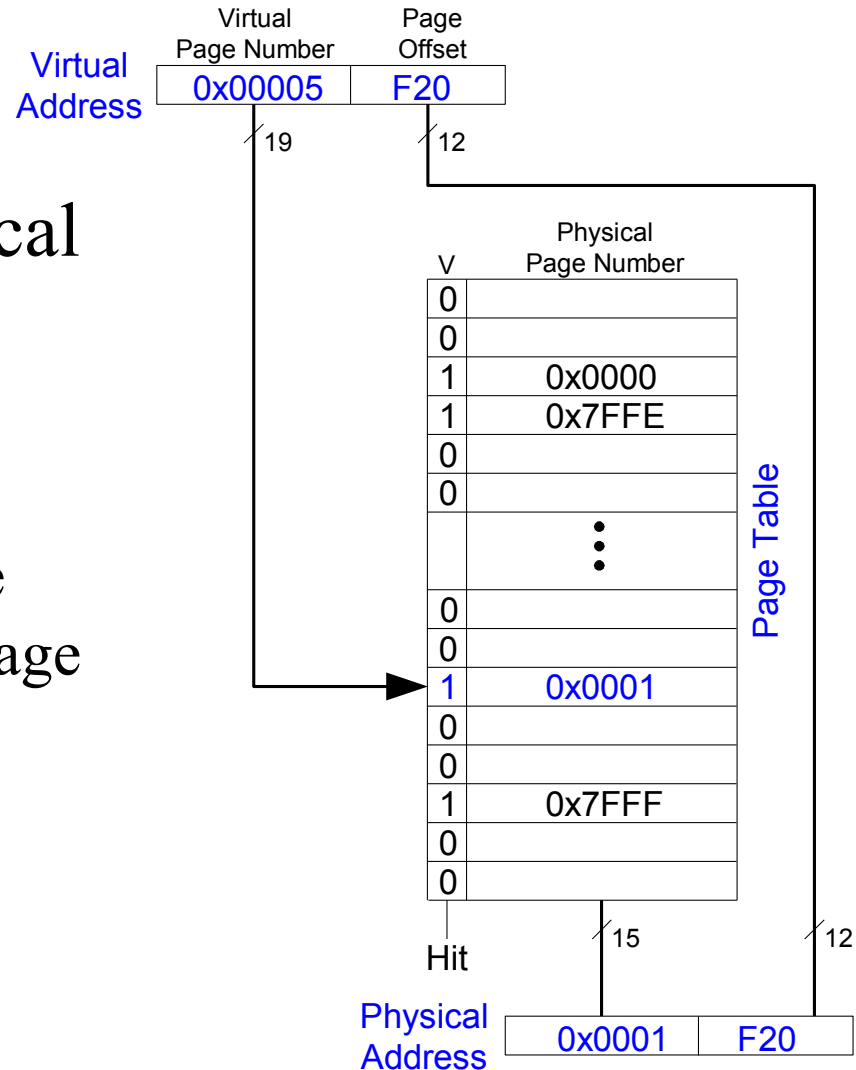
# Page Table Example 1

What is the physical address of virtual address **0x5F20**?

-VPN = **5**

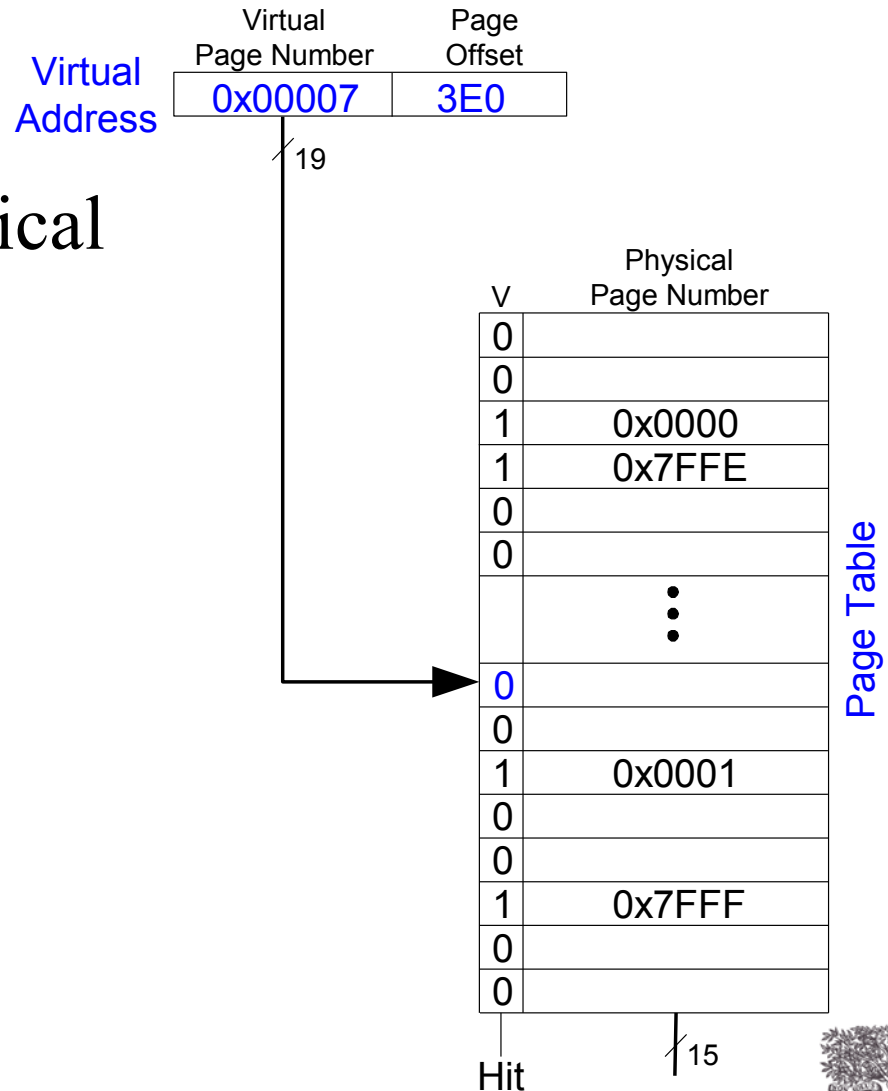
-Entry 5 in page table  
VPN 5 => physical page **1**

-Physical address:  
**0x1F20**



# Page Table Example 2

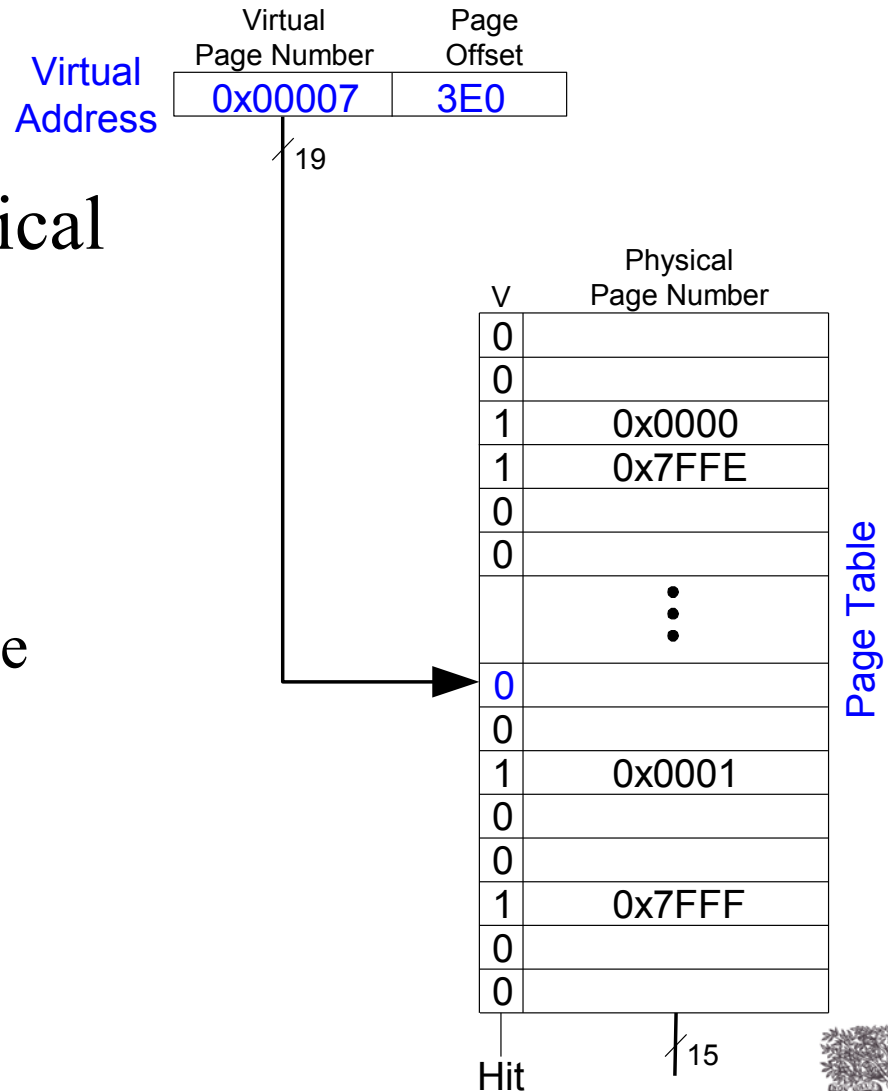
What is the physical address of virtual address **0x73E0**?



# Page Table Example 2

What is the physical address of virtual address **0x73E0**?

- VPN = 7
- Entry 7 is invalid
- Virtual page must be *paged* into physical memory from disk



# Page Table Challenges

- **Page table is large**
  - usually located in physical memory
- Load/store requires 2 main memory accesses:
  - one for translation (page table read)
  - one to access data (after translation)
- Cuts memory performance in half
  - *Unless we get clever...*

# Translation Lookaside Buffer (TLB)

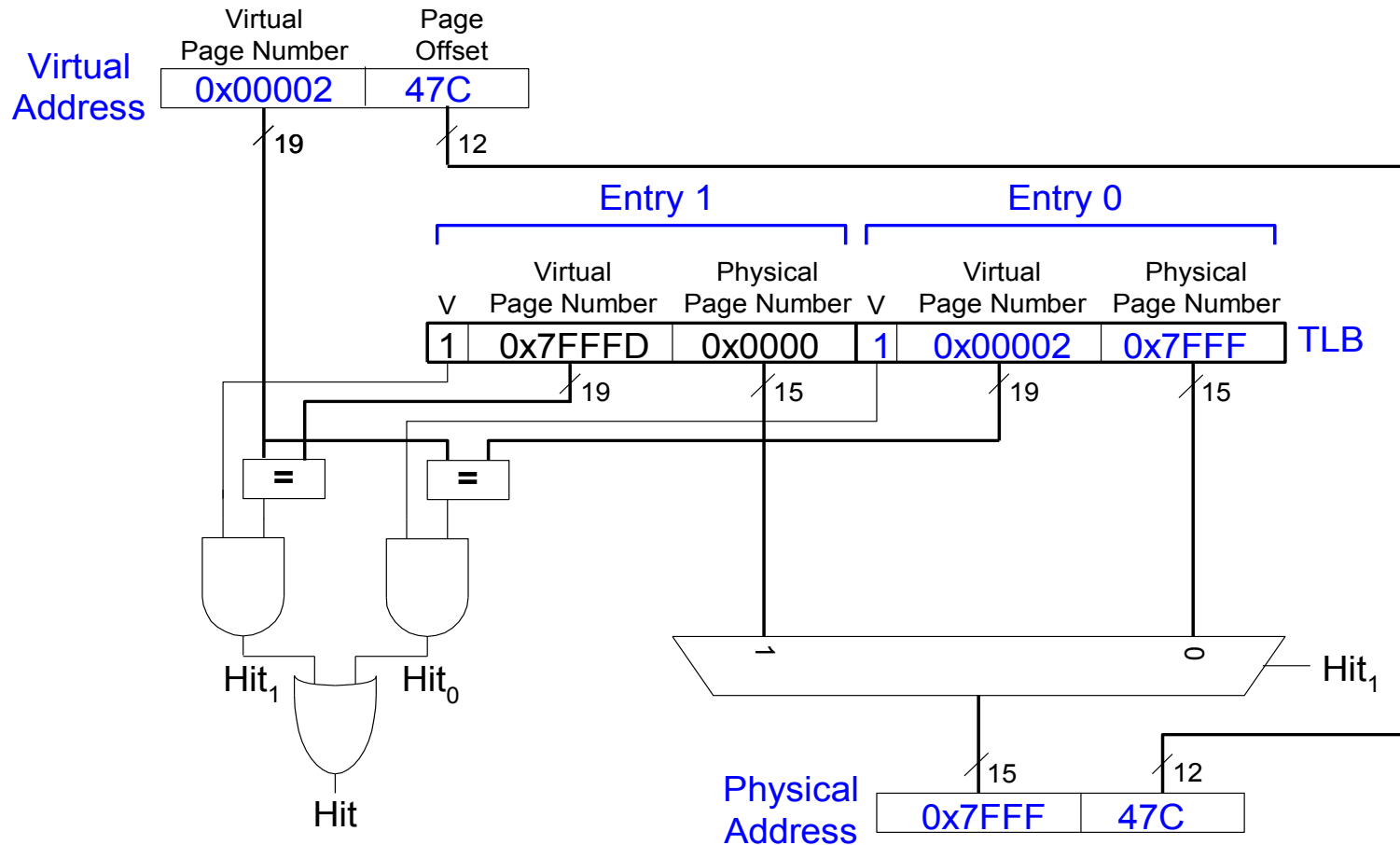
- Small cache of most recent translations
- Reduces # of memory accesses for *most* loads/stores from 2 to 1



# TLB

- Page table accesses: high temporal locality
  - Large page size, so consecutive loads/stores likely to access same page
- TLB
  - Small: accessed in  $< 1$  cycle
  - Typically 16 - 512 entries
  - Fully associative
  - $\rightarrow$  99 % hit rates typical
  - Reduces # of memory accesses for most loads/stores from 2 to 1

# Example 2-Entry TLB



# Memory Protection

- Multiple processes (programs) run at once
- Each process has its own page table
- Each process can use entire virtual address space
- A process can only access physical pages mapped in its own page table

# Virtual Memory Summary

- Virtual memory increases **capacity**
- A subset of virtual pages in physical memory
- **Page table** maps virtual pages to physical pages
  - address translation
- A **TLB** speeds up address translation
- Different page tables for different programs provides **memory protection**