



### Continuous Assessment Test – I

**Programme Name & Branch:B.Tech & CSE**

**Slot: G1+TG1**

**Course Code: CSE 2001**

**Exam Duration: 90 mins**

**Course Title: Computer Architecture and Organization**

**Maximum Marks: 50**

**General instruction(s): Answer All Question (5 \* 10 = 50)**

1. Consider two different machines, with two different instruction sets, both of which have a clock rate of 40 MHz processor.

Instruction Type	Instruction Count	Cycles per Instruction
<b>Machine A</b>		
Integer arithmetic	45000	1
Data transfer	32000	2
Floating point	15000	2
Control transfer	8000	2
<b>Machine B</b>		
Integer arithmetic	48000	1
Data transfer	45000	2
Floating point	15000	4
Control transfer	8000	3

- (a) Determine the effective CPI, MIPS rate, and execution time for this program.  
 (b) Comment on the results.

Solution

(a)

**Machine A:**

$CPI_A=1.55$ ,  $MIPS_A=25.8$ , Execution Time= $CPU_A=3.87ms$

**Machine B:**

$CPI_B=1.91$ ,  $MIPS_B=20.94$ , Execution Time= $CPU_B=5.5ms$

(b) Machine A has higher MIPS rate and it requires higher execution time.

Machine B has lower MIPS rate and it requires lesser execution time.

2. Illustrate expanded structure of IAS computer with neat diagram and also provide description for the following symbolic representations (i) LOAD M(X), (ii) SUB M(X), (iii) JUMP M(X,0:19), (iv) JUMP + M(X,20:39).

Expanded Structure of IAS Computer

**1. Memory buffer register (MBR):-**

Contains a word to be stored in memory or sent to the I/O unit, it is used to receive a word from memory or from the I/O unit.

**2. Memory address Register (MAR):-**

Specifies the address of memory in the word to be written from or read into the MBR.

**3. Instruction register (IR):-** Contains the 8bit Op-code instruction being executed.

**4. Instruction buffer register (IBR):-**

Employed to hold temporarily the right-hand instruction from a word in memory.

**5. Program counter (PC):-**

Contains the address of the next instruction-pair to be fetched from memory.



### **6. Accumulator(AC) and multiplier quotient(MQ):-**

Employed to hold temporarily operands and results of ALU operations. For example, the result of multiplying two 40-bits numbers is an 80bit number, the most significant 40 bits are stored in the AC and the least significant in the MQ.

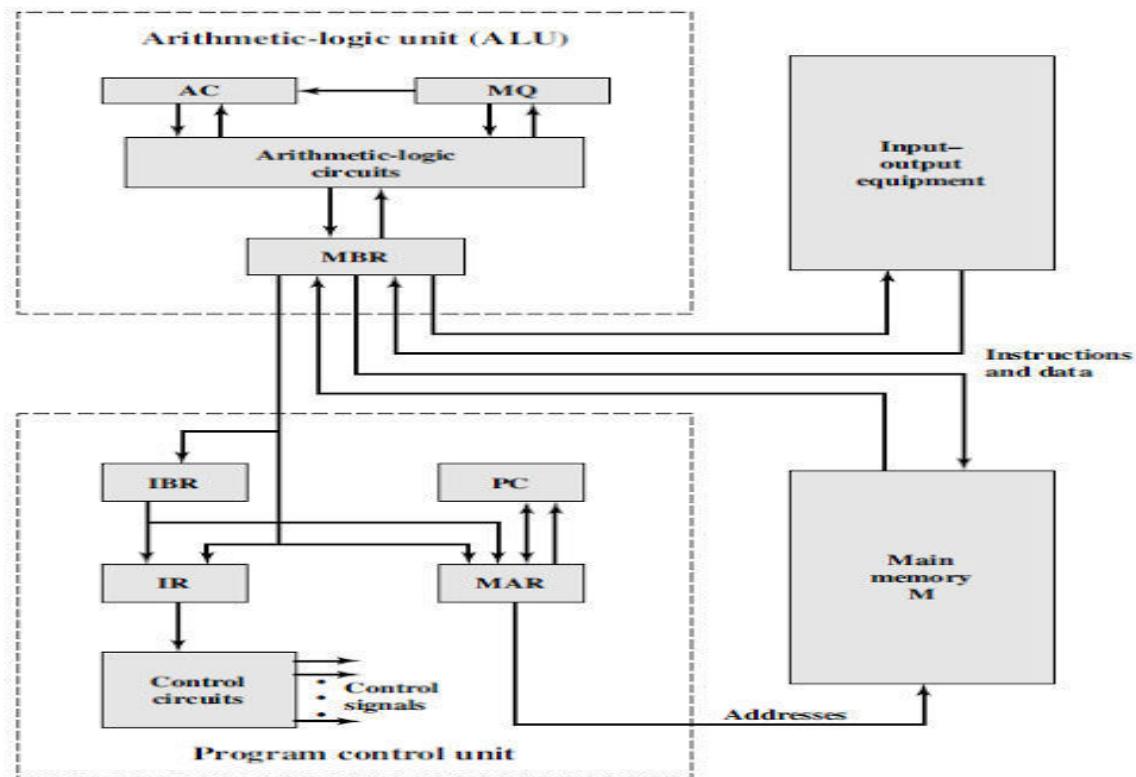


Figure : Expanded Structure of IAS Computer

- (i) LOAD M(X) - Transfer M(X) to the accumulator
- (ii) SUB M(X) - Subtract M(X) from AC; put the result in AC
- (iii) JUMP M(X,0:19) -If number in the accumulator is nonnegative, take next instruction from left half of M(X)
- (iv) JUMP + M(X,20:39)- If number in the accumulator is nonnegative, take next instruction from right half of M(X)

3. Consider the multiplier  $x=0111$  and multiplicand  $y=1011$  in two's complement notation, compute the product of  $p=x*y$  with Booth's algorithm and describe it with flow chart representation.

Ans:

$$x = 0111 = 7$$

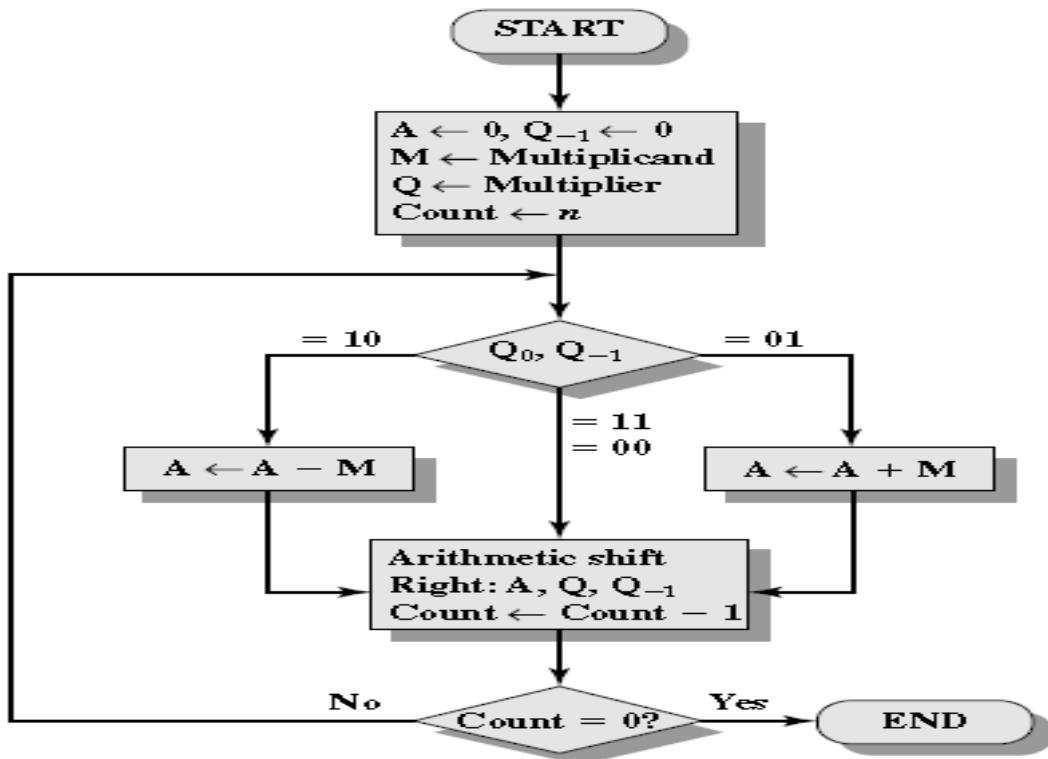
$$y = 1011 = -5$$

$$\text{Ans: } -35$$

$$\text{Product} = 11011101$$

$$00100010 = 1s \text{ Complement}$$

$$\begin{array}{r} + \quad \quad 1 \\ \hline 00100011 = 35 \end{array}$$



4. Write a program to evaluate the arithmetic statement:

$$X = (A + B) * (T + Q)$$

- a. Using a general register computer with three address instructions.
- b. Using a general register computer with two address instructions.
- c. Using an accumulator type computer with one address instructions.
- d. Using a stack organized computer with Zero-address operation instructions.

**a. Three-Address Instructions**

ADD R1, A, B	R1 ← M [A] + M [B]
ADD R2, T, Q	R2 ← M [T] + M [Q]
MUL X, R1, R2	M [X] ← R1 * R2

**b. Two-Address Instructions**

MOV R1, A	R1 ← M [A]
ADD R1, B	R1 ← R1 + M [B]
MOV R2, T	R2 ← M [T]
ADD R2, Q	R2 ← R2 + M [Q]
MUL R1, R2	R1 ← R1 * R2
MOV X, R1	M [X] ← R1

**c. one address instructions**

LOAD A	AC ← M [A]
ADD B	AC ← AC + M [B]
STORE G	M [G] ← AC
LOAD T	AC ← M [T]
ADD Q	AC ← AC + M [Q]
MUL G	AC ← AC * M [G]
STORE X	M [X] ← AC

**d. Zero Address Instruction**

PUSH A	TOS ← A
PUSH B	TOS ← B
ADD	TOS ← (A + B)
PUSH T	TOS ← T



PUSH Q	TOS $\leftarrow$ Q
ADD	TOS $\leftarrow$ (T + Q)
MUL	TOS $\leftarrow$ (T + Q) * (A + B)
POP X	M [X] $\leftarrow$ TOS

5. The two-word instruction at address 300 and 301 is load to Accumulator with address field equal to 400. Program Counter has the value 300 for fetching the instruction. The content of processor register is 500 and the content of index register is 202. The memory content at each of these addresses as shown in Table.

Address	Memory	Address	Memory
300	Load to AC	500	700
301	400	600	250
302	Next Instruction	601	390
400	600	602	900
499	450	700	825

Compute the effective address and operand for the following addressing modes:  
 Direct address, Indirect address, Index address, Relative address, Autoincrement and Autodecrement.

	Effective Address	Operand
Direct address	400	600
Indirect address	600	250
Index address	602	900
Relative address	702	
Autoincrement	500	700
Autodecrement	499	450



### Continuous Assessment Test – I

**Programme Name & Branch:B.Tech & CSE**

**Slot: G1+TG1**

**Course Code: CSE 2001**

**Exam Duration: 90 mins**

**Course Title: Computer Architecture and Organization**

**Maximum Marks: 50**

**General instruction(s): Answer All Question (5 \* 10 = 50)**

1. Consider two different machines, with two different instruction sets, both of which have a clock rate of 40 MHz processor.

Instruction Type	Instruction Count	Cycles per Instruction
<b>Machine A</b>		
Integer arithmetic	45000	1
Data transfer	32000	2
Floating point	15000	2
Control transfer	8000	2
<b>Machine B</b>		
Integer arithmetic	48000	1
Data transfer	45000	2
Floating point	15000	4
Control transfer	8000	3

- (a) Determine the effective CPI, MIPS rate, and execution time for this program.  
 (b) Comment on the results.

Solution

(a)

**Machine A:**

$CPI_A=1.55$ ,  $MIPS_A=25.8$ , Execution Time= $CPU_A=3.87ms$

**Machine B:**

$CPI_B=1.91$ ,  $MIPS_B=20.94$ , Execution Time= $CPU_B=5.5ms$

(b) Machine A has higher MIPS rate and it requires higher execution time.

Machine B has lower MIPS rate and it requires lesser execution time.

2. Illustrate expanded structure of IAS computer with neat diagram and also provide description for the following symbolic representations (i) LOAD M(X), (ii) SUB M(X), (iii) JUMP M(X,0:19), (iv) JUMP + M(X,20:39).

Expanded Structure of IAS Computer

**1. Memory buffer register (MBR):-**

Contains a word to be stored in memory or sent to the I/O unit, it is used to receive a word from memory or from the I/O unit.

**2. Memory address Register (MAR):-**

Specifies the address of memory in the word to be written from or read into the MBR.

**3. Instruction register (IR):-** Contains the 8bit Op-code instruction being executed.

**4. Instruction buffer register (IBR):-**

Employed to hold temporarily the right-hand instruction from a word in memory.

**5. Program counter (PC):-**

Contains the address of the next instruction-pair to be fetched from memory.

### **6. Accumulator(AC) and multiplier quotient(MQ):-**

Employed to hold temporarily operands and results of ALU operations. For example, the result of multiplying two 40-bits numbers is an 80bit number, the most significant 40 bits are stored in the AC and the least significant in the MQ.

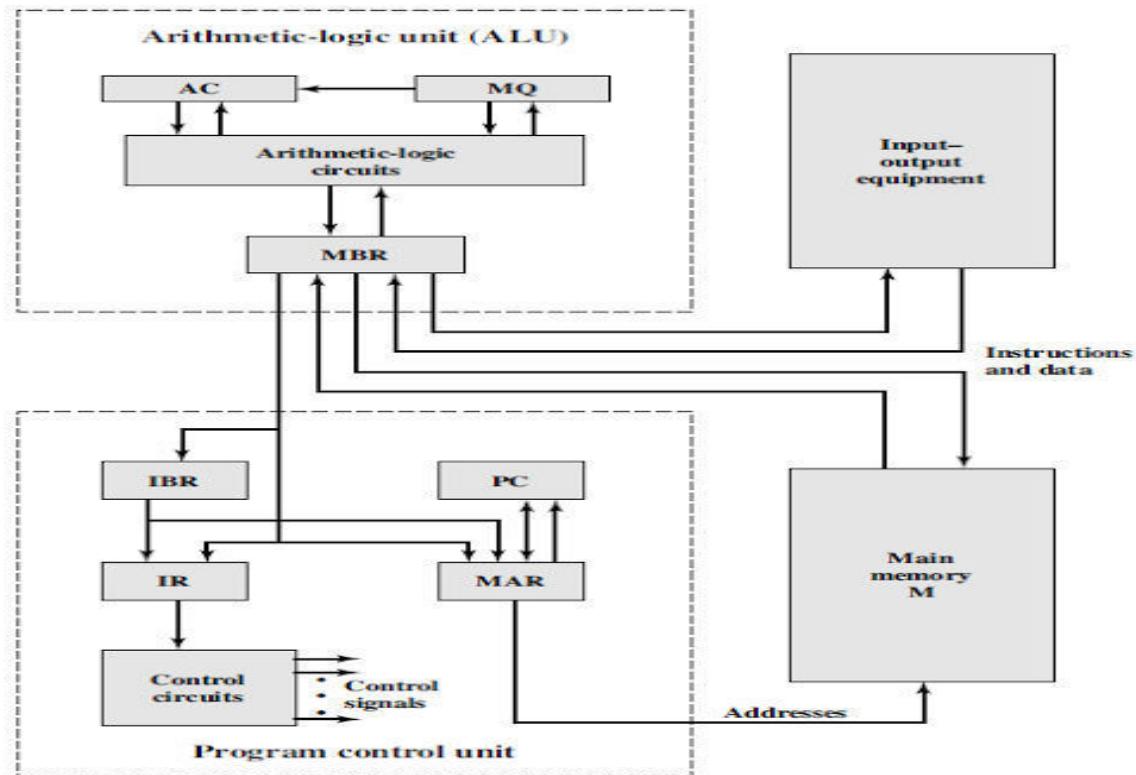


Figure : Expanded Structure of IAS Computer

- LOAD M(X) - Transfer M(X) to the accumulator**
- SUB M(X) - Subtract M(X) from AC; put the result in AC**
- JUMP M(X,0:19) -If number in the accumulator is nonnegative, take next instruction from left half of M(X)**
- JUMP + M(X,20:39)- If number in the accumulator is nonnegative, take next instruction from right half of M(X)**

3. Consider the multiplier  $x=0111$  and multiplicand  $y=1011$  in two's complement notation, compute the product of  $p=x*y$  with Booth's algorithm and describe it with flow chart representation.

Ans:

$$x = 0111 = 7$$

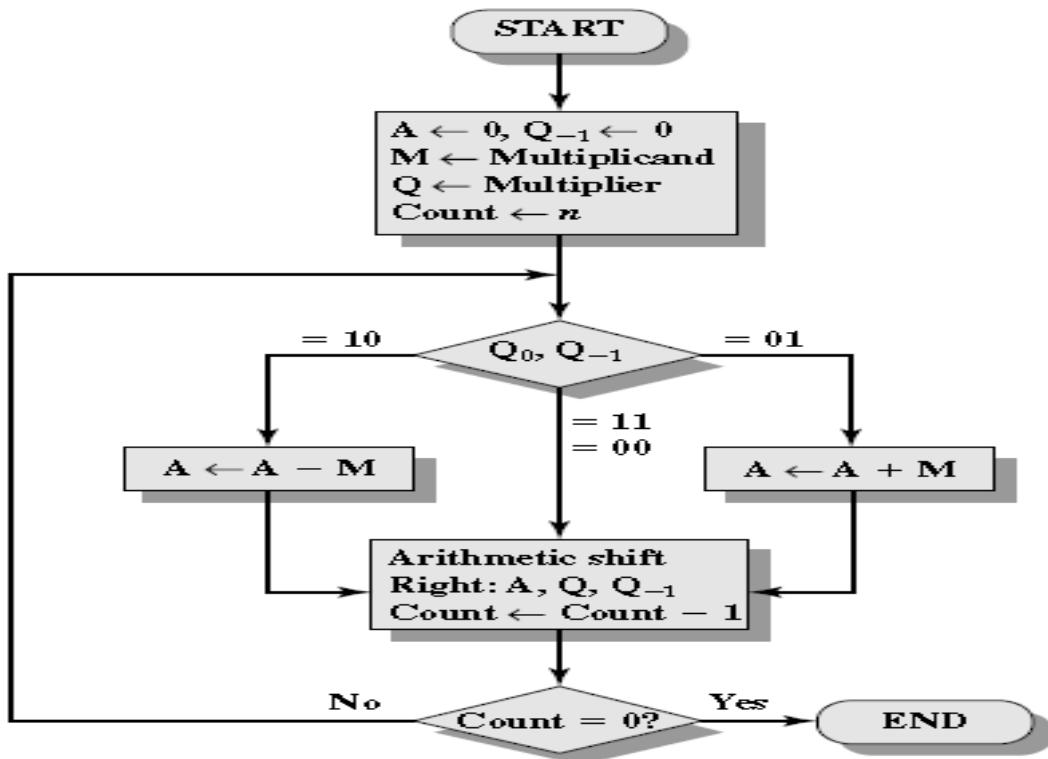
$$y = 1011 = -5$$

$$\text{Ans: } -35$$

$$\text{Product} = 11011101$$

$$00100010 = 1s \text{ Complement}$$

$$\begin{array}{r} + \quad \quad 1 \\ \hline 00100011 = 35 \end{array}$$



4. Write a program to evaluate the arithmetic statement:

$$X = (A + B) * (T + Q)$$

- a. Using a general register computer with three address instructions.
- b. Using a general register computer with two address instructions.
- c. Using an accumulator type computer with one address instructions.
- d. Using a stack organized computer with Zero-address operation instructions.

**a. Three-Address Instructions**

ADD R1, A, B	R1 ← M [A] + M [B]
ADD R2, T, Q	R2 ← M [T] + M [Q]
MUL X, R1, R2	M [X] ← R1 * R2

**b. Two-Address Instructions**

MOV R1, A	R1 ← M [A]
ADD R1, B	R1 ← R1 + M [B]
MOV R2, T	R2 ← M [T]
ADD R2, Q	R2 ← R2 + M [Q]
MUL R1, R2	R1 ← R1 * R2
MOV X, R1	M [X] ← R1

**c. one address instructions**

LOAD A	AC ← M [A]
ADD B	AC ← AC + M [B]
STORE G	M [G] ← AC
LOAD T	AC ← M [T]
ADD Q	AC ← AC + M [Q]
MUL G	AC ← AC * M [G]
STORE X	M [X] ← AC

**d. Zero Address Instruction**

PUSH A	TOS ← A
PUSH B	TOS ← B
ADD	TOS ← (A + B)
PUSH T	TOS ← T



PUSH Q	TOS $\leftarrow$ Q
ADD	TOS $\leftarrow$ (T + Q)
MUL	TOS $\leftarrow$ (T + Q) * (A + B)
POP X	M[X] $\leftarrow$ TOS

5. The two-word instruction at address 300 and 301 is load to Accumulator with address field equal to 400. Program Counter has the value 300 for fetching the instruction. The content of processor register is 500 and the content of index register is 202. The memory content at each of these addresses as shown in Table.

Address	Memory	Address	Memory
300	Load to AC	500	700
301	400	600	250
302	Next Instruction	601	390
400	600	602	900
499	450	700	825

Compute the effective address and operand for the following addressing modes:  
 Direct address, Indirect address, Index address, Relative address, Autoincrement and Autodecrement.

	Effective Address	Operand
Direct address	400	600
Indirect address	600	250
Index address	602	900
Relative address	702	
Autoincrement	500	700
Autodecrement	499	450


**CAT II-G1-AK**
**School of Computer Science and Engineering**
**CSE2001-Computer Architecture and Organization**
**Exam Duration: 90 minutes**
**Maximum Marks: 50**


---

**Answer all the questions**

1. A system is following Non-restoring algorithm for its division operation. Add sufficient changes to the Non-restoring algorithm to perform floating point division. Show the revised algorithm and division steps for the given data.

$$(10010 * 2^{-8}) / (00011 * 2^{-3}) \quad 10$$

**Ans:** Include the floating point addition steps with the non-restoring division algorithm.

	A	D	C
	00000	10010	5
SHL	00001	0010-	
A_M	11110	00100	4
SHL	11100	0100-	
A+M	11111	01000	3
SHL	11110	1000-	
A+M	00001	10001	2
SHL	00011	0001-	
A_M	00000	00011	1
SHL	00000	0011-	
A_M	11101	00110	0
A+M	00000	00110	

2. A) The processor needs to transfer a file of 32768 kilobytes from disk to main memory. The memory is byte addressable. The size of the data count register of a DMA controller is 16 bits. What is the minimum number of times the DMA controller needs to get the control of the system bus from the processor to transfer the file from the disk to main memory in the following transfer mode?

- a. Cycle stealing mode :  $32768 * 1024$  times  
 b. Block transfer/ Burst transfer mode :  $2^{15} * 2^{10} / 2^{16} = 512$  5

B) Burst mode of transfer make CPU idle for a long time, Cycle stealing mode request bus for each byte transfer and transparent mode need a complex circuitry to monitor and use bus when CPU is idle. But DMA is using to save processing time. How DMA achieve it? 5

Ans: working principle of DMA

3. A) Consider a computer system with 2 level caches. Access times of Level 1 cache, Level 2 cache and main memory are 1 ns, 10ns, and 1000ns, respectively. The hit ratio of Level 1 and Level 2 caches are 70% and 90%, respectively. What is the average access time of the system ignoring the search time within the cache? 5

$$\text{Ans: } (0.7 * 1) + (0.3(0.9 * 10)) + (0.3 * 0.1 * 1000) = 33.4 \text{ ns}$$

B) A computer employs RAM chips of  $256 \times 8$  and ROM chips of  $128 \times 8$ . The computer system needs 4K bytes of RAM, 1K x 16 of ROM, and two interface units with 256 registers each. Then, calculate the following,

- a. Number of RAM chips : 16 chips
- b. Number of ROM chips : 16 chips
- c. Value of x, y and z : 

	x	y	z
RAM	8	4	2
ROM	7	3	2
I/F	8	1	2

4. A) The 12bit data stored in the memory is 101110000000 and this data met with a single data bit error. Assume that the system followed the hamming code error for error check. Identify the error bit position and correct it. 5

Ans: Check bits in the received data = 1 0 0 0 → A

Data bits are 1 0 1 1 0 0 0 0

Check bits from the data bits = 1 1 1 1 → B

A X-OR B = 0111 = 7

7<sup>th</sup> position / D4 is in error

B) The main memory of a system has 32bit physical address space and the page size of virtual memory is 4Kbytes. The maximum size of the page table is 12MByte and each entry of page table has 1dirty bit, 1 valid bit, 2 permission bits and frame number. Calculate the size of virtual address. 5

Page size	= 4KB
Page tables size	= 12MB
Single page table entry size	= $1db + 1vb + 2pb + \text{no. of bits for frame number}$
No. of bits for frame number	= bits in physical Address – bits for addressing within a page
	= $32 - 12 = 20$
Single page table entry size	= $1+1+2+20 = 24$
No. of page table entries	= page table size / an entry size = $12MB / 24 \text{ bits} = (12 * 2^{20}) / 3 = 2^{22}$
Virtual address space	= No. of pagetable entry * page table size = $2^{22} * 2^{12}$ = $2^{34}$
Virtual address size	= 34 bits

5. A computer has the main memory with  $2^{32}$  bytes size and has the word size  $2^2$  bytes. The cache memory size is  $2^{20}$  bytes and each cache line is  $2^8$  words. What is the length of tag field of its physical address, for a 4-way set-associative cache memory? 10

Ans: x=32, w=2, y=20 m=8 N=4

No of bits in the address = X-W

No. of bits in the word field = M

No. of sets =  $2^{Y-M-W}/N$

No. of bits in the set field = Y-M-W-logN

No. of bits in the tag field = X-W-(Y-M-W-logN+M)

$$= X-W-Y+M+W+\log N-M$$

$$= X-Y+\log N$$

$$= 32-20+\log 4$$

$$= 14$$

\*\*\*\*\*

# Introduction

# Why this subject?

- To understand the functional components, characteristics, performance and interactions of a computer system.
- Need to understand computer architecture in order to structure a program so that it runs more efficiently on a real machine.
- To understand the tradeoff among various components such as CPU clock speed vs. memory size.

# Text Books

- W. Stallings, Computer organization and architecture, Prentice-Hall,2000
- M. M. Mano, Computer System Architecture, Prentice-Hall
- J. P. Hayes, Computer system architecture, McGraw Hill
- J. L. Hennessy & D.A. Patterson, Computer architecture: A quantitative approach, Fourth Edition, Morgan Kaufman, 2004.

# Evaluation procedure

- Quiz – (10M)
- CAT – 30M
- Assignment - ( $10 \times 2 = 20M$ )
- Term End Exam – 40M
- Additional learning – 10M

## Contact Information

Name :	Jasmin T. Jose
Designation :	Assistant Professor
Email :	<a href="mailto:jasminlijo@vit.ac.in"><u>jasminlijo@vit.ac.in</u></a>
Cabin :	SJT 411 A-33
Open Hours :	Mon [3.00 - 4.00 pm] Wed [02.00 - 03.00 pm]
Contact No:	9486096411 [ only 9.00 -5.00 pm]

# Introduction

- What is computer?
  - A **computer** is a programmable machine that receives input, stores and manipulates data, and provides output in a useful format.

# History of Computers

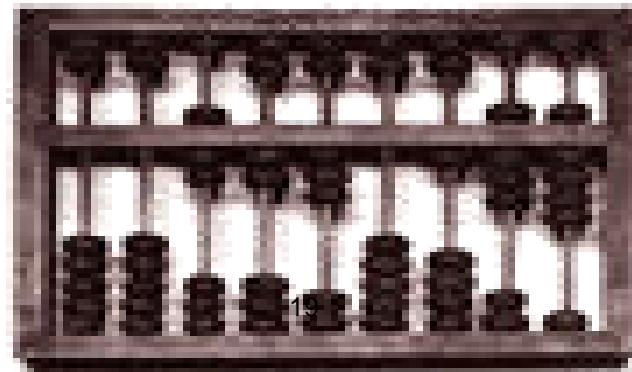
# ABACUS -4<sup>th</sup> Century B.C.

---

---

💡 The abacus, a simple counting aid, may have been invented in Babylonia (now Iraq) in the fourth century B.C.

💡 This device allows users to make computations using a system of sliding beads arranged on a rack.



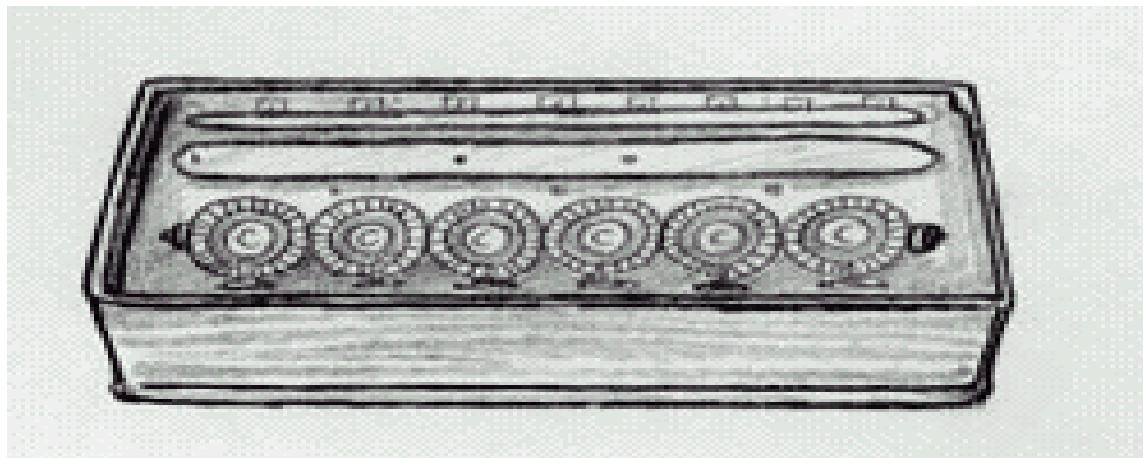
# BLAISE PASCAL

(1623 - 1662)

---

---

💡 In 1642, the French mathematician and philosopher Blaise Pascal invented a calculating device that would come to be called the "Adding Machine".



# CHARLES BABBAGE (1791 - 1871)

---

---

- 💡 Born in 1791, Charles Babbage was an English mathematician and professor.
- 💡 In 1822, he persuaded the British government to finance his design to build a machine that would calculate tables for logarithms.
- 💡 With Charles Babbage's creation of the "Analytical Engine", (1833) computers took the form of a general purpose machine.

# HOWARD AIKEN

## (1900 - 1973)

---

---

 Aiken thought he could create a modern and functioning model of Babbage's Analytical Engine.

 He succeeded in securing a grant of 1 million dollars for his proposed Automatic Sequence Calculator; the Mark I for short. From IBM.

 In 1944, the Mark I was "switched" on. Aiken's colossal machine spanned 51 feet in length and 8 feet in height. 500 meters of wiring were required to connect each component.

- The Mark I *did* transform Babbage's dream into reality and *did* succeed in putting IBM's name on the forefront of the burgeoning computer industry. From 1944 on, modern computers would forever be associated with digital intelligence.

# ENIAC 1946

---

---



## Electronic Numerical Integrator And Computer

Under the leadership of J. Presper Eckert (1919 - 1995) and John W. Mauchly (1907 - 1980) the team produced a machine that computed at speeds 1,000 times faster than the Mark I was capable of only 2 years earlier.

Using 18,000-19,000 vacuum tubes, 70,000 resistors and 5 million soldered joints this massive instrument required the output of a small power station to operate it.

It could do nuclear physics calculations (in two hours) which it would have taken 100 engineers a year to do by hand.

The system's program could be changed by rewiring a panel.

# ENIAC

1946



# TRANSISTOR 1948

---

---

💡 In the laboratories of Bell Telephone, John Bardeen, Walter Brattain and William Shockley discovered the "transfer resistor"; later labelled the transistor.

💡 Advantages:

- 💡 increased reliability
- 💡 1/13 size of vacuum tubes
- 💡 consumed 1/20 of the electricity of vacuum tubes
- 💡 were a fraction of the cost

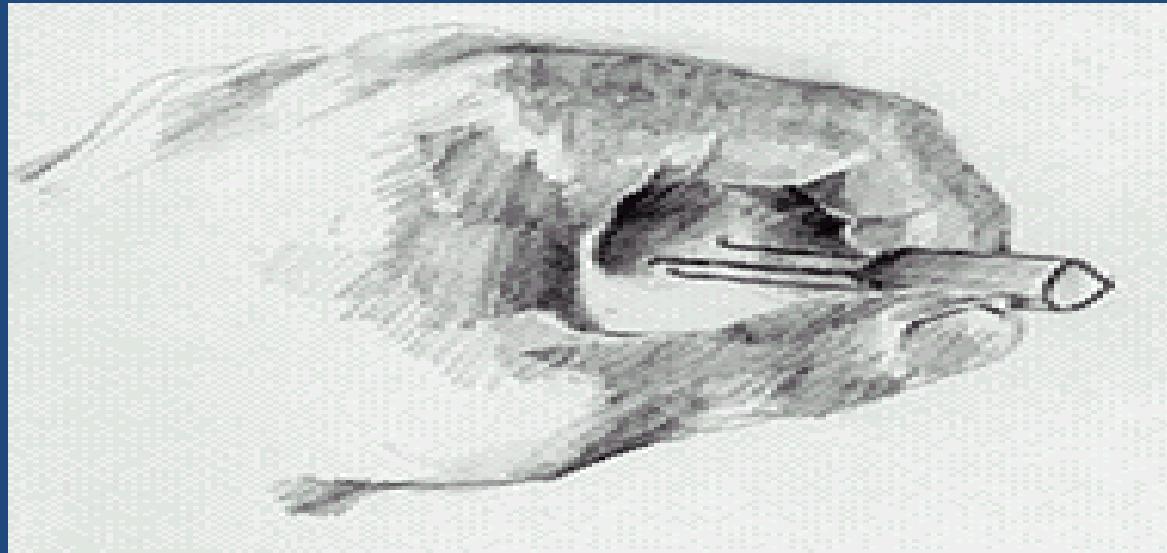
# TRANSISTOR

1948

---



This tiny device had a huge impact on and extensive implications for modern computers. In 1956, the transistor won its creators the Noble Peace Prize for their invention.



# ALTAIR 1975

---

---



The invention of the transistor made computers smaller, cheaper and more reliable. Therefore, the stage was set for the entrance of the computer into the domestic realm. In 1975, the age of personal computers commenced.



Under the leadership of Ed Roberts the Micro Instrumentation and Telemetry Company (MITS) wanted to design a computer 'kit' for the home hobbyist.

Based on the Intel 8080 processor, capable of controlling 64 kilobytes of memory, the MITS Altair - as the invention was later called - was debuted on the cover of the January edition of *Popular Electronics* magazine.

Presenting the Altair as an unassembled kit kept costs to a minimum. Therefore, the company was able to offer this model for only \$395. Supply could not keep up with demand.

# IBM (PC)

## 1981

---

---

💡 On August 12, 1981 IBM announced its own personal computer.

💡 Using the 16 bit Intel 8088 microprocessor, allowed for increased speed and huge amounts of memory.

💡 Unlike the Altair that was sold as unassembled computer kits, IBM sold its "ready-made" machine through retailers and by qualified salespeople.

# IBM (PC)

## 1981

---

---

 To satisfy consumer appetites and to increase usability, IBM gave prototype IBM PCs to a number of major software companies.

 For the first time, small companies and individuals who never would have imagined owning a "personal" computer were now opened to the computer world.

# Computer Generations

# FIRST GENERATION

## (1945-1956)



💡 First generation computers were characterized by the fact that operating instructions were made-to-order for the specific task for which the computer was to be used.

💡 Each computer had a different binary-coded program called a machine language that told it how to operate. This made the computer difficult to program and limited its versatility and speed.

💡 Other distinctive features of first generation computers were the use of vacuum tubes (responsible for their breathtaking size) and magnetic drums for data storage.

# First Generations

- Vacuum Tubes
- Magnetic Drum
- 4,000 bits
- Hard Wire Programs in computers
- IBM 650, Univac I
- ENIAC

# Vacuum Tubes



# SECOND GENERATION (1956-1963)



Throughout the early 1960's, there were a number of commercially successful second generation computers used in

- business,
- universities, and
- government from companies such as Burroughs, Control Data, Honeywell, IBM, Sperry-Rand, and others.

These second generation computers were also of solid state design, and contained transistors in place of vacuum tubes.<sup>34</sup>

# THIRD GENERATION

## (1965-1971)



Though transistors were clearly an improvement over the vacuum tube, they still generated a great deal of heat, which damaged the computer's sensitive internal parts.

The quartz rock eliminated this problem. Jack Kilby, an engineer with Texas Instruments, developed the integrated circuit (IC) in 1958.

The IC combined three electronic components onto a small silicon disc, which was made from quartz.

Scientists later managed to fit even more components on a single chip, called a semiconductor.

As a result, computers became ever smaller as more components were squeezed onto the chip. Another third-generation development included the use of an operating system that allowed machines to run many different programs at once with a central program that monitored and coordinated the computer's memory.

# FOURTH GENERATION

## (1971-Present)

---

---

-  In 1981, IBM introduced its personal computer (PC) for use in the home, office and schools.
-  The 1980's saw an expansion in computer use in all three arenas as clones of the IBM PC made the personal computer even more affordable.
-  The number of personal computers in use more than doubled from 2 million in 1981 to 5.5 million in 1982.

# FOURTH GENERATION

## (1971-Present)

---

---

 Ten years later, 65 million PCs were being used. Computers continued their trend toward a smaller size, working their way down from desktop to laptop computers (which could fit inside a briefcase) to palmtop (able to fit inside a breast pocket).

 In direct competition with IBM's PC was Apple's Macintosh line, introduced in 1984. Notable for its user-friendly design, the Macintosh offered an operating system that allowed users to move screen icons instead of typing instructions.

# FIFTH GENERATION

## (Future)

---

---

Many advances in the science of computer design and technology are coming together to enable the creation of fifth-generation computers.

Two such engineering advances are **parallel processing**, which replaces von Neumann's single central processing unit design with a system harnessing the power of many CPUs to work as one.

Another advance is **superconductor technology**, which allows the flow of electricity with little or no resistance, greatly improving the speed of information flow.

# FIFTH GENERATION

## (Future)

---

---

 Computers today have some attributes of fifth generation computers.

 For example, expert systems assist doctors in making diagnoses by applying the problem-solving steps a doctor might use in assessing a patient's needs.

 It will take several more years of development before expert systems are in widespread use.

# FIFTH GENERATION

## ➤ CHARACTERISTICS

- 1) Super large scale integrated chips.
- 2) They will have artificial intelligence.
- 3) They will be able to recognize image and graphs.
- 4) To be able to solve highly complex problem including decision making, logical reasoning.
- 5) More than one CPU for faster processing.
- 6) To work with natural language.

# Introduction

# Do U know?

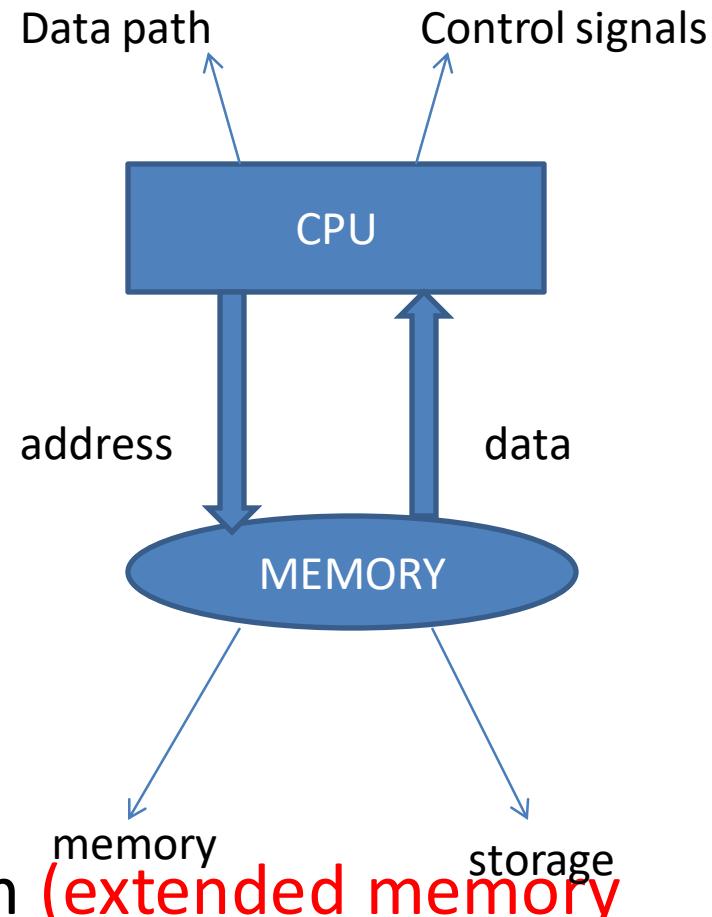
- Nobel prize related to computer system?
- Who is the latest prize winner?
- What is Moor's law?
- What is Principle of Equivalence of Hardware and Software?
- Who is Von Neumann?
- What is multiple core device?
- Is processor and core are differ?

# Fundamental Aspects

- Differentiate Computer Organization and Architecture.
- C.O:-
  - User's point of view
  - Study from s/w point of view
  - Eg. Car, How can be used? Drive?
- C.A:-
  - Designer's point of view
  - Study from h/w point of view
  - Eg. Car, How can be implemented? Repair?

# Root of computing

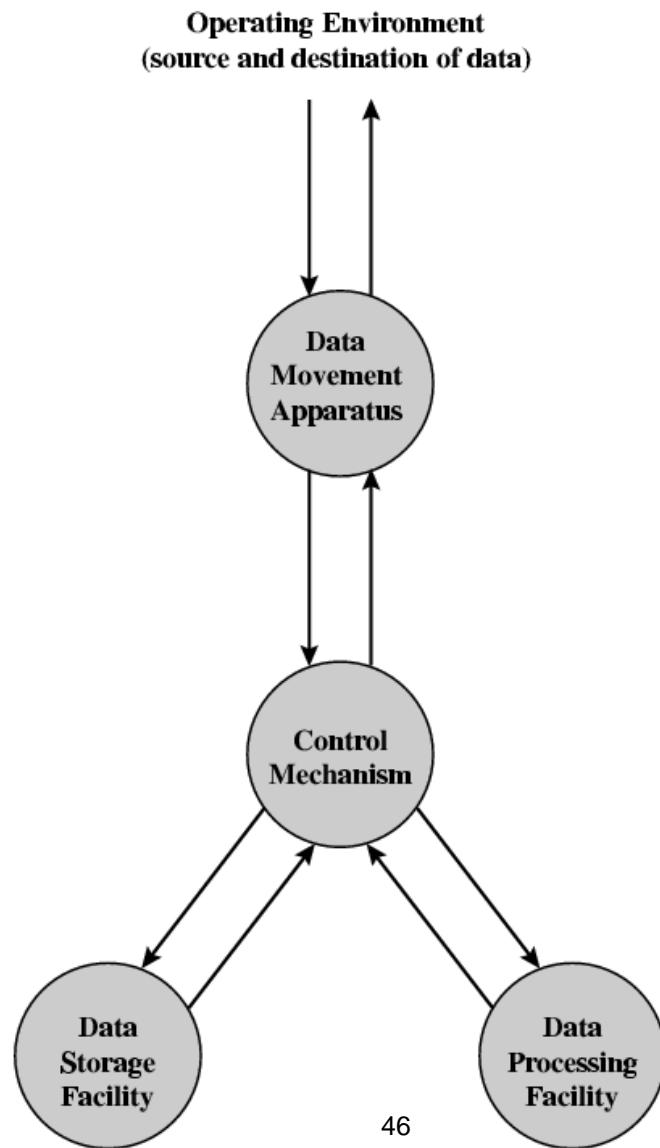
- Started with numbers.
- Processing the numbers.
- Any computing system,
  - Processor- process data
  - Memory- store data
  - I/O- user interacts with system (**extended memory for real data**)



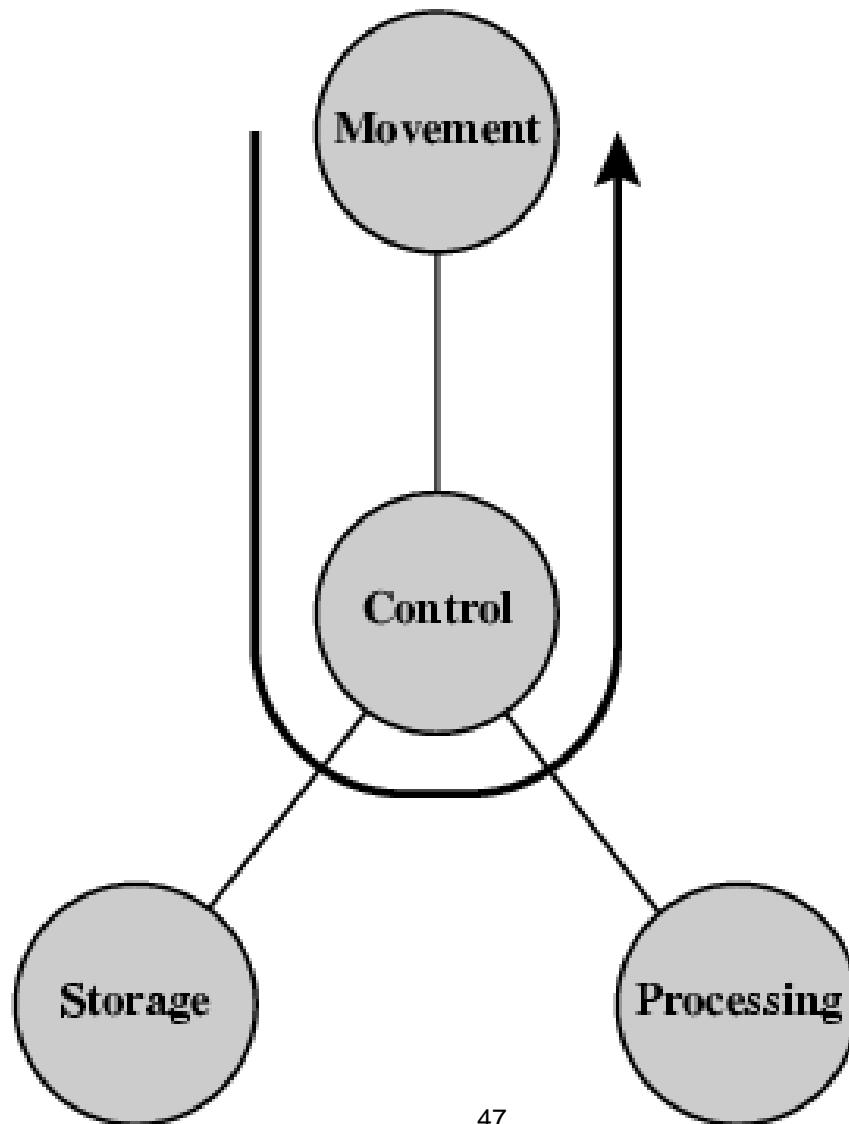
# Structure and Function

- **Structure**:- The way in which the components are interrelated.
  - CPU:- Registers, ALU, Control unit, Buses
  - Memory
  - I/O
- **Function**:- Function is the operation of individual components as part of the structure
  - Data processing
  - Data storage
  - Data movement
  - Control

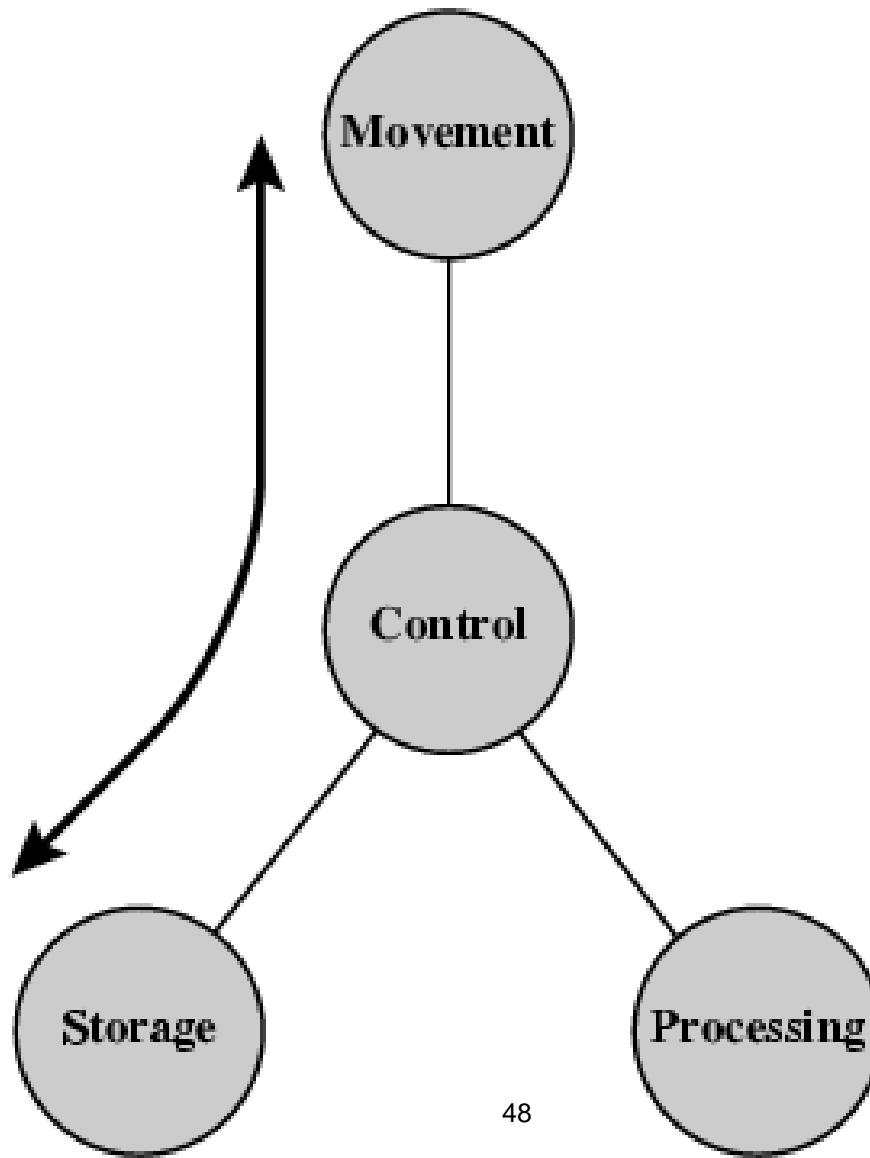
# Functional view



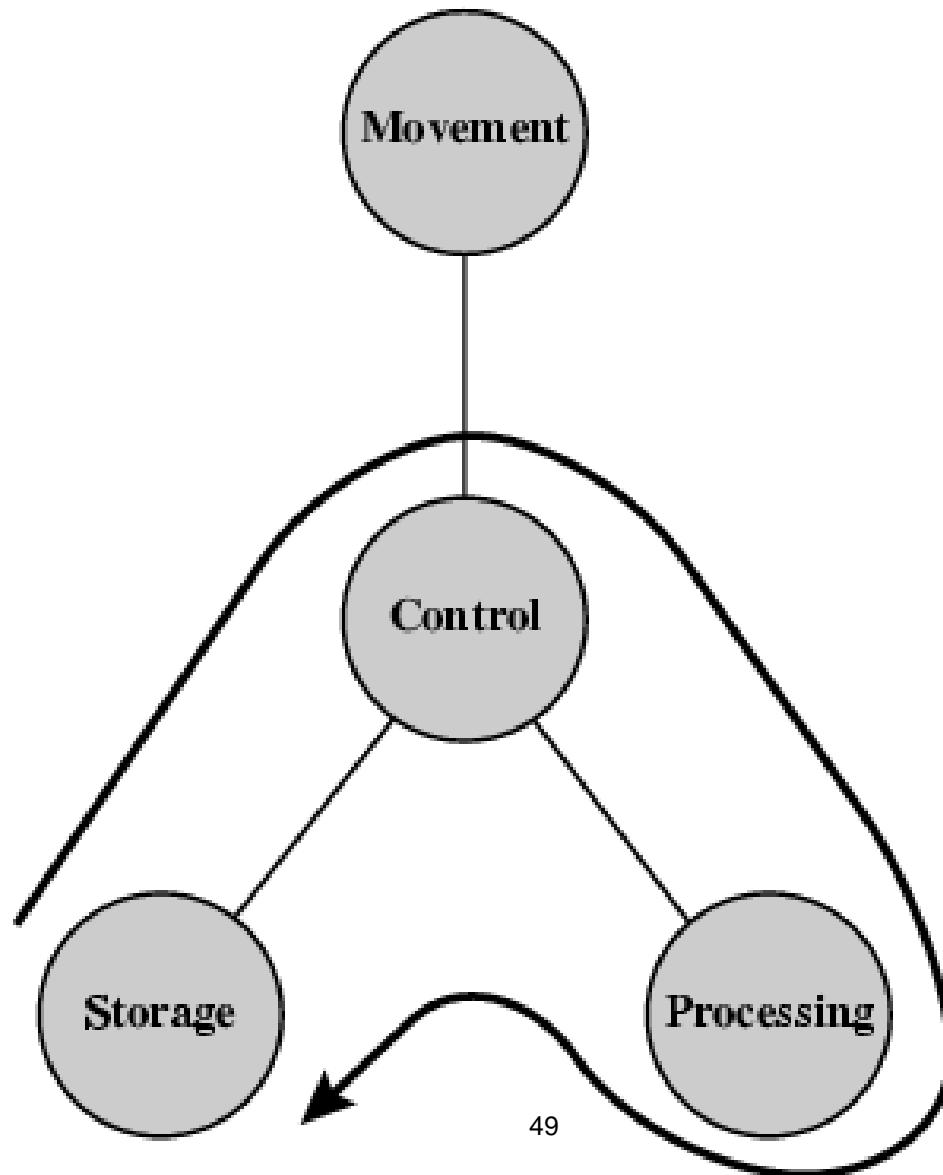
# Operations (1) Data movement



# Operations (2) Storage

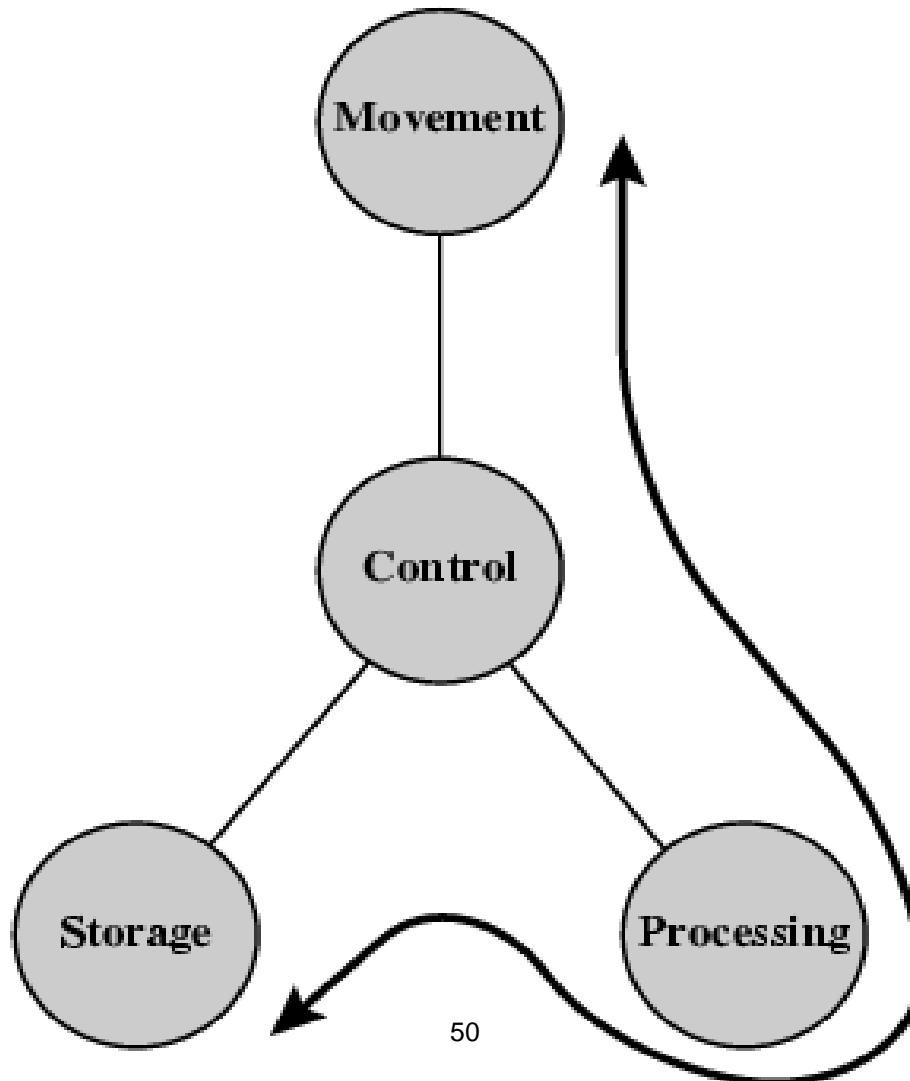


# Operation (3) Processing from/to

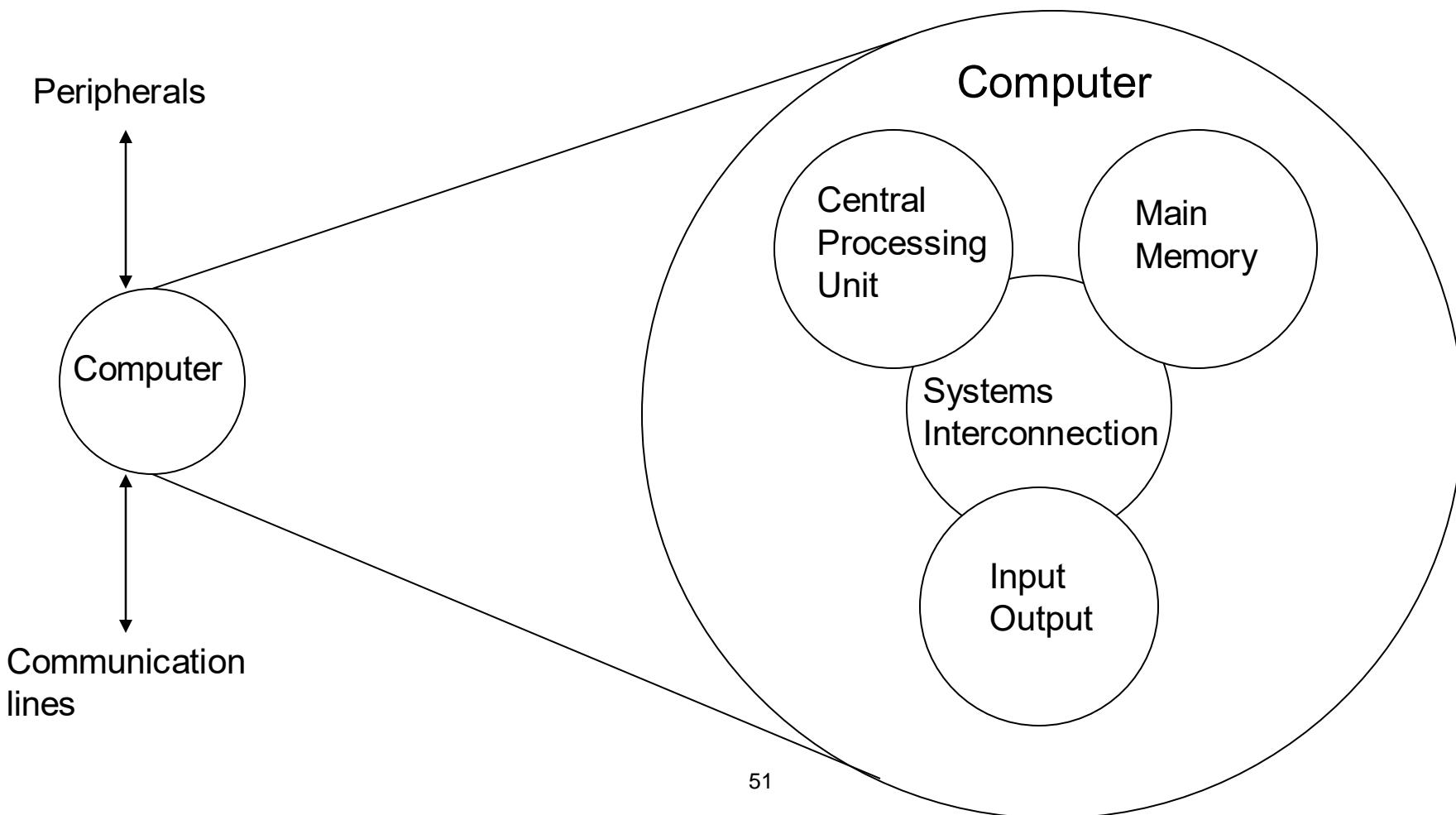


# Operation (4)

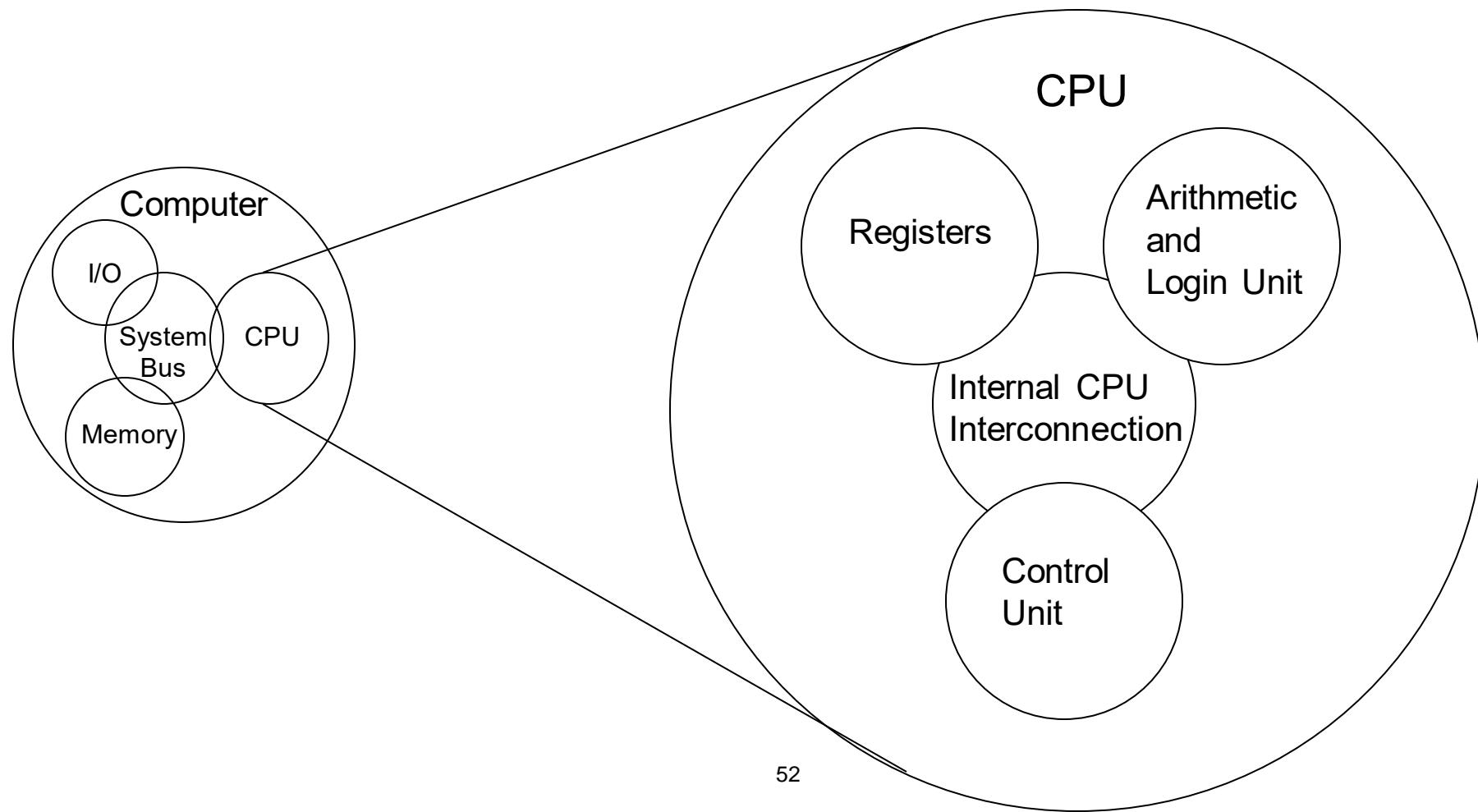
## Processing from storage to I/O



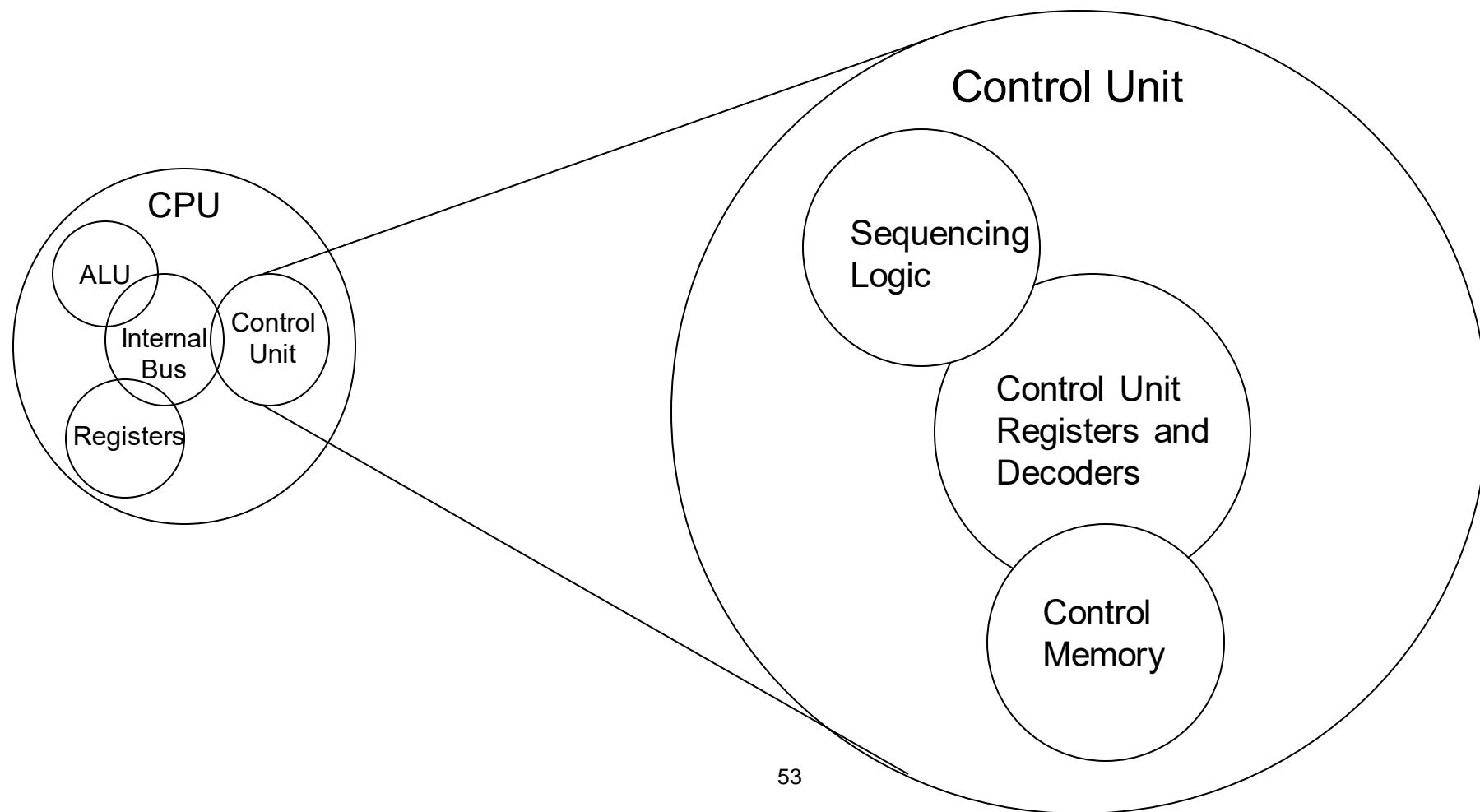
# Structure - Top Level



# Structure - The CPU

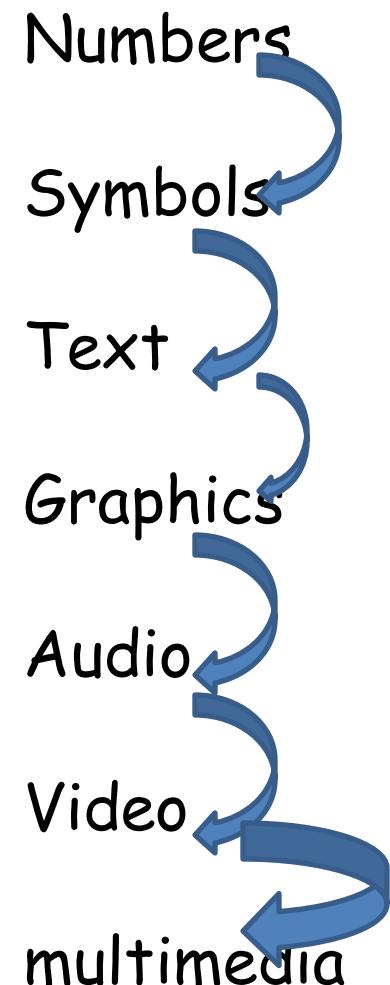


# Structure - The Control Unit



# Computing and Communicating

- Computing
  - Numbers → Arithmetic processing
  - Symbols → Logic processing
  - ALU → core of CPU.
- Communicating
  - Line communication
  - Voice communication
  - Voice + coded data

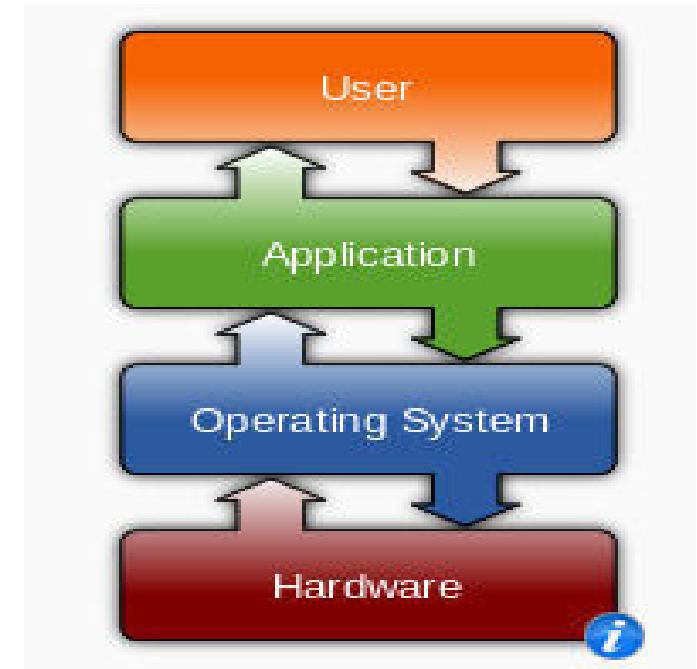


# H/W and S/W

- H/W:- core of the system, hard to understand.
  - S/W:- soften the hard aspects of the system.
  - LLL
  - ALL
  - HLL
  - Translator:- Compiler, Interpreter, Assembler
- ? What is device driver?
- ? Differentiate System S/W and Application S/W.  
Give some eg.
- ? Example for each translators.

# Operating System

- Manages computer hardware resources.
- Provide platform for Application software.
- It's a system software.



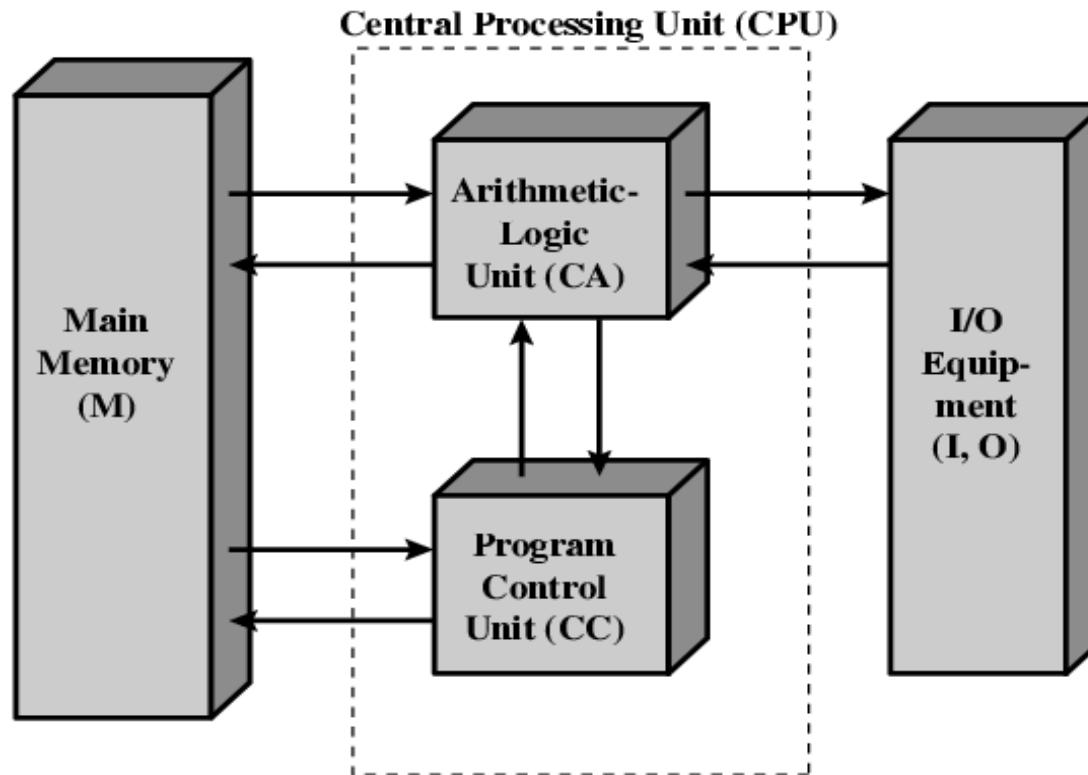
- Interface between application and the hardware.
- ? What are the popular modern operating systems?
- ? Is single processor with multiple OS possible?

# IAS Machine

# Organization of the von Neumann machine

**IAS** – a prototype developed by John Von Neumann in 1946 at Princeton University. (Institute for Advanced Study)

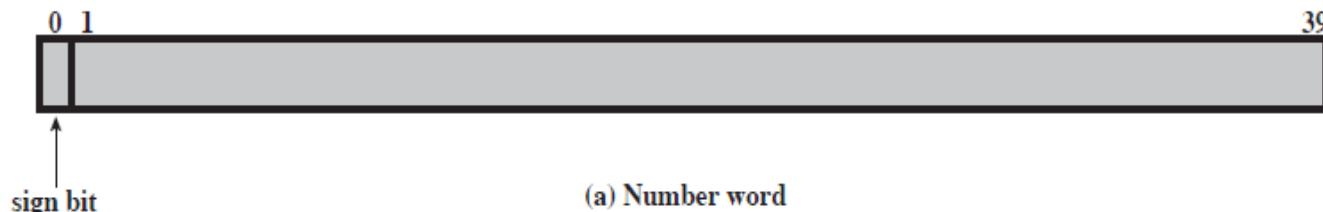
## Structure of Von Neumann machine(IAS Computer)



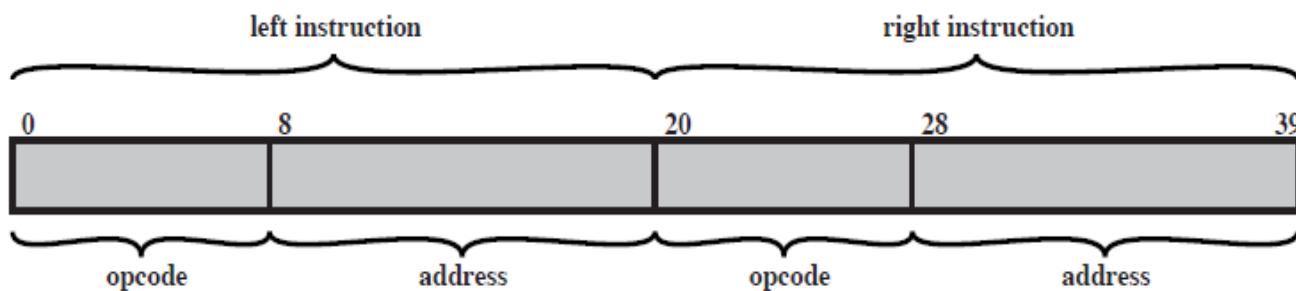
# IAS – Contd..

## Memory Formats

1



(a) Number word



(b) Instruction word

# IAS – memory format

- 1000 x 40 bit words ( 1000 storage locations of 40 binary bits each)
  - Binary number( both data and instructions are stored here)
- Number Format:
  - Each number is represented by a sign bit and a 39 bit value.
- Instruction Format:
  - A word may contain 20 bit instruction with each instruction consisting of an 8 bit operation code (opcode) specifying the operation to be performed and a 12 bit address designating one of the words in memory (0 to 999)

# IAS – Contd..

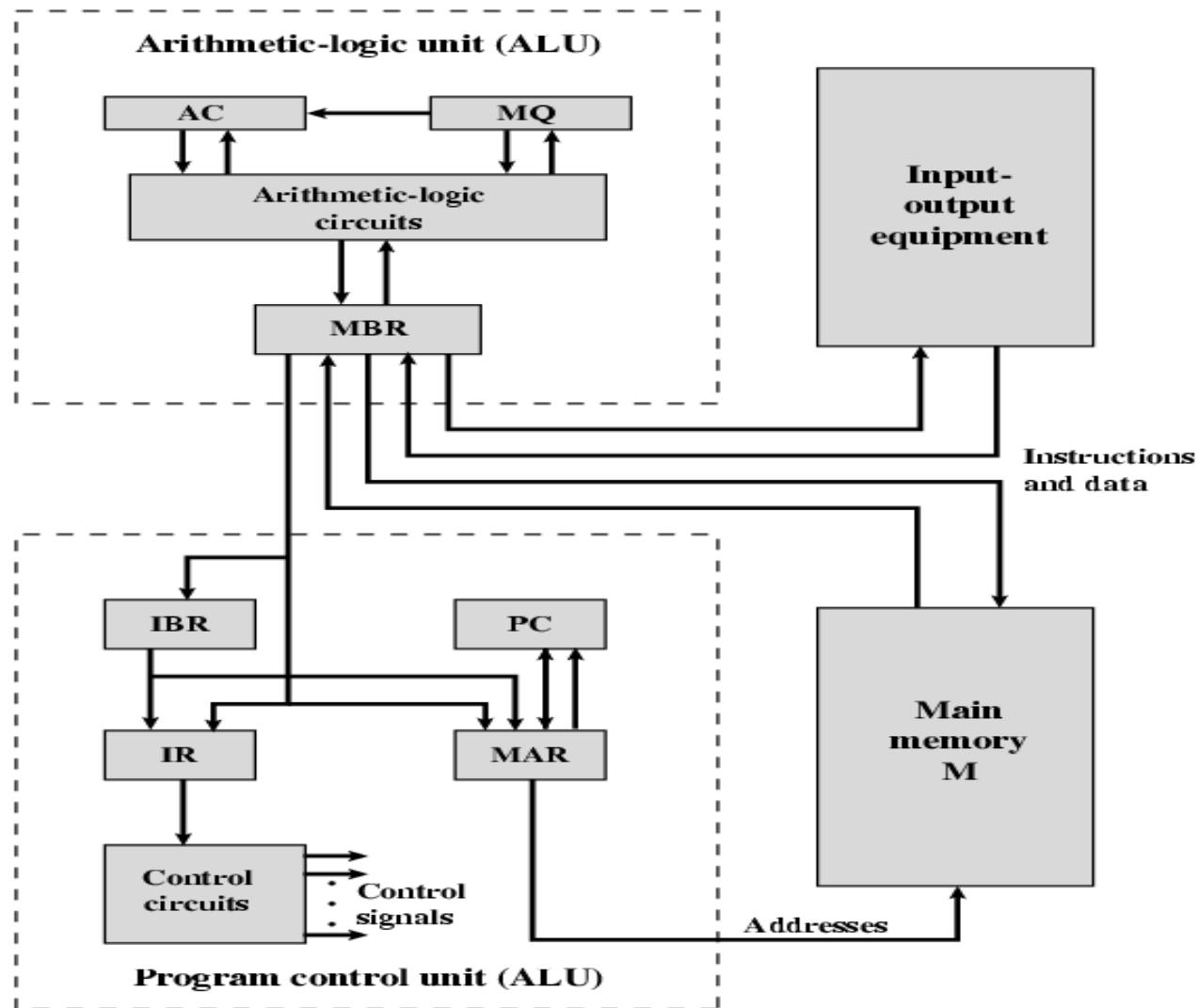
---

## IAS – Total 21 Instructions

- Data Transfer
- Unconditional Branch Instruction
- Conditional Branch Instruction
- Arithmetic
- Address Modify Instruction

# Expanded Structure of IAS

---



# Expanded structure of IAS computer

- **Set of registers (storage in CPU)**
  - **Memory Buffer Register (MBR)**
    - Contains a word to be stored in memory or sent to the I/O unit, or it is used to receive a word from memory or from the I/O unit.
  - **Memory Address Register (MAR)**
    - Specifies the address in memory of the word to be written from or read into the MBR.
  - **Instruction Register (IR)**
    - Contains the 8 bit opcode instruction being executed.
  - **Instruction Buffer Register (IBR)**
    - Employed to hold temporarily the right hand instruction from a word in memory
  - **Program Counter (PC)**
    - Contains the address of the next instruction pair to be fetched from memory
  - **Accumulator (AC) & Multiplier Quotient (MQ)**
    - Employed to hold temporarily the right hand instruction from a word in memory. For eg. The result of multiplying two 40 bit numbers is an 80 bit number, the most significant 40 bits are stored in the AC and the least significant in the MQ.

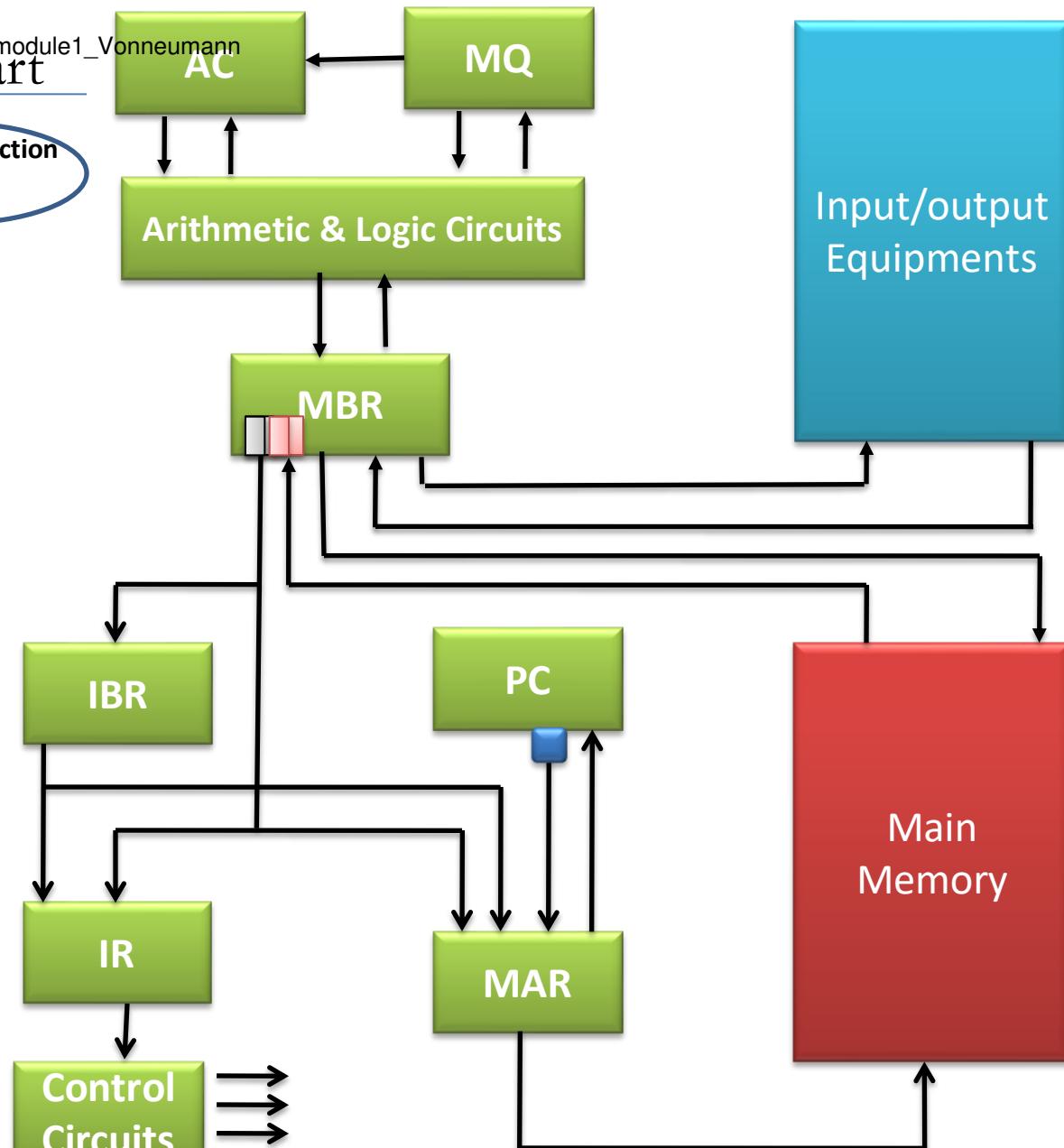
# IAS Operation - Flowchart

$\text{MAR} \leftarrow \text{PC}$   
 $\text{MBR} \leftarrow M[\text{MAR}]$   
 $\text{IBR} \leftarrow \text{MBR}[20..39]$   
 $\text{IR} \leftarrow \text{MBR}[0..7]$   
 $\text{MAR} \leftarrow \text{MBR}[8..19]$   
 $\text{MBR} \leftarrow M[\text{MAR}]$   
 $\text{AC} \leftarrow \text{MBR}$   
 $\text{IR} \leftarrow \text{IBR}[0..7]$   
 $\text{MAR} \leftarrow \text{IBR}[8..19]$   
  
 $\text{PC} \leftarrow \text{PC} + 1$   
 $\text{PC} \leftarrow \text{MAR}$   
 $\text{MBR} \leftarrow M[\text{MAR}]$   
 $\text{AC} \leftarrow \text{MBR}$

Is next instruction  
in IBR?

No

yes



MEMORY

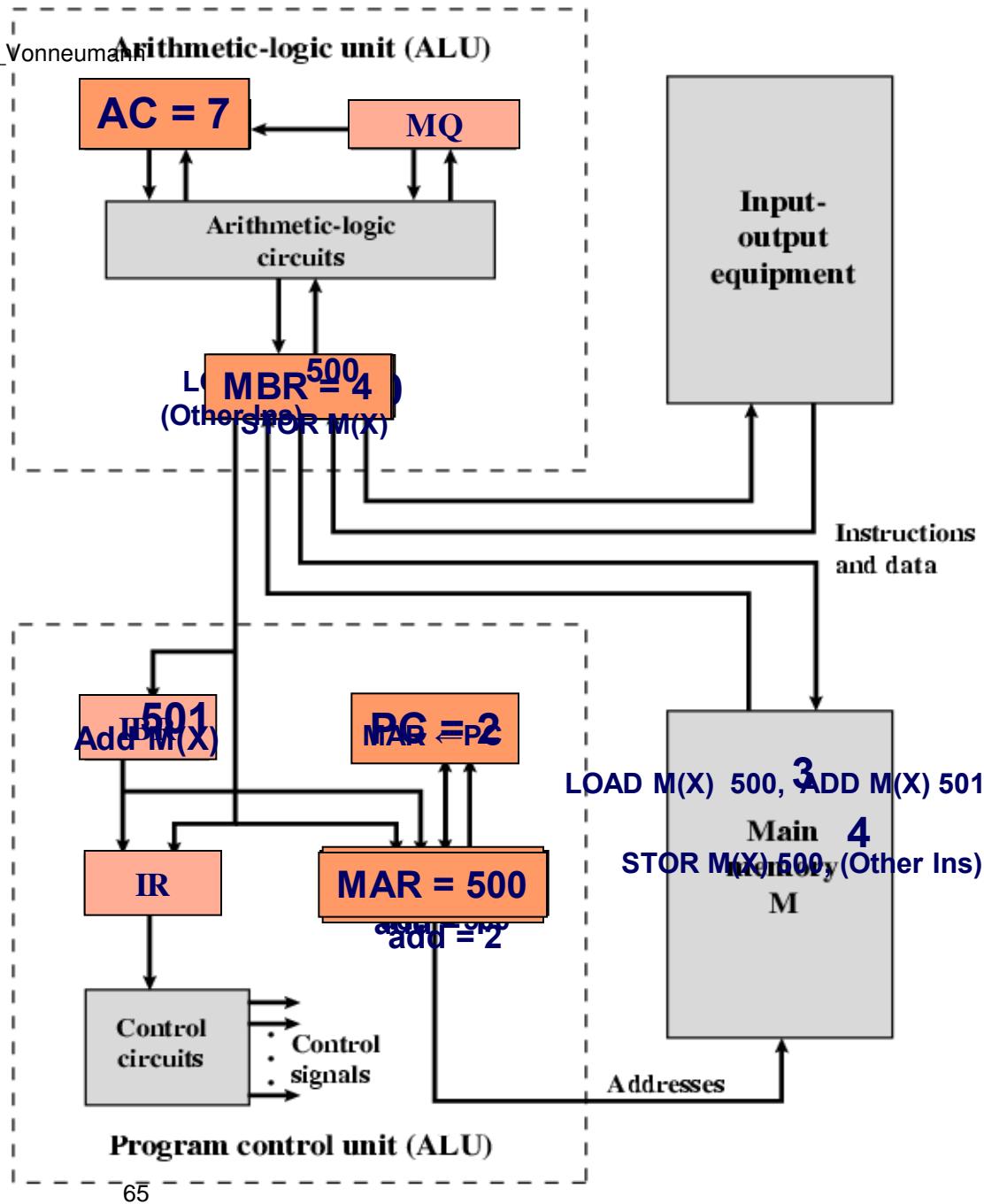
1. LOAD M(X) 500, ADD M(X) 501
  2. STOR M(X) 500, (Other Ins)

••••

500. 3

501. 4

<b>PC</b>	<b>2</b>
<b>MAR</b>	<b>500</b>
<b>MBR</b>	<b>STOR M(X) 500, (Other Ins)</b>
<b>IR</b>	<b>STOR M(X)</b>
<b>IBR</b>	<b>(Other Ins)</b>
<b>AC</b>	<b>7</b>

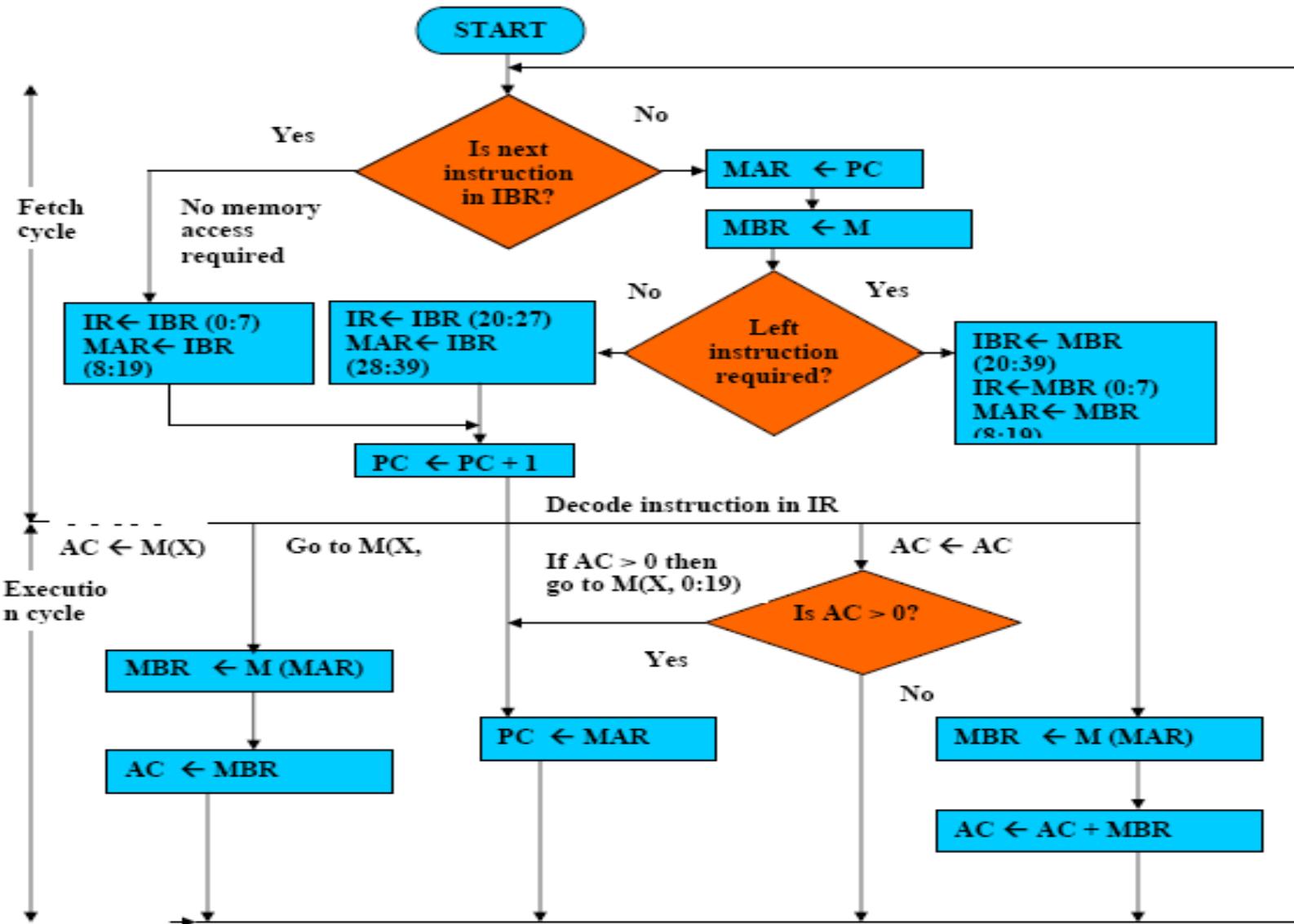


# Register transfer operation for addition operation

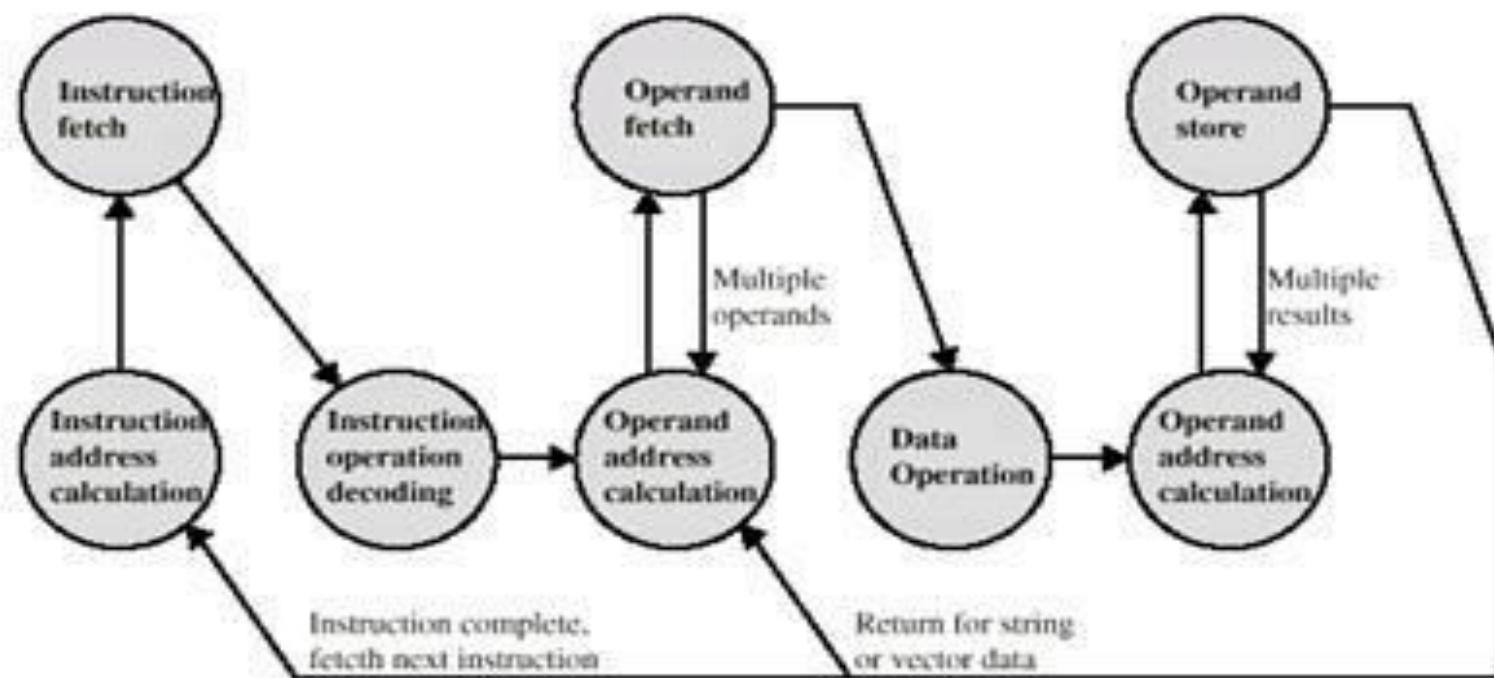
## 1. LOAD M(X) 500, ADD M(X) 501

- Register transfer operations: (PC = 1)
  - MAR  $\leftarrow$  PC
  - MBR  $\leftarrow$  M[MAR]
  - IBR  $\leftarrow$  MBR[20:39]
  - IR  $\leftarrow$  MBR[0:7]
  - MAR  $\leftarrow$  MBR[8:19]
  - MBR  $\leftarrow$  M[MAR]
  - AC  $\leftarrow$  MBR
  - IR  $\leftarrow$  IBR[0:7]
  - MAR  $\leftarrow$  IBR[8:19]
  - MBR  $\leftarrow$  M[MAR]
  - AC  $\leftarrow$  AC + MBR

16-Jul-2019 module1 Von Neumann Fetch / Execute Cycle



# Instruction Cycle State Diagram



**Instruction set:** Collection of instructions that the CPU can execute

## **What an Instruction set should specify?**

- Which Operation to perform (Opcode)
- Where to find the operand or operands (CPU registers, main memory or I/O port)
- Where to put the result, if there is result
- Where to find the next instruction

# IAS Instruction set

---

<b>Instruction Type</b>	<b>Opcode</b>	<b>Symbolic Representation</b>	<b>Description</b>
Data transfer	00001010	LOAD MQ	Transfer contents of register MQ to the accumulator AC
	00001001	LOAD MQ,M(X)	Transfer contents of memory location X to MQ
	00100001	STOR M(X)	Transfer contents of accumulator to memory location X
	00000001	LOAD M(X)	Transfer M(X) to the accumulator
	00000010	LOAD -M(X)	Transfer -M(X) to the accumulator
	00000011	LOAD  M(X)	Transfer absolute value of M(X) to the accumulator
	00000100	LOAD - M(X)	Transfer - M(X)  to the accumulator
Unconditional branch	00001101	JUMP M(X,0:19)	Take next instruction from left half of M(X)
	00001110	JUMP M(X,20:39)	Take next instruction from right half of M(X)
Conditional branch	00001111	JUMP+ M(X,0:19)	If number in the accumulator is nonnegative, take next instruction from left half of M(X)
	00010000	JUMP+ M(X,20:39)	If number in the accumulator is nonnegative, take next instruction from right half of M(X)

# IAS Instruction set Contd..

---

Arithmetic	00000101	ADD M(X)	Add M(X) to AC; put the result in AC
	00000111	ADD  M(X)	Add  M(X)  to AC; put the result in AC
	00000110	SUB M(X)	Subtract M(X) from AC; put the result in AC
	00001000	SUB  M(X)	Subtract  M(X)  from AC; put the remainder in AC
	00001011	MUL M(X)	Multiply M(X) by MQ; put most significant bits of result in AC, put least significant bits in MQ
	00001100	DIV M(X)	Divide AC by M(X); put the quotient in MQ and the remainder in AC
	00010100	LSH	Multiply accumulator by 2, i.e., shift left one bit position
	00010101	RSH	Divide accumulator by 2, i.e., shift right one position
Address modify	00010010	STOR M(X,8:19)	Replace left address field at M(X) by 12 rightmost bits of AC
	00010011	STOR M(X,28:39)	Replace right address field at M(X) by 12 rightmost bits of AC

- Write an ALP for the given expression using Von Neumann instruction set and write the register transfer notation for the same. Assume that B,C and D are available in the memory locations 450, 451 and 452 respectively.
- $A = (B + C) * D$
- $X = A + B/2$

# ALP

- $A = (B + C) * D$
- Load M [450], Add M [451]
- Store M [453], Load MQ, M [453]
- Mul M [452], Store M[453]
- Load MQ,  
Store M[454]
- Complete the Register Transfer Notation

# ALP

- $X = A + B/2$
- Load M [450], RSH
- ADD M [451], Store M[452]
- Complete the Register Transfer Notation

# ALP

If A<0

$$A = A+B$$

Else

$$A = A-B$$

AP

Assume M[450] = A

M[451] = B

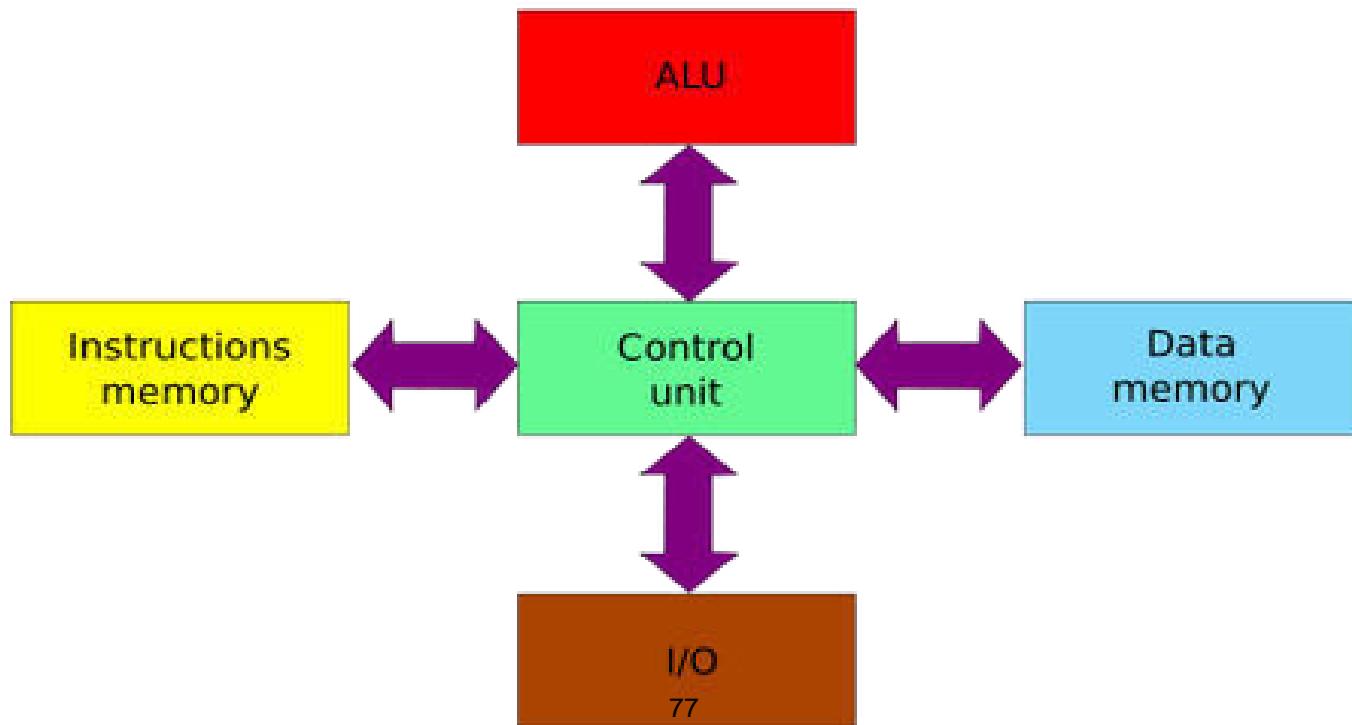
100: Load M [450], JUMP +M[500, 0:19]

101: SUB M[451] , RHI

500: ADD M[451], RHI

# Harvard Architecture

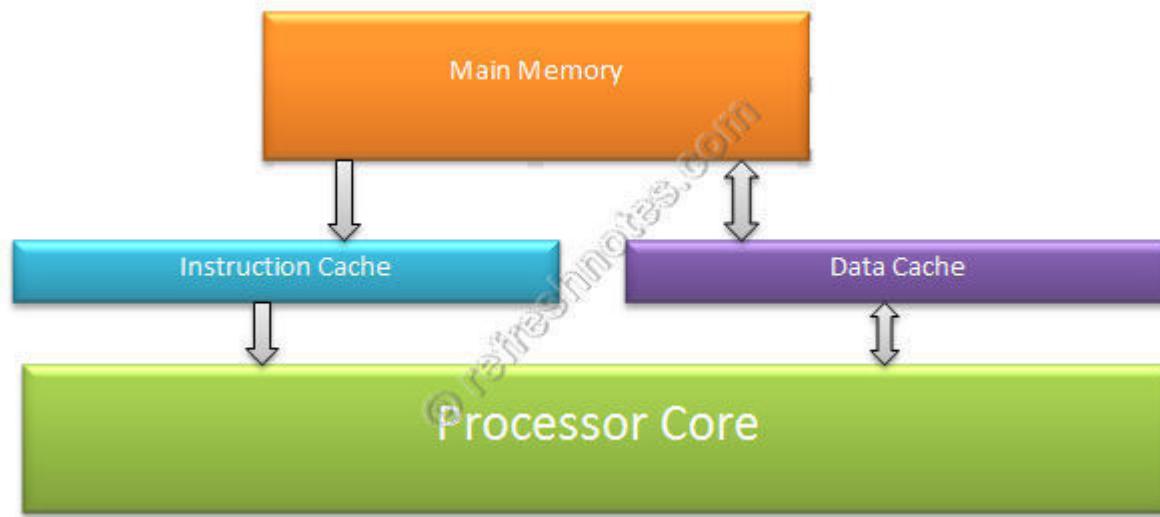
- The Harvard architecture is a computer architecture with physically separate storage and signal pathways for instructions and data.



# Von Neumann Vs. Harvard Architecture

Von Neumann	Harvard Architecture
Based on Stored Program Concept	Modern Architecture based on Harvard Mark I Model
Same physical memory for instruction and data	Separate memory space for instruction and data
Processor takes two cycles to execute an instruction	Processor takes one cycle to execute an instruction
Simpler, cheaper control unit design	Complicated design, as it uses two buses
Data transfer and instruction fetch are not simultaneous	Data transfer and instruction fetch are simultaneous
Used in PCs, laptops etc.	Used in microcontrollers....

# Modified Harvard Architecture



- Split Cache Architecture

# REGISTERS AND REGISTER FILES

# Outline

- Registers
  - User Visible Register
  - Control and Status register
- Register Files

# Introduction

- CPU must have some working space (temporary storage) called registers
- Number and function vary between processor designs
- Top level of memory hierarchy

- **User visible register**
  - Used by the Programmer
  - To minimize the memory reference by optimizing the use of registers.
- **Control and Status register**
  - Used by:
    - The control unit to control the operations of the processor
    - The OS to control execution of programs.

# User Visible Registers

- General Purpose
- Data
- Address
- Condition Codes

## General Purpose Registers (1)

- May be true general purpose
- May be restricted-(dedicated registers-floating point or stack)
- Data
  - Accumulator
- Addressing
  - Segment pointer, Index, stack pointer

## General Purpose Registers (2)

- Make them general purpose
  - Increase flexibility and programmer options
- Make them specialized
  - Smaller (faster) instructions
  - Less flexibility

# How Many GP Registers?

- Between 8 - 32
- Fewer = more memory references
- More-does not reduce memory references

# How big to be the GP register?

- Large enough to hold full address
- Large enough to hold full word
- Often possible to combine two data registers

# Condition Code Registers

- Bits set by hardware processor as a result of some operations.
- Sets of individual bits
  - e.g. result of last operation was zero
- Can be read (implicitly) by programs
  - e.g. Jump if zero –simplifies branch taking.

# Control & Status Registers

- Not visible to the user
- May be visible in a control or operating system mode (supervisor mode)
- Registers essential to instruction execution:
  - Program Counter (PC)
  - Instruction Register (IR)-Contains the inst most recently fetched
  - Memory Address Register (MAR) – contains the addr of loc in mem
  - Memory Buffer Register (MBR) – contains a word of data to be written to mem or the word most recently read

# Program Status Word(PSW)

- Registers or set of register is known as PSW
- Contains status information
- Includes Condition Codes
- Sign of last result
- Zero
- Carry
- Equal
- Overflow
- Interrupt enable/disable
- Supervisor

# Example Register Organizations

<b>Data Registers</b>	
D0	
D1	
D2	
D3	
D4	
D5	
D6	
D7	

<b>Address Registers</b>	
A0	
A1	
A2	
A3	
A4	
A5	
A6	
A7	
A7'	

<b>Program Status</b>	
Program Counter	
Status Register	

(a) MC68000

## General Registers

AX	Accumulator
BX	Base
CX	Count
DX	Data

## Pointer & Index

SP	Stack Pointer
BP	Base Pointer
SI	Source Index
DI	Dest Index

## Segment

CS	Code
DS	Data
SS	Stack
ES	Extra

## Program Status

Instr Ptr
Flags

(b) 8086

## General Registers

EAX	AX
EBX	BX
ECX	CX
EDX	DX

ESP	SP
EBP	BP
ESI	SI
EDI	DI

## Program Status

FLAGS Register
Instruction Pointer

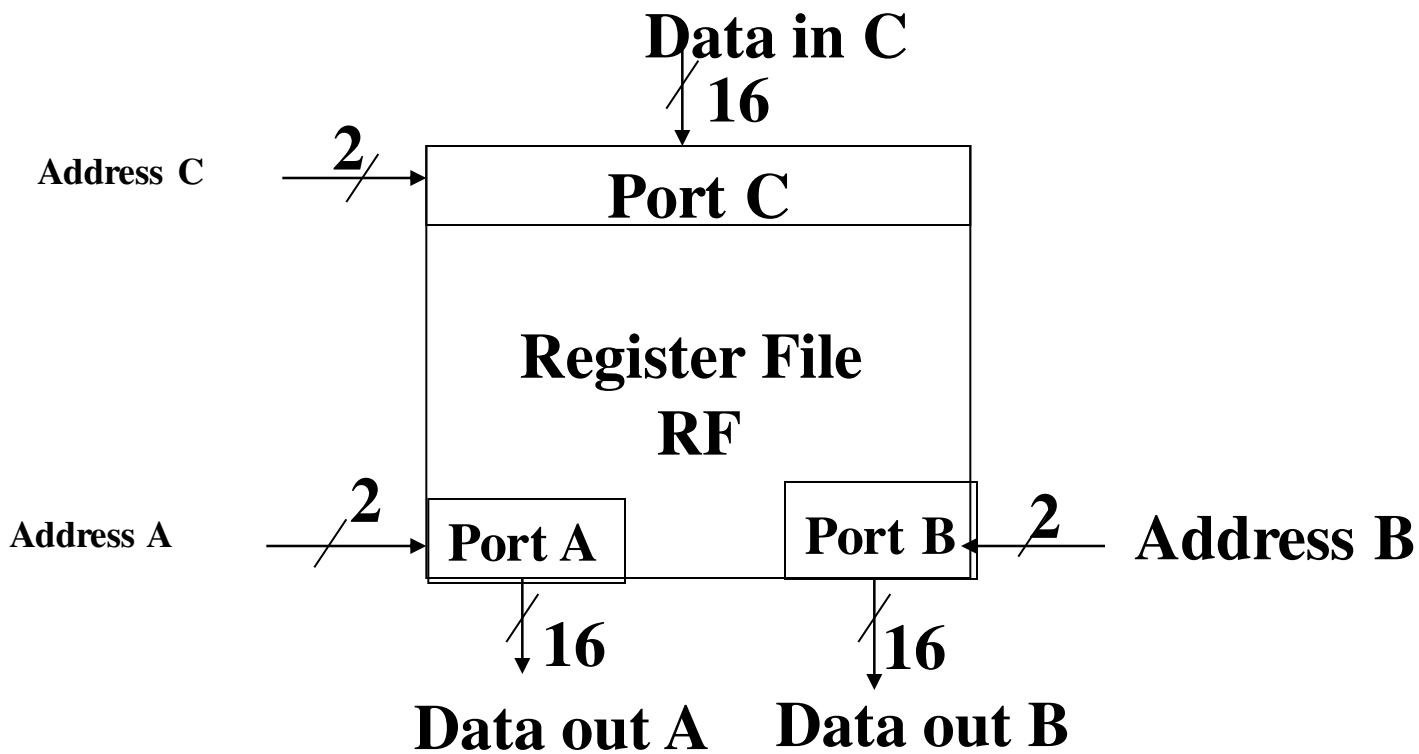
(c) 80386 - Pentium II

**Understatement:** There is no universally accepted philosophy for organizing the processor registers

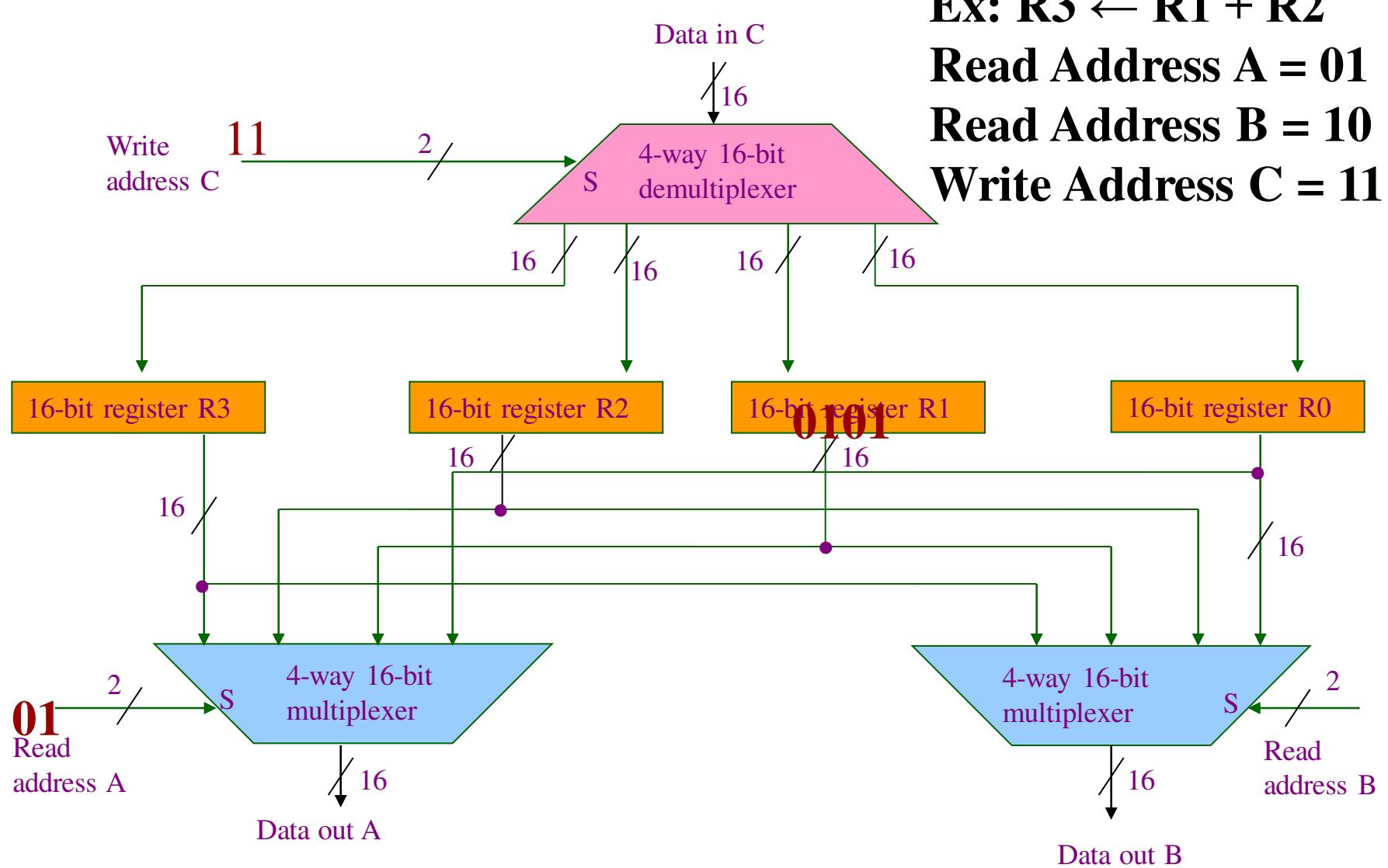
# Register Files (RF)

- Set of general purpose registers.
- It functions as small RAM and implemented using fast RAM technology.
- RF needs several access ports for simultaneously reading from or writing to several different registers. Hence RF is realized as **multiport RAM**.
- A standard RAM has just one access port with an associated address bus and data bus.

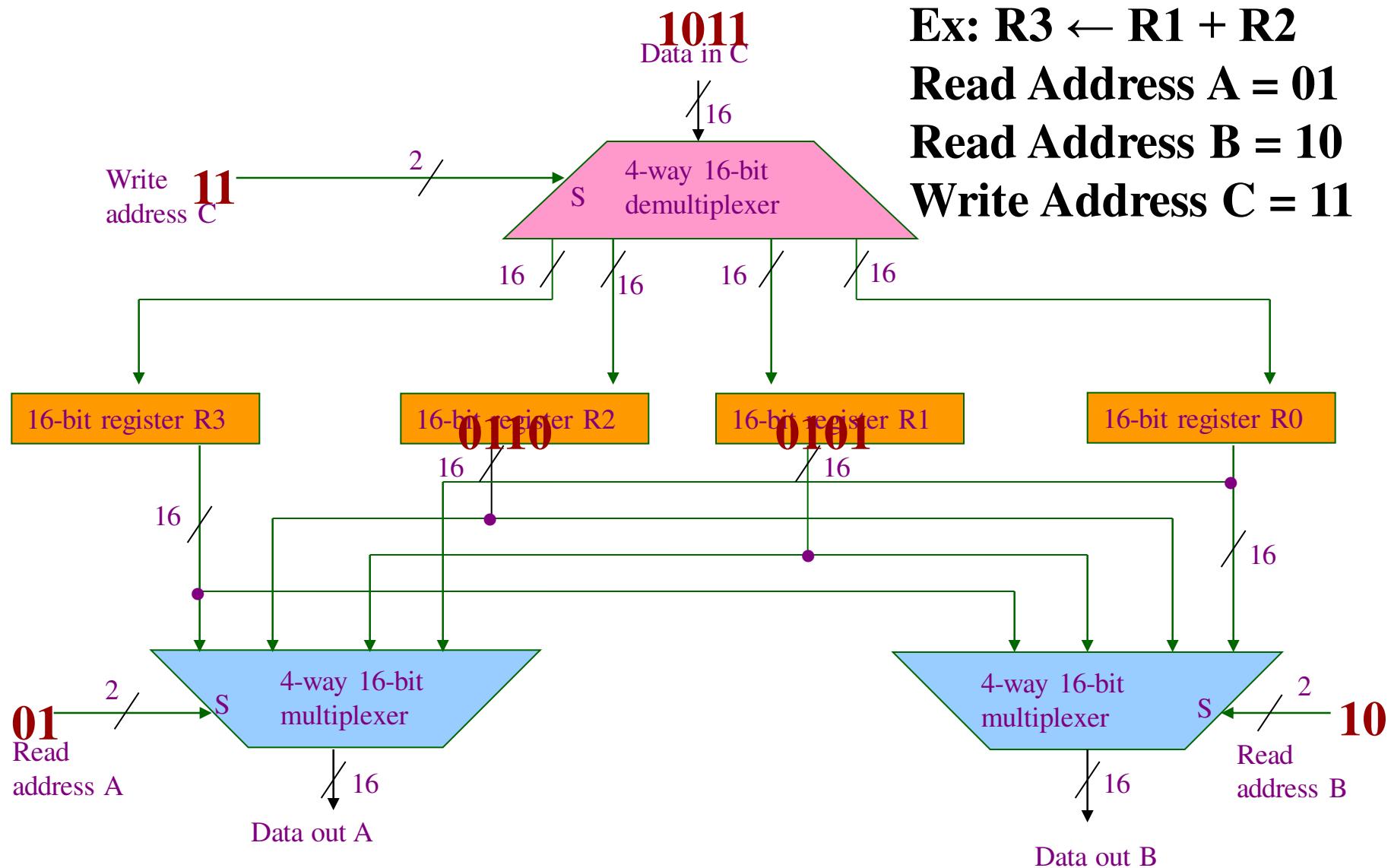
# A register file with three access ports - symbol



# 6-IAS Machine, Registers 18-Jul-2019 Material\_1 18-Jul-2019 module1\_registers



# 6-IAS Machine, Registers 18-Jul-2019 Material\_1 18-Jul-2019 module1\_registers



## **Exercise 1:**

If 8 registers are used

- How many bits are needed for read/write address?
- What is the size of the de-multiplexer and multiplexer required?
- If 4 multiplexers are used, how many parallel reads can be performed?
- Give an example with 4 parallel reads and 1 write.
- List all types of registers for the processor MC6800 and explain them briefly.
- Ref: Vincent .P. Heuring, Harry F. Jordan “ Computer System design and Architecture” Pearson, 2<sup>nd</sup> Edition, 2003.

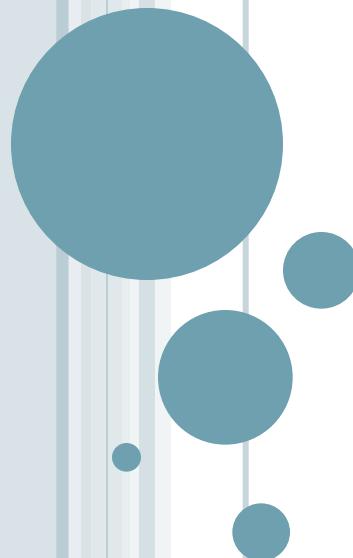
## Exercise 2: Draw your design of a register file:

- Three registers, each is 2-bits wide
  - Two source buses, one destination bus
- How many & what size:
  - Muxes did you use?
  - Demuxes did you use?

# References

- W. Stallings, Computer organization and architecture, Prentice-Hall,2000
- J. P. Hayes, Computer system architecture, McGraw Hill

## SUBROUTINE CALL & RETURN



## SUBROUTINE

- Subroutine is a self-contained sequence of instructions that performs a given computational task.
- Call Subroutine.
- When called,
  - Push [PC] to TOS
  - $[PC] \leftarrow$  1<sup>st</sup> address of subroutine
  - Execute SUB
  - Finally execute RET instruction
    - Pop TOS to PC
    - Continue with the instruction which is next to SUB call.

## LOCATIONS TO STORE THE RETURN ADDRESS

- First memory location of the subroutine
- Fixed location in memory
- Processor registers
- Memory stack – best option
  - Adv: In the case of sequential calls to subroutines. So, the top of the stack always has the return address of the subroutine which to be returned first.

## MICRO-OPERATIONS

### Call:

```
SP ← SP – 1          // decrement stack pointer  
M[SP] ← PC           // push content of PC onto the stack  
PC ← effective address /* transfer control to the subroutine */
```

### Return:

```
PC ← M[SP] // pop stack and transfer to PC  
SP ← SP + 1 // increment stack pointer
```

- **Recursive SUB:-** Subroutine that calls itself
- **Subroutine nesting:-** One subroutine that calls another.
- If only one register or memory location is used to hold the return address, when subroutine is called recursively, it destroys the previous return address.
- So, stack is the good solution for this problem.

## REFERENCES

### Text Book

- William Stallings “Computer Organization and architecture” Prentice Hall, 7th edition, 2006

# Instruction Set Architecture

# Instruction Format

---

The instruction is divided into two fields

- Opcode field
- Operand field

This operand field further divided into one to four fields.

## Simple instruction format

Opcode	Operand Address1	Operand Address2	Result Address1	Next Instruction
--------	---------------------	---------------------	--------------------	---------------------

# Instruction Set is categorized into types based on,

- Operation performed
- Number of operand addresses
- Addressing modes

# Based on Operation

- Data Movement
  - Memory : LOAD, STORE, MOV
- Data Processing
  - Arithmetic : Add, Sub, MUL
- Control Instructions
  - Conditional : JNZ, JZ
  - Un Conditional: Jump
- I\O Instructions: IN, OUT
- Logic Instructions: AND, OR

Based on number of **operand address** in the instruction.

- 4 Address Instruction
- 3 Address Instruction
- 2 Address Instruction
- 1 Address Instruction
- 0 Address Instruction

For a two-operand arithmetic instruction, five items need to be specified

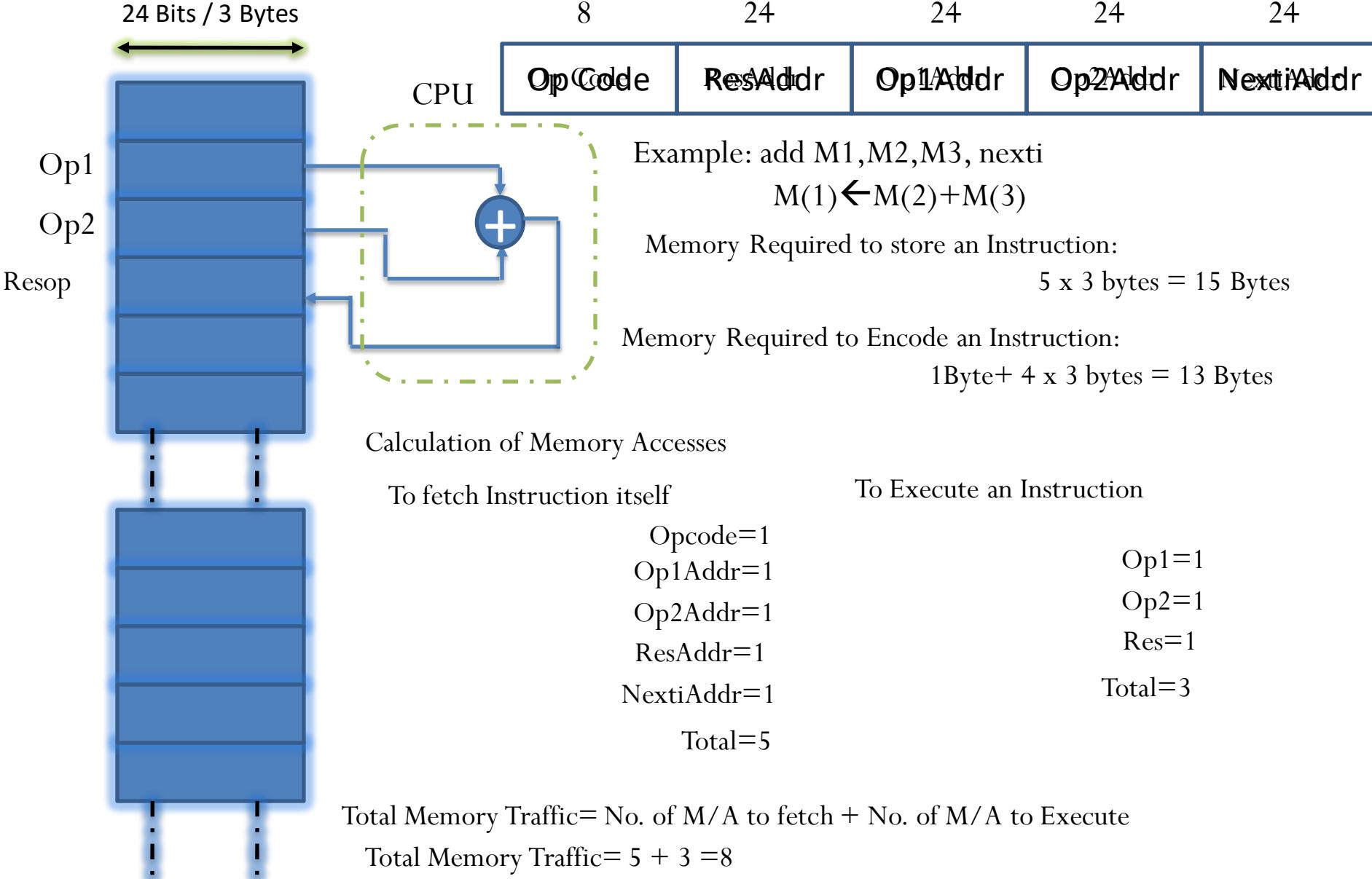
1. Operation to be performed (opcode)
2. Location of the first operand
3. Location of the second operand
4. Place to store the result
5. Location of next instruction to be executed

## **Assumptions**

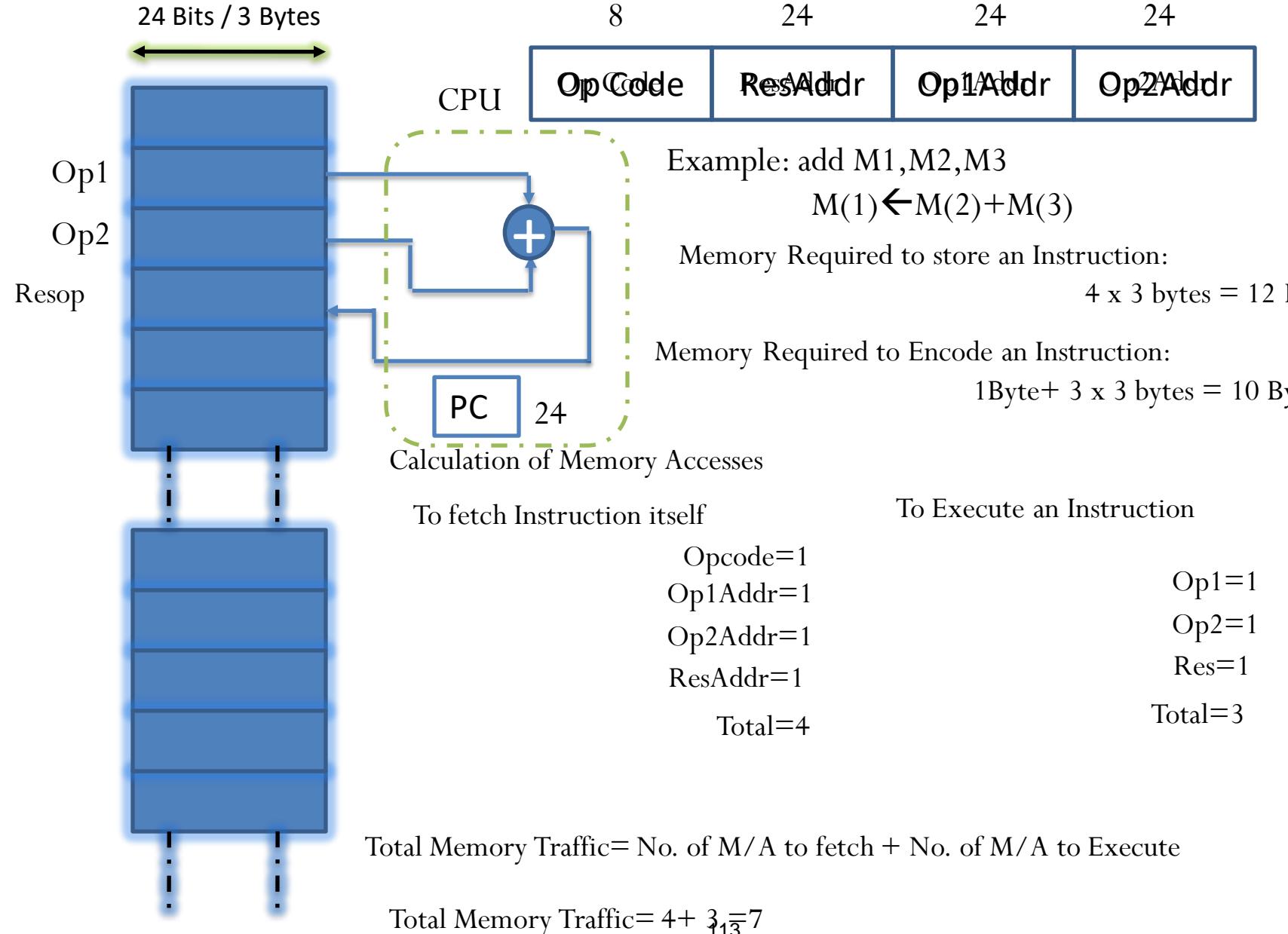
**24-bit memory address (3 bytes)**

**128 instructions (7 bits rounded to 1 byte)**

# 4- Address Instruction



# 3 - Address Instruction



# 2-Address Instruction

24 Bits / 3 Bytes

8

24

24

24

Op Code

Op1 Addr

Op2 Addr

Example: add M2,M3

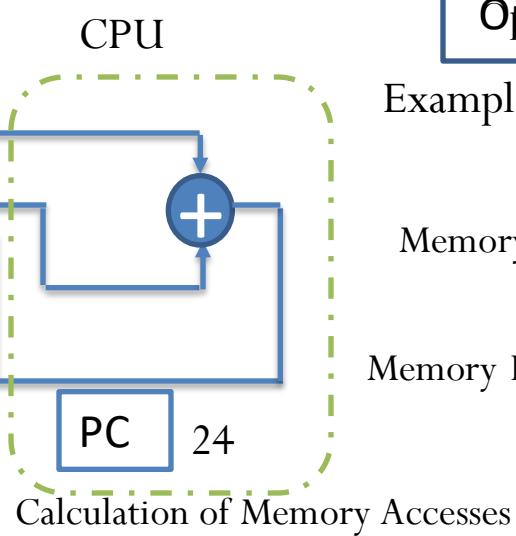
$$M(2) \leftarrow M(2) + M(3)$$

Memory Required to store an Instruction:

$$3 \times 3 \text{ bytes} = 09 \text{ Bytes}$$

Memory Required to Encode an Instruction:

$$1 \text{ Byte} + 2 \times 3 \text{ bytes} = 7 \text{ Bytes}$$



To fetch Instruction itself

Opcode=1  
Op1Addr=1  
Op2Addr=1  
Total=3

To Execute an Instruction

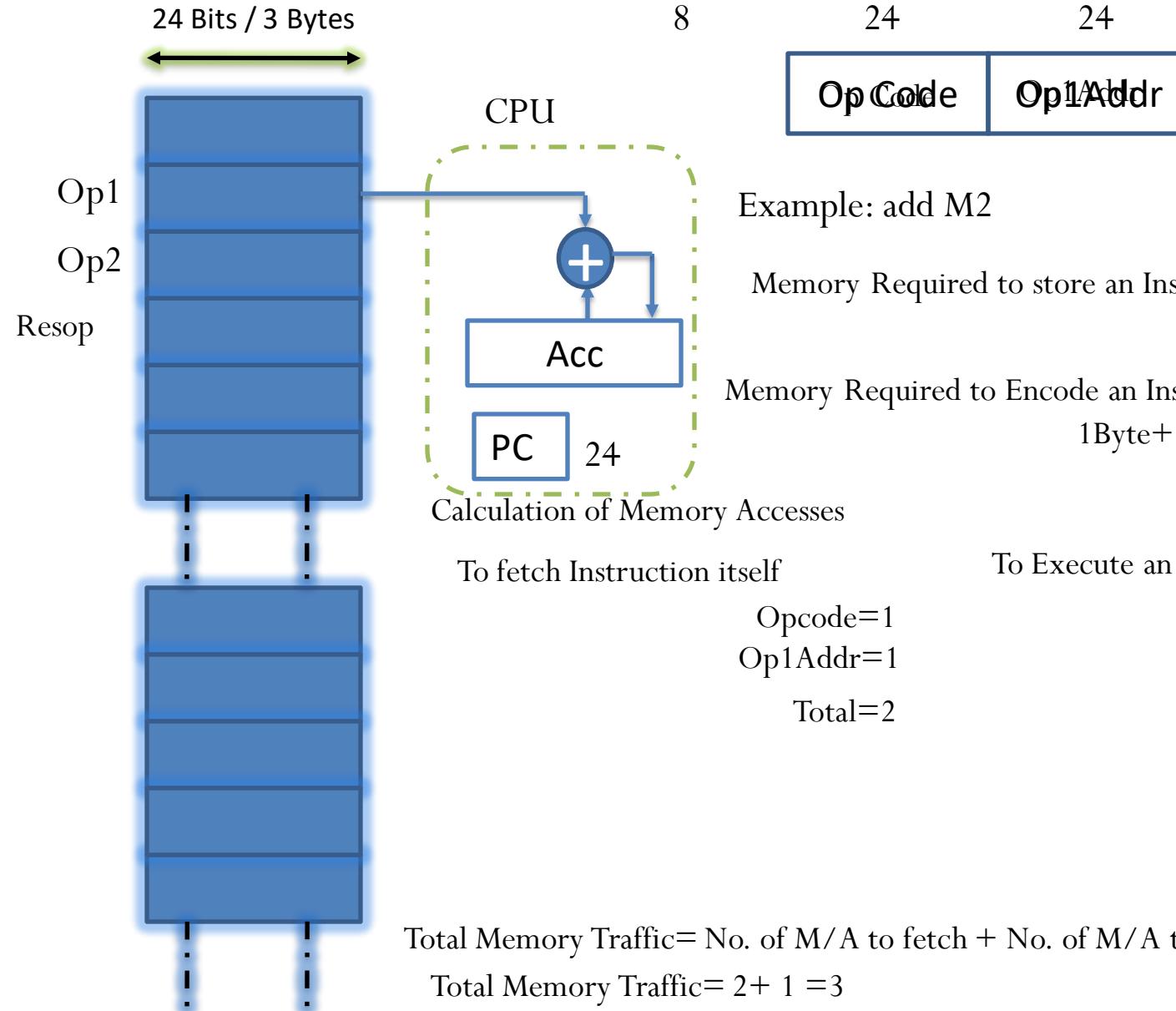
Op1=1  
Op2=1  
Res=1  
Total=3



Total Memory Traffic = No. of M/A to fetch + No. of M/A to Execute

$$\text{Total Memory Traffic} = 3 + 3 = 6$$

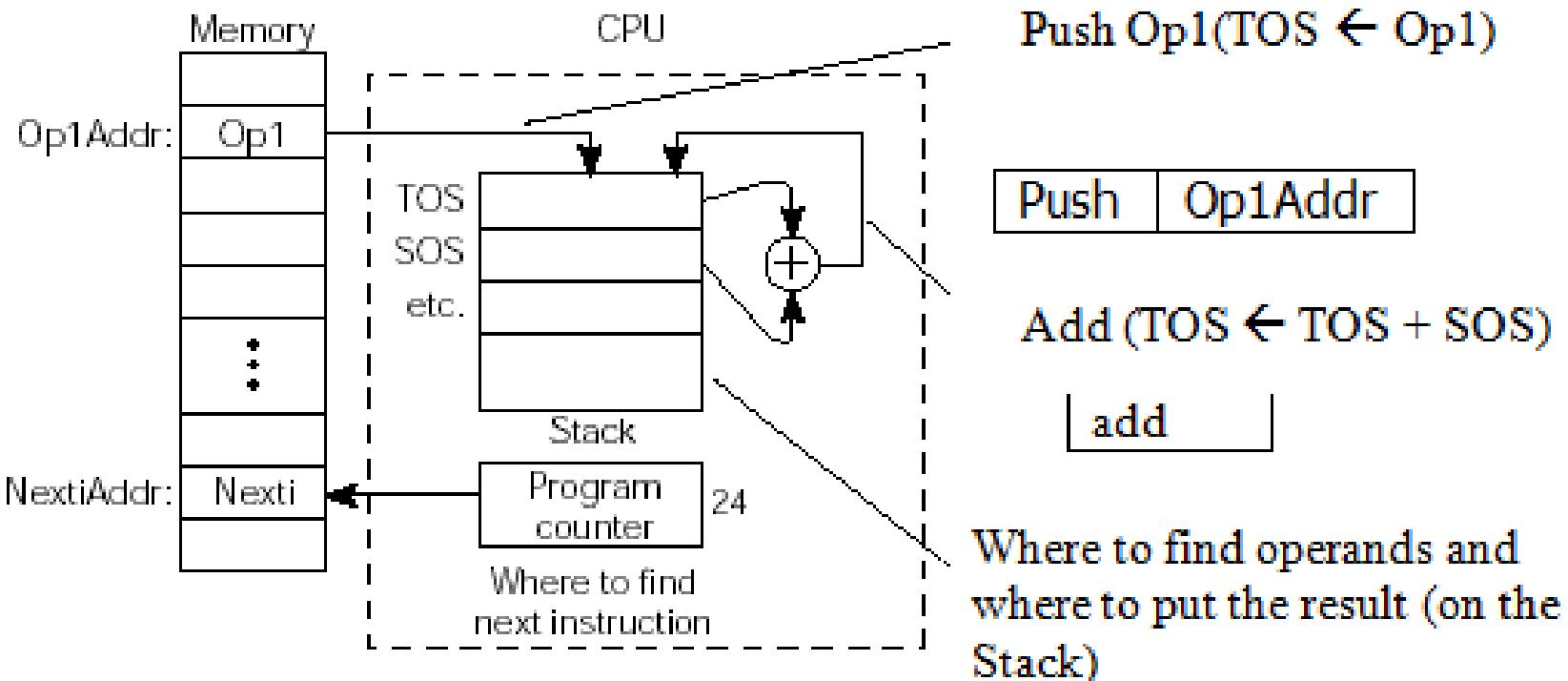
# 1- Address Instruction



# 0-Address Instruction

- Uses a push down stack in CPU
- Arithmetic uses stack for both operands and the result

Computer must have a 1-address instruction to push and pop operands to and from the stack



# Comparisons

Instruction Type	Memory To Store in Bytes	Memory To Encode in Bytes	M/As to fetch an Instruction	M/As to Execute an Instruction	Memory Traffic
4-address	$5 \times 3 = 15$	$1 + (4 \times 3) = 13$	5	3	$5+3=8$
3-Address	$4 \times 3 = 12$	$1 + (3 \times 3) = 10$	4	3	$4+3=7$
2-Address	$3 \times 3 = 09$	$1 + (2 \times 3) = 07$	3	3	$3+3=6$
1-Address	$2 \times 3 = 06$	$1 + (1 \times 3) = 04$	2	1	$2+1=3$
0-Address	$1 \times 3 = 03$	$1 + (0 \times 3) = 01$	1	0	$1+0=1$

# Evaluate $a = (b+c)*d - e$

## 3-Address

add a, b, c	$a \leftarrow b+c$
mpy a, a, d	$a \leftarrow a*d$
sub a, a, e	$a \leftarrow a-e$

## 0-Address

push b  
push c  
add  
push d  
mpy  
push e  
sub

## 2-Address

load a, b	$a \leftarrow b$
add a, c	$a \leftarrow a+c$
mpy a, d	$a \leftarrow a*d$
sub a, e	$a \leftarrow a-e$

pop a

## 1-Address

load b	$Acc \leftarrow b$
add c	$Acc \leftarrow Acc + c$
mpy d	$Acc \leftarrow Acc * d$
sub e	$Acc \leftarrow Acc - e$
store a	$a \leftarrow Acc$

		<b>Memory to Store</b>	<b>Memory to encode</b>	<b>M/As to Fetch</b>	<b>M/As to Execute</b>	<b>Memory Traffic</b>
add a, b, c	a $\leftarrow$ b+c	4*3=12	1+(3*3)=10	4	3	4+3=7
mpy a, a, d	a $\leftarrow$ a*d	4*3=12	1+(3*3)=10	4	3	4+3=7
sub a, a, e	a $\leftarrow$ a-e	4*3=12	1+(3*3)=10	4	3	4+3=7
		<b>36</b>	<b>30</b>	<b>12</b>	<b>9</b>	<b>21</b>

		<b>Memory to Store</b>	<b>Memory to encode</b>	<b>M/As to Fetch</b>	<b>M/As to Execute</b>	<b>Memory Traffic</b>
load a, b	a $\leftarrow$ b	3*3=9	1+(2*3)=7	3	2	3+2=5
add a, c	a $\leftarrow$ a+c	3*3=9	1+(2*3)=7	3	3	3+3=6
mpy a, d	a $\leftarrow$ a*d	3*3=9	1+(2*3)=7	3	3	3+3=6
sub a, e	a $\leftarrow$ a-e	3*3=9	1+(2*3)=7	3	3	3+3=6
		<b>36</b>	<b>28</b>	<b>12</b>	<b>11</b>	<b>23</b>

		<b>Memory to Store</b>	<b>Memory to encode</b>	<b>M/As to Fetch</b>	<b>M/As to Execute</b>	<b>Memory Traffic</b>
load b	Acc <b>←</b> b	2*3=6	1+(1*3)=4	2	1	2+1=3
add c	Acc <b>←</b> Acc+c	2*3=6	1+(1*3)=4	2	1	2+1=3
mpy d	Acc <b>←</b> Acc*d	2*3=6	1+(1*3)=4	2	1	2+1=3
sub e	Acc <b>←</b> Acc-e	2*3=6	1+(1*3)=4	2	1	2+1=3
store a	a <b>←</b> Acc	2*3=6	1+(1*3)=4	2	1	2+1=3
		<b>30</b>	<b>20</b>	<b>10</b>	<b>5</b>	<b>15</b>

push b	<b>6</b>	<b>4</b>	<b>2</b>	<b>1</b>	<b>3</b>
push c					
add	3	1	1	0	1
push d					
mpy					
push e					
sub					
pop a					
	<b>39</b>	<b>23</b>	<b>13</b>	<b>5</b>	<b>18</b>

- **Assume,**
  - Size of memory address is 2bytes
  - Size of operant is 2bytes
  - Size of a memory location is 1byte
  - Size of opcode is 1byte

**Then,**

- Evaluate  $X = (A + B) * (C + D)$
- Evaluate  $X = (a/b + c*d)/(d*e - f + c/a) + g$
- Evaluate  $Y = (A - B) / [C + (D/E)]$

$$\text{Evaluate } a = (b+c)*d - e$$

3-Address

add x, a, b

Add c, c, d

Mul x, x, c

2-Address

Add a, b

add c, d

Mul a, c

Load x, a

1-Address

load a

add b

store x

Load c

Add d

Mul x

Store x

0-Address

push a

Push b

Add

Push c

Push d

Mul

Pop x

# Evaluate $a = (b+c)*d - e$

## 3-Address

add x, a, b 7+6

Add c, c, d

Mul x, x, c

## 2-Address

Add a, b 5+6

add c, d

Mul a, c

Load x, a 5+4

## 1-Address

load a 3+2

add b

store x

Load c

Add d

Mul x

Store x

## 0-Address

push a

Push b

Add

Push c

Push d

Mul

Pop x

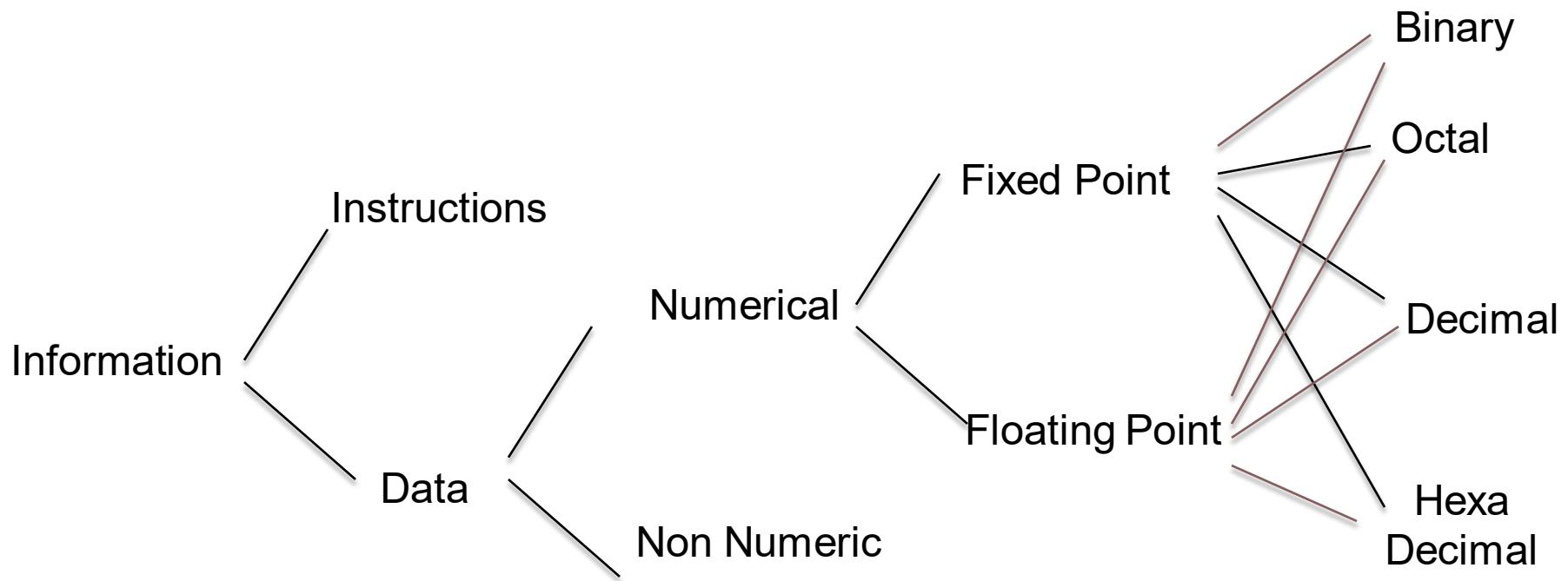
# References

## Reference Book

- W. Stallings, Computer organization and architecture, Prentice-Hall,2000
- J. P. Hayes, Computer system architecture, McGraw Hill,2000
- Vincent .P. Heuring, Harry F. Jordan “Computer System design and Architecture” Pearson, 2<sup>nd</sup> Edition, 2003

# Data Representation

# Data Representation



# Data Representation- Integer-Fixed point

- Unsigned Number
- Signed Number
- One's Complement
- Two's Complement
- Biased one(Not Commonly used)

# Cont

- Unsigned numbers: only non-negative values.
- Signed numbers: include all values (positive and negative).
- There are three common ways of representing signed numbers (positive and negative numbers) for binary numbers:
  - ❖ Sign-and-Magnitude
  - ❖ 1s Complement
  - ❖ 2s Complement

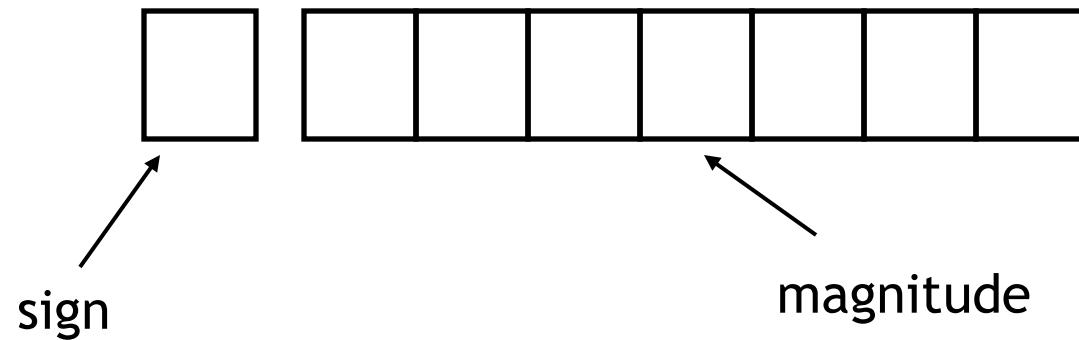
# Sign-and-Magnitude

- Negative numbers are usually written by writing a minus sign in front.
  - ❖ Example:
    - $(12)_{10}$  , -  $(1100)_2$
- In sign-and-magnitude representation, this sign is usually represented by a bit:
  - 0 for +
  - 1 for -

# Negative Numbers:

## Sign-and-Magnitude (Cont')

- Example: an 8-bit number can have 1-bit sign and 7-bit magnitude.



# Signed magnitude representation (cont')

.

- Examples:

$1101_2 = 13_{10}$  (a 4-bit unsigned number)

**0**  $1101 = +13_{10}$  (a positive number in 5-bit signed magnitude)

**1**  $1101 = -13_{10}$  (a negative number in 5-bit signed magnitude)

$0100_2 = 4_{10}$  (a 4-bit unsigned number)

**0**  $0100 = +4_{10}$  (a positive number in 5-bit signed magnitude)

**1**  $0100 = -4_{10}$  (a negative number in 5-bit signed magnitude)

# Negative Numbers:

## Sign-and-Magnitude (Cont')

- To negate a number, just **invert the sign bit**.
- Examples:
  - $(0\ 0100001)_{sm} = (1\ 0100001)_{sm}$
  - $(1\ 0000101)_{sm} = (0\ 0000101)_{sm}$

# 1s and 2s Complement

- Two other ways of representing signed numbers for binary numbers are:
  - ❖ 1s-complement
  - ❖ 2s-complement
- They are preferred over the simple sign-and-magnitude representation.

# One's complement representation

- A different approach, **one's complement**, negates numbers by complementing each bit of the number.
- We keep the sign bits: 0 for positive numbers, and 1 for negative. The sign bit is complemented along with the rest of the bits.
- Examples:

$1101_2 = 13_{10}$  (a 4-bit unsigned number)

**0**  $1101 = +13_{10}$  (a positive number in 5-bit one's complement)

**1**  $0010 = -13_{10}$  (a negative number in 5-bit one's complement)

$0100_2 = 4_{10}$  (a 4-bit unsigned number)

**0**  $0100 = +4_{10}$  (a positive number in 5-bit one's complement)

**1**  $1011 = -4_{10}$  (a negative number in 5-bit one's complement)

# Why is it called “one’s complement?”

- Complementing a single bit is equivalent to subtracting it from 1.

$$0' = 1, \text{ and } 1 - 0 = 1 \quad 1' = 0, \text{ and } 1 - 1 = 0$$

- Similarly, complementing each bit of an n-bit number is equivalent to subtracting that number from  $2^n - 1$ .
- For example, we can negate the 5-bit number 01101.
  - Here  $n=5$ , and  $2^n - 1 = 31_{10} = 11111_2$ .
  - Subtracting 01101 from 11111 yields 10010:

$$\begin{array}{r} 11111 \\ - 01101 \\ \hline 10010 \end{array}$$

# 1s Complement Addition/Subtraction

- **Algorithm for addition, A + B:**
  1. Perform binary addition on the two numbers.
  2. If there is a carry out of the MSB, add 1 to the result.
  3. Check for overflow: Overflow occurs if result is opposite sign of A and B.
- **Algorithm for subtraction, A - B:**
$$A - B = A + (-B)$$
  1. Take 1s complement of B by inverting all the bits.
  2. Add the 1s complement of B to A.

# One's complement addition

- To add one's complement numbers:
  - First do unsigned addition on the numbers, *including* the sign bits.
  - Then take the carry out and add it to the sum.
- Two examples:

$$\begin{array}{r}
 0111 \\
 + 1011 \\
 \hline
 1\textcolor{magenta}{0}010
 \end{array}
 \quad
 \begin{array}{r}
 (+7) \\
 + (-4) \\
 \hline
 \end{array}$$

$$\begin{array}{r}
 0010 \\
 + \textcolor{magenta}{1} \\
 \hline
 0011
 \end{array}
 \quad (+3)$$

$$\begin{array}{r}
 0011 \quad (+3) \\
 + 0010 \quad + (+2) \\
 \hline
 0\textcolor{magenta}{0}101
 \end{array}$$

$$\begin{array}{r}
 0101 \\
 + 0 \\
 \hline
 0101 \quad (+5)
 \end{array}$$

- This is simpler and more uniform than signed magnitude addition.

# Two's complement

- Our final idea is **two's complement**. To negate a number, complement each bit (just as for ones' complement) and then add 1.
- Examples:

$1101_2 = 13_{10}$  (a 4-bit unsigned number)

$0\ 1101 = +13_{10}$  (a positive number in 5-bit two's complement)

$1\ 0010 = -13_{10}$  (a negative number in 5-bit ones' complement)

$1\ 0011 = -13_{10}$  (a negative number in 5-bit two's complement)

$0100_2 = 4_{10}$  (a 4-bit unsigned number)

$0\ 0100 = +4_{10}$  (a positive number in 5-bit two's complement)

$1\ 1011 = -4_{10}$  (a negative number in 5-bit ones' complement)

$1\ 1100 = -4_{10}$  (a negative number in 5-bit two's complement)

# 2s Complement Addition/Subtraction

- **Algorithm for addition, A + B:**
  1. Perform binary addition on the two numbers.
  2. Ignore the carry out of the MSB (most significant bit).
  3. Check for overflow: Overflow occurs if the ‘carry in’ and ‘carry out’ of the MSB are different, or if result is opposite sign of A and B.
- **Algorithm for subtraction, A - B:**
$$A - B = A + (-B)$$
  1. Take 2s complement of B by inverting all the bits and adding 1.
  2. Add the 2s complement of B to A.

# 2s Complement Addition/Subtraction

- Examples: 4-bit binary system

$$\begin{array}{r}
 +3 & 0011 \\
 + +4 & + 0100 \\
 \hline
 & \hline \\
 +7 & 0111 \\
 \hline
 & \hline
 \end{array}$$

$$\begin{array}{r}
 -2 & 1110 \\
 + -6 & + 1010 \\
 \hline
 & \hline \\
 -8 & \textcolor{red}{1}1000 \\
 \hline
 & \hline
 \end{array}$$

$$\begin{array}{r}
 +6 & 0110 \\
 + -3 & + 1101 \\
 \hline
 & \hline \\
 +3 & \textcolor{red}{1}0011 \\
 \hline
 & \hline
 \end{array}$$

$$\begin{array}{r}
 +4 & 0100 \\
 + -7 & + 1001 \\
 \hline
 & \hline \\
 -3 & 1101 \\
 \hline
 & \hline
 \end{array}$$

- Which of the above is/are overflow(s)?

# Overflow

- If the numbers are unsigned, overflow occurs when there is a carry out of the most significant bit
- For signed numbers, overflow detected by comparing the sign of the result against the sign of the numbers
- If one number is positive and another is negative, overflow cannot occur
- If both are positive or both are negative, compare the carry into the sign bit and carry out of the sign bit
- If these two carries are not equal, then there is an overflow

# Signed overflow

- With two's complement and a 4-bit adder, for example, the largest representable decimal number is +7, and the smallest is -8.
- What if you try to compute  $4 + 5$ , or  $(-4) + (-5)$ ?

$$\begin{array}{r}
 0100 \quad (+4) \\
 + 0101 \quad (+5) \\
 \hline
 01001 \quad (-7)
 \end{array}$$

$$\begin{array}{r}
 1100 \quad (-4) \\
 + 1011 \quad (-5) \\
 \hline
 1011 \quad (+7)
 \end{array}$$

- We cannot just include the carry out to produce a five-digit result, as for unsigned addition. If we did,  $(-4) + (-5)$  would result in +23!
- Also, unlike the case with unsigned numbers, the carry out *cannot* be used to detect overflow.
  - In the example on the left, the carry out is 0 but there *is* overflow.
  - Conversely, there are situations where the carry out is 1 but there is *no* overflow.

# Detecting signed overflow

- The easiest way to detect signed overflow is to look at all the sign bits.

$$\begin{array}{r} 0100 \quad (+4) \\ + 0101 \quad (+5) \\ \hline 01001 \quad (-7) \text{ in 2's compl} \end{array}$$

$$\begin{array}{r} 1100 \quad (-4) \\ + 1011 \quad (-5) \\ \hline 10111 \quad (+7) \end{array}$$

- Overflow occurs only in the two situations above:
  - If you add two *positive* numbers and get a *negative* result.
  - If you add two *negative* numbers and get a *positive* result.
- Overflow cannot occur if you add a positive number to a negative number. Do you see why?

# Signed Overflow

$$\begin{array}{r}
 1101 (-3) \\
 +1011 (-5) \\
 \hline
 1000 (-8)
 \end{array}$$

carry generated, but no overflow

$$\begin{array}{r}
 & 1 \\
 & 1101 (-3) \\
 + & 1010 (-6) \\
 \hline
 1 & 0111 (+7)
 \end{array}$$

carry and overflow

$$\begin{array}{r}
 1101 (-3) \\
 +0010 (+2) \\
 \hline
 01111 (-1)
 \end{array}$$

no carry and no overflow

$$\begin{array}{r}
 0111 (+7) \\
 +0011 (+3) \\
 \hline
 01010 (-6)
 \end{array}$$

no carry and overflow

# Comparing the signed number systems

- Here are all the 4-bit numbers in the different systems.
- Positive numbers are the same in all three representations.*
- Signed magnitude and one's complement have *two* ways of representing 0. This makes things more complicated.
- Two's complement has asymmetric ranges; there is one more negative number than positive number.  
Here, you can represent -8 but not +8.
- However, two's complement is preferred because it has only one 0, and its addition algorithm is the simplest.

Decimal	S.M.	1's comp.	2's comp.
7	0111	0111	0111
6	0110	0110	0110
5	0101	0101	0101
4	0100	0100	0100
3	0011	0011	0011
2	0010	0010	0010
1	0001	0001	0001
0	0000	0000	0000
-0	1000	1111	-
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	-	-	1000

# Ranges of the signed number systems

- How many negative and positive numbers can be represented in each of the different systems on the previous page?

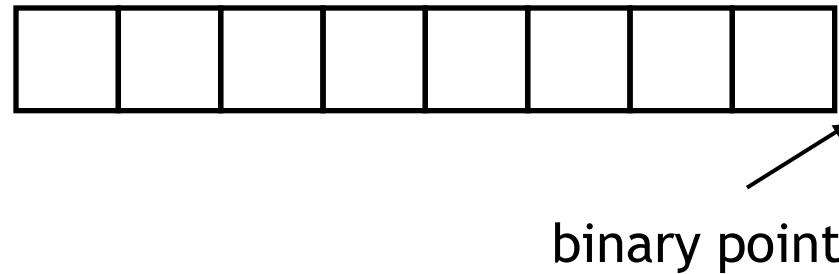
	Unsigned	Signed Magnitude	One's complement	Two's complement
Smallest	0000 (0)	1111 (-7)	1000 (-7)	1000 (-8)
Largest	1111 (15)	0111 (+7)	0111 (+7)	0111 (+7)

- In general, with n-bit numbers including the sign, the ranges are:

	Unsigned	Signed Magnitude	One's complement	Two's complement
Smallest	0	$-(2^{n-1}-1)$	$-(2^{n-1}-1)$	$-2^{n-1}$
Largest	$2^n-1$	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$

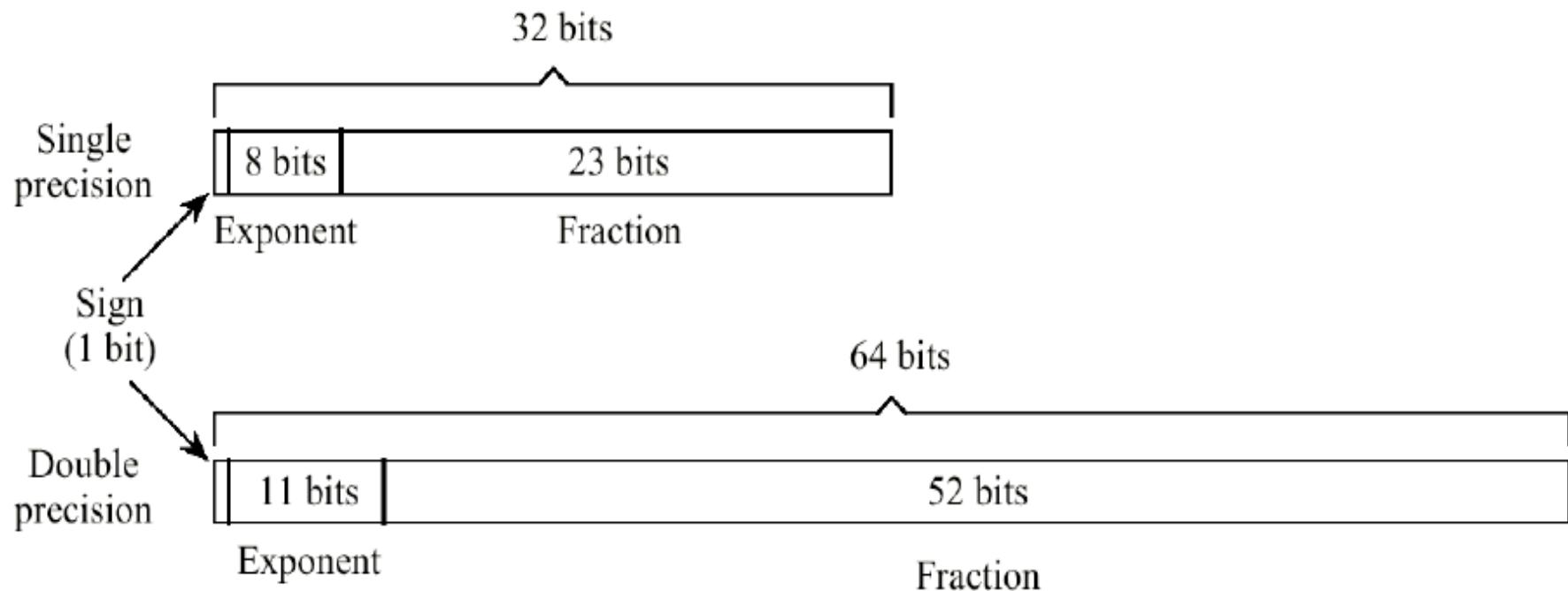
# Fixed Point Numbers

- The signed and unsigned numbers representation given are **fixed point numbers**.
- The **binary point** is assumed to be at a fixed location, say, at the end of the number:

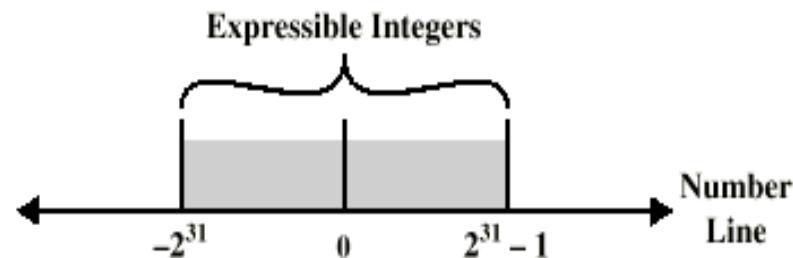


- Can represent all integers between -128 to 127 (for 8 bits).

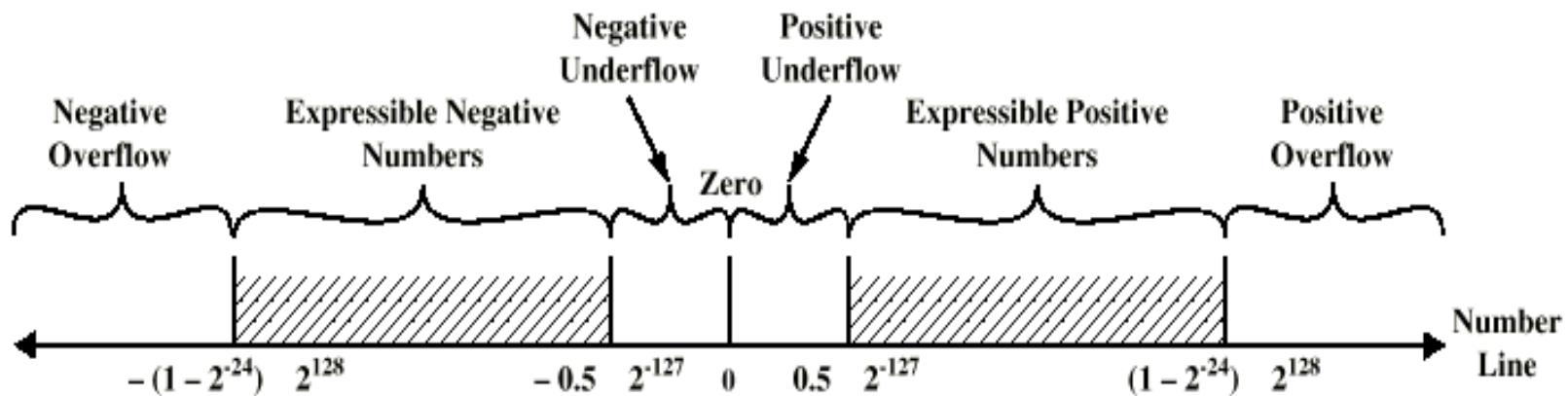
# IEEE-754 Floating Point Formats



# Expressible Numbers



(a) Twos Complement Integers



(b) Floating-Point Numbers

# IEEE-754 Conversion Example

Represent -12.62510 in single precision IEEE-754 format.

- Step #1: Convert to target base.  $-12.625_{10} = -1100.101_2$
- Step #2: Normalize.  $-1100.101_2 = -1.100101_2 \times 2^3$
- Step #3: Fill in bit fields. Sign is negative, so sign bit is 1. Exponent is in excess 127 (not excess 128!), so exponent is represented as the unsigned integer  $3 + 127 = 130$ . Leading 1 of significant is hidden, so final bit pattern is:

1 1000 0010 . 1001 0100 0000 0000 0000 000

# Character Representation ASCII

## ASCII (American Standard Code for Information Interchange) Code

		MSB (3 bits)							
		0	1	2	3	4	5	6	7
LSB (4 bits)	0	NUL	DLE	SP	0	@	P	'	P
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	\$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	'	7	G	W	g	w
	8	BS	CAN	(	8	H	X	h	x
	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	
	D	CR	GS	-	=	M	]	m	}
	E	SO	RS	.	>	N	m	n	~
	F	SI	US	151 /	?	O	n	o	DEL

# Control Character Representation (ASCII)

<b>NUL</b>	Null	<b>DC1</b>	Device Control 1
<b>SOH</b>	Start of Heading (CC)	<b>DC2</b>	Device Control 2
<b>STX</b>	Start of Text (CC)	<b>DC3</b>	Device Control 3
<b>ETX</b>	End of Text (CC)	<b>DC4</b>	Device Control 4
<b>EOT</b>	End of Transmission (CC)	<b>NAK</b>	Negative Acknowledge (CC)
<b>ENQ</b>	Enquiry (CC)	<b>SYN</b>	Synchronous Idle (CC)
<b>ACK</b>	Acknowledge (CC)	<b>ETB</b>	End of Transmission Block (CC)
<b>BEL</b>	Bell	<b>CAN</b>	Cancel
<b>BS</b>	Backspace (FE)	<b>EM</b>	End of Medium
<b>HT</b>	Horizontal Tab. (FE)	<b>SUB</b>	Substitute
<b>LF</b>	Line Feed (FE)	<b>ESC</b>	Escape
<b>VT</b>	Vertical Tab. (FE)	<b>FS</b>	File Separator (IS)
<b>FF</b>	Form Feed (FE)	<b>GS</b>	Group Separator (IS)
<b>CR</b>	Carriage Return (FE)	<b>RS</b>	Record Separator (IS)
<b>SO</b>	Shift Out	<b>US</b>	Unit Separator (IS)
<b>SI</b>	Shift In	<b>DEL</b>	Delete
<b>DLE</b>	Data Link Escape (CC)		

(CC) Communication Control

(FE) Format Effector

(IS) Information Separator

# The EBCDIC character code, shown with hexadecimal indices

11-Module 2 - data representation-01-Aug-2019Material\_I\_01-Aug-2019\_Data\_Representation

00	NUL	20	DS	40	SP	60	-	80		A0		C0	{	E0	\
01	SOH	21	SOS	41		61	/	81	a	A1	~	C1	A	E1	
02	STX	22	FS	42		62		82	b	A2	s	C2	B	E2	S
03	ETX	23		43		63		83	c	A3	t	C3	C	E3	T
04	PF	24	BYP	44		64		84	d	A4	u	C4	D	E4	U
05	HT	25	LF	45		65		85	e	A5	v	C5	E	E5	V
06	LC	26	ETB	46		66		86	f	A6	w	C6	F	E6	W
07	DEL	27	ESC	47		67		87	g	A7	x	C7	G	E7	X
08		28		48		68		88	h	A8	y	C8	H	E8	Y
09		29		49		69		89	i	A9	z	C9	I	E9	Z
0A	SMM	2A	SM	4A	¢	6A	'	8A		AA		CA		EA	
0B	VT	2B	CU2	4B		6B	,	8B		AB		CB		EB	
0C	FF	2C		4C	<	6C	%	8C		AC		CC		EC	
0D	CR	2D	ENQ	4D	(	6D	-	8D		AD		CD		ED	
0E	SO	2E	ACK	4E	+	6E	>	8E		AE		CE		EE	
0F	SI	2F	BEL	4F		6F	?	8F		AF		CF		EF	
10	DLE	30		50	&	70		90		B0		D0	}	F0	0
11	DC1	31		51		71		91	j	B1		D1	J	F1	1
12	DC2	32	SYN	52		72		92	k	B2		D2	K	F2	2
13	TM	33		53		73		93	l	B3		D3	L	F3	3
14	RES	34	PN	54		74		94	m	B4		D4	M	F4	4
15	NL	35	RS	55		75		95	n	B5		D5	N	F5	5
16	BS	36	UC	56		76		96	o	B6		D6	O	F6	6
17	IL	37	EOT	57		77		97	p	B7		D7	P	F7	7
18	CAN	38		58		78		98	q	B8		D8	Q	F8	8
19	EM	39		59		79		99	r	B9		D9	R	F9	9
1A	CC	3A		5A	!	7A	:	9A		BA		DA		FA	
1B	CU1	3B	CU3	5B	\$	7B	#	9B		BB		DB		FB	
1C	IFS	3C	DC4	5C	.	7C	@	9C		BC		DC		FC	
1D	IGS	3D	NAK	5D	)	7D	'	9D		BD		DD		FD	
1E	IRS	3E		5E	:	7E	158	9E		BE		DE		FE	
1F	IUS	3F	SUB	5F	-	7F	"	9F		BF		DF		FF	

# The EBCDIC control character representation

STX	Start of text	RS	Reader Stop	DC1	Device Control 1	BEL	Bell
DLE	Data Link Escape	PF	Punch Off	DC2	Device Control 2	SP	Space
BS	Backspace	DS	Digit Select	DC4	Device Control 4	IL	Idle
ACK	Acknowledge	PN	Punch On	CU1	Customer Use 1	NUL	Null
SOH	Start of Heading	SM	Set Mode	CU2	Customer Use 2		
ENQ	Enquiry	LC	Lower Case	CU3	Customer Use 3		
ESC	Escape	CC	Cursor Control	SYN	Synchronous Idle		
BYP	Bypass	CR	Carriage Return	IFS	Interchange File Separator		
CAN	Cancel	EM	End of Medium	EOT	End of Transmission		
RES	Restore	FF	Form Feed	ETB	End of Transmission Block		
SI	Shift In	TM	Tape Mark	NAK	Negative Acknowledge		
SO	Shift Out	UC	Upper Case	SMM	Start of Manual Message		
DEL	Delete	FS	Field Separator	SOS	Start of Significance		
SUB	Substitute	HT	Horizontal Tab	IGS	Interchange Group Separator		
NL	New Line	VT	Vertical Tab	IRS	Interchange Record Separator		
LF	Line Feed	UC	Upper Case	IUS	Interchange Unit Separator		

# Pros and cons of integer representation

- Signed magnitude representation:
  - 2 representations for 0
  - Simple
  - 255 different numbers can be represented.
  - Need to consider both sign and magnitude in arithmetic
  - Different logic for addition and subtraction
- 1's complement representation:
  - 2 representations for 0
  - Complexity in performing addition and subtraction
  - 255 different numbers can be represented.
- 2's complement representation:
  - Only one representation for 0
  - 256 different numbers can be represented.
  - Arithmetic works easily
  - Negating is fairly easy

# Reference

- Morris Mano, “Computer System Architecture”, Pearson Education, 3<sup>rd</sup> edition (Chapter 3)
- William Stallings “Computer Organization and architecture”, Prentice Hall, 7<sup>th</sup> edition, 2006.(chapter 9)
- <http://acad.intranet.vit.ac.in/CoursePage/COURSEPAGE-DUMP/EEE116%20Digital%20Logic%20System%20Design/Prof.%20Balamurugan%20%20S/Unit-I-Number%20system%20and%20Codes.ppt>

# Booth Multiplication Algorithm

# Booth Algorithm

- An efficient way to multiply two signed binary numbers expressed in 2's complement notation :
- Reduces the number of operations by relying on blocks of consecutive 1's
- Example:
- $Y \times 00111110 = Y \times (2^5 + 2^4 + 2^3 + 2^2 + 2^1)$ .
- $Y \times 00111110 = Y \times (01000000 - 00000010) = Y \times (2^6 - 2^1)$ .  
One addition and one subtraction

# Description and Hardware for Booth Multiplication

- $QR$  multiplier
- $Q_n$  least significant bit of  $QR$
- $Q_{n+1}$  previous least significant bit of  $QR$
- $BR$  multiplicand
- $AC = 0$
- $SC$  number of bits in multiplier

## Algorithm

Do  $SC$  times

$$Q_n Q_{n+1} = 10$$

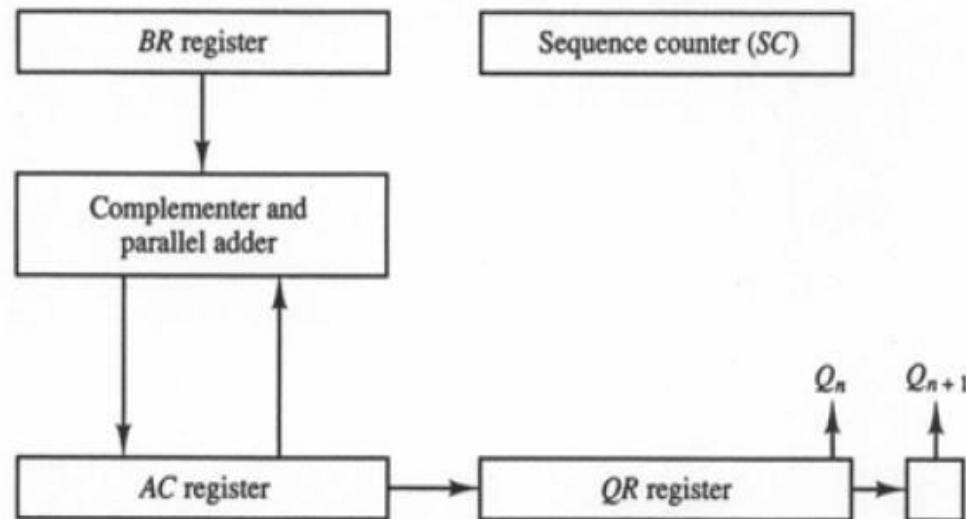
$$AC \leftarrow AC + \overline{BR} + 1$$

$$Q_n Q_{n+1} = 01$$

$$AC \leftarrow AC + BR$$

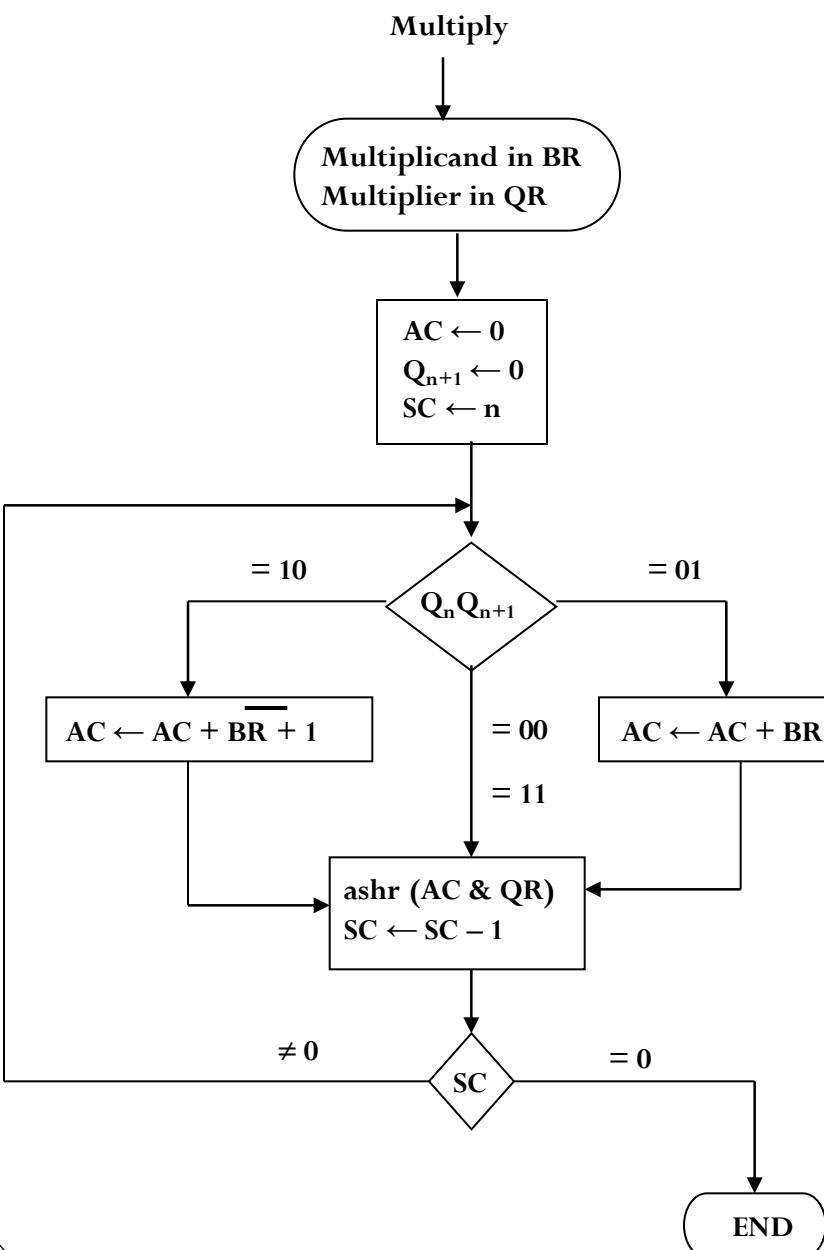
Arithmetic shift right  $AC$  &  $QR$

$$SC \leftarrow SC - 1$$



- For our example, and multiply **(-9) x (-13)**
  - The numerically larger **operand (13) would require 4 bits** to represent in binary (1101). So we must use **AT LEAST 5 bits** to represent the operands, to allow for the sign bit.

# Flowchart for Booth Multiplication



Example:  $-9 \times -13 = 117$   
 $BR = 10111, \overline{BR} + 1 = 01001$

Comment	AC	QR	Q <sub>n+1</sub>	SC
	00000	10011	0	5
Subtract BR	01001			
		01001		
Ashr	00100	11001	1	4
Ashr	00010	01100	1	3
Add BR	10111			
		11001		
		11100	0	2
Ashr	11100	10110	0	1
Ashr	11110	01011	0	
Subtract BR	01001			
		00111		
Ashr	00011	10101	1	0

# Multiply $7 \times 3$ using above signed 2's compliment binary multiplication.

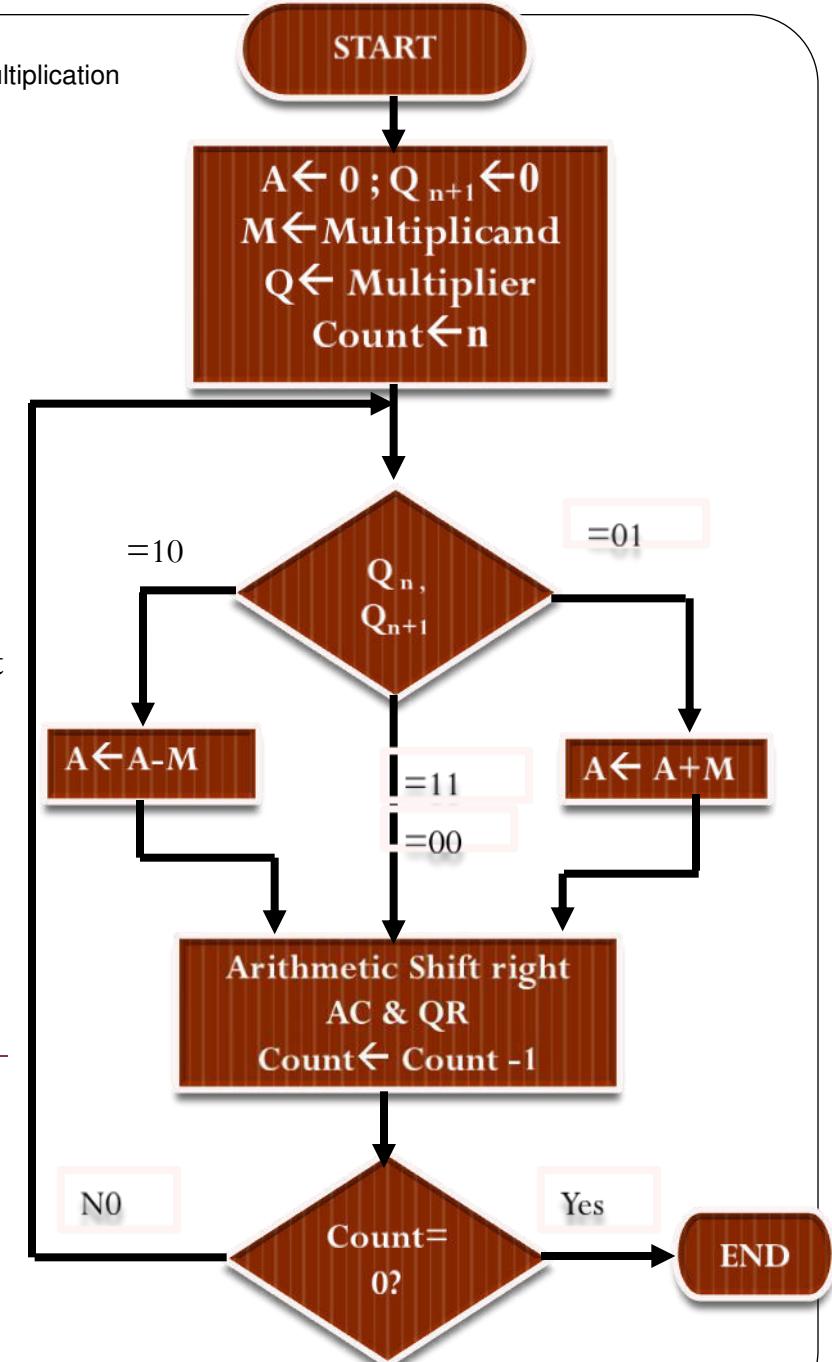
Multiplicand = 7 → Binary equivalent is 0111 → M

Multiplier = 3 → Binary equivalent is 0011 → Q

-7 → Binary equivalent is 1001 → -M

$$\begin{array}{r}
 \begin{array}{r} A \ 0 \ 0 \ 0 \ 0 \\ + M \ 1 \ 0 \ 0 \ 1 \\ \hline A \ 1 \ 0 \ 0 \ 1 \end{array}
 \quad
 \begin{array}{r} A \ 1 \ 1 \ 1 \ 0 \\ + M \ 0 \ 1 \ 1 \ 1 \\ \hline A \ 0 \ 1 \ 0 \ 1 \end{array}
 \end{array}$$

Step	A	Q	$Q_{n+1}$	Action	Count
1	0 0 0 0	0 0 1 1	0	Initial	4
2	1 0 0 1	0 0 1 1	0	$A \leftarrow A - M$	
2	1 1 0 0	1 0 0 1	1	Shift	3
3	1 1 1 0	0 1 0 0	1	Shift	2
4	0 1 0 1	0 1 0 0	1	$A \leftarrow A + M$	
4	0 0 1 0	1 0 1 0	0	Shift	1
5	0 0 0 1	0 1 0 1	0	Shift	0



# Examples

- Multiply the following using Booth's algorithm

$$7 \times -3$$

$$-7 \times 3$$

$$-7 \times -3$$

$$11 \times 13$$

$$-11 \times 13$$

$$11 \times -13$$

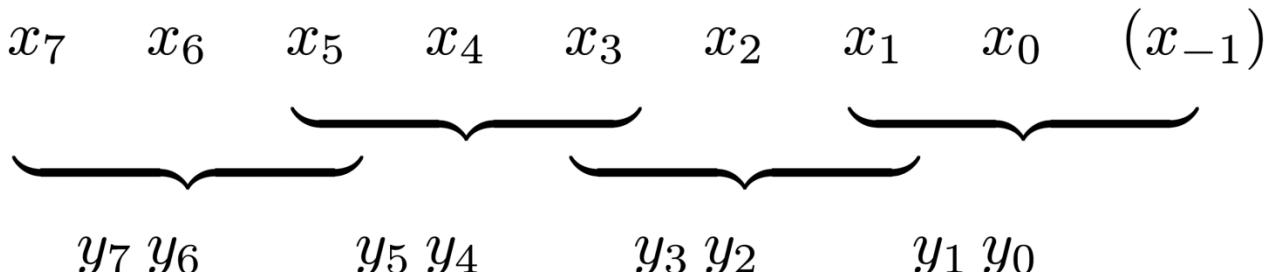
$$-11 \times -13$$

# Reference

- Morris Mano, “Computer System Architecture”, Pearson Education, 3<sup>rd</sup> edition (Chapter 10)

# Modified Booth's Algorithm

- Guarantees that the maximum number of summands that must be added is  $n/2$  for  $n$ -bit operands.
- Bit pair recoding technique
- Observe the following:
  - The pair  $(+1, -1)$  is equivalent to the pair  $(0, +1)$
  - $(+1, 0)$  is equivalent to  $(0, +2)$
  - $(-1, \bullet \quad \bullet \quad \bullet)$
- Exam



# Modified Booth's Algorithm

$$\begin{array}{r} 101101 \\ \times -1\ 1 \\ \hline 1101101 \\ 010011 \\ \hline 10010011 \end{array}$$

$$\begin{array}{r} 101101 \\ \times -1 \\ \hline 010011 \\ 0010011 \\ \hline 0010011 \end{array}$$

$$\begin{array}{r} 101101 \\ \times 1\ -1 \\ \hline 0010011 \\ 101101 \\ \hline 1101101 \end{array}$$

$$\begin{array}{r} 101101 \\ \times 1 \\ \hline 101101 \\ 1101101 \\ \hline 1101101 \end{array}$$

# Modified Booth's Algorithm

$$\begin{array}{r} 101101 \\ \times 10 \\ \hline 0000000 \\ 101101 \\ \hline 1011010 \end{array}$$

$$\begin{array}{r} 101101 \\ \times +2 \\ \hline 1011010 \\ 1011010 \\ \hline \end{array}$$

$$\begin{array}{r} 101101 \\ \times -10 \\ \hline 0000000 \\ 010011 \\ \hline 0100110 \end{array}$$

$$\begin{array}{r} 101101 \\ \times -2 \\ \hline 0100110 \\ 0100110 \\ \hline \end{array}$$

# Table of Multiplicand and Selection decisions

Multiplier Bit-Pair		Multiplier bit on the right i - 1	Booths Represenation		Multiplicand selected at position i
i + 1	i		0	0	
0	0	0	0	0	$0 \times M$
0	0	1	0	1	$+1 \times M$
0	1	0	1	-1	$+1 \times M$
0	1	1	1	0	$+2 \times M$
1	0	0	-1	0	$-2 \times M$
1	0	1	-1	1	$-1 \times M$
1	1	0	0	-1	$-1 \times M$
1	1	1	0	0	$0 \times M$

Select Line (encoding)	Partial Products (operation)
000	add 0
001	add multiplicand
010	add multiplicand
011	add $2^*$ multiplicand
100	subtract $2^*$ multiplicand
101	subtract multiplicand
110	subtract multiplicand
111	subtract 0

# Multiplication requiring only n/2 summands

$$\begin{array}{r}
 0 1 1 0 1 (+13) \\
 \times 1 1 0 1 0 (-6) \\
 \hline
 \end{array}$$



$$\begin{array}{r}
 & 0 1 1 0 1 \\
 & 0 -1 +1 -1 0 \\
 \hline
 0 & 0 0 0 0 0 & 0 0 0 0 0 \\
 1 & 1 1 1 1 1 & 0 0 1 1 1 \\
 0 & 0 0 0 0 0 & 0 1 1 0 1 \\
 1 & 1 1 1 0 0 & 1 0 0 1 1 \\
 0 & 0 0 0 0 0 & 0 0 0 0 0 \\
 \hline
 1 & 1 1 0 1 1 0 0 1 0 & (-78)
 \end{array}$$

$$\begin{array}{r}
 & 0 1 1 0 1 \\
 & 0 -1 -1 -2 \\
 \hline
 1 & 1 1 1 1 1 & 0 0 1 1 0 \\
 1 & 1 1 1 0 0 & 0 0 1 1 1 \\
 0 & 0 0 0 0 0 & 0 0 0 0 0 \\
 \hline
 1 & 1 1 1 0 1 1 0 0 1 0
 \end{array}$$



# Modified Booth's Multiplication - Example

$$\text{Example: } -9 \times -13 = 117$$

$$M = 110111, \overline{M} + 1 = 001001$$

Comment	A	Q	Q <sub>-1</sub>	SC
	000000	1100	11 0	3
Subtract M	001001			
	001001			
Ashr	000100	111001	1	
Ashr	000010	011100	1	2
Add M	<u>110111</u>			
	111001			
Ashr	111100	101110	0	
Ashr	111110	0101	11 0	1
Subtract M	<u>001001</u>			
	000111			
Ashr	000011	101011	1	
Ashr	000001	110101	1	0

# Modified Booth's Multiplication - Example

Example:  $13 \times 9 = 117$

$M = 001101$ ,  $\overline{M} + 1 = 110011$

Comment	A	Q	Q <sub>-1</sub>	SC
	000000	001001	0	3
Add M	001101			
	001101			
Ashr	000110	100100	1	
Ashr	000011	010010	0	2
Add shifted 2's	100110			
	101001			
Ashr	110100	101001	0	
Ashr	111010	010100	1	1
Add M	001101			
	000111			
Ashr	000011	101010	0	
Ashr	000001	110101	0	0

# Performance of a Processor - Instruction Execution Rate

- Processor's *throughput*—the amount of work the CPU completes in a given period of time.
- A processor is driven by a clock with a constant **frequency  $f$**  or, equivalently, a constant **cycle time  $T$** , where  $T = 1/f$ .
- **Instruction count –  $I_c$**  – The number of machine instructions executed for that program until it runs to completion or for some defined time interval
- **CPI**- Average cycles per instruction CPI for a program.
- If all instructions required the same number of clock cycles, then CPI would be a constant value for a processor.
- However, on any give processor, the number of clock cycles required varies for different types of instructions, such as load, store, branch, and so on.

# Instruction Execution Rate

- $i$  – Instruction type
- $CPI_i$  - The number of cycles required
- $I_i$  – The number of executed instructions of type  $i$  for a given program.
- Then we can calculate an overall CPI as follows

$$CPI = \frac{\sum_{i=1}^n (CPI_i \times I_i)}{I_c}$$

# Processor Time

- The processor time  $T$  needed to execute a given program can be expressed as

$$T = I_c \times CPI \times \tau$$

- During the execution of an Instruction, part of the work is done by the processor, and part of the time a word is being transferred to or from memory.
- In this latter case, the time to transfer depends on the memory cycle time, which may be greater than the processor cycle time.
- We can rewrite the preceding equation as

$$T = I_c \times [p + (m \times k)] \times \tau$$

- Where,
- $p$  – the number of processor cycles needed to decode and execute the instruction,
- $m$  – the number of memory references needed
- $k$  – The ratio between memory cycle time and processor cycletime

# MIPS

- A common measure of performance for a processor is the rate at which instructions are executed, expressed as Millions of Instructions Per Second (MIPS), referred to as the **MIPS rate**.
- We can express the MIPS rate in terms of the clock rate and CPI as follows:

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6}$$

# MIPS Example

- Consider the execution of a program which results in the execution of 2 million instructions on a 400-MHz processor.
- The instruction mix and the CPI for each instruction type are given below based on the result of a program trace experiment:

Instruction Type	CPI	Instruction Mix
Arithmetic and logic	1	60%
Load/store with cache hit	2	18%
Branch	4	12%
Memory reference with cache miss	8	10%

$$CPI = 0.6 + (2 \times 0.18) + (4 \times 0.12) + (8 \times 0.1) = 2.24.$$

MIPS rate is  $(400 \times 10^6)/(2.24 \times 10^6) \approx 178$ .

# MFLOPS

- Floating point performance is expressed as millions of floating-point operations per second (**MFLOPS**), defined as follows:

$$\text{MFLOPS rate} = \frac{\text{Number of executed floating-point operations in a program}}{\text{Execution time} \times 10^6}$$

# Benchmarks

- Measures such as MIPS and MFLOPS have proven inadequate to evaluate the performance of processors.
- Because of differences in instruction sets, the instruction execution rate is not a valid means of comparing the performance of different architectures.
- For example, consider this high-level language statement:

$$A = B + C$$

- With the traditional instruction set architecture, referred to as a complex instruction set computer (CISC), this instruction can be compiled into one processor instruction:

add mem(B), mem(C), mem (A)

# Benchmarks

- On a typical RISC machine, the compilation would look something like this:

```
load mem(B), reg(1);
          load mem(C), reg(2);
          add reg(1), reg(2), reg(3);
          store reg(3), mem (A)
```
- Because of the nature of the RISC architecture, both machines may execute the original high-level language instruction in about the same time.
- If this example is representative of the two machines, then if the CISC machine is rated at 1 MIPS, the RISC machine would be rated at 4 MIPS.
- But both do the same amount of high-level language work in the same amount of time.

# Benchmarks

- Measure the performance of systems using a set of benchmark programs.
- The same set of programs can be run on different machines and the execution times compared.
- Collection of benchmark suites is defined and maintained by the System Performance Evaluation Corporation (SPEC), an industry consortium.
- Ex: SPECCPU2006, SPECjvm98, SPECweb99

# Amdahl's Law

- Amdahl's law was first proposed by Gene Amdahl in [AMDA67] and deals with the potential speedup of a program using multiple processors compared to a single processor.
- Consider a program running on a single processor such that a fraction  $(1 - f)$  of the execution time involves code that is inherently serial and a fraction  $f$  that involves code that is infinitely parallelizable with no scheduling overhead.
- Let  $T$  be the total execution time of the program using a single processor.

# Speedup

- Then the speedup using a parallel processor with  $N$  processors that fully exploits the parallel portion of the program is as follows:

$$\begin{aligned}\text{Speedup} &= \frac{\text{time to execute program on a single processor}}{\text{time to execute program on } N \text{ parallel processors}} \\ &= \frac{T(1 - f) + Tf}{T(1 - f) + \frac{Tf}{N}} = \frac{1}{(1 - f) + \frac{f}{N}}\end{aligned}$$

# Speedup

- Two important conclusions can be drawn:
  - When  $f$  is small, the use of parallel processors has little effect.
  - As  $N$  approaches infinity, speedup is bound by  $1/(1 - f)$ , so that there are diminishing returns for using more processors.

# Speedup

- Consider any enhancement to a feature of a system that results in a speedup.
- The speedup can be expressed as

$$\text{Speedup} = \frac{\text{Performance after enhancement}}{\text{Performance before enhancement}} = \frac{\text{Execution time before enhancement}}{\text{Execution time after enhancement}}$$

- Suppose that a feature of the system is used during execution a fraction of the time  $f$ , before enhancement, and that the speedup of that feature after enhancement is  $SU_f$ . Then the overall speedup of the system is

$$\text{Speedup} = \frac{1}{(1 - f) + \frac{f}{SU_f}}$$

# Speedup – Example

- Suppose that a task makes extensive use of floating-point operations, with 40% of the time is consumed by floating-point operations.
- With a new hardware design, the floating-point module is speeded up by a factor of K.
- Then the overall speedup is:

$$\text{Speedup} = \frac{1}{0.6 + \frac{0.4}{K}}$$

- Thus, independent of K, the maximum speedup is 1.67.

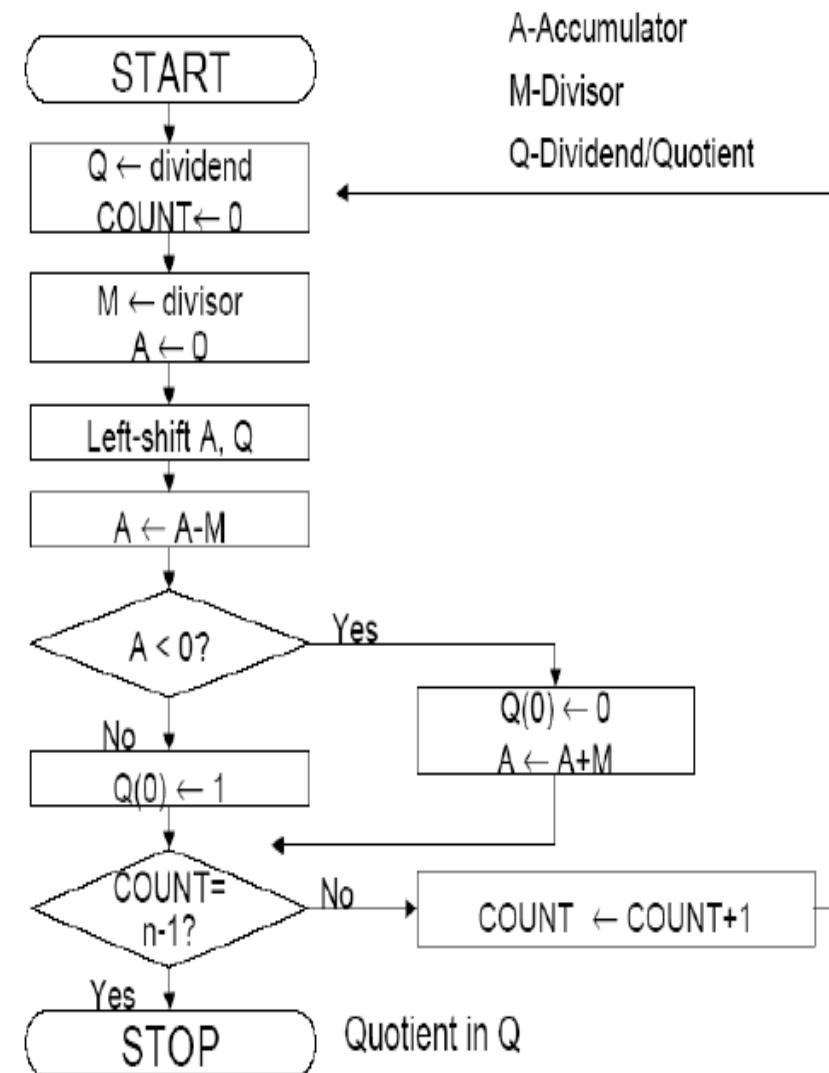
# References

- **Text Book** – William Stallings, “Computer Organization and Architecture, Designing for performance”, 8<sup>th</sup> edition, Prentice Hall.
- Internet Sources
- <https://www.coursehero.com/file/pna2ec/D3-resides-in-bit-position-6-Dr-VSaritha-Associate-Professor-SCOPE-VIT/>

# DIVISION ALGORITHMS

# Restoring Division

- Input:
  - M – positive divisor (n-bit)
  - Q – positive dividend (n-bit)
- Output:
  - Q – Quotient
  - A – Remainder
- Begin
  - A is set to 0.
  - Shift A and Q left one binary position
  - $A \leftarrow A - M$
  - If sign of A is 1
    - $q_0 \leftarrow 0$  and  $A \leftarrow A + M$  (Restore A)
  - Else
    - $q_0 \leftarrow 1$
- End



$$\begin{array}{r} 10 \\ 11 \overline{)1000} \\ 11 \\ \hline 10 \end{array}$$

Initially	0 0 0 0 0	1 0 0 0	
Shift	0 0 0 1 1	0 0 0 □	
Subtract	1 1 1 0 1		
Setq <sub>0</sub>	1 1 1 1 0		
Restore	1 1		
	0 0 0 0 1	0 0 0 0	
Shift	0 0 0 1 0	0 0 0 □	
Subtract	1 1 1 0 1		
Setq <sub>0</sub>	1 1 1 1 1		
Restore	1 1		
	0 0 0 1 0	0 0 0 0	
Shift	0 0 1 0 0	0 0 0 □	
Subtract	1 1 1 0 1		
Setq <sub>0</sub>	0 0 0 0 1		
Shift	0 0 0 1 0	0 0 0 1	
Subtract	1 1 1 0 1	0 0 1 1	
Setq <sub>0</sub>	1 1 1 1 1		
Restore	1 1		
	0 0 0 1 0	0 0 1 0	
Remainder		Quotient	

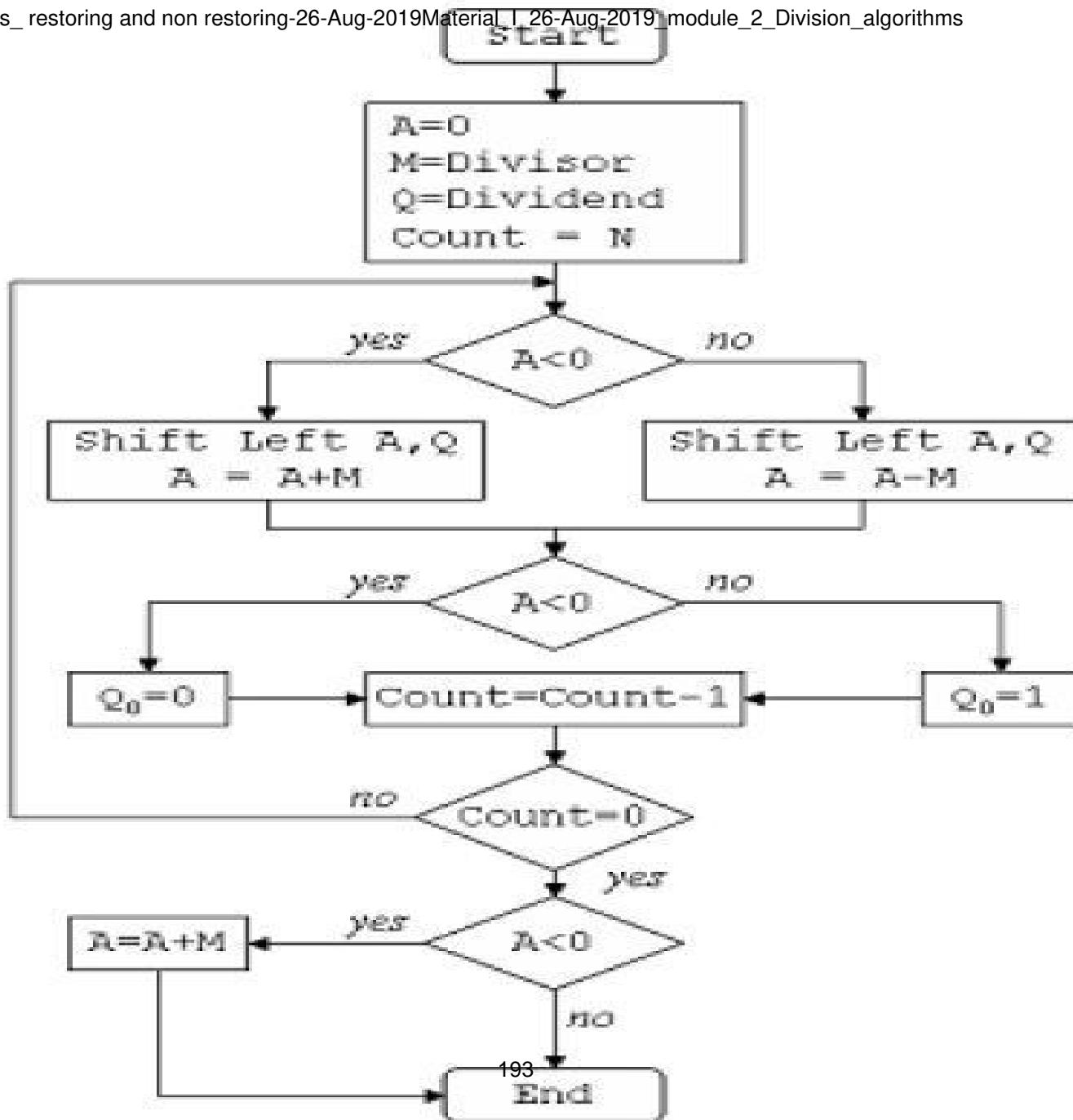
7/3

<b>A</b>	<b>Q</b>	<b>M = 0011</b>
<b>0000</b>	<b>0111</b>	Initial values
<b>0000</b>	<b>1110</b>	Shift
<b>1101</b>		$A = A - M$
<b>0000</b>	<b>1110</b>	$A = A + M$
<b>0001</b>	<b>1100</b>	Shift
<b>1110</b>		$A = A - M$
<b>0001</b>	<b>1100</b>	$A = A + M$
<b>0011</b>	<b>1000</b>	Shift
<b>0000</b>		$A = A - M$
<b>0000</b>	<b>1001</b>	$Q_0 = 1$
<b>0001</b>	<b>0010</b>	Shift
<b>1110</b>		$A = A - M$
<b>0001</b>	<b>0010</b>	$A = A + M$

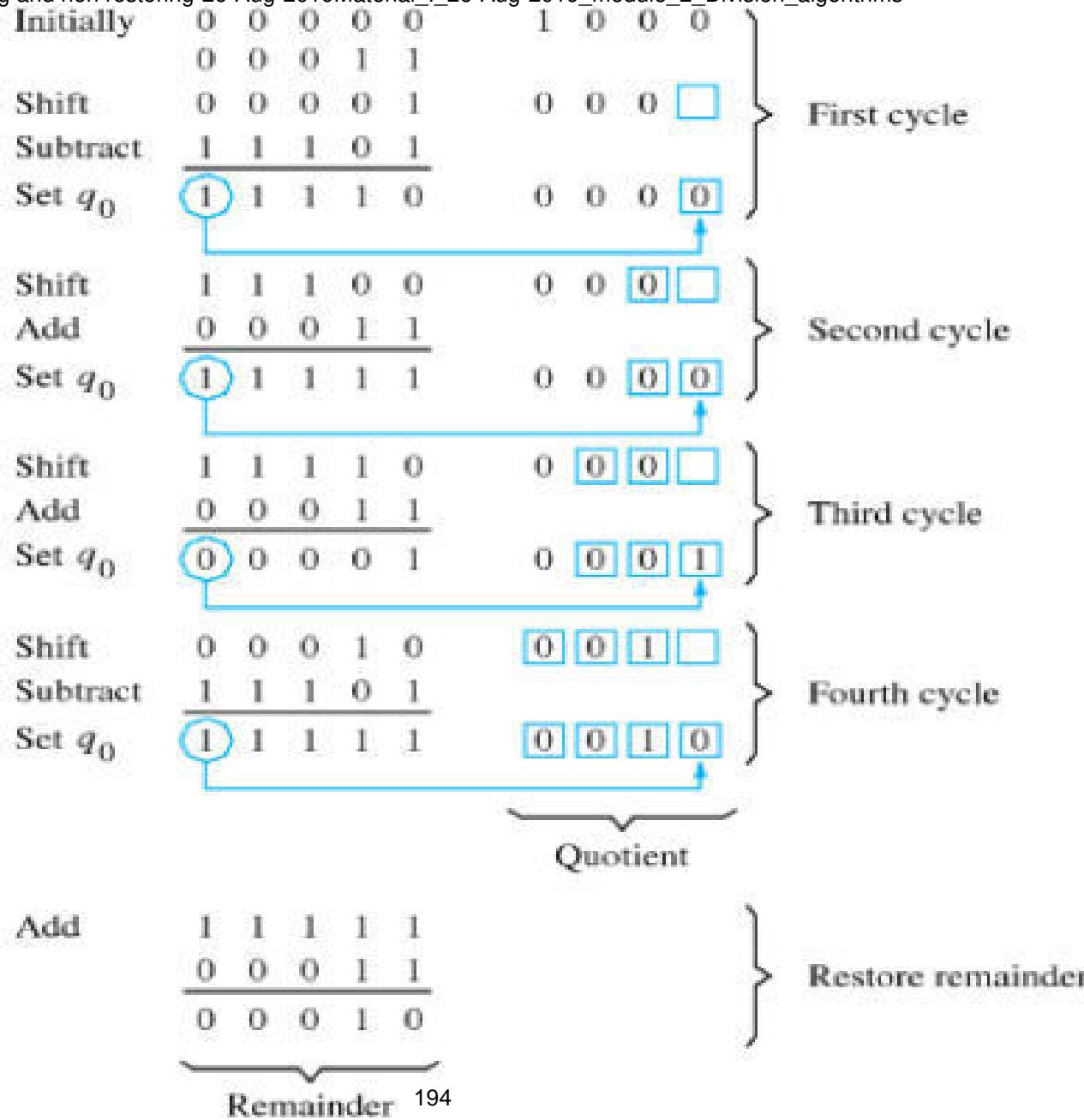
# Non-Restoring Division

---

- **Input:**
  - M – positive divisor (n-bit)
  - Q – positive dividend (n-bit)
- **Output:**
  - Q – Quotient
  - A – Remainder
- **Begin**
  - $A \leftarrow 0$
  - Do n times
    - If the sign of A is 0
      - Shift A and Q left one bit position and  $A \leftarrow A - M$
    - else
      - Shift A and Q left one bit position and  $A \leftarrow A + M$
    - If Sign of A is 0
      - $q_0 \leftarrow 1$
    - Else
      - $q_0 \leftarrow 0$
    - If sign of A is 1
      - $A \leftarrow A + M$
- End



8/3



# FLOATING POINT OPERATION

# Floating Point Number representation

- \* The location of the fractional point is not fixed to a certain location
- \* The range of the representable numbers is wide

$$F = EM$$

$m_n$	$e_k e_{k-1} \dots e_0$	$m_{n-1} m_{n-2} \dots m_0 \cdot m_{-1} \dots m_{-m}$
sign	exponent	mantissa

- Mantissa

Signed fixed point number, either an integer or a fractional number

- Exponent

Designates the position of the radix point

# Floating Point Number Representation

## Example

sign	
0	.1234567
mantissa	

$$\Rightarrow +.1234567 \times 10^{+04}$$

sign	
0	04
exponent	

## Note:

In Floating Point Number representation, only Mantissa(M) and Exponent(E) are explicitly represented. The Radix(R) and the position of the Radix Point are implied.

## Example

A binary number +1001.11 in 16-bit floating point number representation (6-bit exponent and 10-bit fractional mantissa)

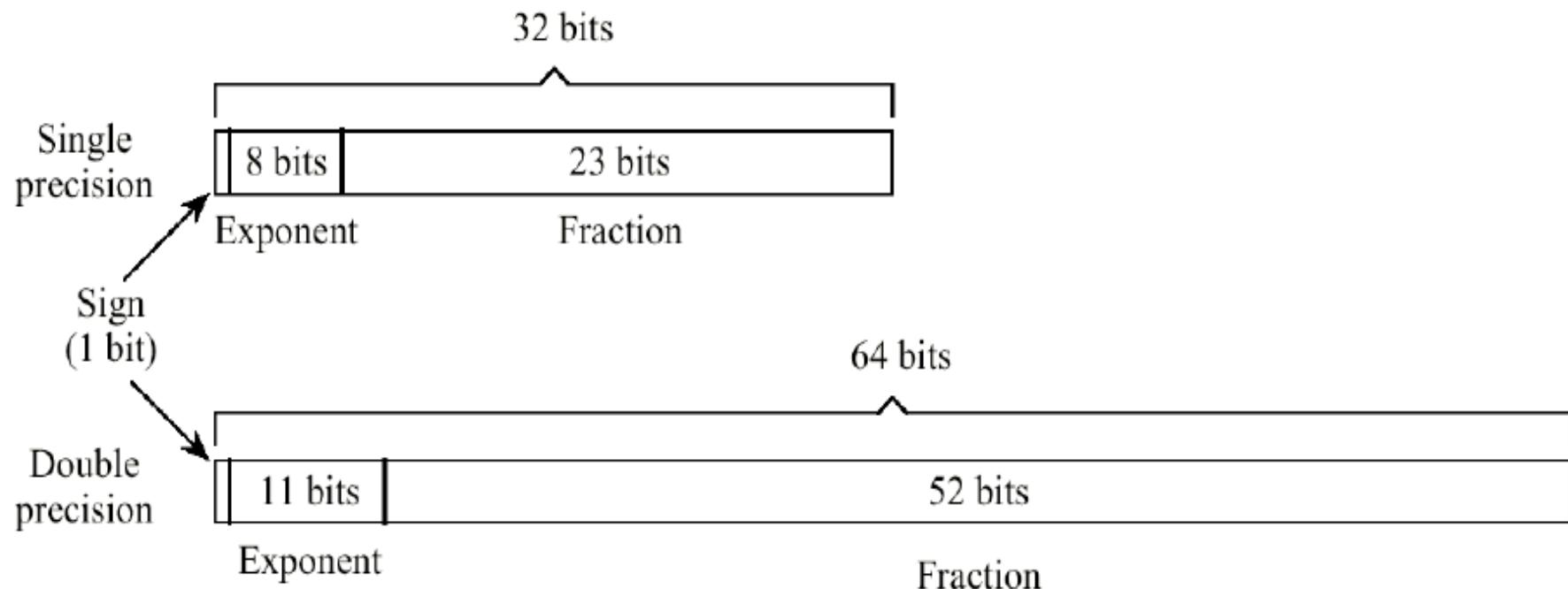
or

Sign	Exponent	Mantissa
0	0 00100	100111000
197		

## Normal Form

- There are many different floating point number representations of the same number
  - Need for a unified representation in a given computer
- *the most significant position of the mantissa contains a non-zero digit*

# IEEE-754 Floating Point Formats



# IEEE-754 Examples

	<b>Value</b>	<b>Bit Pattern</b>		
		<b>Sign</b>	<b>Exponent</b>	<b>Fraction</b>
(a)	$+1.101 \times 2^5$	0	1000 0100	101 0000 0000 0000 0000 0000
(b)	$-1.01011 \times 2^{-126}$	1	0000 0001	010 1100 0000 0000 0000 0000
(c)	$+1.0 \times 2^{127}$	0	1111 1110	000 0000 0000 0000 0000 0000
(d)	$+0$	0	0000 0000	000 0000 0000 0000 0000 0000
(e)	$-0$	1	0000 0000	000 0000 0000 0000 0000 0000
(f)	$+\infty$	0	1111 1111	000 0000 0000 0000 0000 0000
(g)	$+2^{-128}$	0	0000 0000	010 0000 0000 0000 0000 0000
(h)	$+NaN$	0	1111 1111	011 0111 0000 0000 0000 0000

# IEEE-754 Conversion Example

Represent -12.62510 in single precision IEEE-754 format.

- Step #1: Convert to target base.  $-12.62510 = -1100.101_2$
- Step #2: Normalize.  $-1100.101_2 = -1.100101_2 \times 2^3$
- Step #3: Fill in bit fields. Sign is negative, so sign bit is 1.  
Exponent is in excess 127 (not excess 128!), so exponent is represented as the  
unsigned integer  $3 + 127 = 130$ . Leading 1 of significant is hidden,  
so

final bit pattern is:

1 1000 0010 . 1001 0100 0000 0000 0000 000

# Character Representation ASCII

## ASCII (American Standard Code for Information Interchange) Code

		MSB (3 bits)							
		0	1	2	3	4	5	6	7
LSB (4 bits)	0	NUL	DLE	SP	0	@	P	'	P
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	\$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	'	7	G	W	g	w
	8	BS	CAN	(	8	H	X	h	x
	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	
	D	CR	GS	-	=	M	]	m	}
	E	SO	RS	.	>	N	m	n	~
	F	SI	US	2021	?	O	n	o	DEL

# Control Character Representation (ASCII)

<b>NUL</b>	Null	<b>DC1</b>	Device Control 1
<b>SOH</b>	Start of Heading (CC)	<b>DC2</b>	Device Control 2
<b>STX</b>	Start of Text (CC)	<b>DC3</b>	Device Control 3
<b>ETX</b>	End of Text (CC)	<b>DC4</b>	Device Control 4
<b>EOT</b>	End of Transmission (CC)	<b>NAK</b>	Negative Acknowledge (CC)
<b>ENQ</b>	Enquiry (CC)	<b>SYN</b>	Synchronous Idle (CC)
<b>ACK</b>	Acknowledge (CC)	<b>ETB</b>	End of Transmission Block (CC)
<b>BEL</b>	Bell	<b>CAN</b>	Cancel
<b>BS</b>	Backspace (FE)	<b>EM</b>	End of Medium
<b>HT</b>	Horizontal Tab. (FE)	<b>SUB</b>	Substitute
<b>LF</b>	Line Feed (FE)	<b>ESC</b>	Escape
<b>VT</b>	Vertical Tab. (FE)	<b>FS</b>	File Separator (IS)
<b>FF</b>	Form Feed (FE)	<b>GS</b>	Group Separator (IS)
<b>CR</b>	Carriage Return (FE)	<b>RS</b>	Record Separator (IS)
<b>SO</b>	Shift Out	<b>US</b>	Unit Separator (IS)
<b>SI</b>	Shift In	<b>DEL</b>	Delete
<b>DLE</b>	Data Link Escape (CC)		

(CC) Communication Control  
 (FE) Format Effector  
 (IS) Information Separator

# The EBCDIC character code, shown with hexadecimal indices

00	NUL	20	DS	40	SP	60	-	80		A0		C0	{	E0	\
01	SOH	21	SOS	41		61	/	81	a	A1	~	C1	A	E1	
02	STX	22	FS	42		62		82	b	A2	s	C2	B	E2	S
03	ETX	23		43		63		83	c	A3	t	C3	C	E3	T
04	PF	24	BYP	44		64		84	d	A4	u	C4	D	E4	U
05	HT	25	LF	45		65		85	e	A5	v	C5	E	E5	V
06	LC	26	ETB	46		66		86	f	A6	w	C6	F	E6	W
07	DEL	27	ESC	47		67		87	g	A7	x	C7	G	E7	X
08		28		48		68		88	h	A8	y	C8	H	E8	Y
09		29		49		69		89	i	A9	z	C9	I	E9	Z
0A	SMM	2A	SM	4A	¢	6A	'	8A		AA		CA		EA	
0B	VT	2B	CU2	4B		6B	,	8B		AB		CB		EB	
0C	FF	2C		4C	<	6C	%	8C		AC		CC		EC	
0D	CR	2D	ENQ	4D	(	6D	-	8D		AD		CD		ED	
0E	SO	2E	ACK	4E	+	6E	>	8E		AE		CE		EE	
0F	SI	2F	BEL	4F		6F	?	8F		AF		CF		EF	
10	DLE	30		50	&	70		90		B0		D0	}	F0	0
11	DC1	31		51		71		91	j	B1		D1	J	F1	1
12	DC2	32	SYN	52		72		92	k	B2		D2	K	F2	2
13	TM	33		53		73		93	l	B3		D3	L	F3	3
14	RES	34	PN	54		74		94	m	B4		D4	M	F4	4
15	NL	35	RS	55		75		95	n	B5		D5	N	F5	5
16	BS	36	UC	56		76		96	o	B6		D6	O	F6	6
17	IL	37	EOT	57		77		97	p	B7		D7	P	F7	7
18	CAN	38		58		78		98	q	B8		D8	Q	F8	8
19	EM	39		59		79		99	r	B9		D9	R	F9	9
1A	CC	3A		5A	!	7A	:	9A		BA		DA		FA	
1B	CU1	3B	CU3	5B	\$	7B	#	9B		BB		DB		FB	
1C	IFS	3C	DC4	5C	.	7C	@	9C		BC		DC		FC	
1D	IGS	3D	NAK	5D	)	7D	'	9D		BD		DD		FD	
1E	IRS	3E		5E	:	7E	204	9E		BE		DE		FE	
1F	IUS	3F	SUB	5F	-	7F	"	9F		BF		DF		FF	

# The EBCDIC control character representation

STX	Start of text	RS	Reader Stop	DC1	Device Control 1	BEL	Bell
DLE	Data Link Escape	PF	Punch Off	DC2	Device Control 2	SP	Space
BS	Backspace	DS	Digit Select	DC4	Device Control 4	IL	Idle
ACK	Acknowledge	PN	Punch On	CU1	Customer Use 1	NUL	Null
SOH	Start of Heading	SM	Set Mode	CU2	Customer Use 2		
ENQ	Enquiry	LC	Lower Case	CU3	Customer Use 3		
ESC	Escape	CC	Cursor Control	SYN	Synchronous Idle		
BYP	Bypass	CR	Carriage Return	IFS	Interchange File Separator		
CAN	Cancel	EM	End of Medium	EOT	End of Transmission		
RES	Restore	FF	Form Feed	ETB	End of Transmission Block		
SI	Shift In	TM	Tape Mark	NAK	Negative Acknowledge		
SO	Shift Out	UC	Upper Case	SMM	Start of Manual Message		
DEL	Delete	FS	Field Separator	SOS	Start of Significance		
SUB	Substitute	HT	Horizontal Tab	IGS	Interchange Group Separator		
NL	New Line	VT	Vertical Tab	IRS	Interchange Record Separator		
LF	Line Feed	UC	Upper Case	IUS	Interchange Unit Separator		

# References

## Text Book

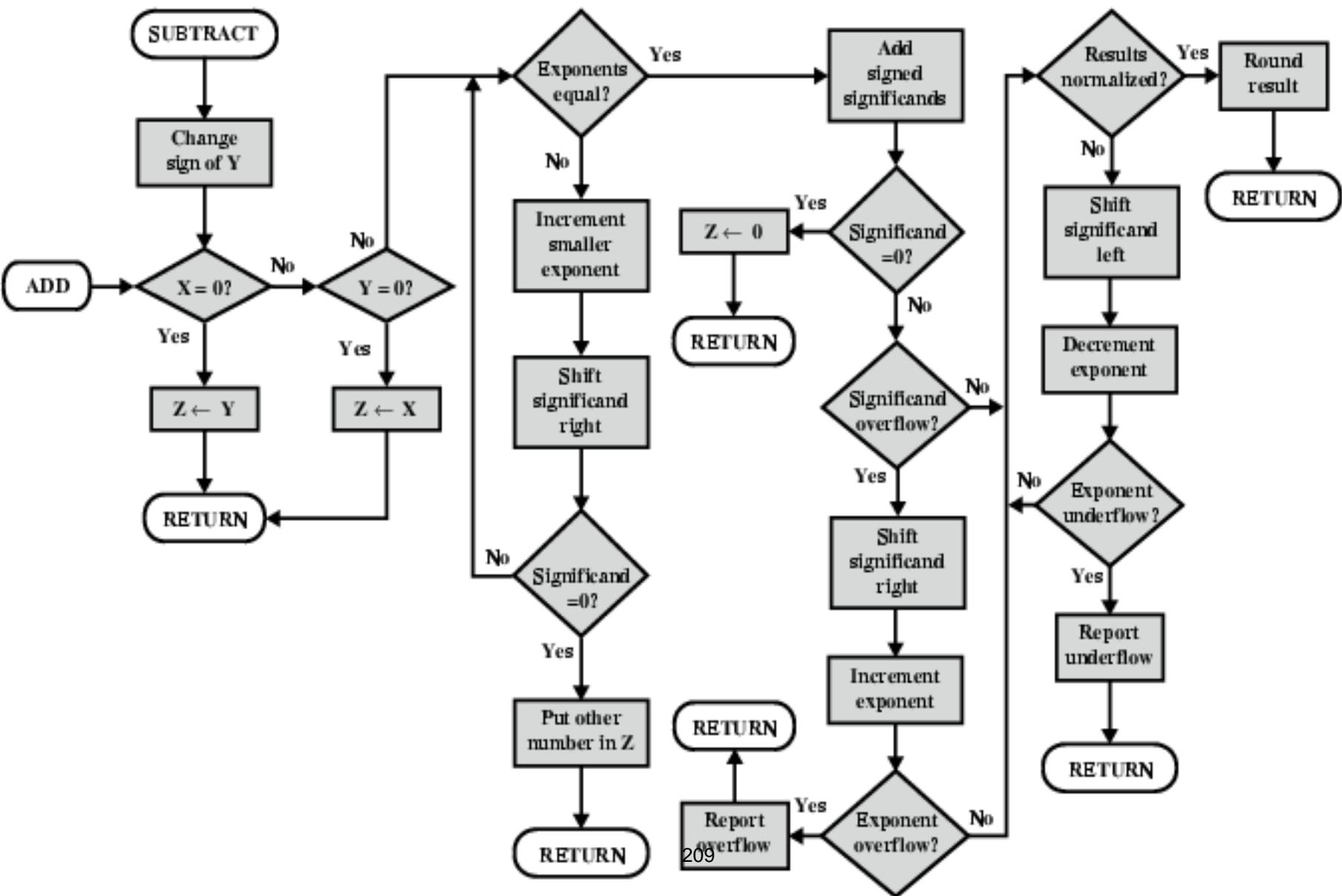
- M. M. Mano, Computer System Architecture, Prentice-Hall, 2004
- William Stallings “Computer Organization and architecture” Prentice Hall, 7th edition, 2006

# Floating Point Operations

# FP Arithmetic +/-

- Check for zeros
- Align Mantissa (adjusting exponents)
- Add or subtract Mantissa's
- Normalize the result

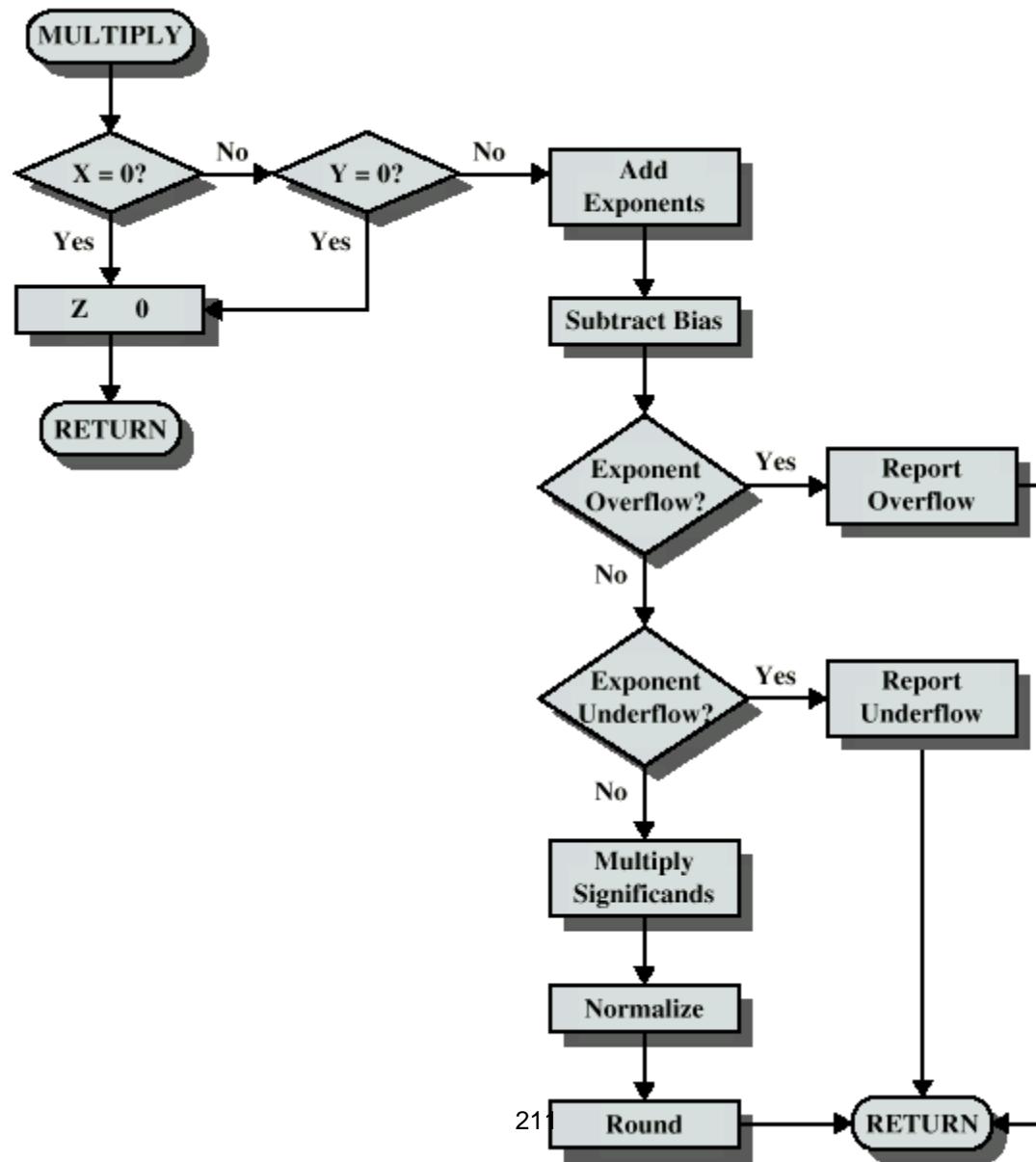
# FP Addition & Subtraction Flowchart



# Floating Point Multiplication

- Check for zero
- Add exponents
- Multiply Mantissa's
- Normalize
- Round
- All intermediate results should be in double length storage

# Floating Point Multiplication

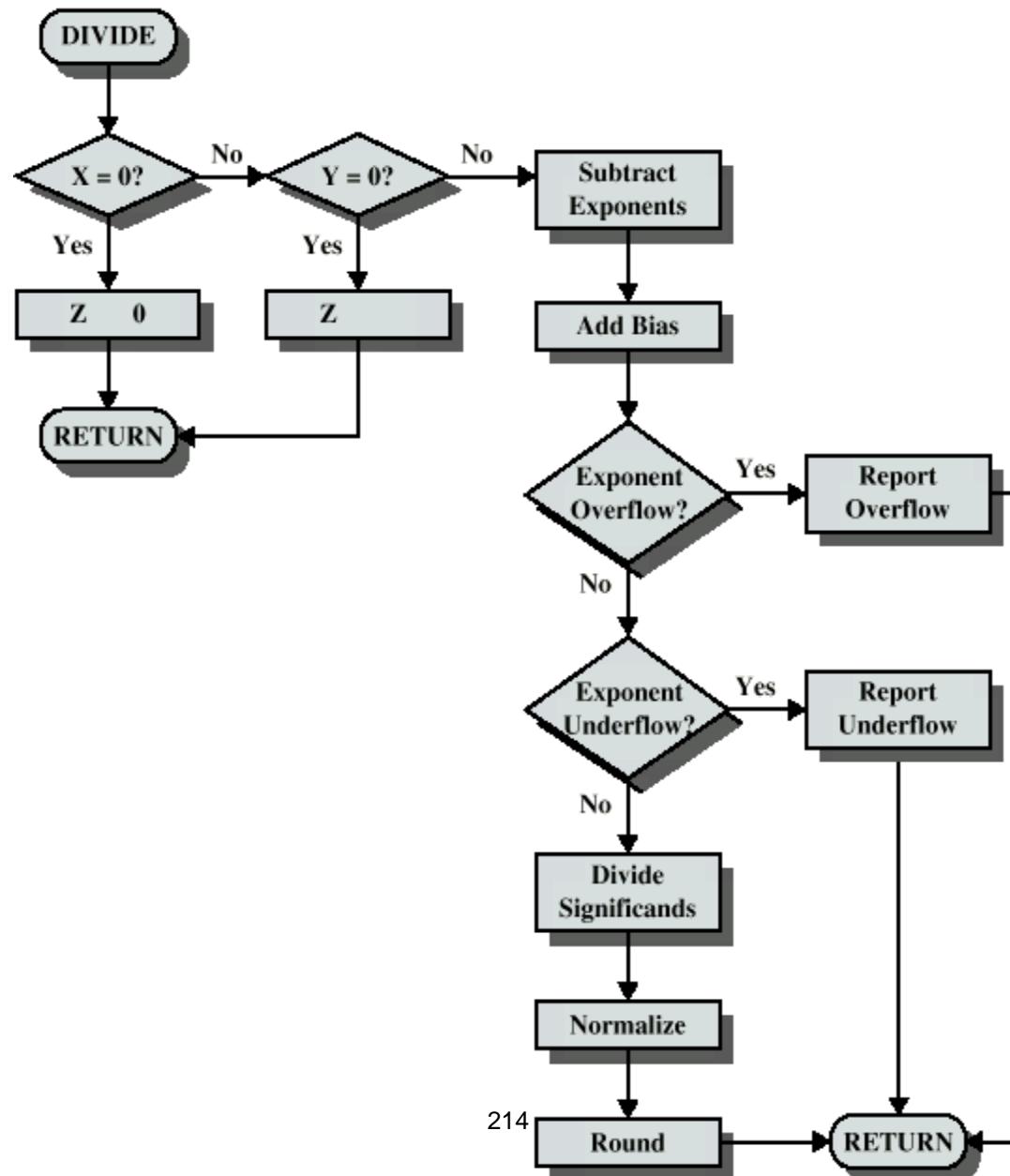


- Perform floating point multiplication:
  - 0 10000011 1100000.....
  - 0 11000000 1010000.....
  - 1 00000111 1000000.....
  - 0 11100000 1000000.....

# Floating Point Division

- Check for zero
- Subtract exponents
- Divide Mantissa's
- Normalize
- Round

# Floating Point Division



# References

## Text Book

- William Stallings “Computer Organization and architecture” Prentice Hall, 7th edition, 2006
- <http://courses.cs.tamu.edu/rabi/cpsc321/lectures/lec06.ppt>

# MEMORY ORGANIZATION

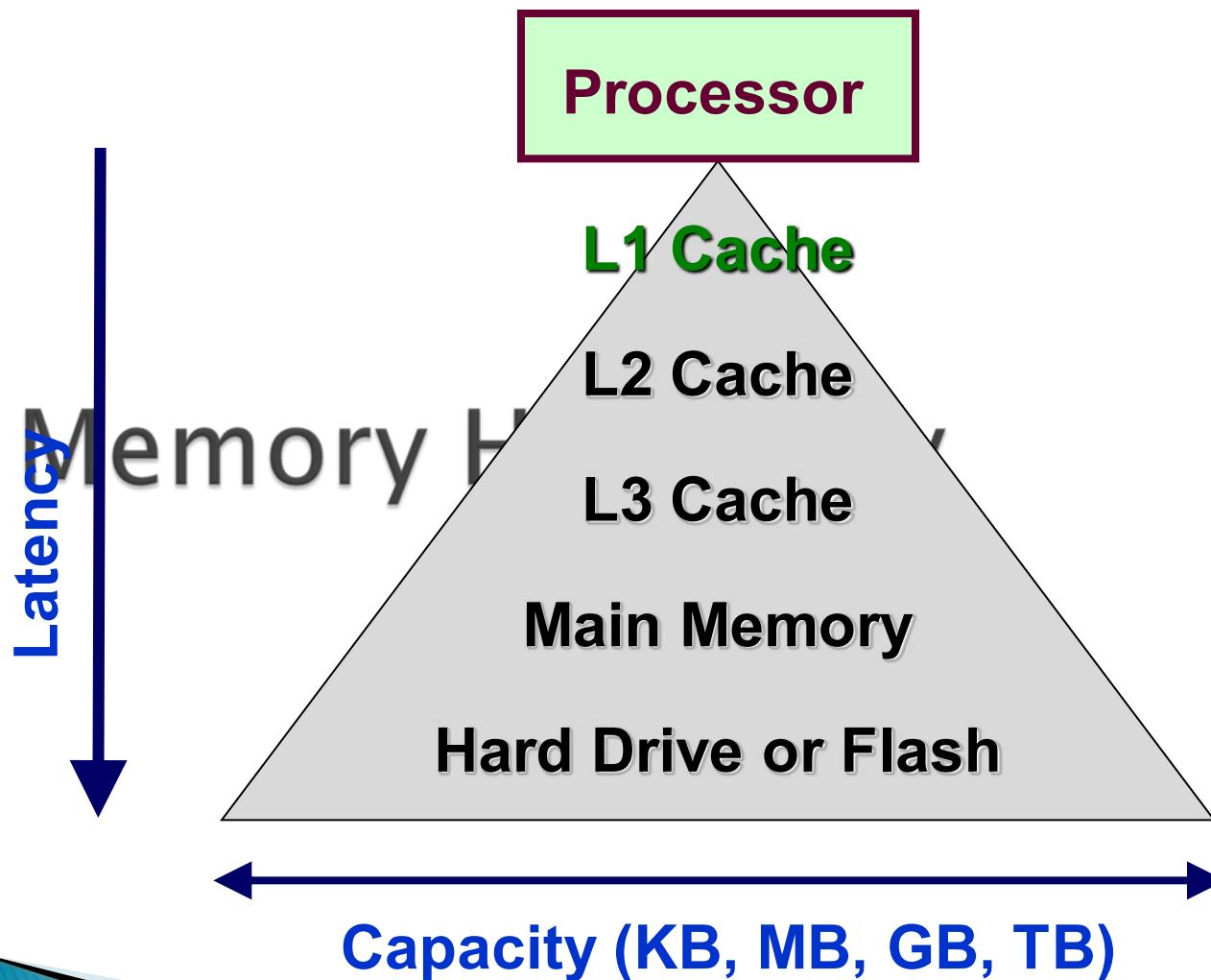
LIJO V. P

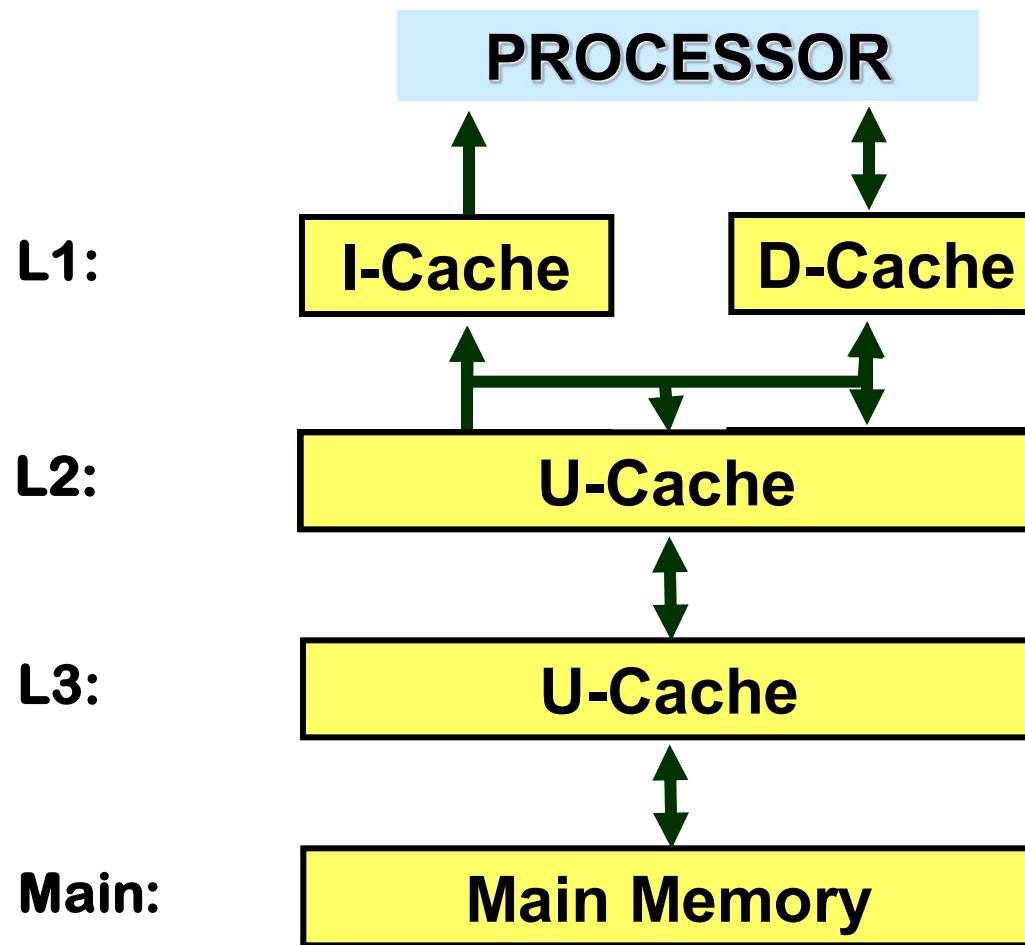
ASST. PROFESSOR, SCSE  
VIT UNIVERSITY, VELLORE

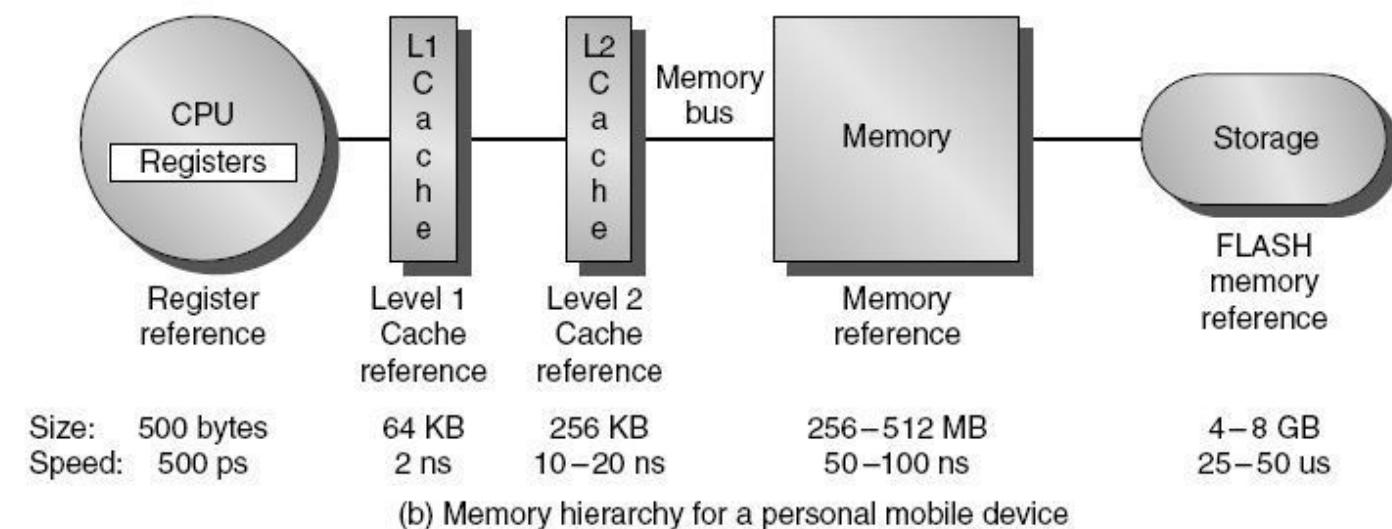
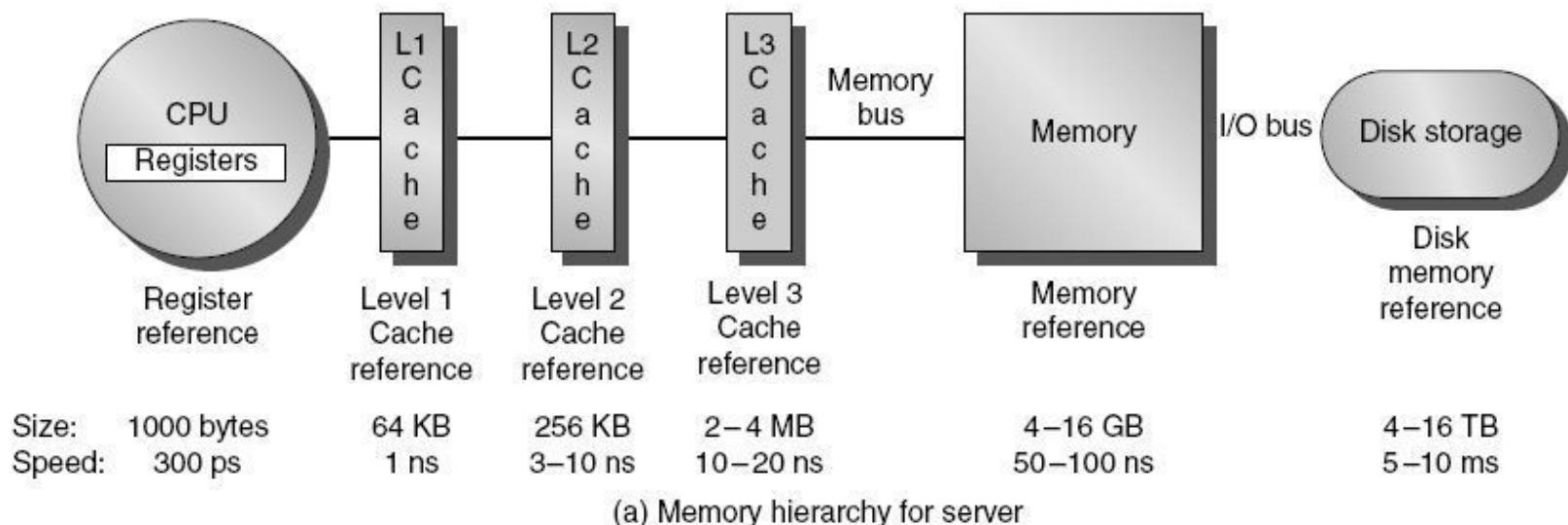
# Introduction

- ▶ Programmers want **very large memory** with **low latency**
- ▶ **Fast memory** technology is more **expensive** per bit than slower memory
- ▶ Solution: organize memory system into a hierarchy
  - Entire addressable memory space available in largest, slowest memory
  - Incrementally smaller and faster memories, **each containing a subset of the memory below it**, proceed in steps up toward the processor
- ▶ Temporal and spatial locality insures that nearly all references can be found in smaller memories
  - Gives the allusion of a large, fast memory being presented to the processor

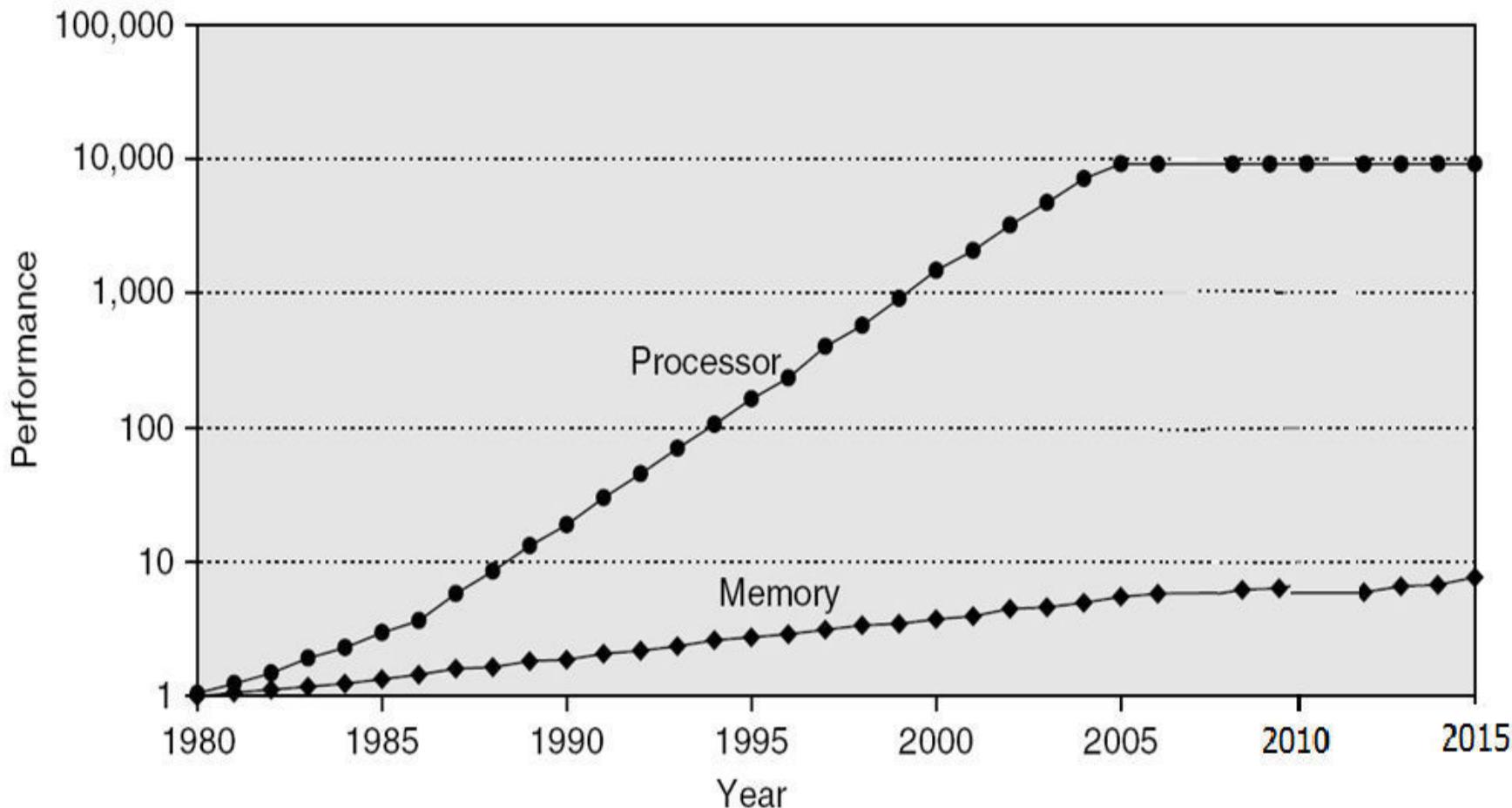
# Memory Hierarchy







# Memory Performance Gap



# Memory organization

## Byte Ordering

Ordering of bytes within a multi-byte data item

### Types:

- Big-endian
- Little-endian

# Byte Storage Methods (Byte Ordering)

## ▶ Big-Endian

- Assigns MSB to least address and LSB to highest address
- Ex: 0 × DEADBEEF

Memory Location	Value
Base Address + 0	DE
Base Address + 1	AD
Base Address + 2	BE
Base Address + 3	EF

# Byte Storage Methods contd.,

- ▶ Little Endian
  - Assigns MSB to highest address and LSB to least address
  - Ex: 0 × DEADBEEF

Memory Location	Value
Base Address + 0	EF
Base Address + 1	BE
Base Address + 2	AD
Base Address + 3	DE

# Example

- ▶ **Example:** Show the contents of memory at word address 24 if that word holds the number given by 122E 5F01H in both the big-endian and the little-endian schemes?

Big Endian				Little Endian					
MSB	----->			LSB	MSB	----->			LSB
24	25	26	27		27	26	25	24	
Word 24	12	2E	5F	01	Word 24	12	2E	5F	01

## Some points about endian-ness

- Computer systems, in use today are split between those that are big-endian, and those that are little-endian.
- This leads to problems when a big-endian computer wants to transfer data to a little-endian computer.
- Some architectures, for example the PowerPC and ARM, allow the **endian-ness** of the architecture to be changed programmatically.

# References

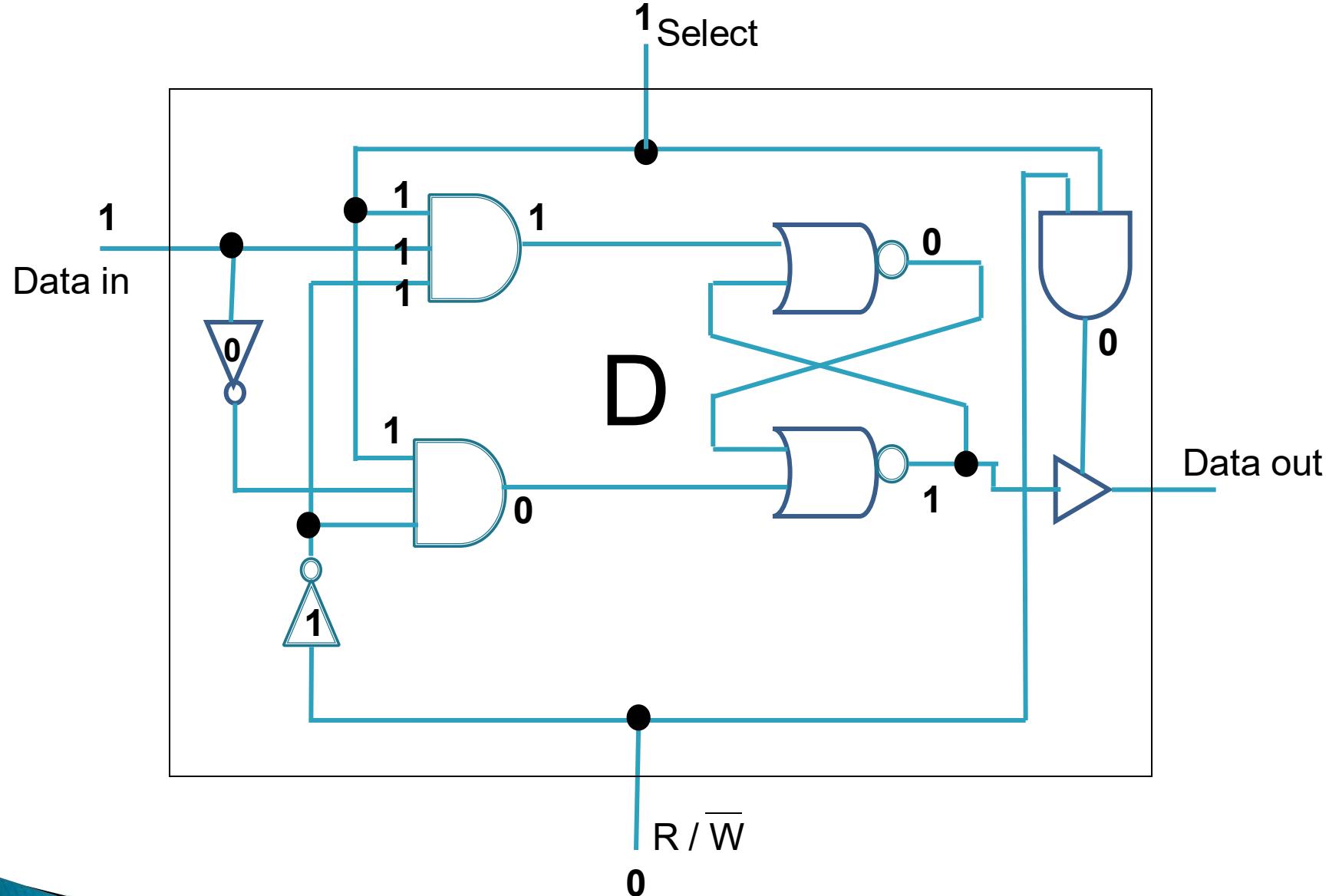
## Text Book

- ▶ William Stallings “Computer Organization and architecture” Prentice Hall, 8th edition, 2009

# Thank You

# Conceptual memory cell – static RAM cell

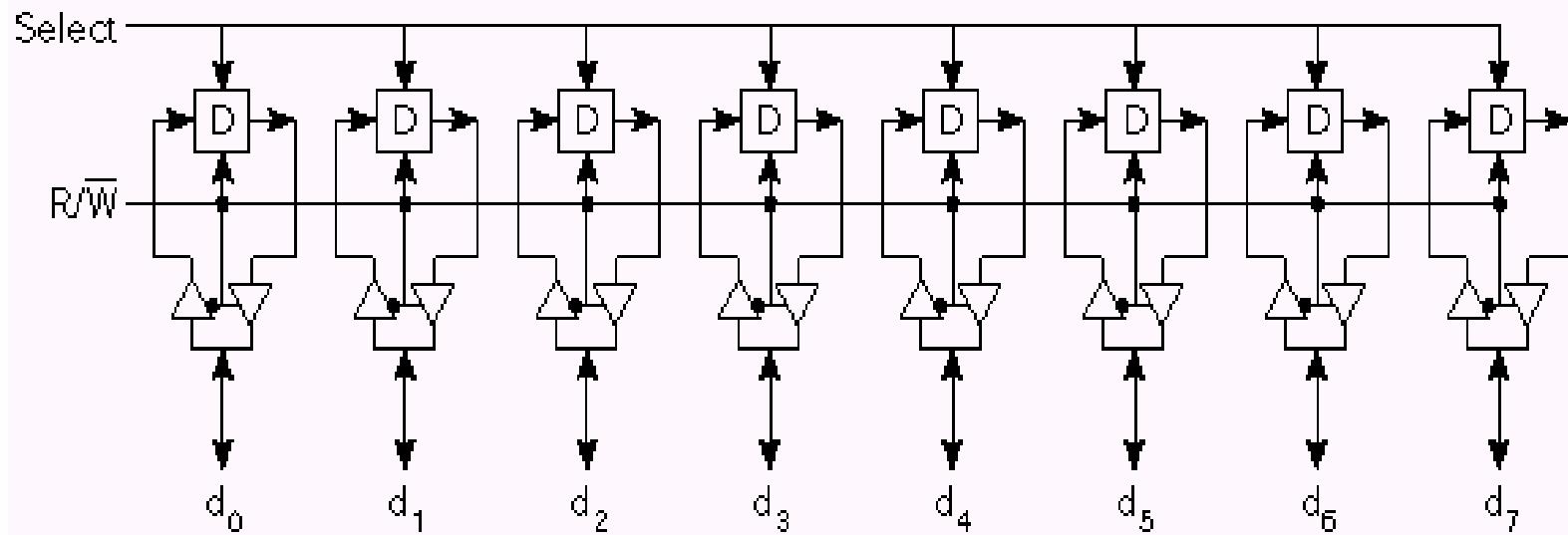
17-memory organization and types of memory-29-Aug-2019Material\_1\_29-Aug-2019\_module\_4\_Memory\_organisation



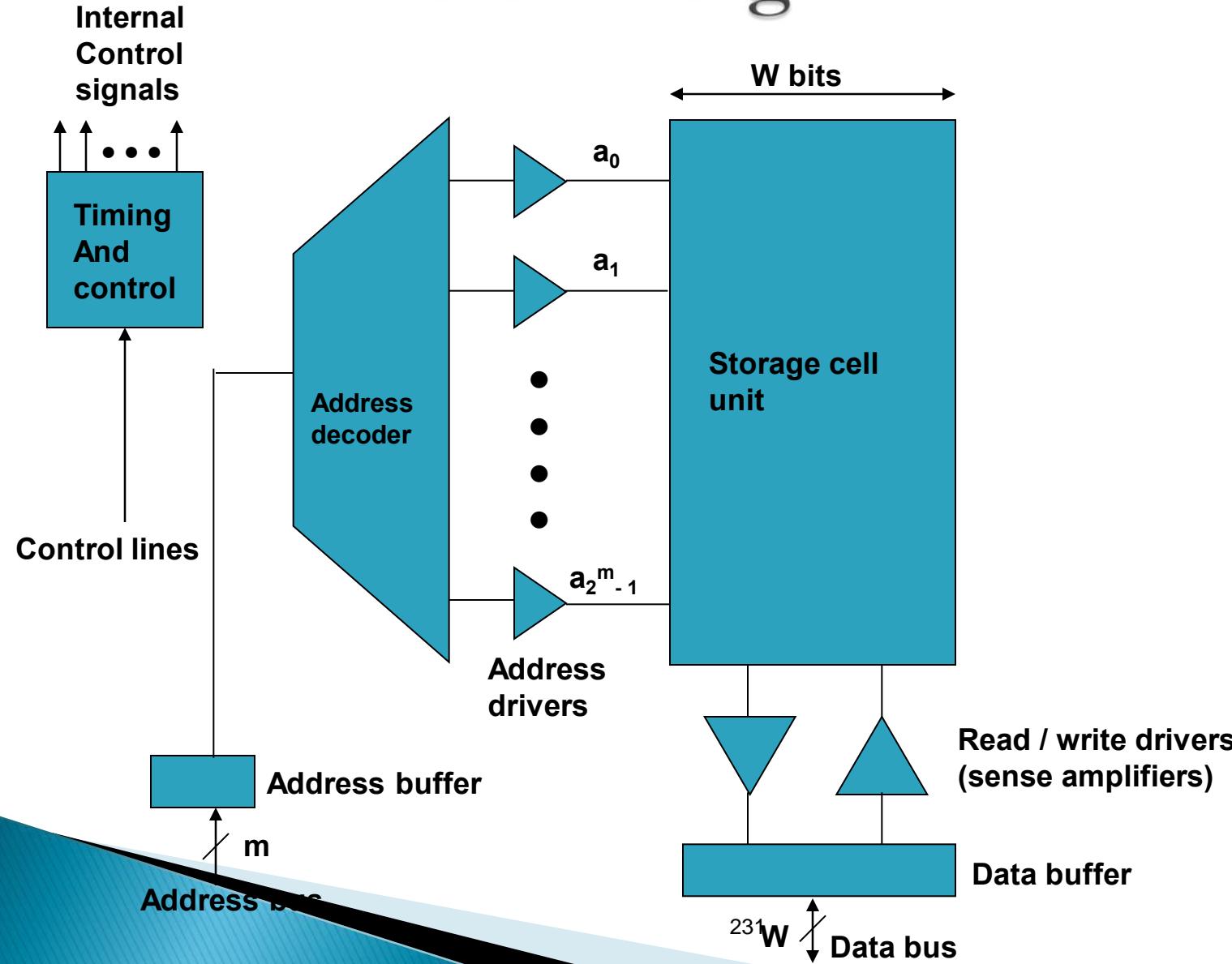
So, Tri-state buffer is in high impedance state and buffers the value

# 1 – dimensional organization

- ▶ Buffers would not be required, since the output of each cell is already buffered. They are shown to indicate buffering of the register from the data bus



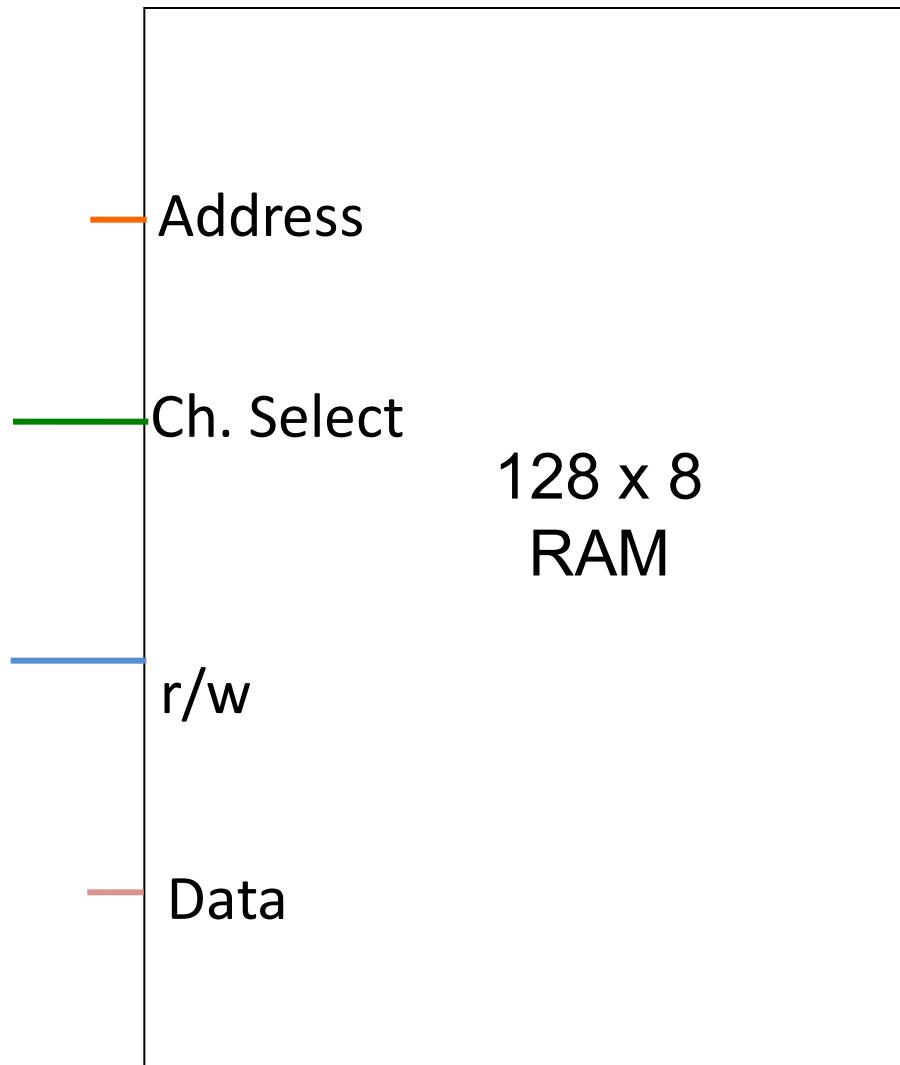
# 1 – dimensional organization



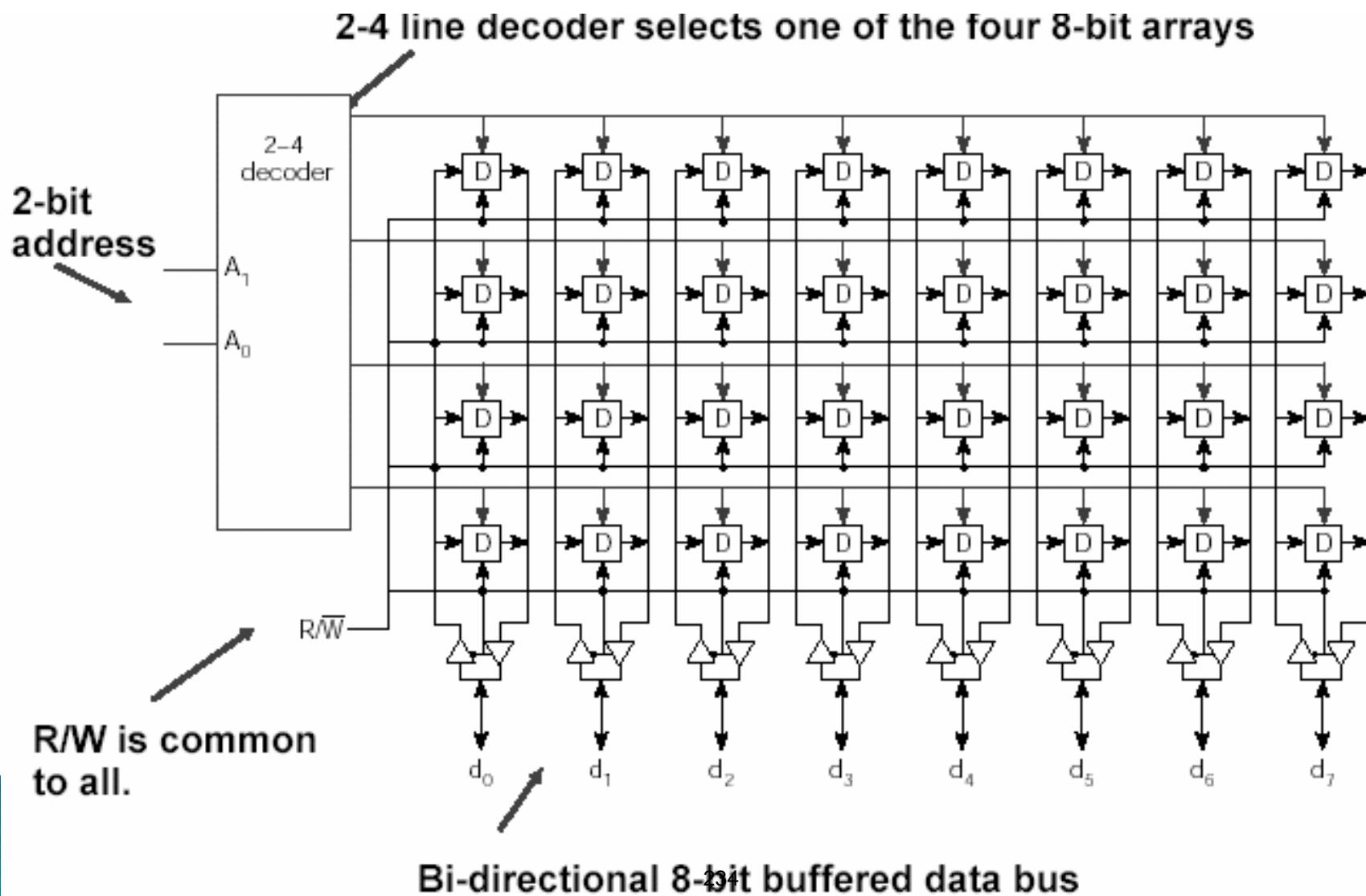


**128x 8**

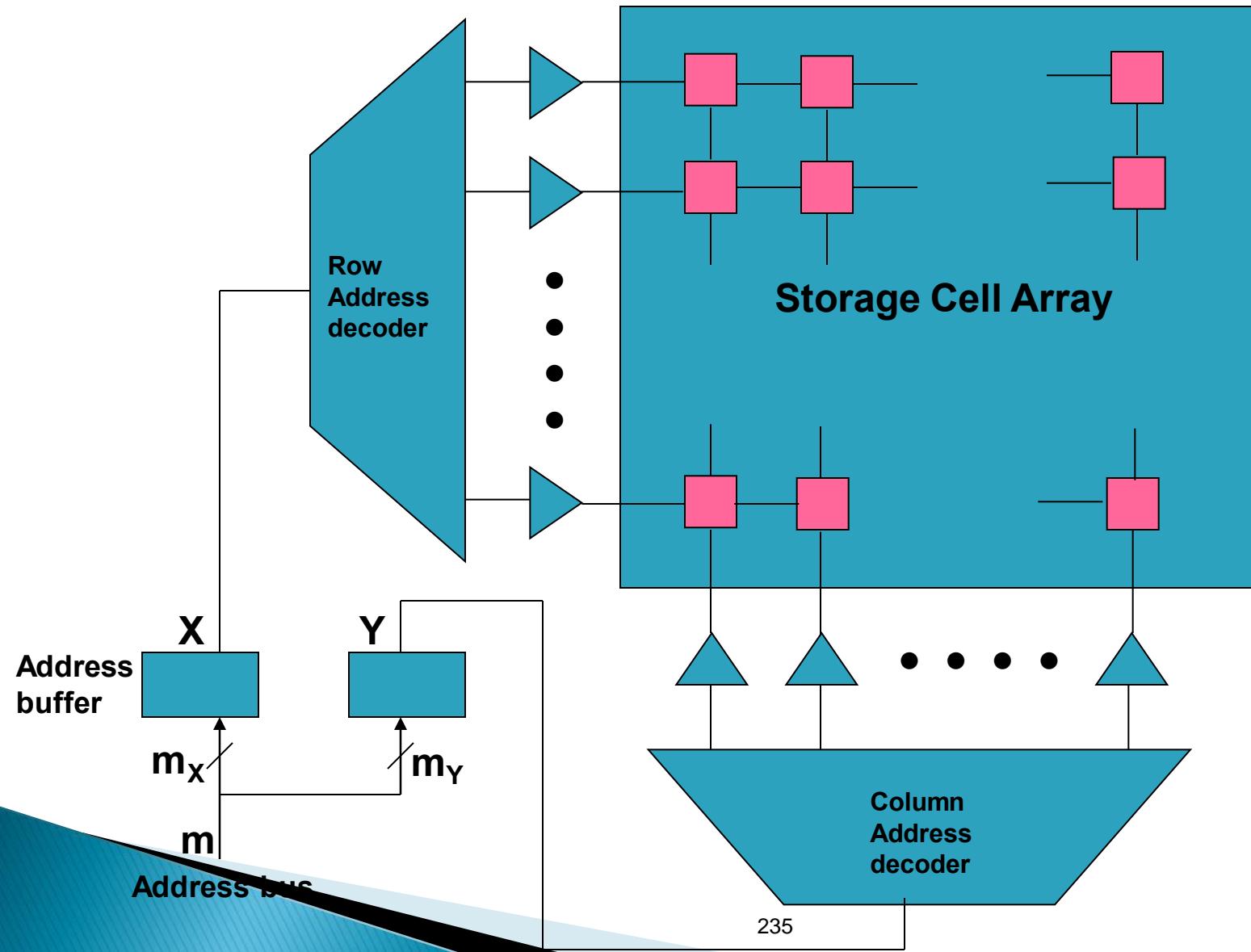
**RAM**



# 2 – dimensional organization



# 2 – dimensional organization



# References

## Text Book

- ▶ William Stallings “Computer Organization and architecture” Prentice Hall, 7th edition, 2006
- ▶ J. P. Hayes, Computer system architecture, McGraw Hill,2000

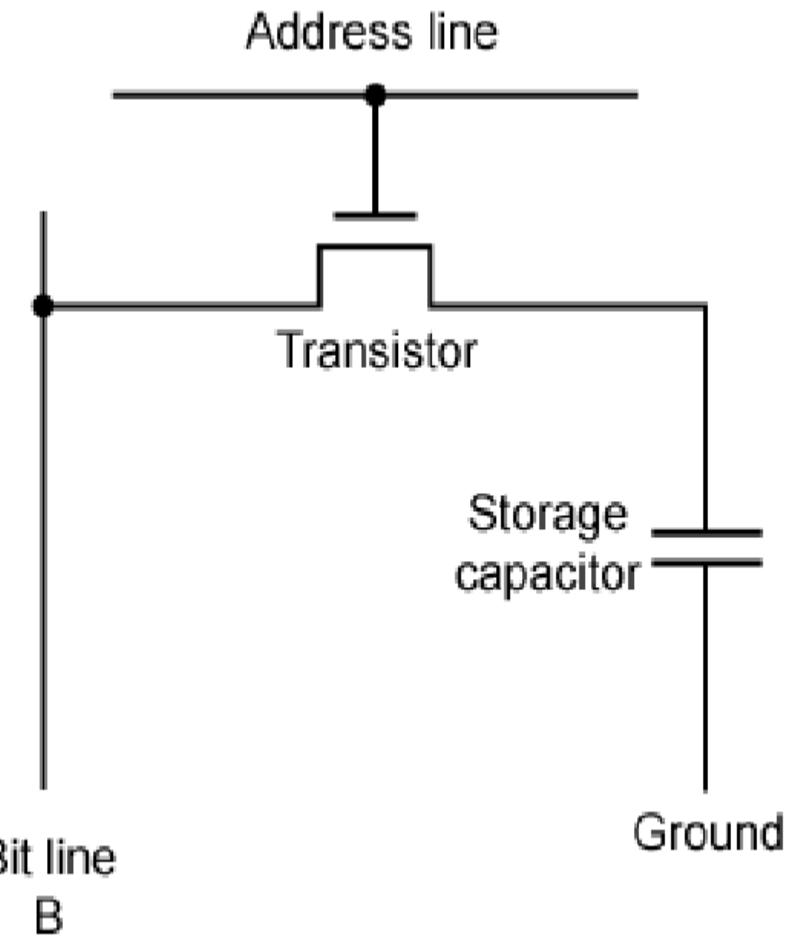


# D- RAM

- Stores data as charge on capacitors
- If capacitor is charged
  - Data is 1
- Else
  - Data is 0.
- Needs refreshing cycle as capacitors have a tendency of discharging.
- The term *dynamic* refers to this tendency of the stored charge to leak away, even with power continuously applied.
- Volatile
- When read, data is lost. So, restoring need to be done.

# D- RAM

- DRAM cell
  - Consists of a transistor and a capacitor.
  - Transistor acts a switch
    - If transistor is closed
      - Allows current to flow
    - Else
      - No current flows



# D- RAM

- Write

- Voltage signal is applied to the bit line.
  - High voltage – 1
  - Low voltage – 0
- Address line is activated allowing the charge to be transferred to the capacitor

- Read

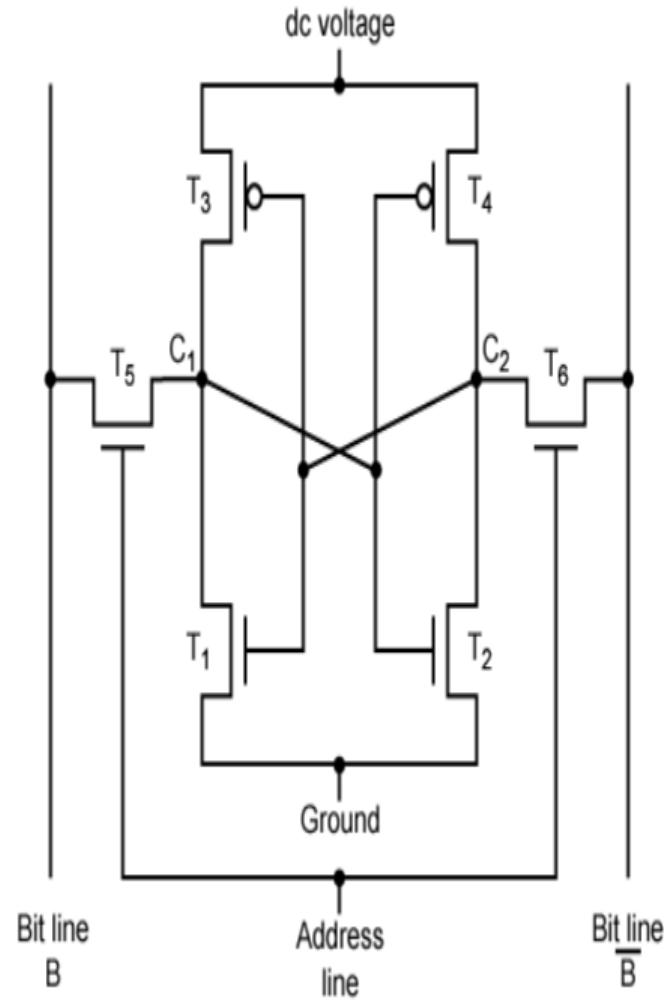
- Address line is activated
- Charge on capacitor is fed out onto a bit line and to a sense amplifier.
- Sense amplifier compares with reference value and determines if the cell contains 0 or 1.
- The value is restored
- Used for large memory requirements

# S-RAM

- Holds the data as long as the power is supplied
- Read operation don't destroy the original data.
- Expensive than DRAM but shorter cycle times.
- Used for faster small memories like cache memory.
- Uses 4 – 6 transistors to store a single bit of data
- Less power consumption than DRAM
- Complex construction

# S-RAM

- Transistor arrangement gives stable logic state
- Logic State 1
  - $C_1$  high,  $C_2$  low
  - $T_1 T_4$  off,  $T_2 T_3$  on
- Logic State 0
  - $C_2$  high,  $C_1$  low
  - $T_2 T_3$  off,  $T_1 T_4$  on
- Address line controls two transistors  $T_5 T_6$ .
- When signal is applied to address line,  $T_5$  and  $T_6$  are on.
- Write – apply value to B & complement to  $\bar{B}$
- Read – bit value is read from line B



# D-RAM Vs S-RAM

## SRAM

- Volatile
- Faster
- Smaller memory units
- Complex construction
- Don't require refreshing circuit
- Cache memory
- Digital
- Expensive

## DRAM

- Volatile
- Slower
- Larger memory units
- Simpler to build
- Require refresh
- Main memory
- Analog
- Less expensive

# MEMORY DESIGN

# Memory Design

- Available Memory chip Size  $M_{N, W}$ :  $N \times W$
- Required memory size:  $N^1 \times W^1$ , Where  $N^1 \geq N$  and  $W^1 \geq W$
- Required number of  $M_{N, W}$  chips:  $p \times q$ , Where  $p = \lceil N^1 / N \rceil$  and  $q = \lceil W^1 / W \rceil$

# Memory design

There are 3 types of organizations of  $N^1 \times W^1$  that can be formed using  $N \times W$

- $N^1 = N$  and  $W^1 > W \Rightarrow$  increasing the word size of the chip
- $N^1 > N$  and  $W^1 = W \Rightarrow$  increasing the number of words in the memory
- $N^1 > N$  and  $W^1 > W \Rightarrow$  increasing both the number of words and number of bits in each word.

There are different types of organization of  $N^1 \times W^1$  –memory using  $N \times W$  –bit chips

How many  $1024 \times 8$  RAM chips are needed to provide a memory capacity of  $2048 \times 8$ ?

Case 1:

If  $N' > N$  &  $W' = W$

Increase number of words by the factor of  $p =$

$$\left\lceil \frac{N'}{N} \right\rceil$$

How many  $1024 \times 4$  RAM chips are needed to provide a memory capacity of  $1024 \times 8$ ?

Case 2:

If  $N' = N$  &  $W' > W$

Increase the word size of a Memory by a factor of  $q =$

$$\left\lceil \frac{W'}{W} \right\rceil$$

How many  $1024 \times 4$  RAM chips are needed to provide a memory capacity of  $2048 \times 8$ ?

Case 3:

If  $N' > N$  &  $W' > W$

Increase number of words by the factor of  $p$  &

Increase the word size of a Memory by a factor of  $q$

# Memory design – Increasing the word size

- **Problem - 1**
- Design  $128 \times 16$  - bit RAM using  $128 \times 4$  - bit RAM
- Solution:  $p = 128 / 128 = 1$ ;  $q = 16 / 4 = 4$
- Therefore,  $p \times q = 1 \times 4 = 4$  memory chips of size  $128 \times 4$  are required to construct  $128 \times 16$  bit RAM

S.No	Memory Type	$N \times W$	$N^1 \times W^1$	p	q	$p * q$	x	y	z	Total
1	RAM	$128 \times 4$	$128 \times 16$	1	4	4	7	0	0	7

**x – number of address lines**

**y ( $p = 2^y$ ) – to select one among the same type of memory**

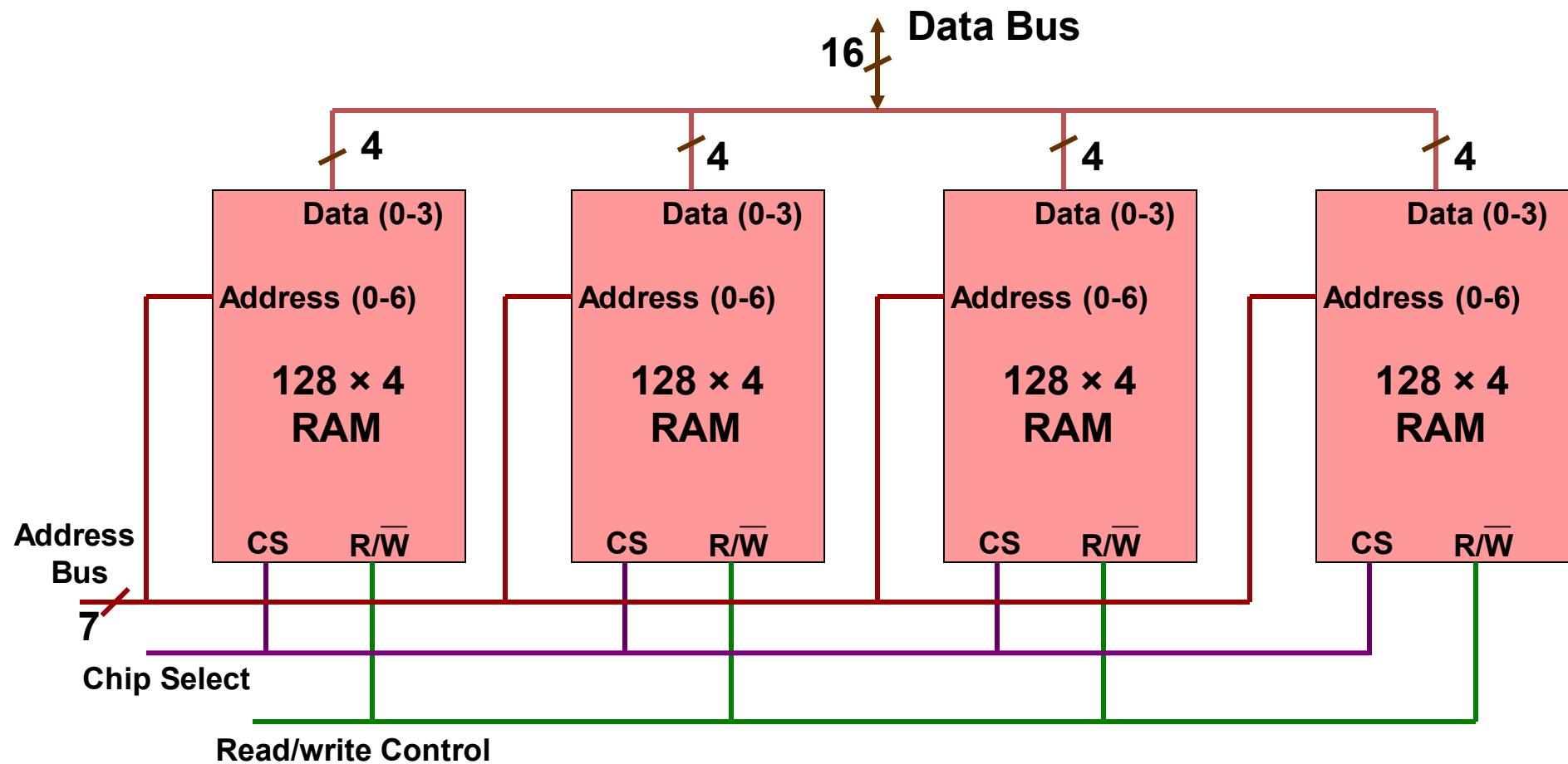
**z – to select the type of memory**

# Memory Address Map

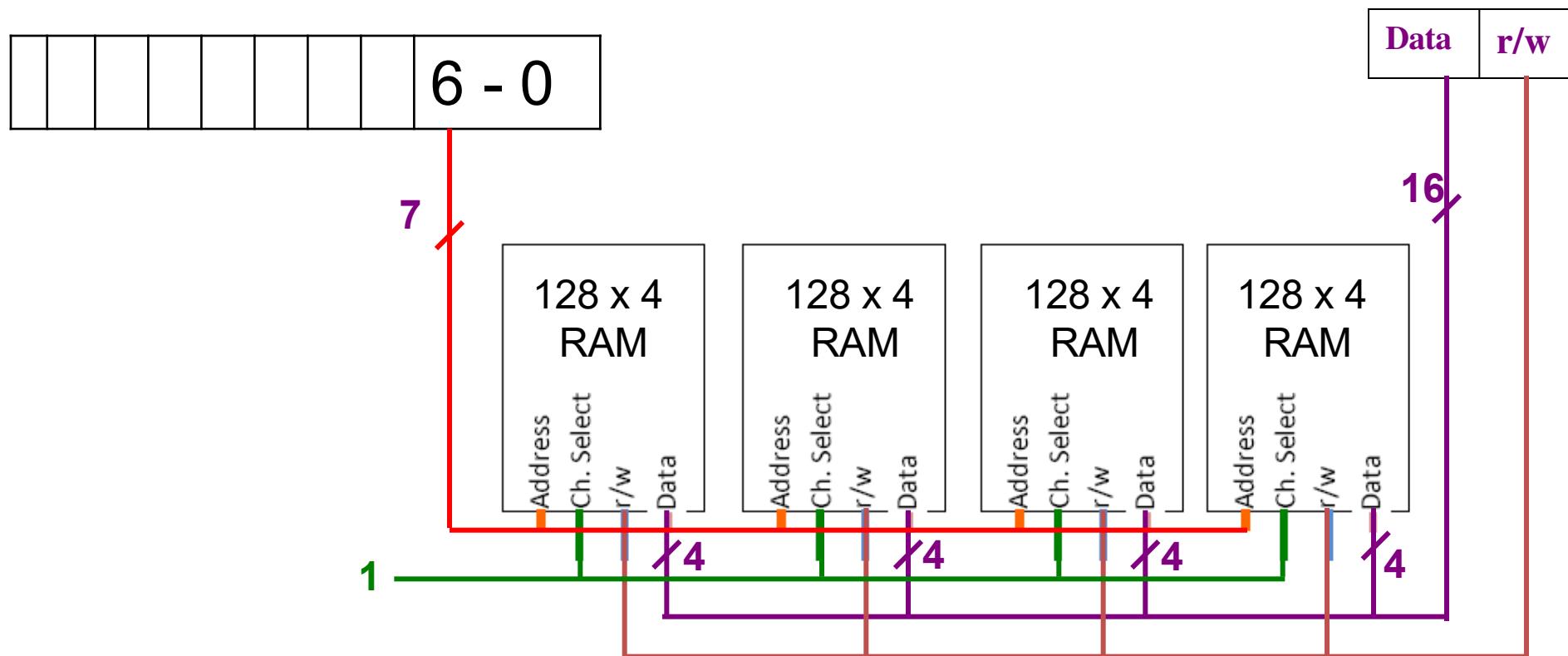
Component	Hexadecimal address		Address Bus														
	From	To	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
RAM 1.1	0000	007F										1	1	1	1	1	1
RAM 1.2	0000	007F										X	X	X	X	X	X
RAM 1.3	0000	007F										X	X	X	X	X	X
RAM 1.4	0000	007F										X	X	X	X	X	X

Substitute 0 in place of x to get 'From' address and 1 to get 'To' address

# Memory design – Increasing the word size



# Memory Design



# Memory Design – Increasing the number of words

- Problem - 2
- Design  $1024 \times 8$  - bit RAM using  $256 \times 8$  - bit RAM
- Solution:  $p = 1024 / 256 = 4$ ;  $q = 8 / 8 = 1$
- Therefore,  $p \times q = 4 \times 1 = 4$  memory chips of size  $256 \times 8$  are required to construct  $1024 \times 8$  bit RAM

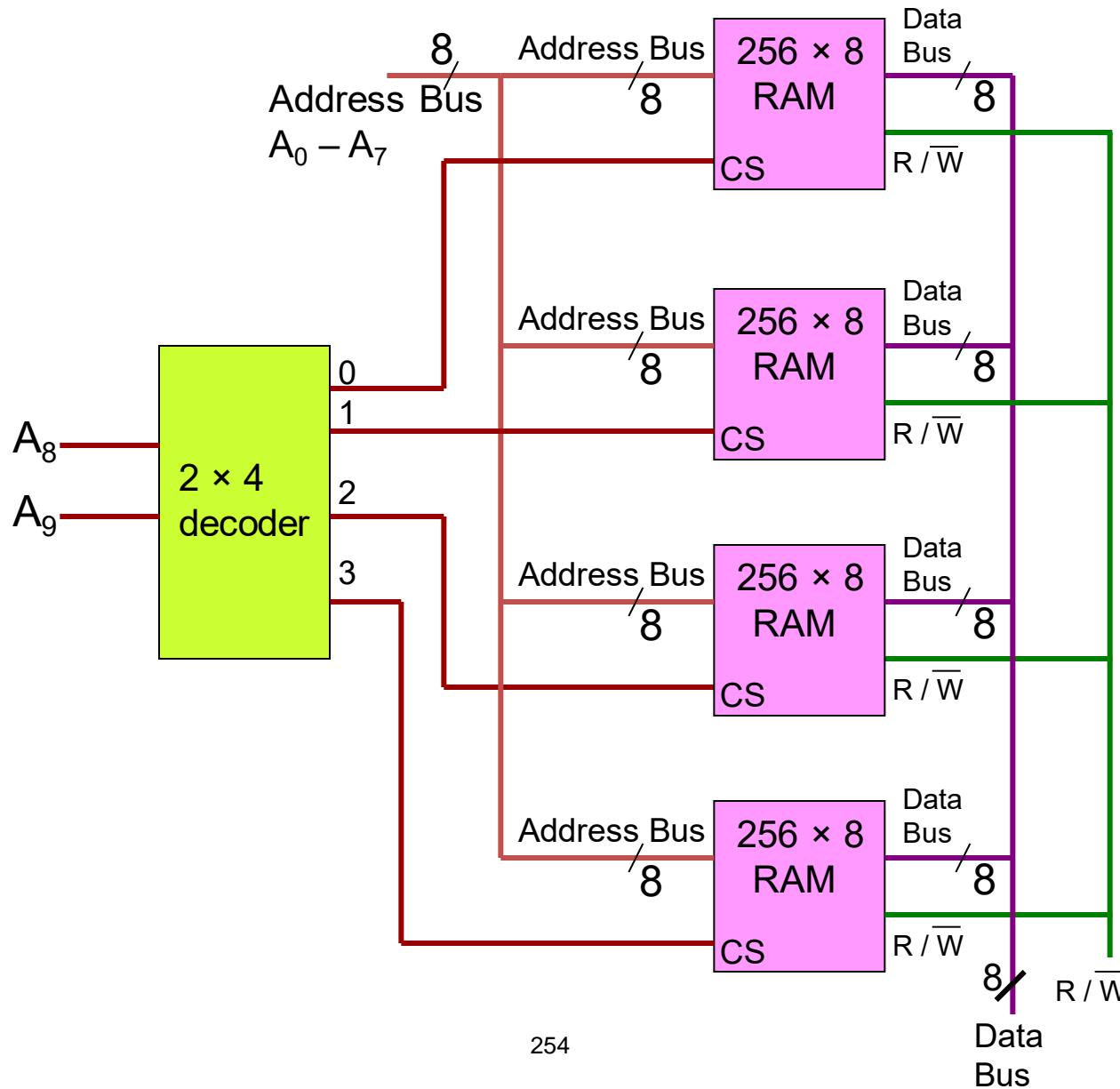
S.NO	Memory	$N \times W$	$N^1 \times W^1$	P	q	$p * q$	x	y	z	Total
1	RAM	$256 \times 8$	$1024 \times 8$	4	1	4	8	2	0	10
2										
3										
4				252						

# Memory Address Map

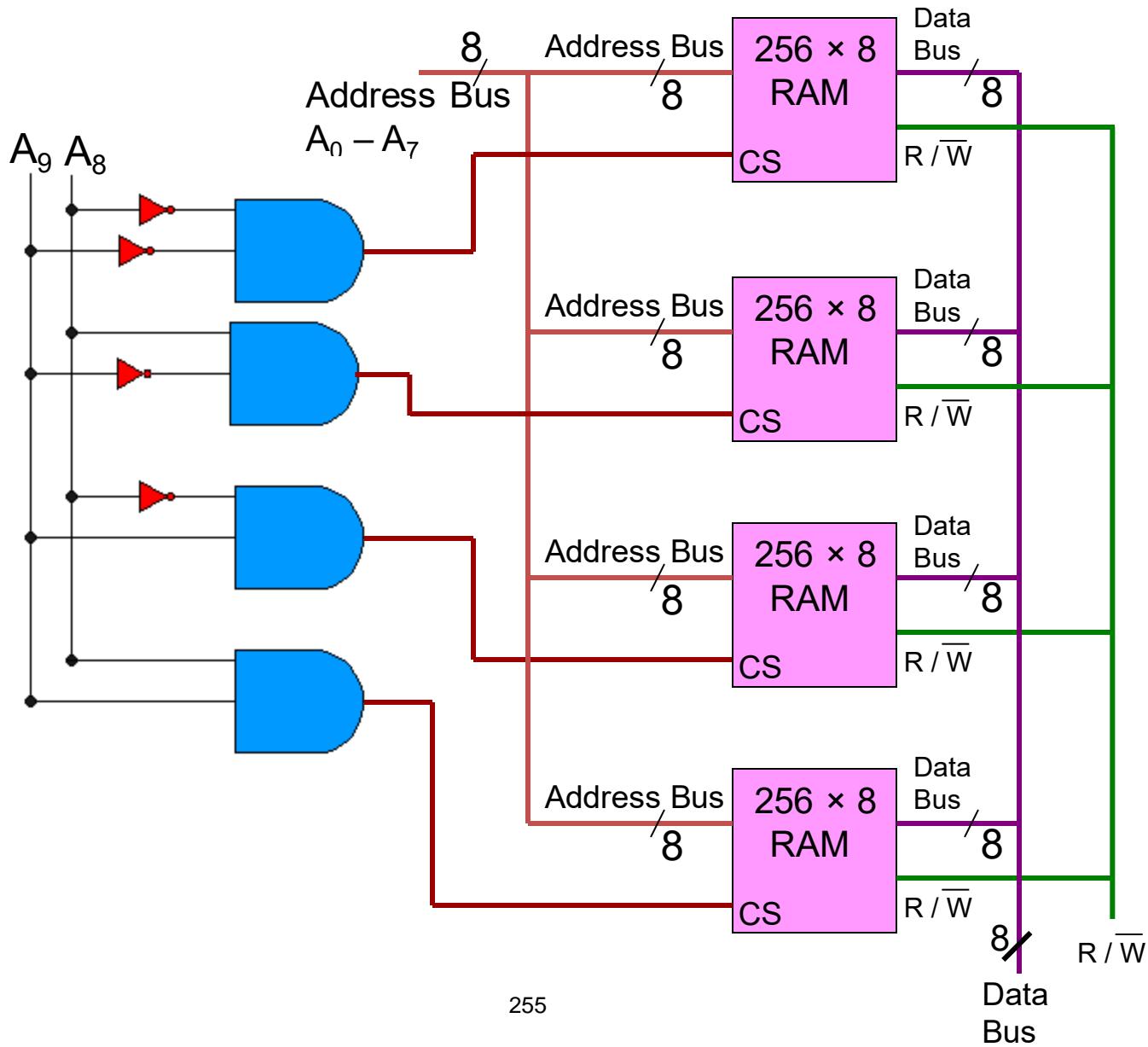
Component	Hexadecimal address		Address Bus														
	From	To	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
RAM 1	0000	00FF							0	0	1	0	0	0	0	0	0
RAM 2	0100	01FF							0	1	x	x	x	x	x	x	x
RAM 3	0200	02FF							1	0	x	x	x	x	x	x	x
RAM 4	0300	03FF							1	1	x	x	x	x	x	x	x

Substitute 0 in place of x to get 'From' address and 1 to get 'To' address

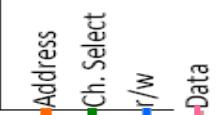
# Memory Design – Increasing the number of words



# Design with gates



**256 × 8**  
**RAM 1**



8

8

8

8

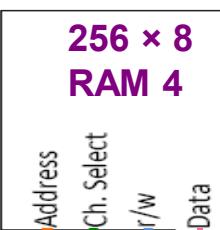
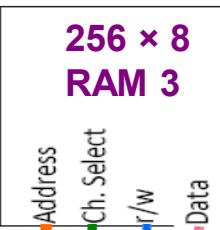
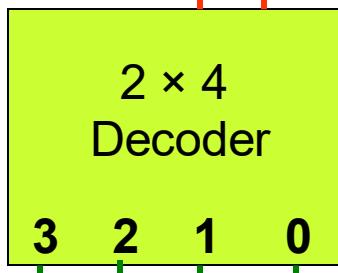
8

8

8

8

8



256

# Memory Design

- Problem - 3
- Design  $256 \times 16$  – bit RAM using  $128 \times 8$  – bit RAM chips

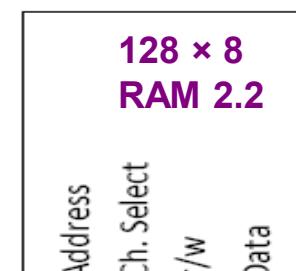
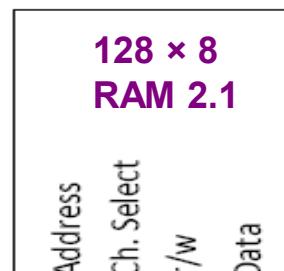
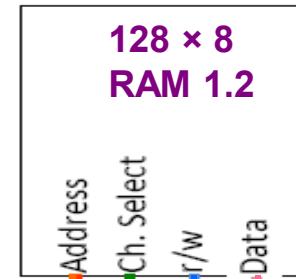
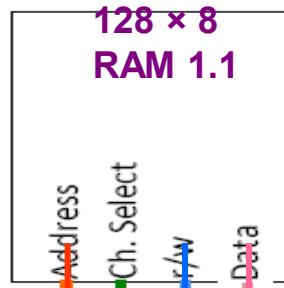
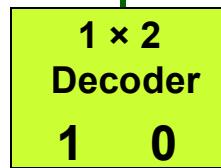
S.NO	Memory	$N \times W$	$N^1 \times W^1$	P	q	$p * q$	x	y	z	Total
1	RAM	$128 \times 8$	$256 \times 16$	2	2	4	7	1	0	8
2										
3										
4										

# Memory Address Map

Component	Hexadecimal address		Address Bus															
	From	To	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RAM 1.1	0000	007F									0	x	x	x	x	x	x	x
RAM 1.2	0000	007F									0	x	x	x	x	x	x	x
RAM 2.1	0080	00FF									1	x	x	x	x	x	x	x
RAM 2.2	0080	00FF									1	x	x	x	x	x	x	x



**Address Bus**



**16**

**16**

**8**

**8**

**8**

**8**

# Memory Design

- **Problem - 4**
- Design  $256 \times 16$  – bit RAM using  $256 \times 8$  – bit RAM chips and  $256 \times 8$  – bit ROM using  $128 \times 8$  – bit ROM chips.

# Memory Design

- **Problem - 4**
- Design  $256 \times 16$  – bit RAM using  $256 \times 8$  – bit RAM chips and  $256 \times 8$  – bit ROM using  $128 \times 8$  – bit ROM chips.

# Memory Design

- Problem - 4
- Design  $256 \times 16$  – bit RAM using  $256 \times 8$  – bit RAM chips and  $256 \times 8$  – bit ROM using  $128 \times 8$  – bit ROM chips.

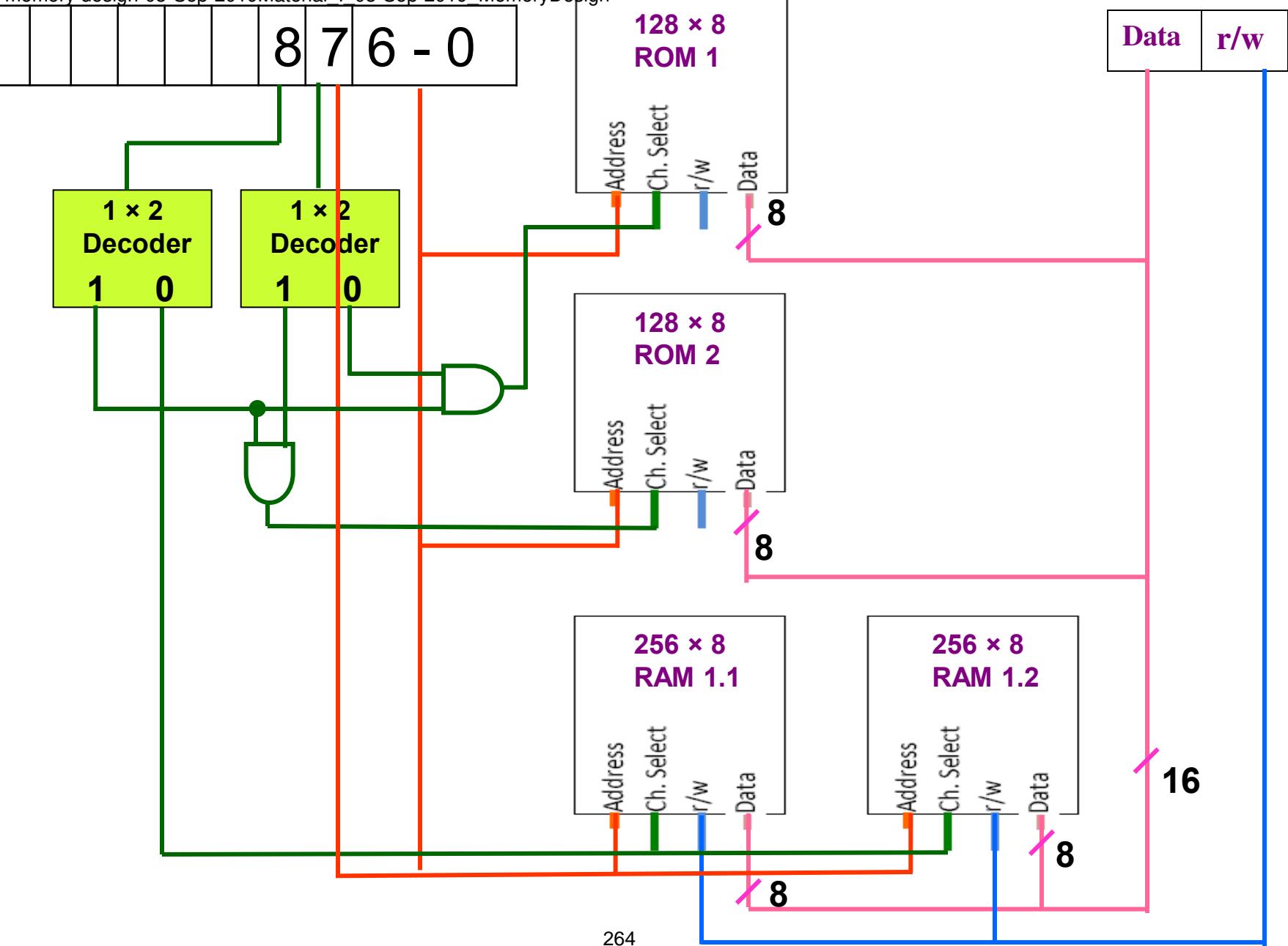
S.NO	Memory	$N \times W$	$N^1 \times W^1$	P	q	$p * q$	x	y	z	Total
1	RAM	$256 \times 8$	$256 \times 16$	1	2	2	8	0	1	9
2	Rom	$128 \times 8$	$256 \times 8$	2	1	2	7	1	1	9
3										
4										

# Memory Address Map

Component	Hexadecimal address		Address Bus														
	From	To	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
RAM 1.1	0000	00FF								0	x	x	x	x	x	x	x
RAM 1.2	0000	00FF								0	x	x	x	x	x	x	x
ROM 1	0100	017F								1	0	x	x	x	x	x	x
ROM 2	0180	01FF								1	1	x	x	x	x	x	x

# Address Bus

19-memory-design-08-Sep-2019Material | 03-Sep-2019 MemoryDesign



# Memory design

- Problem – 5
- A computer employs RAM chips of  $128 \times 8$  and ROM chips of  $512 \times 8$ . The computer system needs 256 bytes of RAM, 1024  $\times$  16 of ROM, and two interface units with 256 registers each. A memory mapped I/O configuration is used. The two higher - order bits of the address bus are assigned 00 for RAM, 01 for ROM, and 10 for interface registers.
- a. Compute total number of decoders are needed for the above system?
- b. Design a memory-address map for the above system
- c. Show the chip layout for the above design

# Requirements

S.NO	Memory	$N \times W$	$N^1 \times W^1$	P	q	$p * q$	x	y	z	Total
1	RAM	$128 \times 8$	$256 \times 8$	2	1	2	7	1	2	10
2	ROM	$512 \times 8$	$1024 \times 16$	2	2	4	9	1	2	12
3	Interface	256		2	1	2	8	1	2	11
4										

**q is 1 always for interfaces.**

**Number of registers =  $2^x$**

**P = number of interfaces**

**Number of data lines = size of registers**

# Memory Address Map

# Address Bus

19-memory design -03-Sep-2019 Material 1\_03-Sep-2019\_MemoryDesign



**128 × 8  
RAM 1**

**128 × 8  
RAM 2**

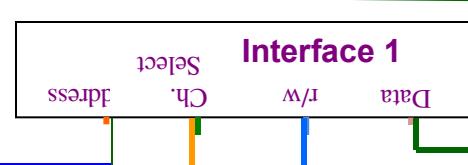
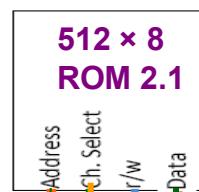
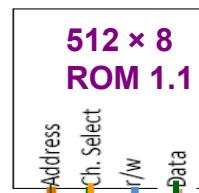
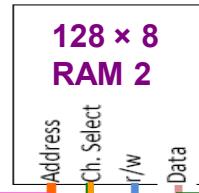
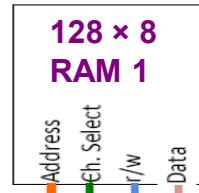
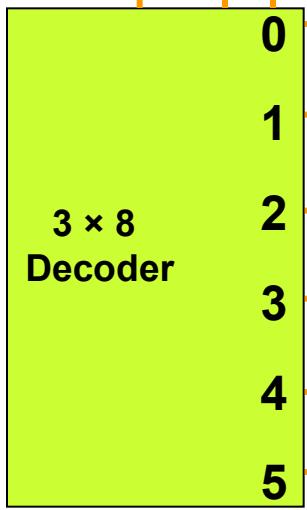
**512 × 8  
ROM 1.1**

**512 × 8  
ROM 2.1**

**Interface 1**

**Interface 2**

**Data**      **r/w**



# Example

A computer employs RAM chips of 1024 x 8 and ROM chips of 2048 x 4. The computer system needs 2K bytes of RAM, and 2K bytes of ROM and an interface unit with 256 registers each. A memory-mapped I/O configuration is used. The two higher -order bits of the address bus are assigned 00 for RAM, 01 for ROM, and 10 for interface.

- a). How many RAM and ROM chips are needed?
- b). How many lines of the address bus must be used to access Computer system memory? How many of these lines will be common to all chips?
- c). How many lines must be decoded for chip select? Specify the size of the decoder
- d). Draw a memory-address map for the system and Give the address range in hexadecimal for RAM, ROM
- e). Develop a chip layout for the above said specifications

# Example 2

A computer employs RAM chips of  $1024 \times 8$  and ROM chips of  $2048 \times 4$ . The computer system needs  $2K \times 16$  of RAM, and  $2K \times 16$  ROM and an interface unit with 256 registers each. A memory-mapped I/O configuration is used. The two higher -order bits of the address bus are assigned 00 for RAM, 01 for ROM, and 10 for interface..

- a). How many RAM and ROM chips are needed?
- b). How many lines of the address bus must be used to access total memory? How many of these lines will be common to all chips?
- c). How many lines must be decoded for chip select? Specify the size of the decoder
- d). Draw a memory-address map for the system.
- e). Draw a memory-address map for the system and Give the address range in hexadecimal for RAM, ROM
- f). Develop a chip layout for the above said specifications

# References

## Text Book(s)

- M. M. Mano, Computer System Architecture, Prentice-Hall, 2004
- J. P. Hayes, Computer system architecture, McGraw Hill, 2000

# Memory Key Characteristics

# Introduction to Memory

- **Charles Babbage started Difference engine in 1821 but failed its test in 1833, Why?**

**Due to unavailability of Memory**

- **What is Memory?**
- **A single separate storage structure that holds information in the form of bits called as Memory**
- **The binary information may be instructions and data**
- **Stored program concept was introduced with the advent of vacuum tubes by John Von Neumann 1940**



# Memory Capacity

- Number of bytes that can be stored

Term	Normal Usage	Usage as Power of 2
K ( Kilo)	$10^3$	$2^{10} = 1,024$
M (Mega)	$10^6$	$2^{20} = 1,048,576$
G (Giga)	$10^9$	$2^{30} = 1,073,741,824$
T (Tera)	$10^{12}$	$2^{40} = 1,099,511,627,776$

# Key Characteristics



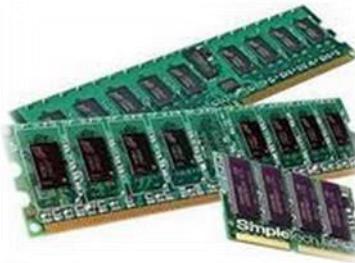
- Location
  - CPU
  - Internal (main)
  - External (secondary)
- Capacity
  - Word size
  - Number of words
- Unit of transfer
  - Word
  - Block
- Access methods
  - Sequential access
  - Direct access
  - Random access
  - Associative access
- Performance
  - Access time
  - Cycle time
  - Transfer rate

# Key Characteristics contd.,

- Physical Type
  - Semiconductor
  - Magnetic surface
  - Optical
- Physical Characteristics
  - Volatile / Non-Volatile
  - Erasable / Non-erasable
- Organization

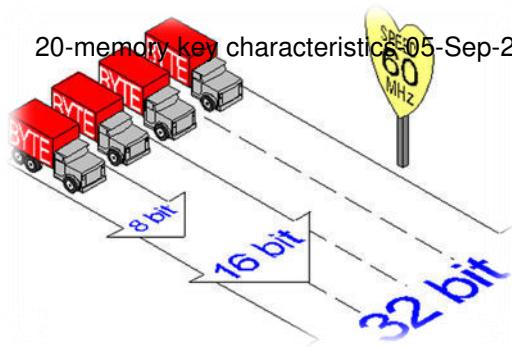
# Location

- Three locations of memories
  - CPU
    - Registers – used by CPU as its local memory
  - Internal memory
    - Main memory
    - Cache memory
  - External memory
    - Peripheral devices – disk, tape – accessible to CPU via I/O controllers



# Capacity

- Internal memory capacity is expressed in terms of bytes or words.
- External memory capacity is expressed in terms of blocks (depends on words in memory)
- Total memory = number of words × word length
- Number of words =  $2^{\text{address bus width}}$
- Word length = Data bus width



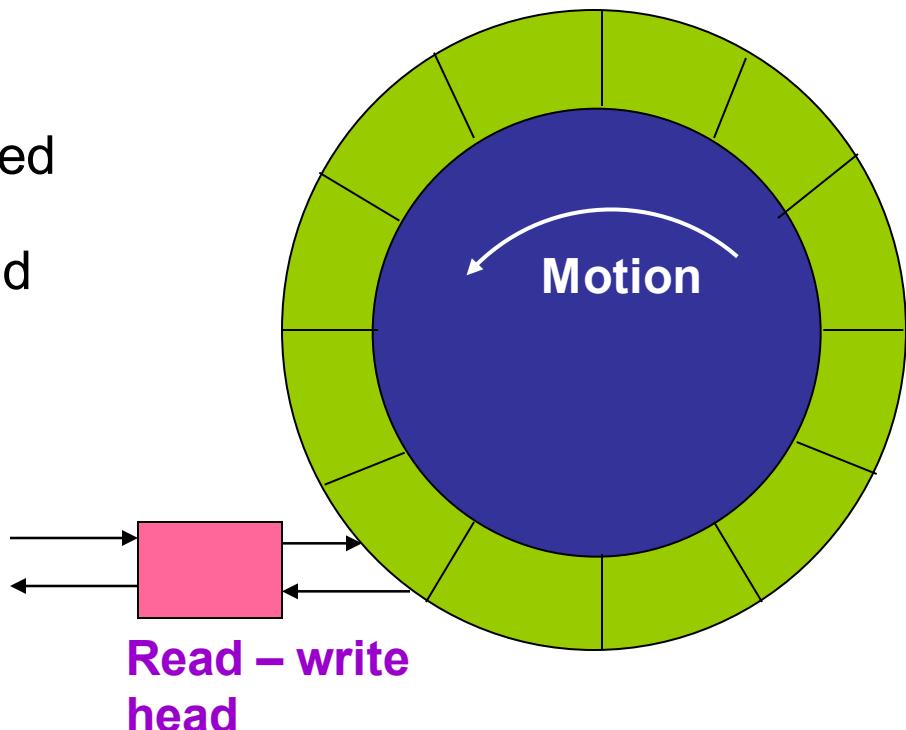
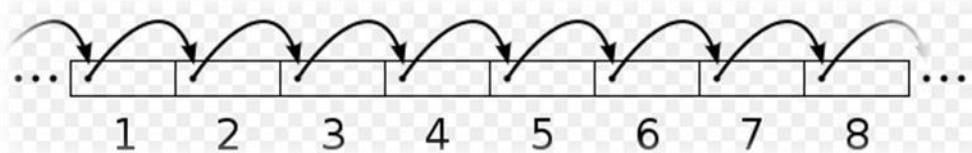
# Unit of transfer

- Internal memory – number of data lines into and out of the main memory module
- External memory – blocks – longer units than a word

# Access Methods

- Four types
  - Sequential Access

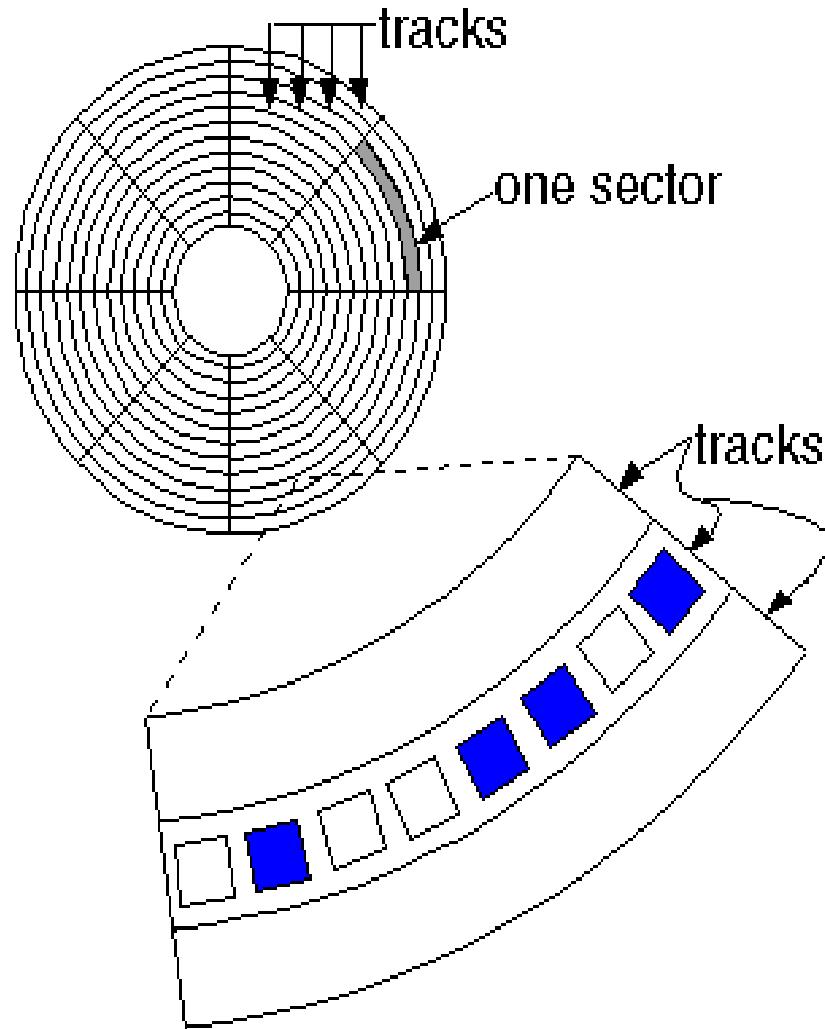
- Accesses the memory in predetermined sequence
- Shared read/write head is used, and this must be moved its current location to the desired location, passing and rejecting each intermediate record.
- So, the time to access an arbitrary record is highly variable
- Slower than random access memory
- Ex: Magnetic Tapes, data in memory array



# Access Methods contd.,

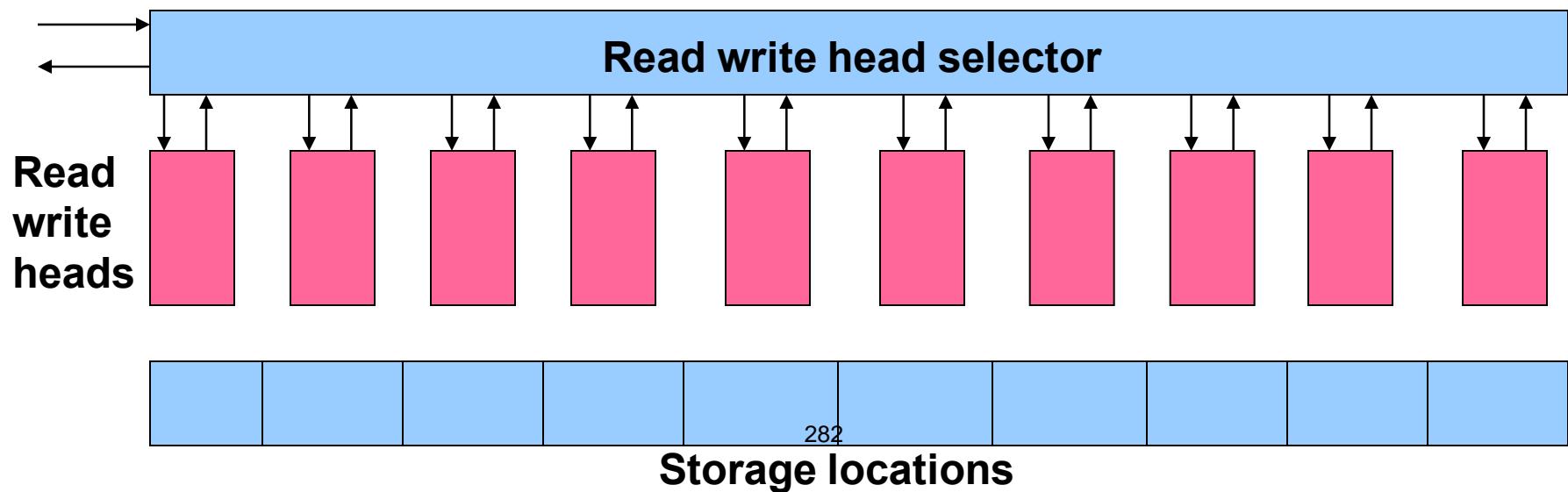
From Computer Desktop Encyclopedia  
© 1998 The Computer Language Co., Inc.

- Direct access
  - Also referred as semi random access memory
  - Access time is variable
  - The track is accessed randomly but access within each track is serial
  - Access is accomplished by general access to reach a general vicinity plus sequential searching, counting, waiting to reach the final location.
  - Ex: Magnetic Disk



# Access methods contd.,

- Random Access
  - Each addressable location in memory has unique, physically wired – in addressing mechanism
  - Time to access a location is independent of the sequences of prior access and is constant
  - Main memory systems are a random access
  - Storage locations can be accessed in any order
  - Semiconductor memories



# Access Methods contd.,



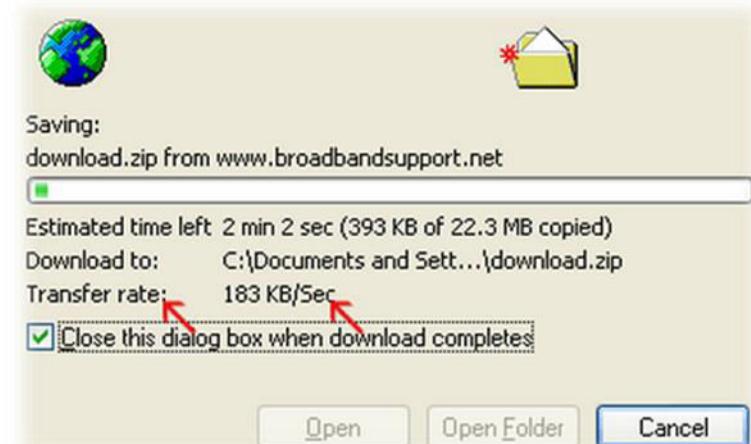
- Associate Access
  - Word is retrieved based on portion of its contents rather than its address
  - This enables one to make a comparison of desired bit locations within a word for specific match
  - Has own addressing mechanism
  - Retrieval time is constant
  - Access time is independent of location or prior access patterns
  - Cache memories

# Performance

- Access time
  - The time required to read / write the data from / into desired record
  - Depends on the amount of data to be read / write
  - If the amount data is uniform for all records then the access time is same for all records.
  - Time from the instant that an address is presented to the memory to the instant that data have been stored or made available for use.
- Memory Cycle time
  - Access time + time required before a second access can commence
  - For Random access method ,this memory cycle time is same for all records
  - The sequential access and direct access ,the memory cycle time is different

# Performance contd.,

- Transfer rate / Throughput
  - Rate at which the data can be transferred into or out of a memory unit
  - Random access memory
    - 1/cycle time
  - Non-Random access memory
    - $T_n = T_a + (N/R)$ , where
      - $T_n$  – average time to read or write N bits
      - $T_a$  – average access time
      - N – Number of bits
      - R – Transfer rate,  
in bits per second (BPS)



# Physical type

## Semiconductor

Semiconductor memory uses semiconductor-based integrated circuits to store information.



## Magnetic surface



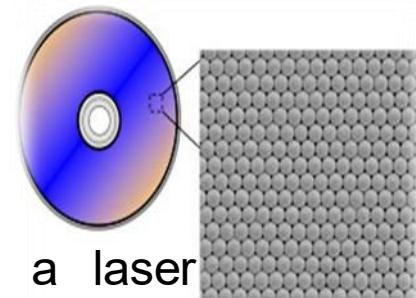
Magnetic storage uses different patterns of magnetization on a magnetically coated surface to store information.

Example:

Magnetic disk, Floppy disk, Hard disk drive

## Optical

The typical optical disc, stores information in deformities on the surface of a circular disc and reads this information by illuminating the surface with a laser diode and observing the reflection.



# Physical characteristics

- Volatile memory
  - Information decays naturally or lost when electrical power is switched off
- Non-volatile memory
  - Once recorded is retained until deliberately changed
  - No electrical power is needed to retain information
  - Magnetic surface memories
- Semiconductor memories may be either volatile or non-volatile
  - A type of non-volatile semiconductor memory known as flash memory
  - A type of volatile semiconductor memory is random access memory
- Non-erasable memory
  - Cannot be altered, except by destroying the storage unit (ROM)
  - A practical non-erasable memory must also be non-volatile
  - Ex: CD-R, Flash Memories
- Erasable memory
  - Erase the stored information by writing new information
  - Ex: Magnetic storage is erasable

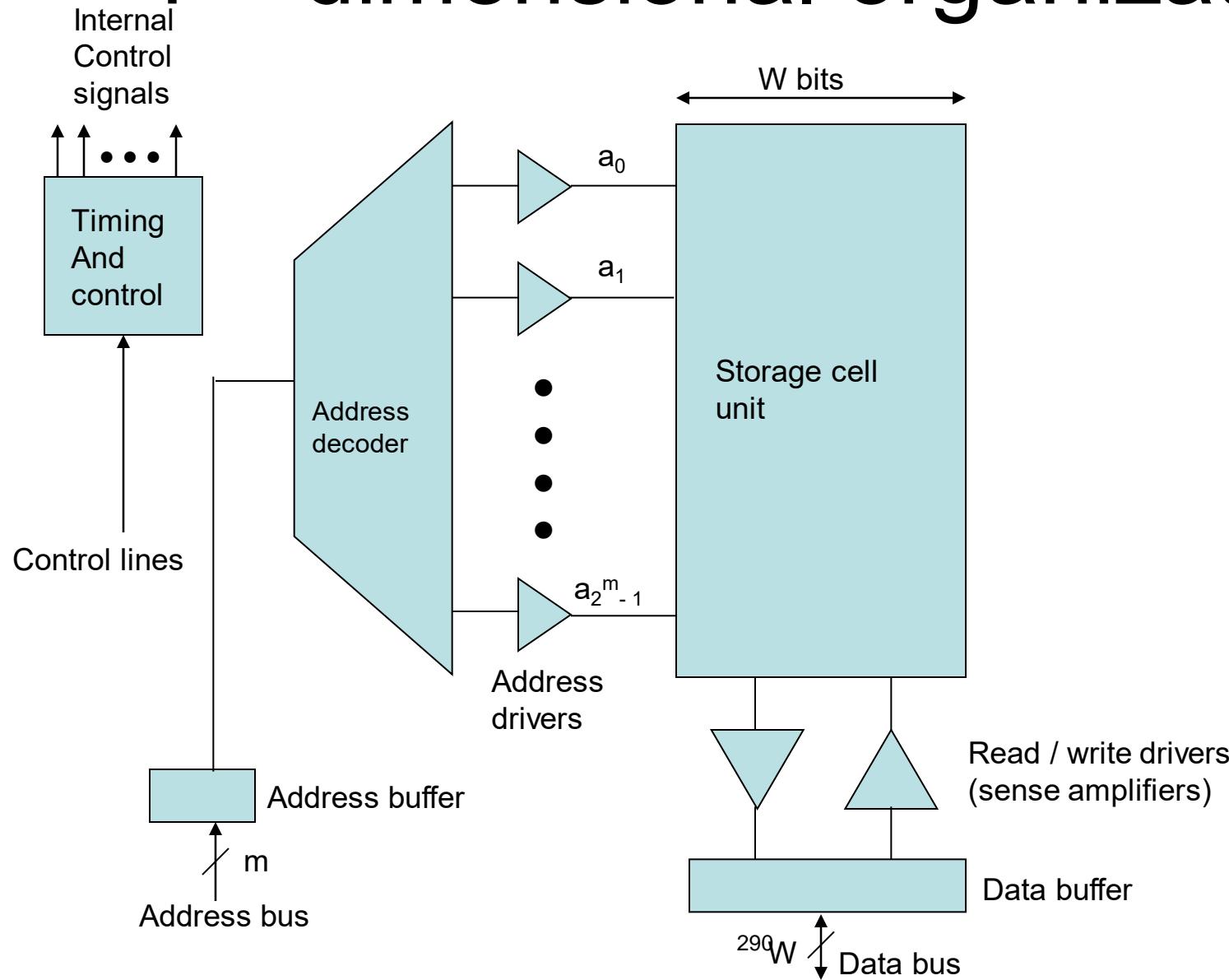
# Organization

- Physical arrangement of bits to form words
- 2 types
  - 1 dimensional
  - 2 dimensional

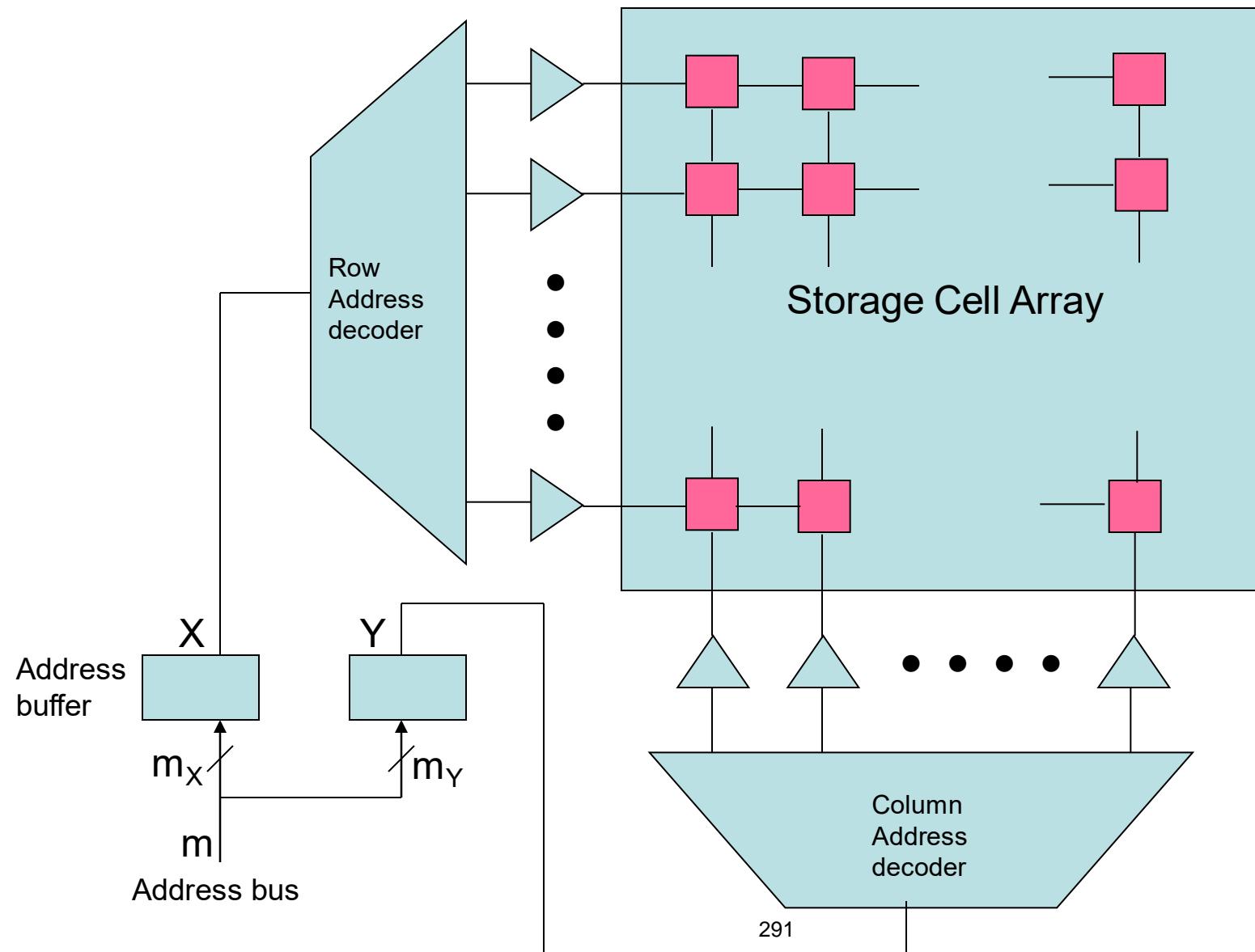
# Memory Organization

- Basic element = memory cell
- Properties of Memory cell:
  - They exhibit two stable states, which can be used to represent binary 1 and 0.
  - They are capable of being written into (atleast once) to set the state.
  - They are capable of being read to sense the state.

# 1 – dimensional organization



# 2 – dimensional organization



# Byte Storage Methods

- Big-Endian
  - Assigns MSB to least address and LSB to highest address
  - Ex: 0 × DEADBEEF

Memory Location	Value
Base Address + 0	DE
Base Address + 1	AD
Base Address + 2	BE
Base Address + 3	EF

# Byte Storage Methods contd.,

- LittleEndian
  - Assigns MSB to highest address and LSB to least address
  - Ex: 0 × DEADBEEF

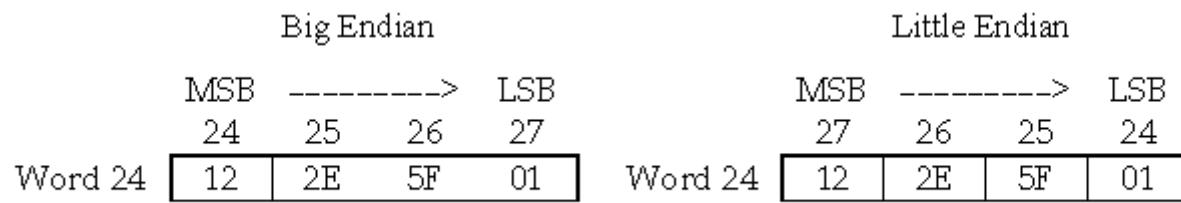
Memory Location	Value
Base Address + 0	EF
Base Address + 1	BE
Base Address + 2	AD
Base Address + 3	DE

# Byte Storage Methods contd.,

- LittleEndian
  - Intel x 86 family
  - Digital equipment corporation architectures (PDP – 11, VAX, Alpha)
- BigEndian
  - Sun SPARC
  - IBM 360 / 370
  - Motorola 68000
  - Motorola 88000
- Bi-Endian
  - Power PC
  - MIPS
  - Intel's 64 IA - 64

# Example

- **Example:** Show the contents of memory at word address 24 if that word holds the number given by 122E 5F01H in both the big-endian and the little-endian schemes?



# References

- William Stallings “Computer Organization and architecture” Prentice Hall, 7th edition, 2006

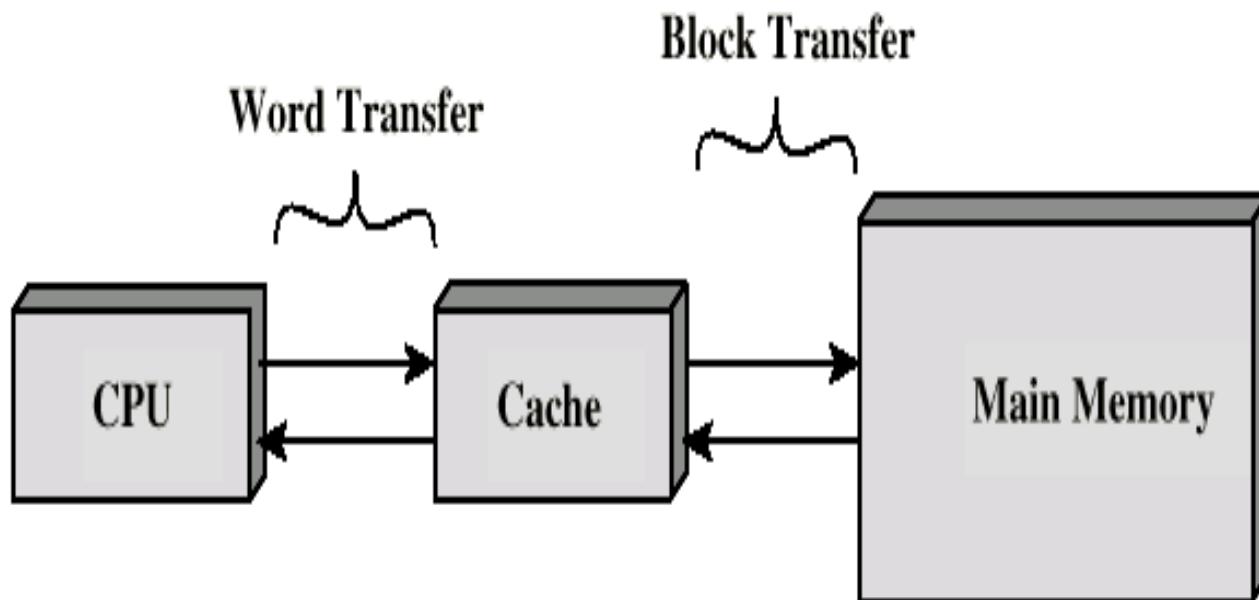
# CACHE MEMORY

**JASMIN T JOSE  
ASST. PROFESSOR, SCSE  
VIT UNIVERSITY, VELLORE**

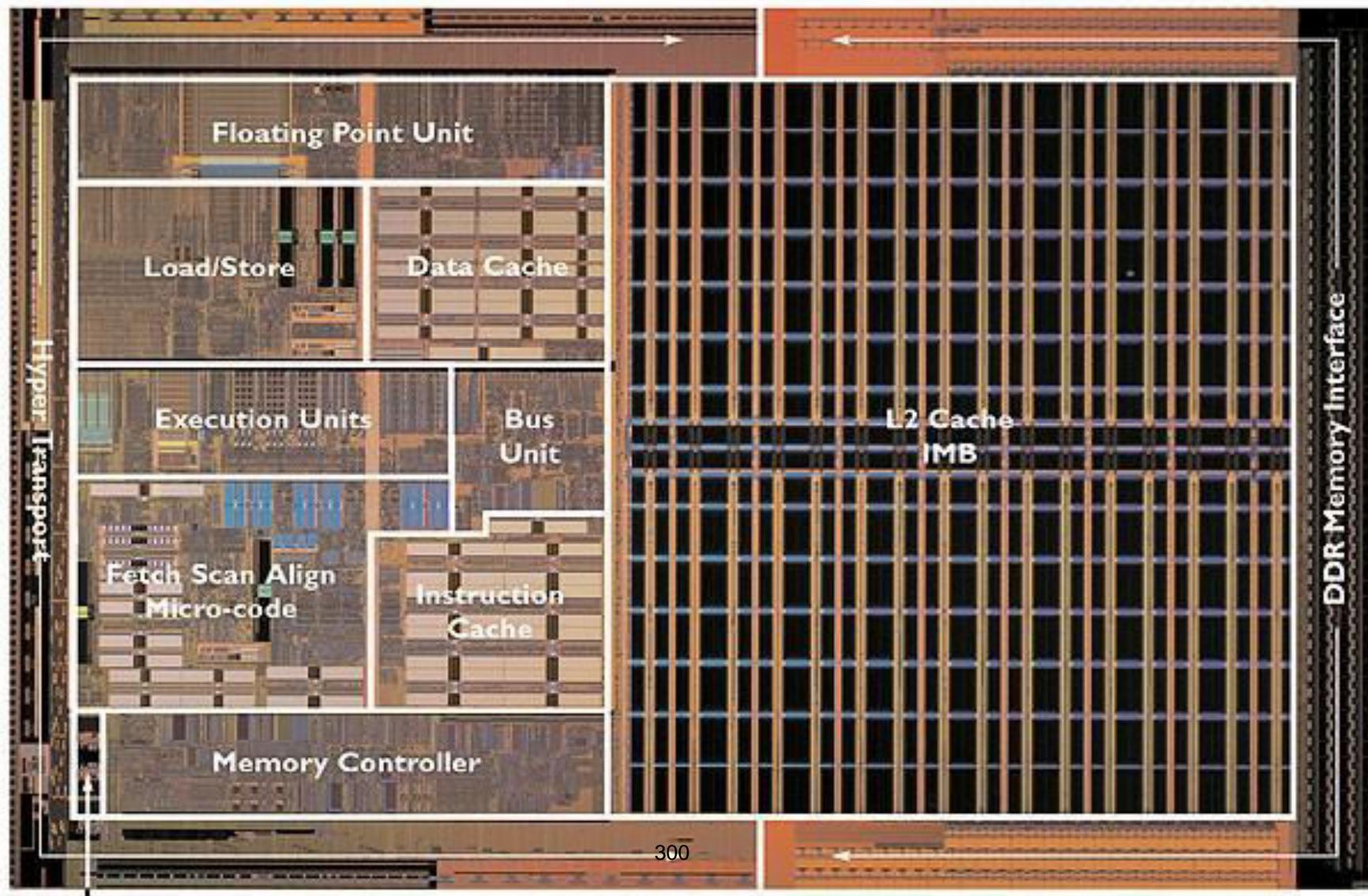
# Cache Memory

A small amount of high speed memory between the main memory and the processor.

May be located on processor chip or separate module

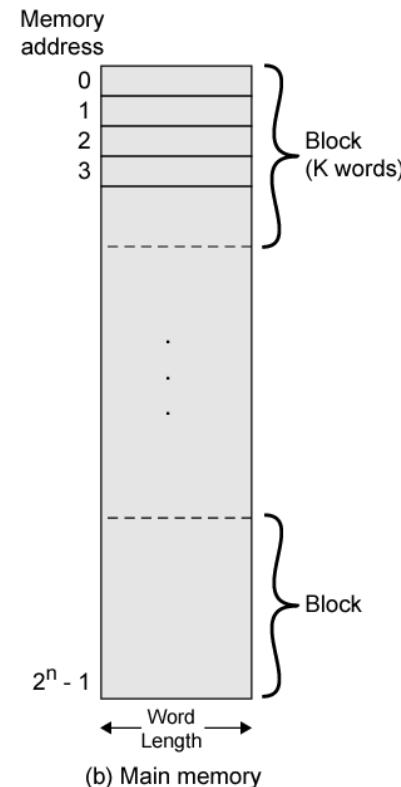
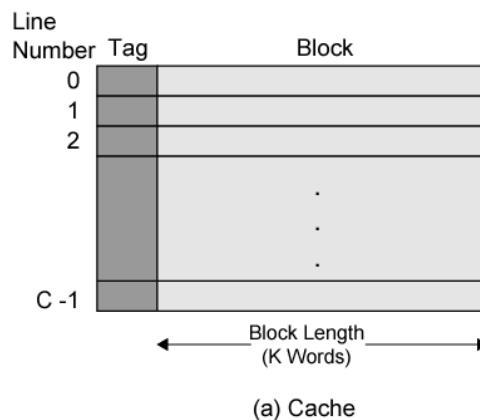


# Cache Memory



# Cache memory Vs Main memory structure

- Cache is partitioned into **lines** (also called **blocks**).
- Each line has 4-64 bytes in it. During data transfer, a whole line is read or written.
- Cache includes **tags** to identify which block of main memory is in each cache line.



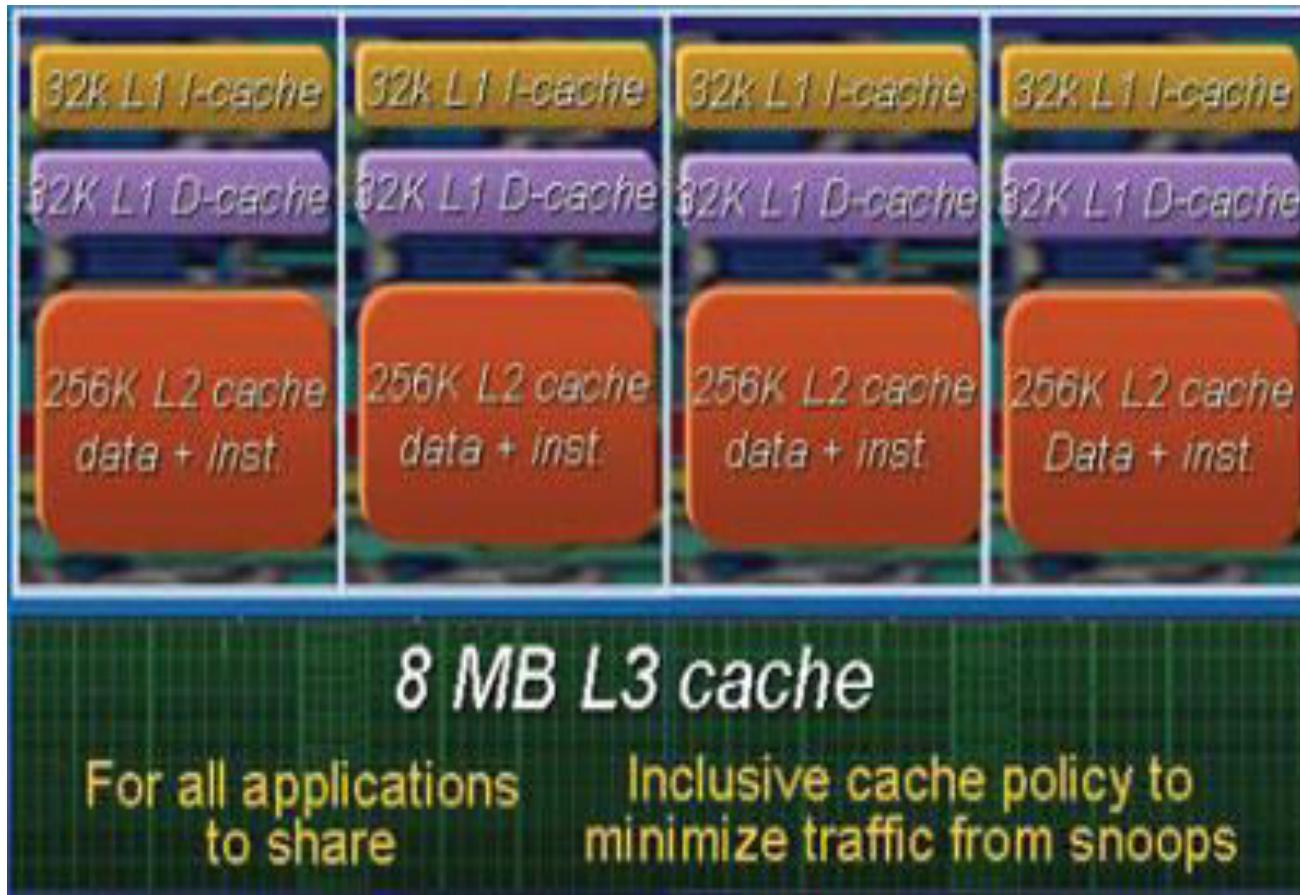
# Cache organization

- Split cache
  - Separate caches for instructions and data
  - I-cache (Instruction) – mostly accessed sequentially
  - D-cache (data) – mostly random access
- Unified cache
  - Same cache for instruction and data
- Higher hit rate for unified cache as it balances between instruction and data
- Split caches eliminate contention for cache between the instruction processor and the execution unit – used for pipelining processes

# Multilevel caches

- The penalty for a cache miss is the extra time that it takes to obtain the requested item from central memory.
- One way in which this penalty can be reduced is to provide another cache, the secondary cache, which is accessed in response to a miss in the primary cache.
- The primary cache is referred to as the L1 (level 1) cache and the secondary cache is called the L2 (level 2) cache.
- Most high-performance microprocessors include an L2 and L3 cache which is often located off-chip, whereas the L1 cache is located on the same chip as the CPU.
- With a two-level cache, central memory has to be accessed only if a miss occurs in both caches.

# Nehalem's micro-architecture showing different cache memory levels



# Small or slow

- Unfortunately there is a tradeoff between speed, cost and capacity.

Storage	Speed	Cost	Capacity
Static RAM	Fastest	Expensive	Smallest
Dynamic RAM	Slow	Cheap	Large
Hard disks	Slowest	Cheapest	Largest

- Fast memory is too expensive for most people to buy a lot of.
- Here are rough estimates of some current storage parameters.

Storage	Delay	Cost/MB	Capacity
Static RAM	1-10 cycles	~\$5	128KB-2MB
Dynamic RAM	100-200 cycles	~\$0.10	128MB-4GB
Hard disks	10,000,000 cycles <sub>305</sub>	~\$0.0005	20GB-400GB

# The principle of locality

- It's usually difficult or impossible to figure out what data will be “most frequently accessed” before a program actually runs, which makes it hard to know what to store into the small, precious cache memory.
- But in practice, most programs exhibit *locality*, which the cache can take advantage of.
  - The principle of **temporal locality** says that if a program accesses one memory address, there is a good chance that it will access the same address again.
  - The principle of **spatial locality** says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses.

# Temporal locality in programs

- The principle of **temporal locality** says that if a program accesses one memory address, there is a good chance that it will access the same address again.
- Loops are excellent examples of temporal locality in programs.
  - The loop body will be executed many times.
  - The computer will need to access those same few locations of the instruction memory repeatedly.
- For example:

```
Loop:    lw      $t0, 0($s1)
          add    $t0, $t0, $s2
          sw      $t0, 0($s1)
          addi   $s1, $s1, -4
          bne   $s1, $0, Loop
```

- Each instruction will be fetched over and over again, once on every loop iteration.

# Spatial locality in programs

- The principle of **spatial locality** says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses.

```
sub    $sp, $sp, 16
sw     $ra, 0($sp)
sw     $s0, 4($sp)
sw     $a0, 8($sp)
sw     $a1, 12($sp)
```

- Nearly every program exhibits spatial locality, because instructions are usually executed in sequence—if we execute an instruction at memory location  $i$ , then we will probably also execute the next instruction, at memory location  $i+1$ .
- Code fragments such as loops exhibit *both* temporal and spatial locality.

# Parameters of cache memory

## i) Cache hit

Data found in cache.

Results in data transfer at maximum speed.

## ii) Cache miss

Data not found in cache. Processor loads data from memory and copies into cache (miss penalty).

## iii) Hit ratio

Ratio of number of hits to total number of references =>  
number of hits / (number of hits + number of Miss)

## iv) Miss penalty

Additional cycles required to serve the miss

Time required for the cache miss depends on both the latency and bandwidth

## v) Latency – time to retrieve the first word of the block

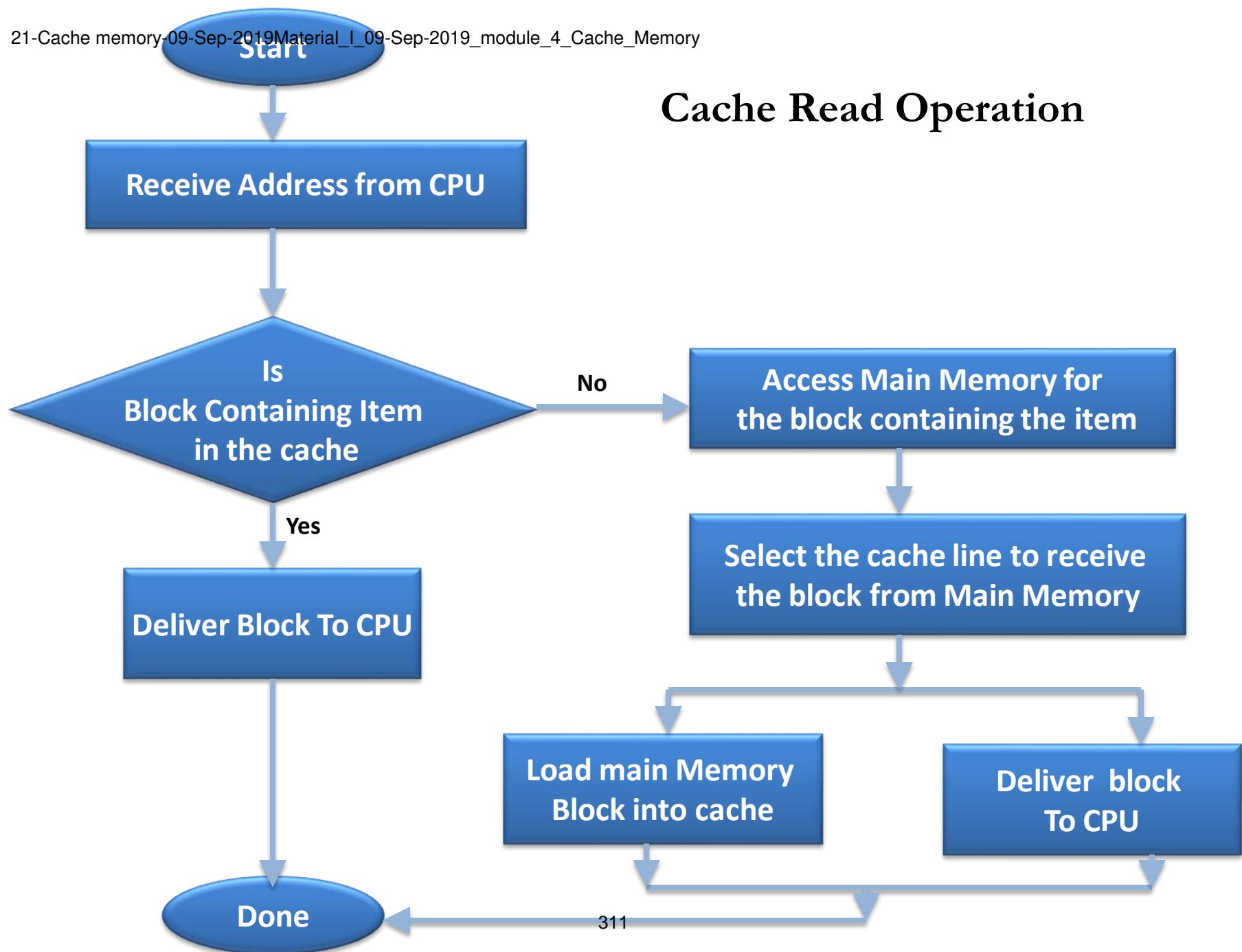
## vi) Bandwidth – time to retrieve the rest of this block

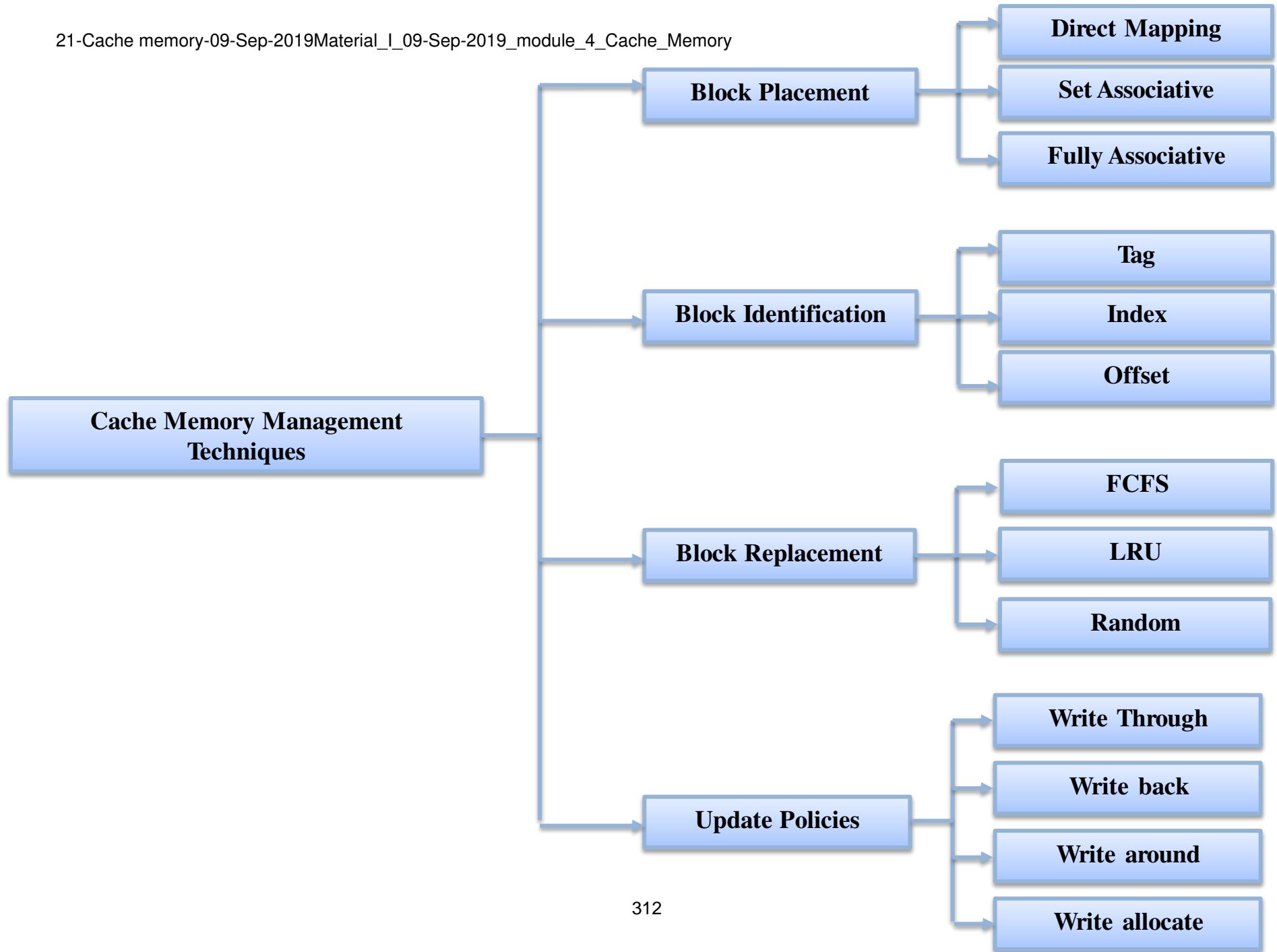
# Sources of Cache Misses (Three C's)

**Compulsory Misses:** These are misses that are caused by the cache being empty initially or by the first reference to a location in memory . Sometimes referred to as **Cold misses**.

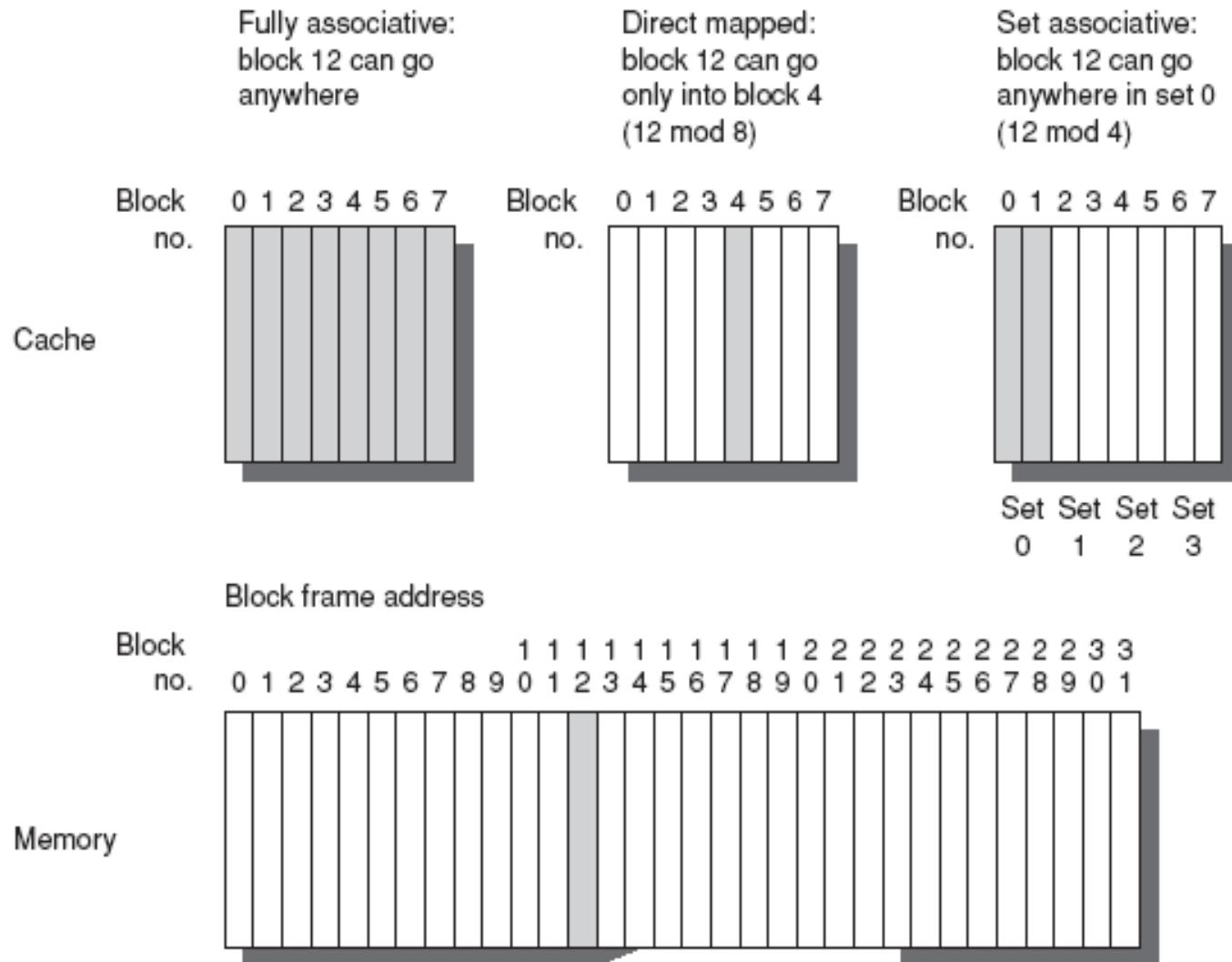
**Capacity Misses :** If the cache cannot contain all the blocks needed during the execution of a program, capacity misses will occur due to blocks being discarded and later retrieved.

**Conflict Misses:** If the cache mapping is such that multiple blocks are mapped to the same cache entry. Common in set associative or direct mapped block placement, where a block can be discarded and later retrieved if too many blocks map to its set. Also called collision or interference misses.



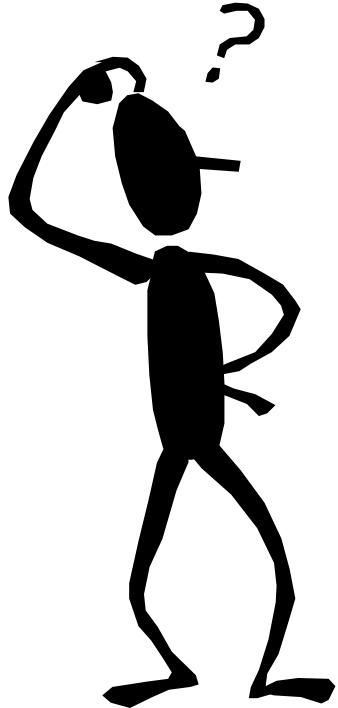


# Example



**Figure C.2** This example cache has eight block frames and memory has 32 blocks.

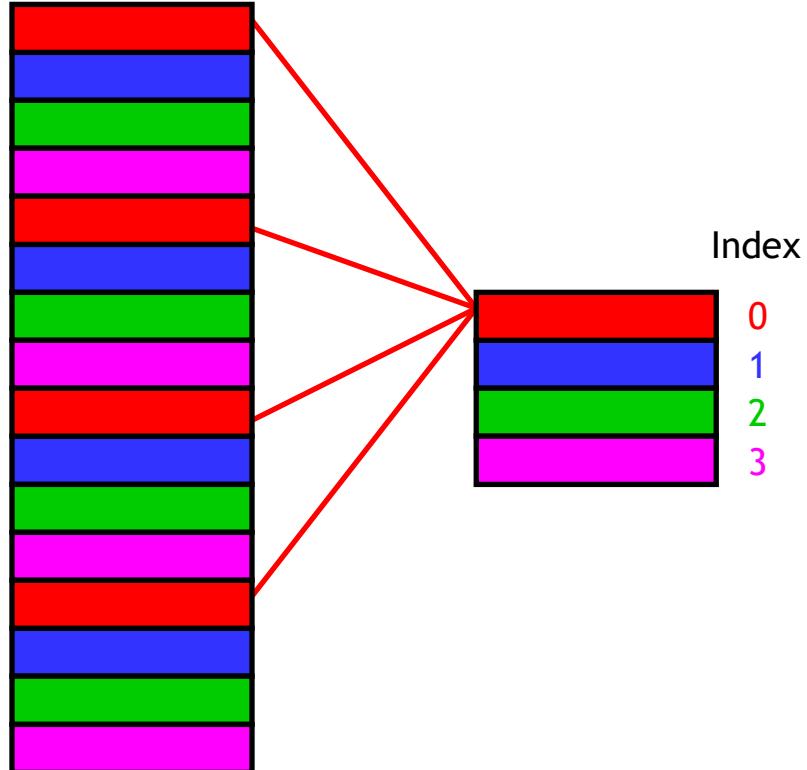
# Four important questions



1. When we copy a block of data from main memory to the cache, where exactly should we put it?
2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?
3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we'd have to replace one of the existing blocks in the cache... which one?
4. How can *write* operations be handled by the memory system?

- Questions 1 and 2 are related—we have to know where the data is placed if we ever hope to find it again later!

# Where should we put data in the cache?

- A **direct-mapped** cache is the simplest approach: each main memory address maps to exactly one cache block.
- For example, on the right is a 16-byte main memory and a 4-byte cache (four 1-byte blocks). 

The diagram illustrates a direct-mapped cache system. On the left, a vertical stack of 16 colored bars represents main memory locations indexed from 0 to 15. The colors follow a repeating pattern: Red (0, 4, 8, 12), Blue (1, 5, 9, 13), Green (2, 6, 10, 14), and Magenta (3, 7, 11, 15). On the right, a smaller vertical stack of four colored bars represents the cache, indexed from 0 to 3. The colors are Red (0), Blue (1), Green (2), and Magenta (3). Red lines connect memory location 0 to cache index 0, memory location 4 to cache index 0, memory location 8 to cache index 0, and memory location 12 to cache index 0. Blue lines connect memory location 1 to cache index 1, memory location 5 to cache index 1, memory location 9 to cache index 1, and memory location 13 to cache index 1. Green lines connect memory location 2 to cache index 2, memory location 6 to cache index 2, memory location 10 to cache index 2, and memory location 14 to cache index 2. Magenta lines connect memory location 3 to cache index 3, memory location 7 to cache index 3, memory location 11 to cache index 3, and memory location 15 to cache index 3.
- Memory locations 0, 4, 8 and 12 all map to cache block 0.
- Addresses 1, 5, 9 and 13 map to cache block 1, etc.
- How can we compute this mapping?

# It's all divisions...

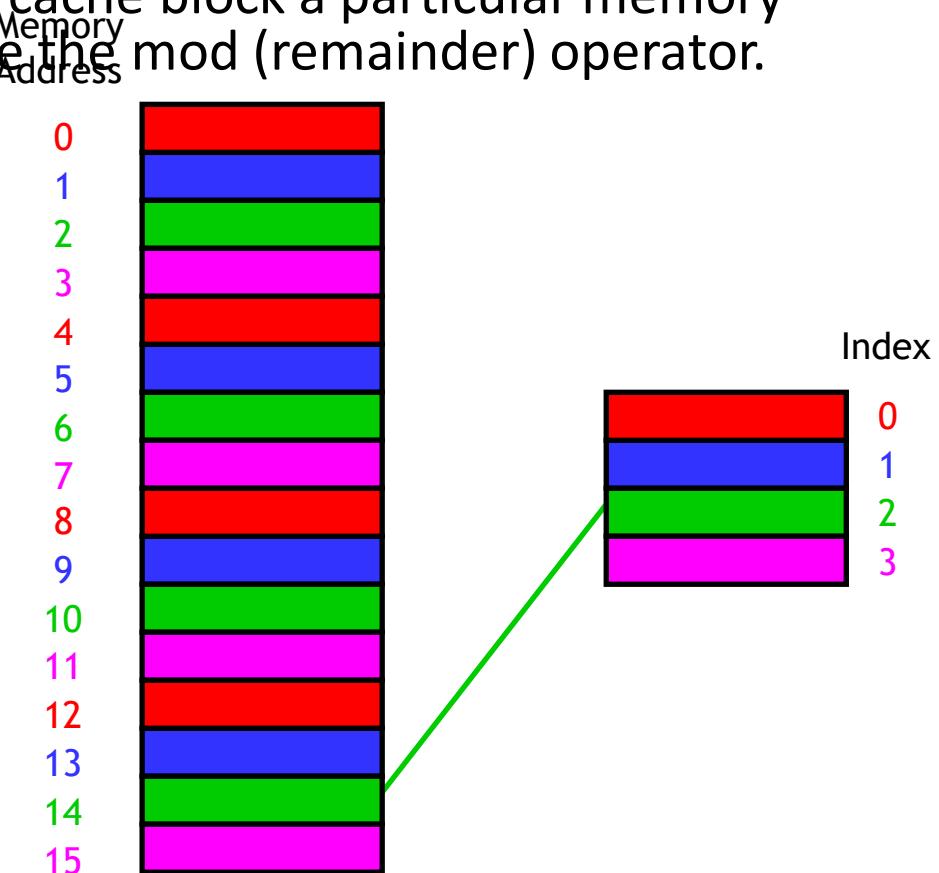
- One way to figure out which cache block a particular memory address should go to is to use the mod (remainder) operator.

- If the cache contains  $2^k$  blocks, then the data at memory address  $i$  would go to cache block index

$$i \bmod 2^k$$

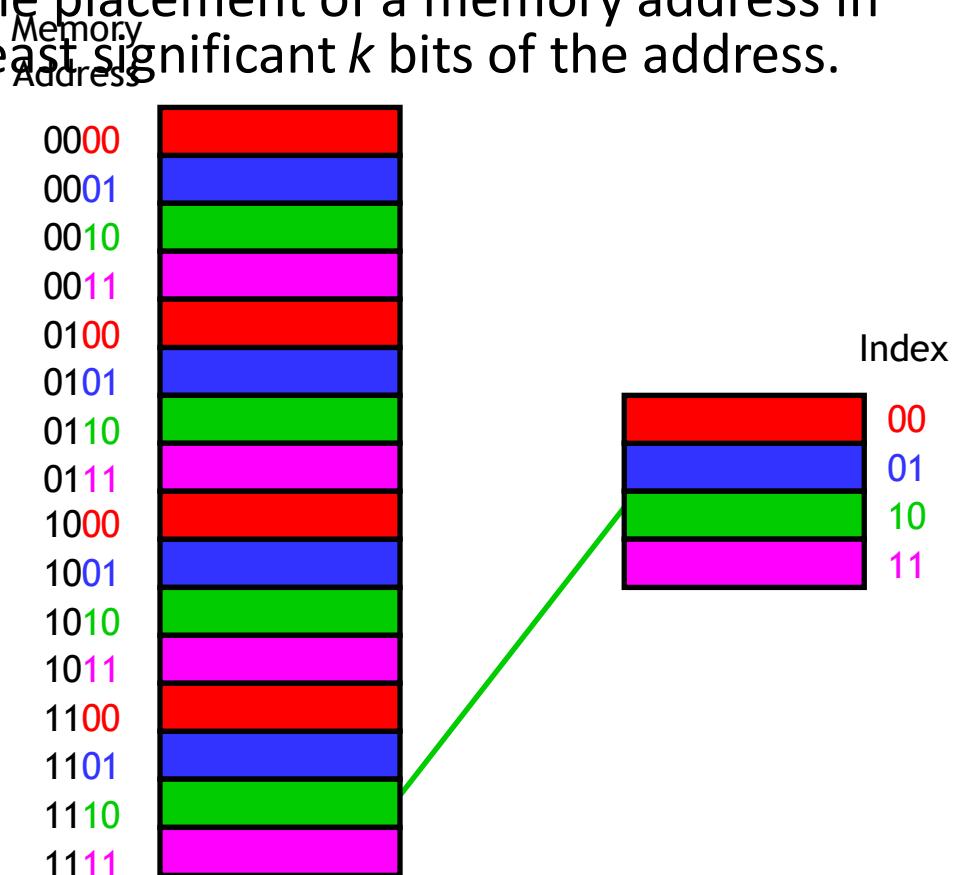
- For instance, with the four-block cache here, address 14 would map to cache block 2.

$$14 \bmod 4 = 2$$



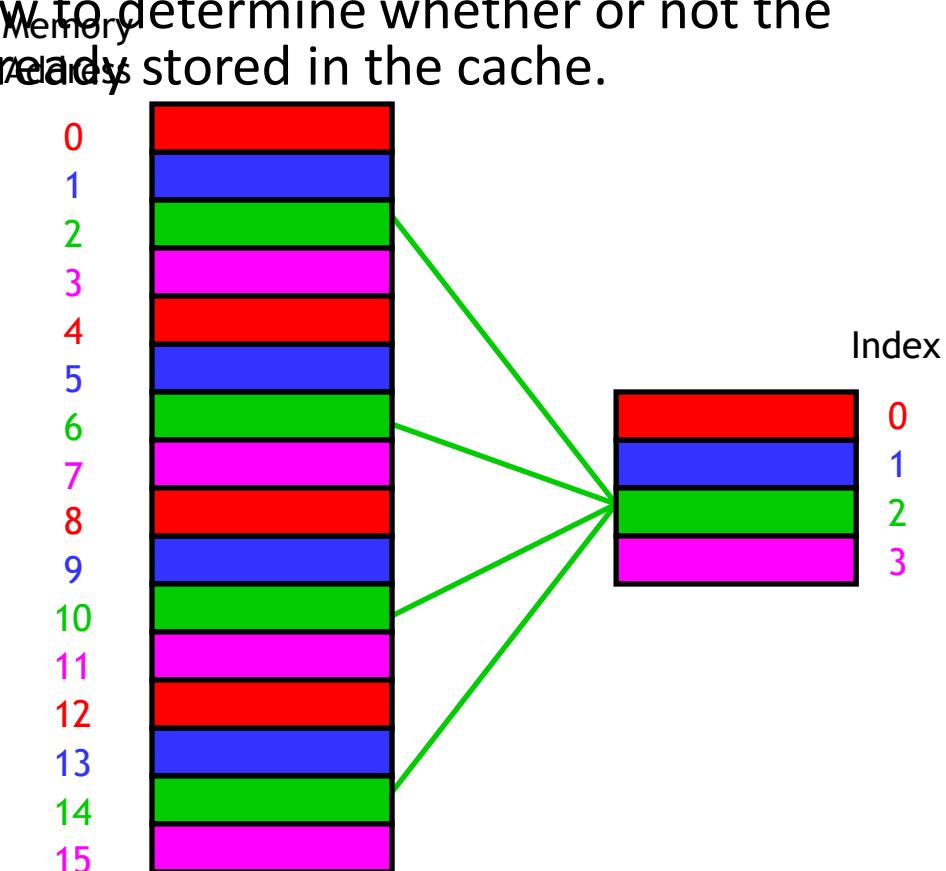
# ...or least-significant bits

- An equivalent way to find the placement of a memory address in the cache is to look at the least significant  $k$  bits of the address.
- With our four-byte cache we would inspect the two least significant bits of our memory addresses.
- Again, you can see that address 14 (1110 in binary) maps to cache block 2 (10 in binary).
- Taking the least  $k$  bits of a binary value is the same as computing that value mod  $2^k$ .



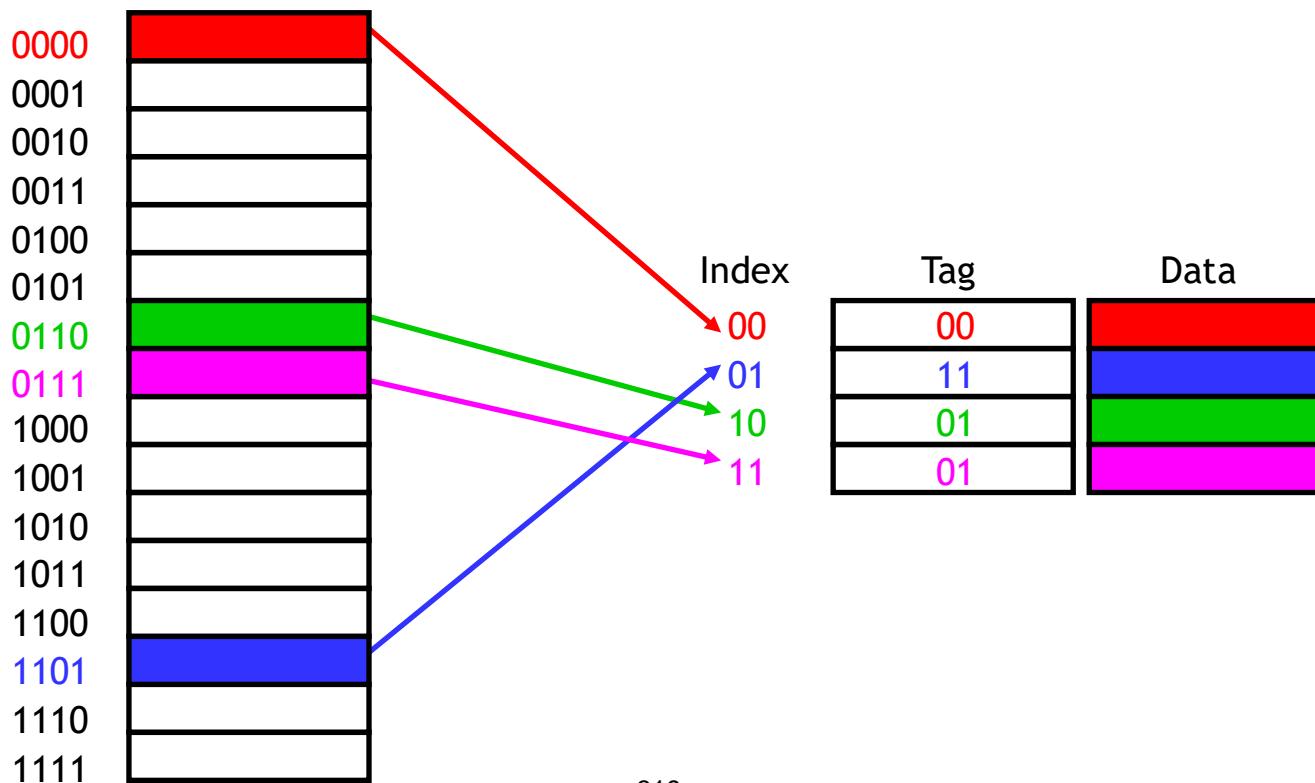
# How can we find data in the cache?

- The second question was how to determine whether or not the data we're interested in is already stored in the cache.
- If we want to read memory address  $i$ , we can use the mod trick to determine which cache block would contain  $i$ .
- But other addresses might *also* map to the same cache block. How can we distinguish between them?
- For instance, cache block 2 could contain data from addresses 2, 6, 10 or 14.



# Adding tags

- We need to add **tags** to the cache, which supply the rest of the address bits to let us distinguish between **different memory locations that map to the same cache block**.



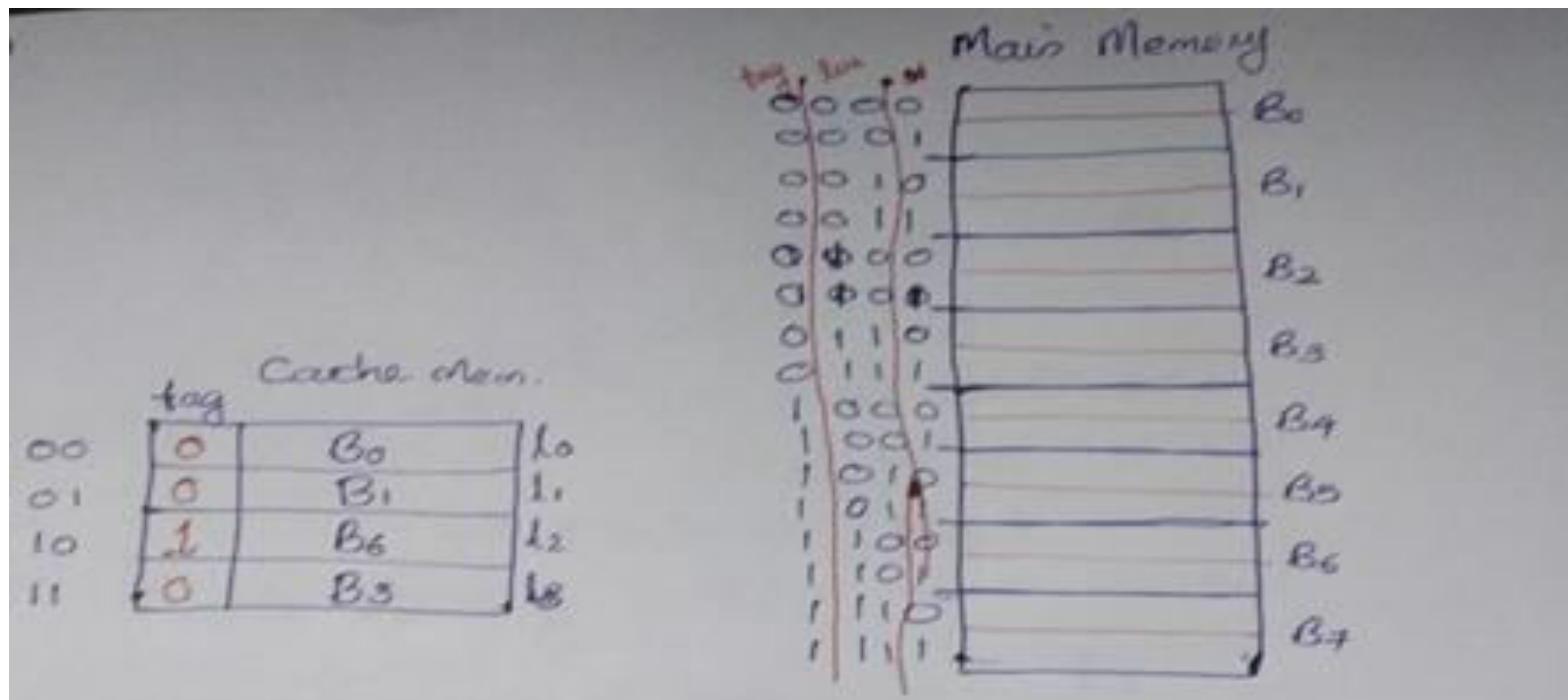
Assume,

Main Memory with,

- 8 blocks,  $B_0 \rightarrow B_7$
- 16 words.
- $2^4 \rightarrow$  4 bit address.  
→ 0000 to 1111
- 2 words / block.

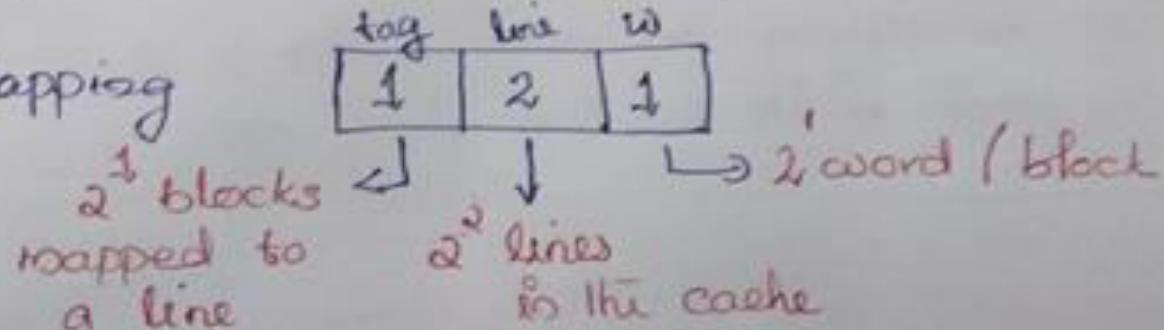
Cache Memory with,

- 4 lines,  $l_0 \rightarrow l_3$
- 1 block / line.
- 2 block can be mapped to a line, but 1 at a time.

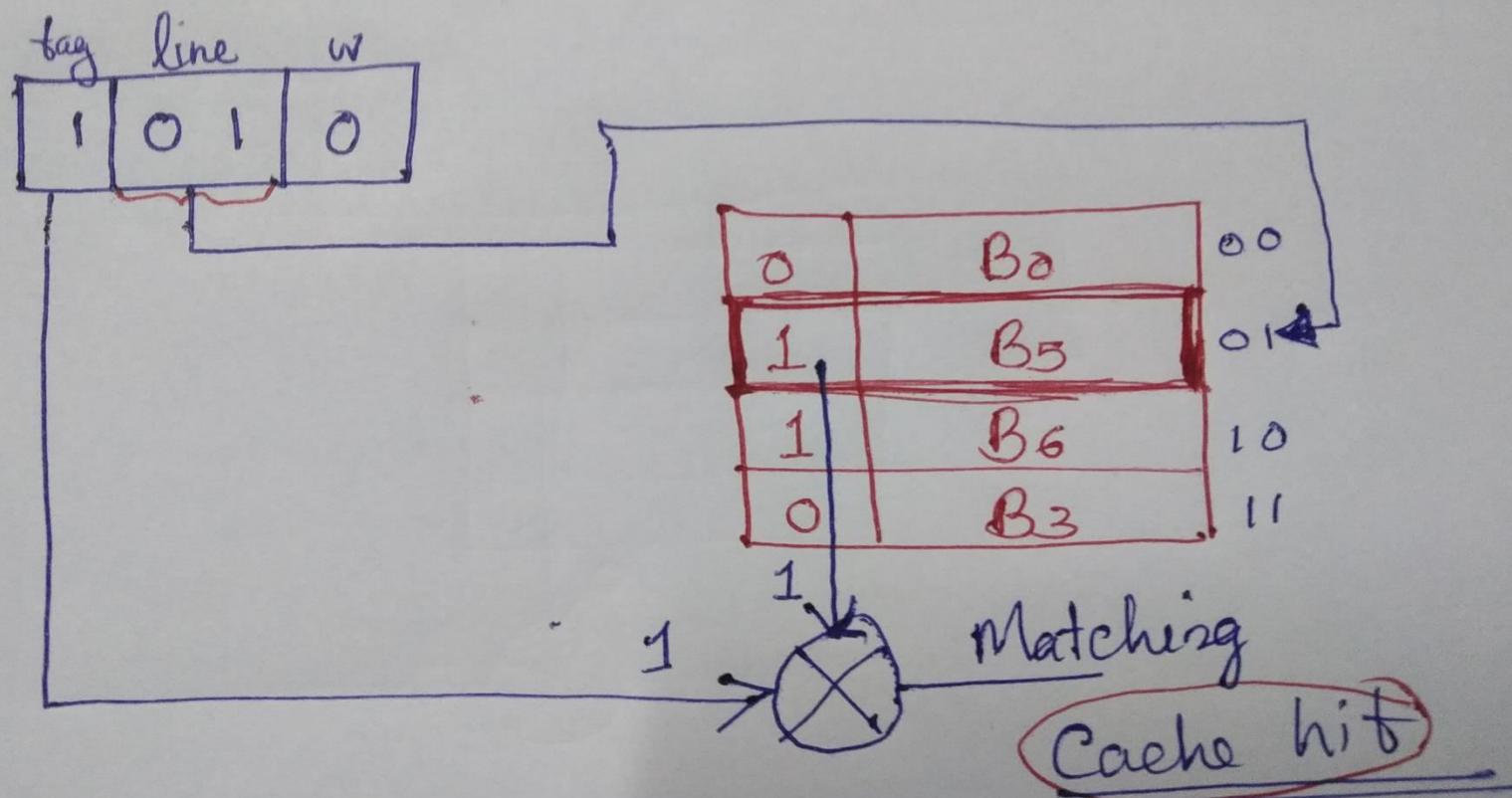


Address from CPU is 1010

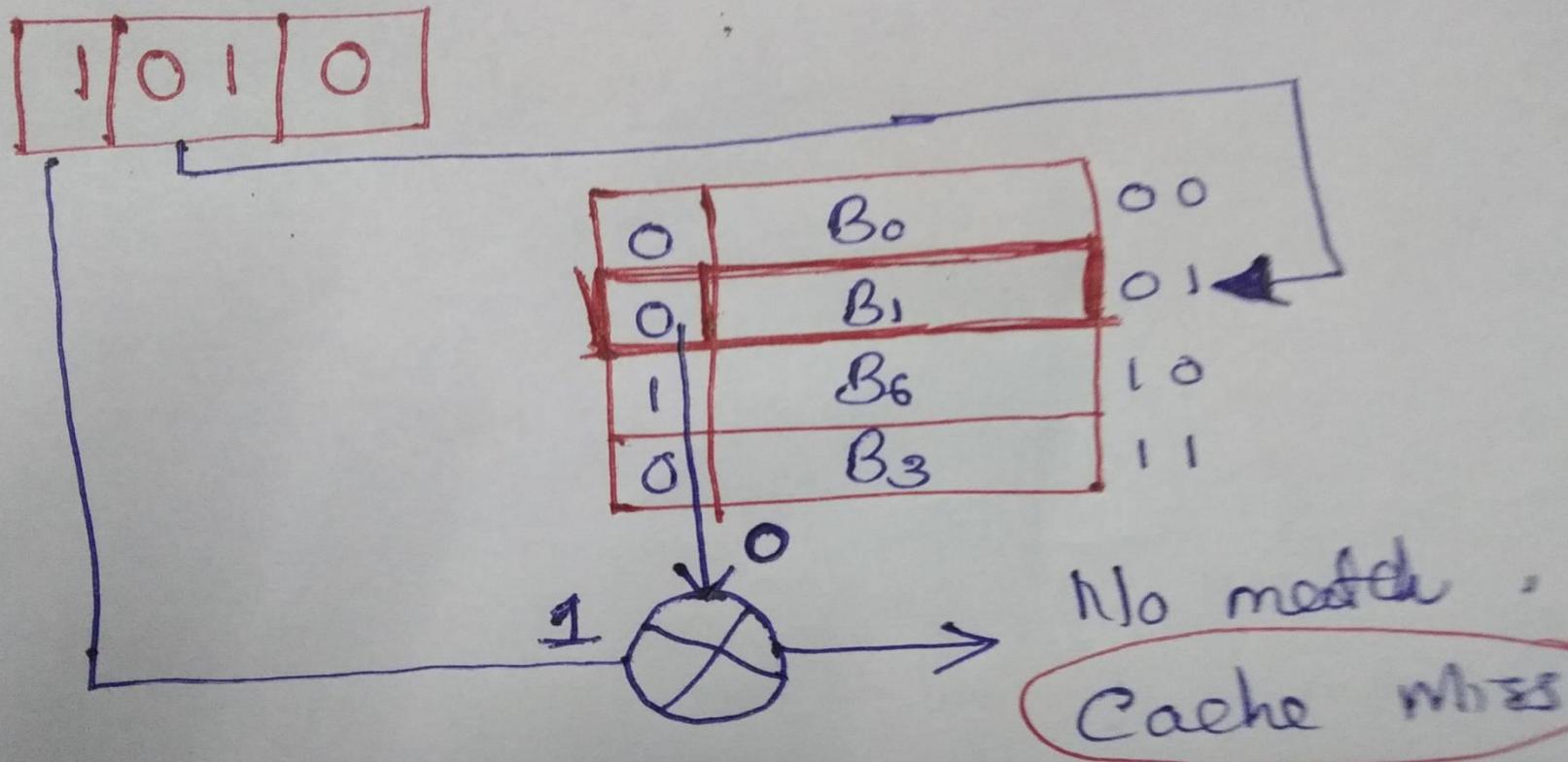
Direct Mapping



# Cache hit

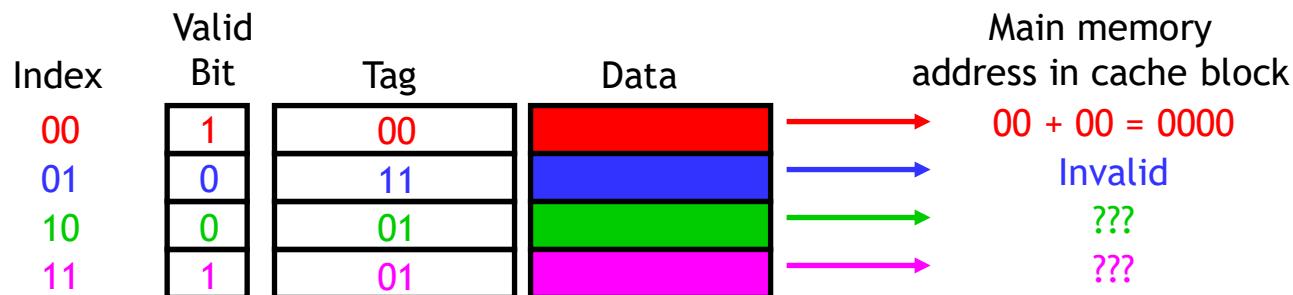


# Cache miss



# One more detail: the valid bit

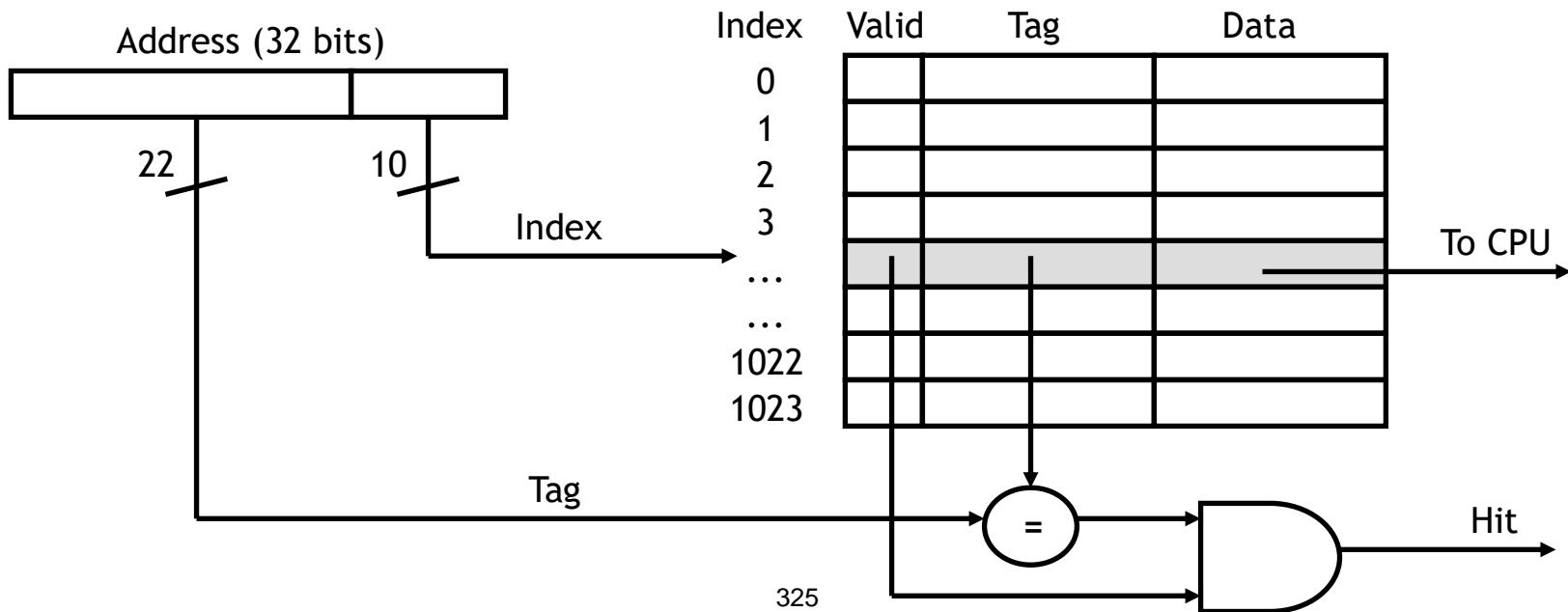
- When started, the cache is empty and does not contain valid data.
- We should account for this by adding a **valid bit** for each cache block.
  - When the system is initialized, all the valid bits are set to 0.
  - When data is loaded into a particular cache block, the corresponding valid bit is set to 1.



- So the cache contains more than just copies of the data in memory; it also has bits to help us find data within the cache and verify its validity.

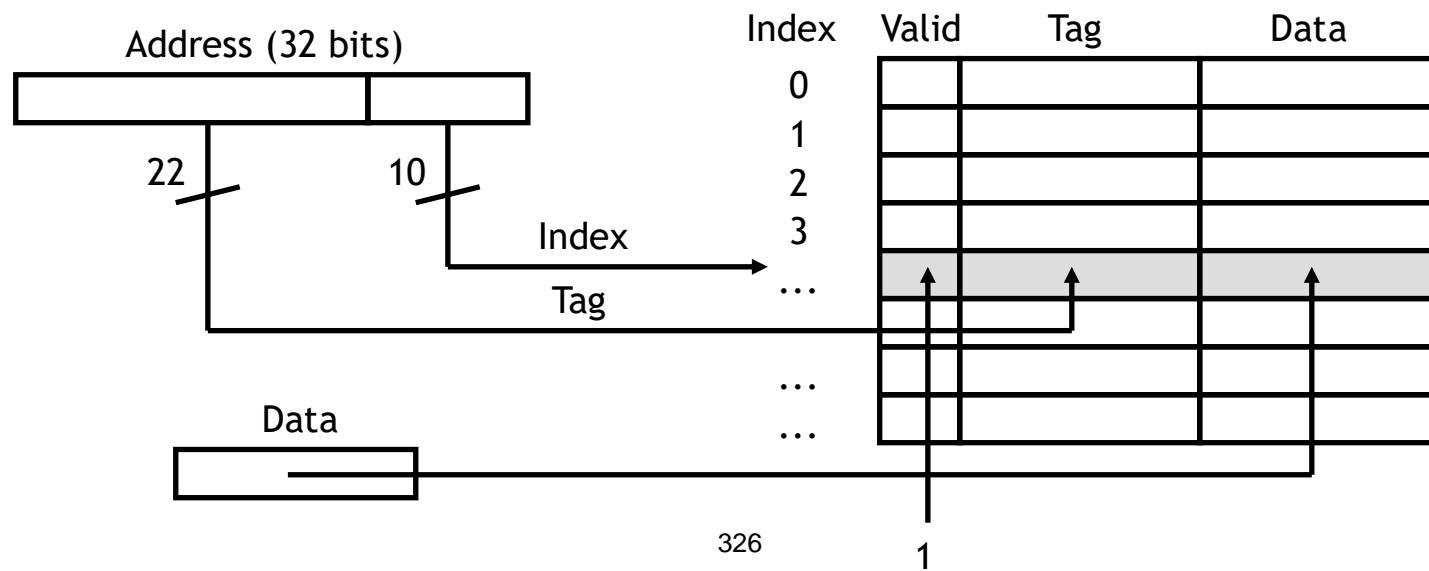
# What happens on a cache hit

- When the CPU tries to read from memory, the address will be sent to a **cache controller**.
  - The lowest  $k$  bits of the address will index a block in the cache.
  - If the block is valid and the tag matches the upper  $(m - k)$  bits of the  $m$ -bit address, then that data will be sent to the CPU.
- Here is a diagram of a 32-bit memory address and a  $2^{10}$ -byte cache.



# Loading a block into the cache

- After data is read from main memory, putting a copy of that data into the cache is straightforward.
  - The lowest  $k$  bits of the address specify a cache block.
  - The upper  $(m - k)$  address bits are stored in the block's tag field.
  - The data from main memory is stored in the block's data field.
  - The valid bit is set to 1.



# Mapping Function

Example 1

Assumption:

No. of lines(blocks) in a cache: 128

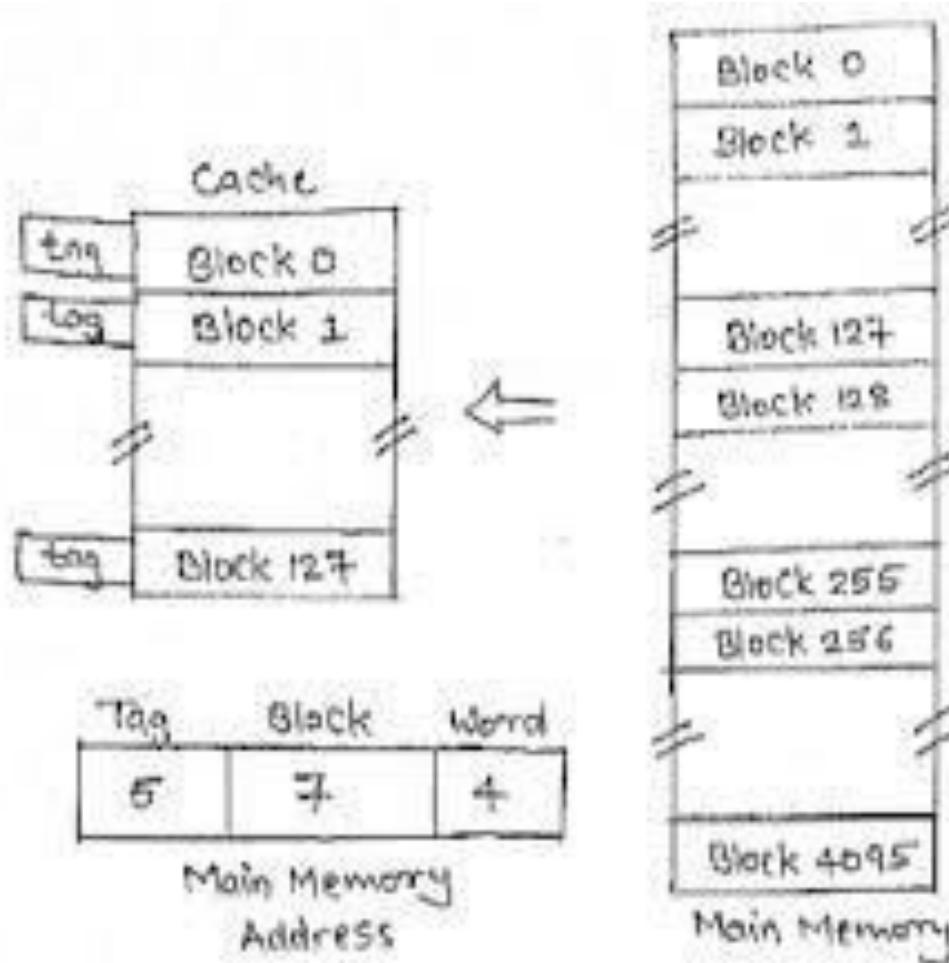
No. of blocks in main memory: 4096 blocks

No. of words in main memory: 64k words

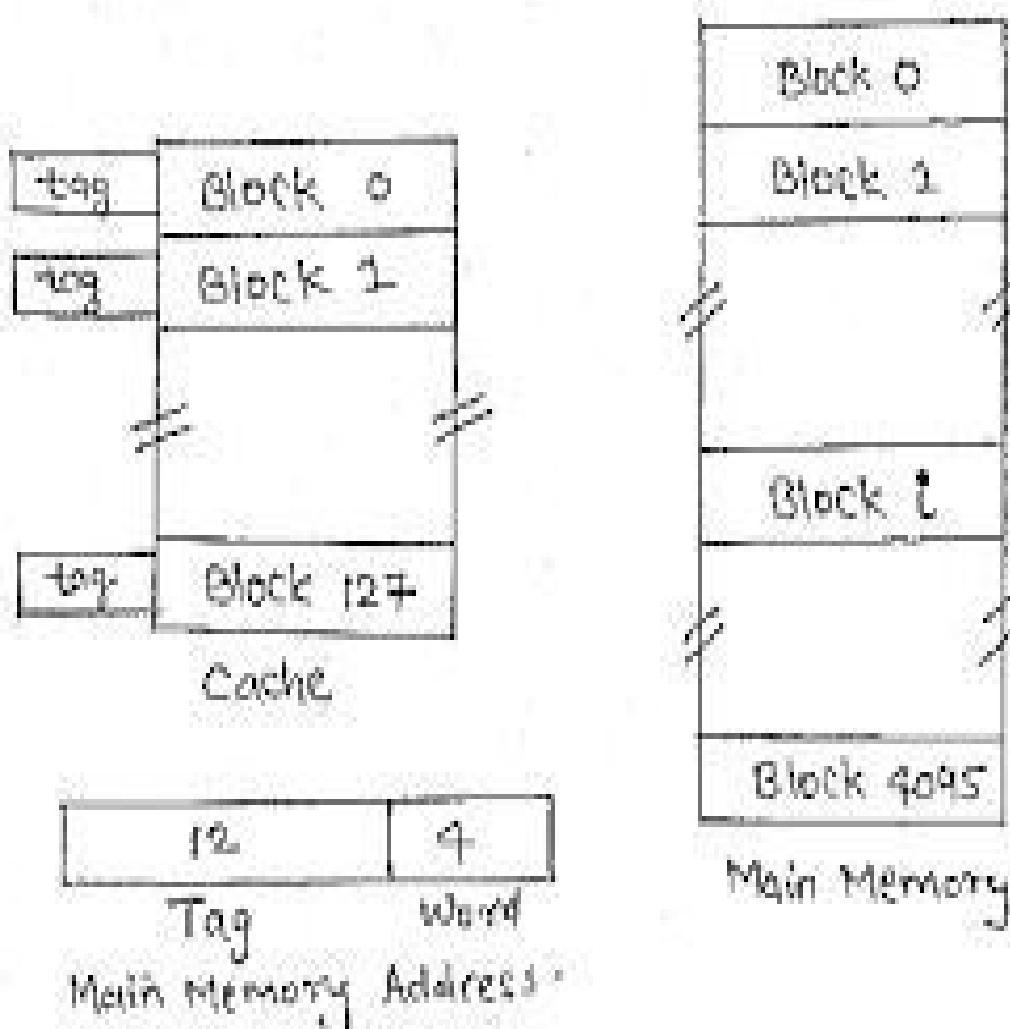
No. of Words in a block: 16

No. address bits: 16 ( $64k = 2^{10} * 2^6$ )

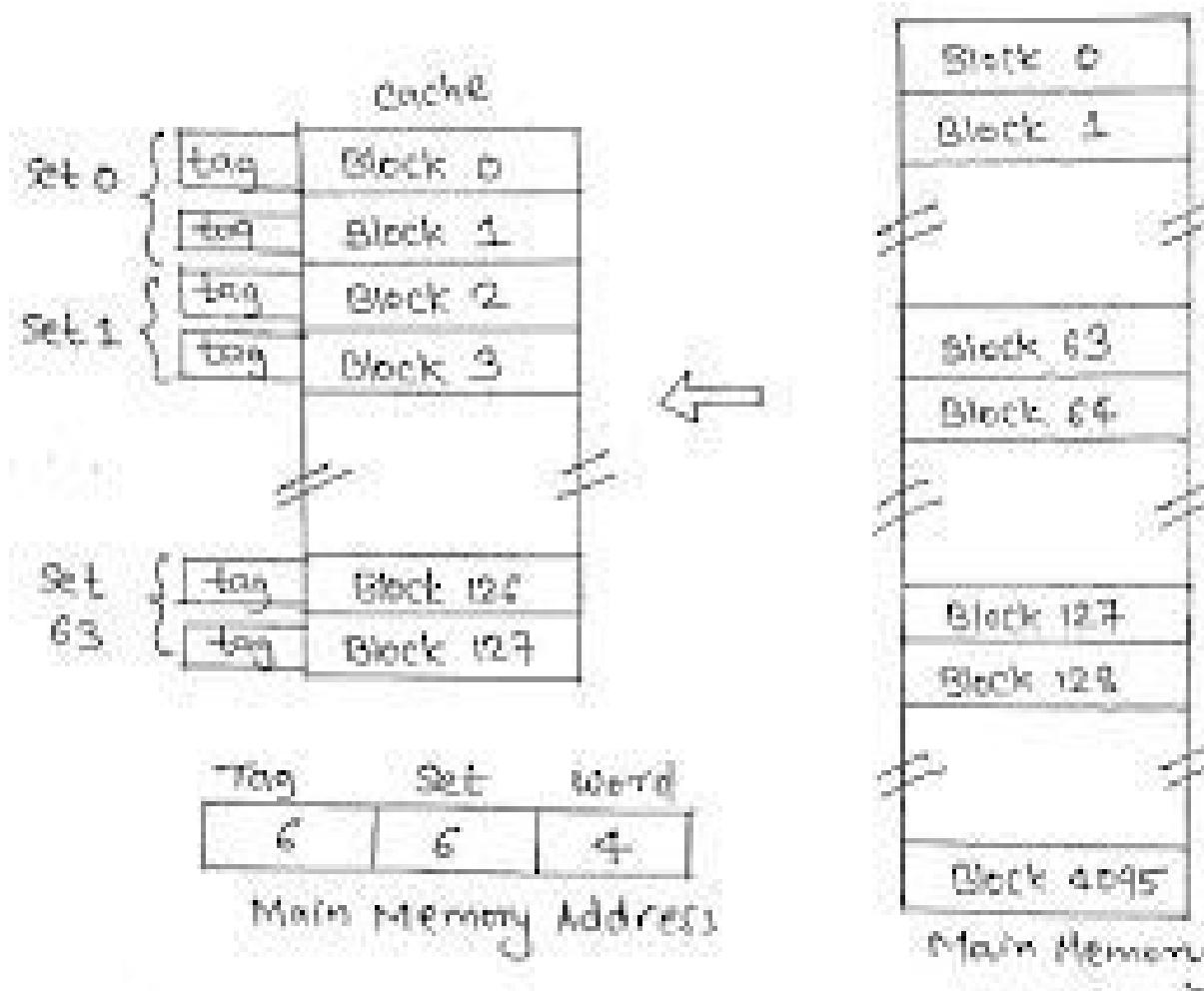
# Direct Mapping



# Fully Associative Mapping



# 2-Way Set Associative Mapping



# Mapping Function

Example 2

Assumption:

No. of lines(blocks) in a cache: 128

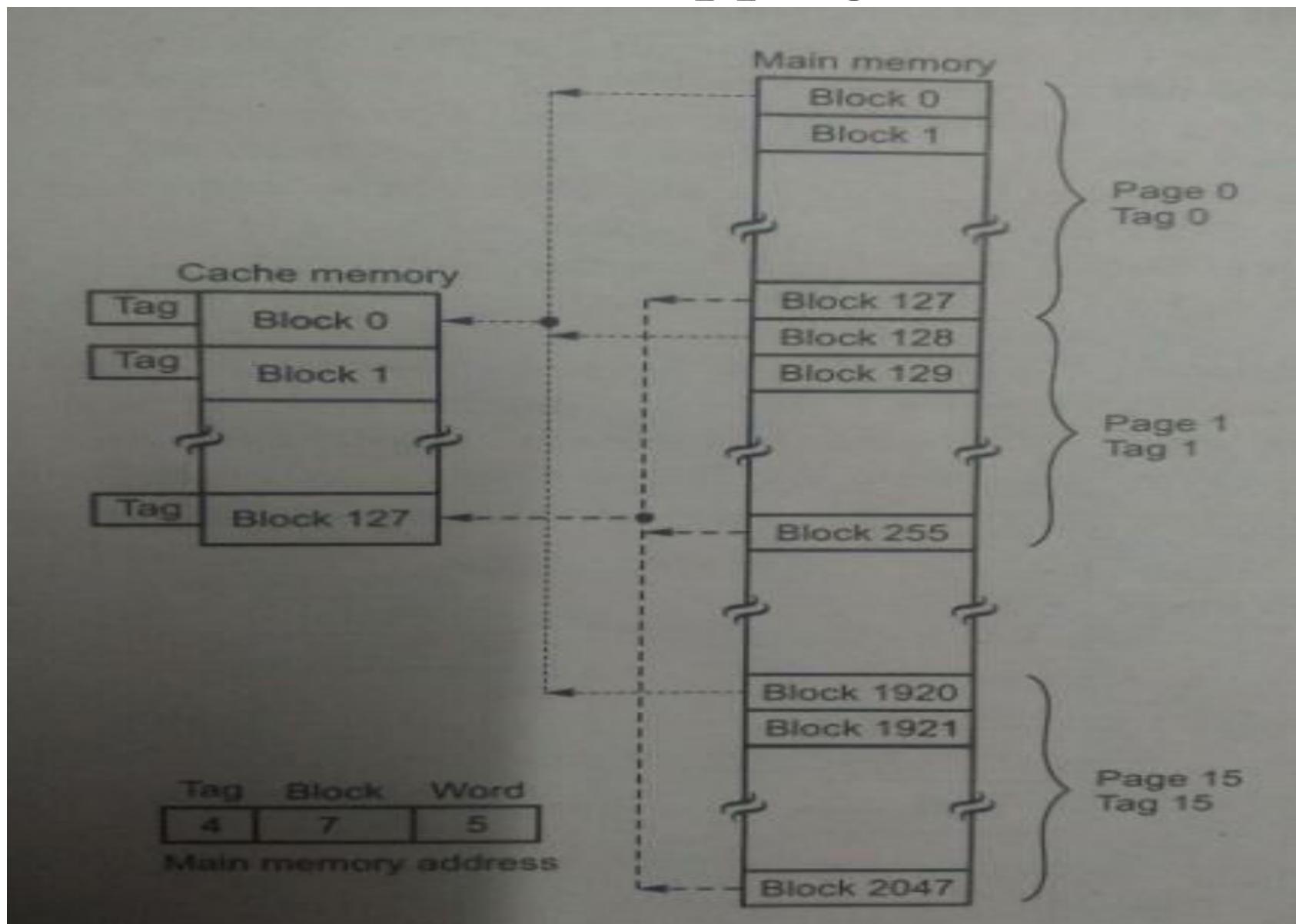
No. of blocks in main memory: 2046 blocks

No. of words in main memory: 64k words

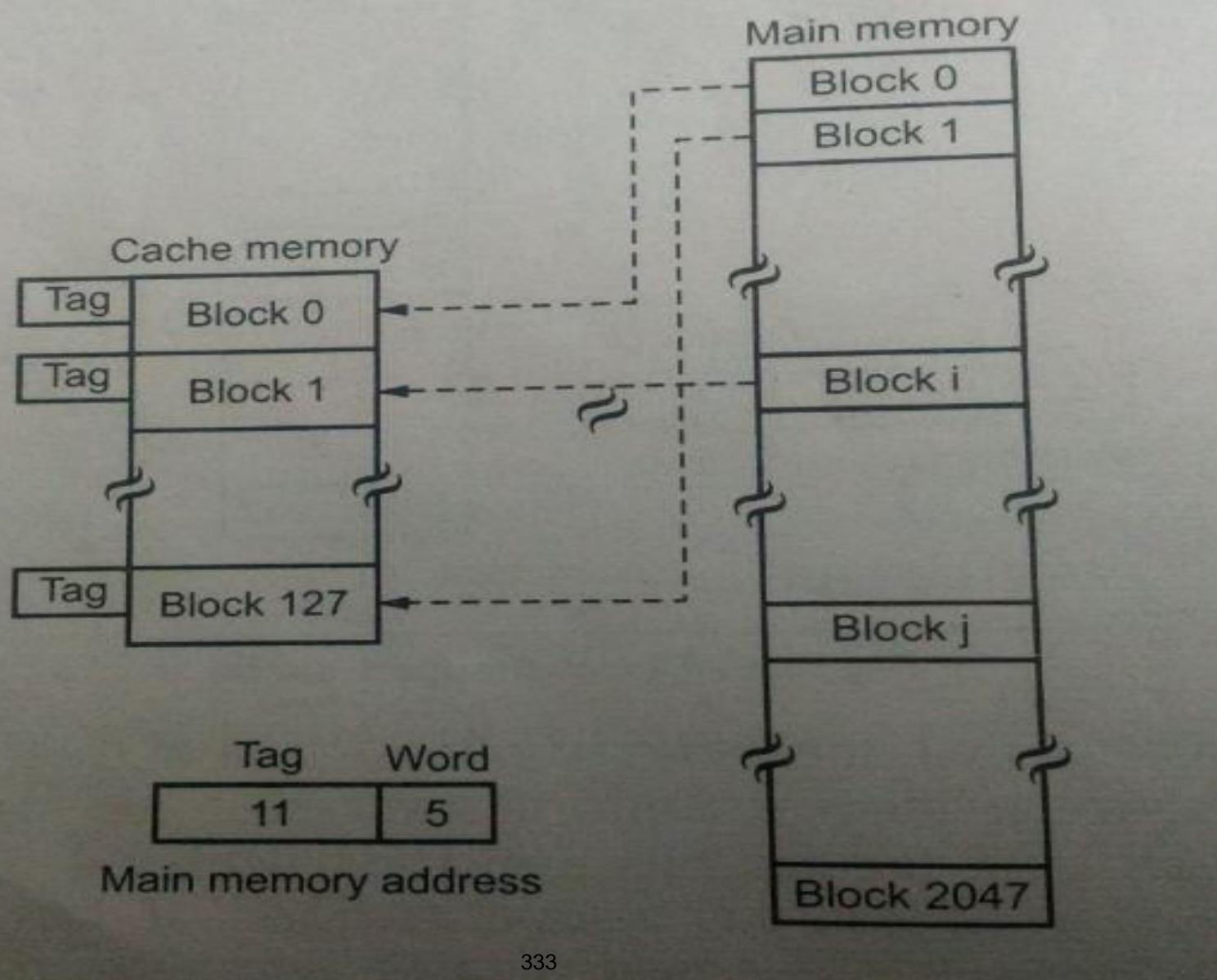
No. of Words in a block: 32

No. address bits: 16 ( $64k = 2^{10} * 2^6$ )

# Direct Mapping



# Fully Associative Mapping



# 2-Way Set Associative Mapping

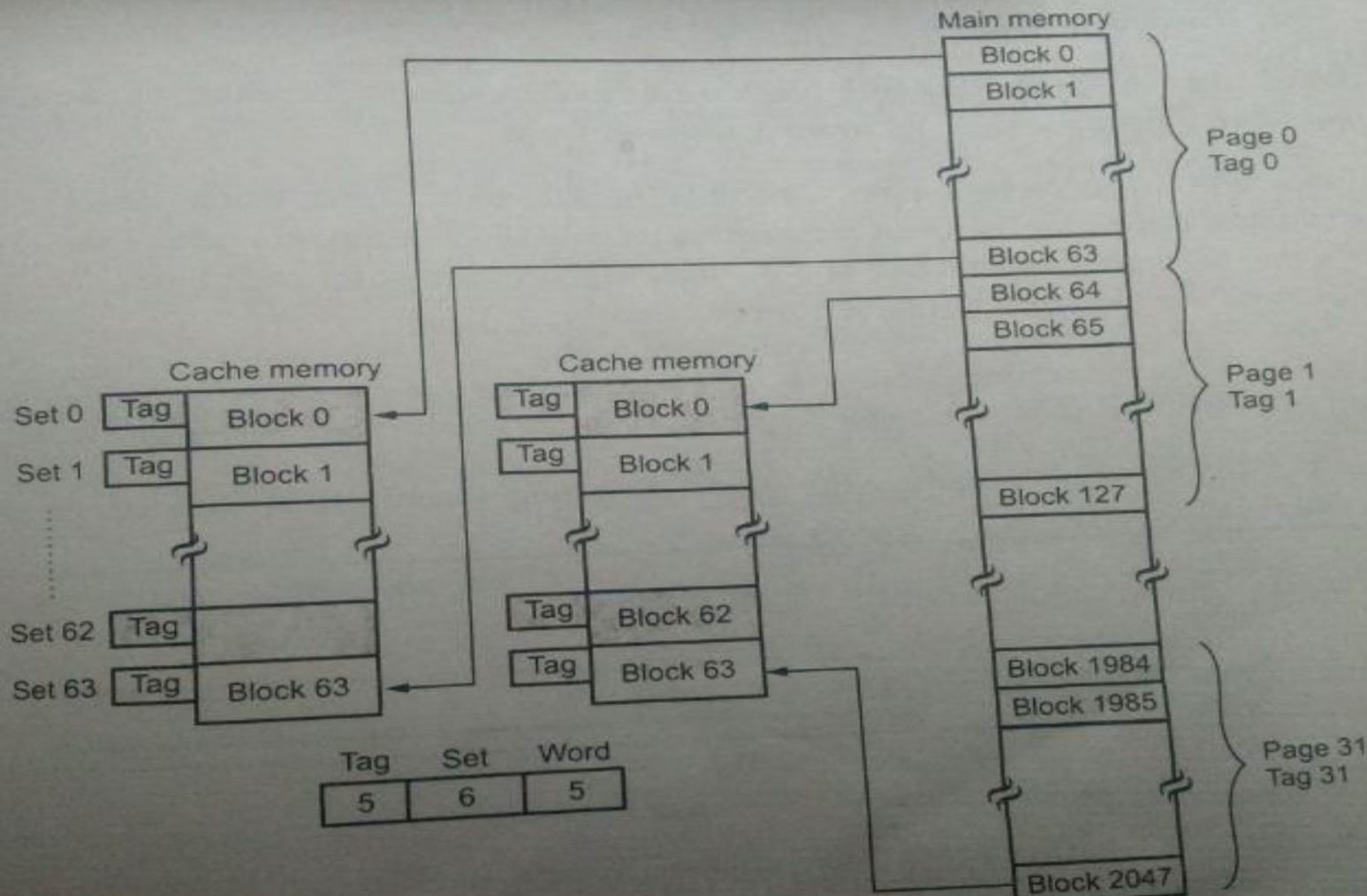


Fig. 4.53 Two-way set associative cache

# Problem 1

- A set associative cache consists of 64 lines or slots, divided into four line sets. Main memory consists 4k blocks of 128 words each. Show the format of main memory addresses.

- A set associative cache consists of 64 lines or slots, divided into four line sets. Main memory consists 4k blocks of 128 words each. Show the format of main memory addresses.

The cache is divided into 16 sets of 4 lines each. Therefore, 4 bits are needed to identify the set number. Main memory consists of  $4K = 2^{12}$  blocks. Therefore, the set plus tag lengths must be 12 bits and therefore the tag length is 8 bits. Each block contains 128 words. Therefore, 7 bits are needed to specify the word.

	TAG	SET	WORD
Main memory address =	8	4	7

- Tag =  $2^{12}/2^4$

# Problem 2

- A two-way set associative cache has lines of 16 bytes and a total size of 8k bytes. The 64-Mbyte main memory is byte addressable. Show the format of main memory addresses.

# Problem 2

- A two-way set associative cache has lines of 16 bytes and a total size of 8k bytes. The 64-Mbyte main memory is byte addressable. Show the format of main memory addresses.

There are a total of  $8\text{ kbytes}/16\text{ bytes} = 512$  lines in the cache. Thus the cache consists of 256 sets of 2 lines each. Therefore 8 bits are needed to identify the set number. For the 64-Mbyte main memory, a 26-bit address is needed. Main memory consists of  $64\text{-Mbyte}/16\text{ bytes} = 2^{22}$  blocks. Therefore, the set plus tag lengths must be 22 bits, so the tag length is 14 bits and the word field length is 4 bits.

Main memory address =	TAG	SET	WORD
	14	8	4

# Problem 3

- Consider a cache consisting of 128 blocks of 16 words each, for a total of 2k words, and assume that the main memory is addressable by a 16 bit address and it consists of 4k blocks. Show the format of main memory address in all three types of mapping.
- Assume 2way set associative for set associative mapping.

**Direct:**

Tag → 5bits →  $2^{12}/2^7 = 2^5$

Block → 7bits →  $2^7 = 128$

Word → 4bits →  $2^4 = 16$  words in a block

**Associative:****Set associative:**

Tag → 6bits →  $2^{12}/2^6 = 2^6$

Set → 6bits →  $2^6 = 64$  sets

Word → 4bits →  $2^4 = 16$  words in a block

# problem4

- A block set associative cache consists of a total of 64-lines divided into four-line sets. The main memory contains 4096 blocks, each consisting of 128 words.
- What is the main memory address size?
- Format of main memory address?
- What is the size of cache memory?

- What is the main memory address size?
  - Total word= $4096 * 128 = 2^{19}$
  - Address size=19
- Format of main memory address?
  - Tag $\rightarrow$ 8bits, set $\rightarrow$ 4bits, word $\rightarrow$ 7bits
- What is the size of cache memory?  $64 * 128$

# Block Replacement

- **Least Recently Used: (LRU)**

Replace that block in the set that has been in the cache longest with no reference to it.

- **First Come First Out: (FIFO)**

Replace that block in the set that has been in the cache longest.

- **Least Frequently Used: (LFU)**

Replace that block in the set that has experienced the fewest references

# Update Policies - Write Through

- Update main memory with every memory write operation
- Cache memory is updated in parallel **if it contains the word** at specified address.
- Advantage:
  - main memory always contains the same data as the cache
  - easy to implement
- Disadvantage: -
  - write is slower
  - every write needs a main memory access

# Write Back

- Only cache is updated during write operation and marked by flag. When the word is removed from the cache (at the time of replacement), it is copied into main memory
- *Advantage:*
  - writes occur at the speed of the cache memory
  - multiple writes within a block require only one write to main memory

*Disadvantage:*

- harder to implement
- main memory is not always consistent with cache

## Update policies – Contd..

- Write-Allocate
  - update the item in main memory and bring the block containing the updated item into the cache.
- Write-Around or Write-no-allocate
  - correspond to items not currently in the cache (i.e. write misses) the item is updated in main memory only without affecting the cache.

- ***Write Through with Write Allocate:***
  - on hits it writes to cache and main memory
  - on misses it updates the block in main memory and brings the block to the cache
- ***Write Through with No Write Allocate:***
  - on hits it writes to cache and main memory;
  - on misses it updates the block in main memory not bringing that block to the cache;

- ***Write Back with Write Allocate:***
  - on hits it writes to cache setting dirty bit for the block, main memory is not updated;
  - on misses it updates the block in main memory and brings the block to the cache;
- ***Write Back with No Write Allocate:***
  - on hits it writes to cache setting dirty bit for the block, main memory is not updated;
  - on misses it updates the block in main memory not bringing that block to the cache;

## Update policies – Contd..

- Write back:-Write only in cache, updating main memory only at the time of replacement.
- Write through:-Both are updating for each write operation.
- Write-Allocate:- first in main memory, then copy the block into cache.
- Write-Around or Write-no-allocate:- when write miss occurred, updated in main memory without affecting the cache.

# Performance analysis

- Look through: The cache is checked first for a hit, and if a miss occurs then the access to main memory is started.
- Look aside: access to main memory in parallel with the cache lookup.

- **Look through**

$$T_A = T_c + (1-h)*T_m$$

$T_A$  – Average read access time

$T_c$  is the average cache access time

$T_m$  is the average main memory access time

- **Look aside**

$$T_A = h*T_c + (1-h)*T_m$$

number of references found in the cache

- hit ratio  $h = \frac{\text{number of references found in the cache}}{\text{total number of memory references}}$

- **Miss Ratio  $m=(1-h)$**

Example: assume that a computer system employs a cache with an access time of 20ns and a main memory with a cycle time of 200ns. Suppose that the hit ratio for reads is 90%,

- a) what would be the average access time for reads if the cache is a look through-cache?

The average read access time ( $T_A$ ) =  $T_c + (1-h)*T_m$

$$20\text{ns} + 0.10*200\text{ns} = 40\text{ns}$$

- b) what would be the average access time for reads if the cache is a “look-Aside” cache?

The average read access time in this case ( $T_A$ )

$$= h*T_c + (1-h)*T_m = 0.9*20\text{ns} + 0.10*200\text{ns} = 38\text{ns}$$

# Problem 1

- Consider a memory system with  $T_c = 100\text{ns}$  and  $T_m = 1200\text{ns}$ . If the effective access time is 10% greater than the cache access time, what is the hit ratio  $H$  in look-through cache?

$$\Rightarrow T_A = T_C + (1-h)*T_M$$

$$\Rightarrow 1.1 T_c = T_c + (1-h)*T_M$$

$$\Rightarrow 0.1 T_C = (1-h) * T_M$$

$$\Rightarrow 0.1 * 100 = (1-h) * 1200$$

$$\Rightarrow 1-h = 10/1200$$

$$\Rightarrow h = 1190/1200$$

## Problem 2

- A computer system employs a write-back cache with a 70% hit ratio for writes. The cache operates in look-aside mode and has a 90% read hit ratio. Reads account for 80% of all memory references and writes account for 20%. If the main memory cycle time is 200ns and the cache access time is 20ns, what would be the average access time for all references (reads as well as writes)?



The average access time for reads

$$= 0.9 \times 20\text{ns} + 0.1 \times 200\text{ns} = 38\text{ns}.$$

The average write time

$$= 0.7 \times 20\text{ns} + 0.3 \times 200\text{ns} = 74\text{ns}$$

Hence the overall average access time for combined reads and writes is

$$= 0.8 \times 38\text{ns} + 0.2 \times 74\text{ns} = 45.2\text{ns}$$

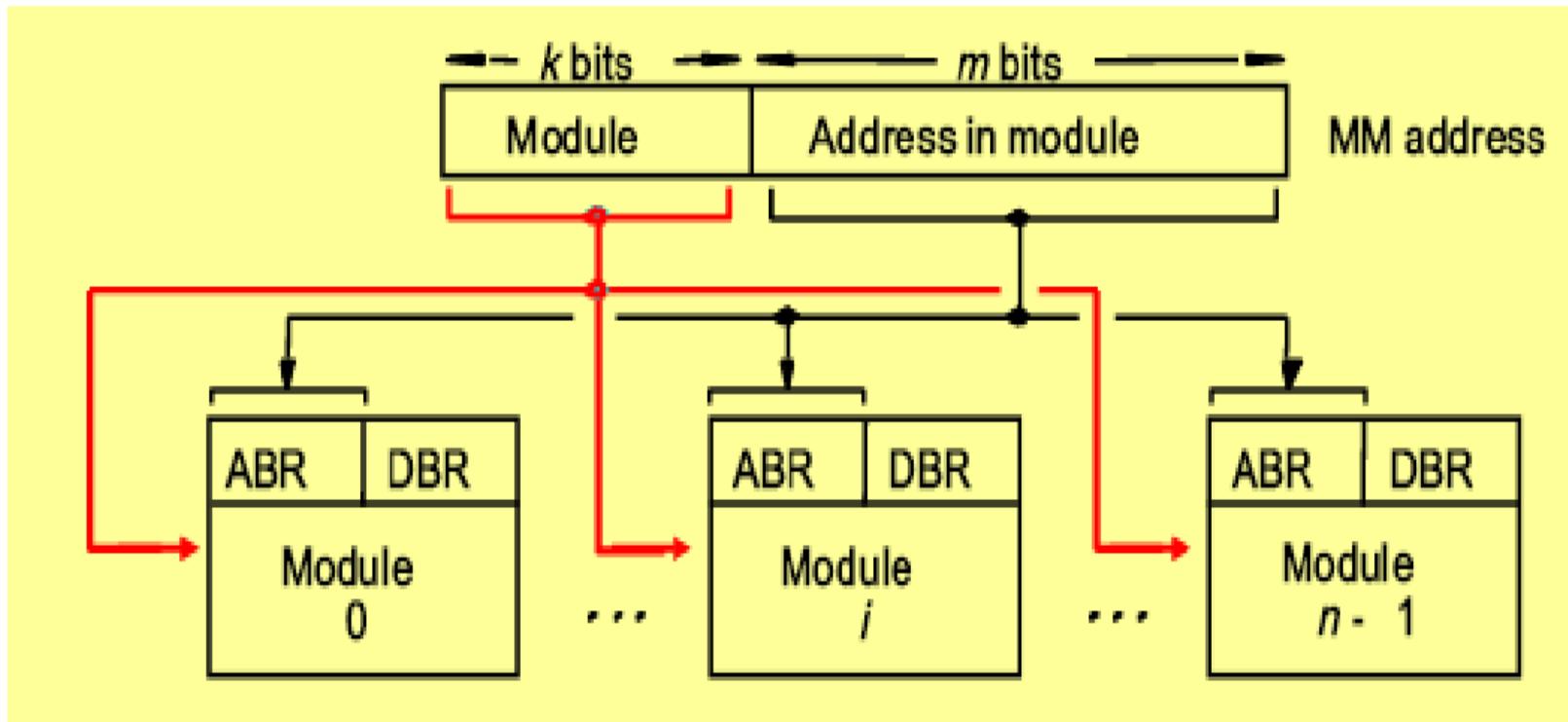
# References

- J. L. Hennessy & D.A. Patterson, Computer architecture: A quantitative approach, Fourth Edition, Morgan Kaufman, 2004.

# Memory Interleaving

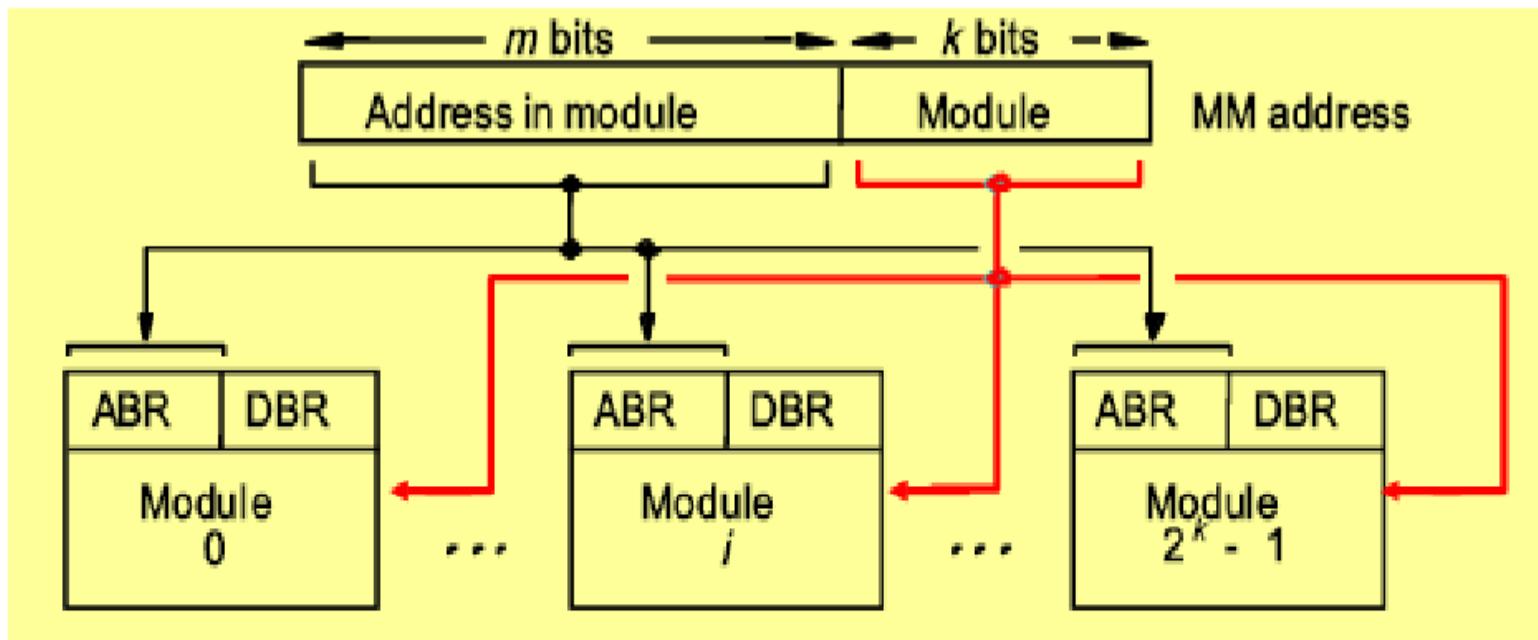
- If the main memory is structured as a collection of physically separate modules, each with its own address buffer register (ABR) and data buffer register (DBR), memory access operations may proceed in more than one module at the same time.
- Hence, aggregate rate of transmission of words to and from the memory can be increased.
- Two methods of distribution of words among modules
  - Consecutive words in a module
  - Consecutive words in consecutive modules

# Consecutive words in a module



- When consecutive locations are accessed, as happens when a block of data is transferred to a cache, only one module is involved.

# Consecutive words in consecutive modules



- This method is called memory interleaving
- Parallel access is possible. Hence, faster
- Higher average utilization of the memory system

# Cache Coherence Problem

# The Cache Coherence Problem

- Caches in multiprocessing environment introduce the cache coherence problem.
- When multiple processors maintain locally cached copies of a unique-shared memory location, any local modification of the location can result in globally inconsistent view of memory.
- Cache coherence protocols prevent this problem by maintaining a uniform state for each cached block of data.

# Causes of Cache Inconsistency

- Cache inconsistency only occurs when there are multiple caches capable of storing (potentially modified) copies of the same objects.
- There are three frequent sources of this problem:
  - Sharing of writable data
  - Process migration
  - I/O activity

# Inconsistency in Data Sharing

- Suppose two processors each use (read) a data item X from a shared memory. Then each processor's cache will have a copy of X that is consistent with the shared memory copy.
- Now suppose one processor modifies X (to X'). Now that processor's cache is inconsistent with the other processor's cache and the shared memory.
- With a write-through cache, the shared memory copy will be made consistent, but the other processor still has an inconsistent value (X).
- With a write-back cache, the shared memory copy will be updated eventually, when the block containing X (actually X') is replaced or invalidated.

# Inconsistency After Process Migration

- If a process accesses variable X (resulting in it being placed in the processor cache), and is then moved to a different processor and modifies X (to X'), then the caches on the two processors are inconsistent.
- This problem exists regardless of whether write-through caches or write-back caches are used

# Inconsistency Caused by I/O

- Data movement from an I/O device to a shared primary memory usually does not cause cached copies of data to be updated.
- As a result, an input operation that writes X causes it to become inconsistent with a cached value of X.
- Likewise, writing data to an I/O device usually use the data in the shared primary memory, ignoring any potential cached data with different values.

# Cache Coherence Protocols

- Snoopy protocols
- MSI Protocols
- MESI Protocols
- MOSI Protocols
- MOESI Protocols
- MERSI Protocol
- MESIF Protocol
- Write-once Protocol
- Firefly Protocol
- Dragon Protocol

# Cache Coherence Problem

# The Cache Coherence Problem

- Caches in multiprocessing environment introduce the cache coherence problem.
- When multiple processors maintain locally cached copies of a unique-shared memory location, any local modification of the location can result in globally inconsistent view of memory.
- Cache coherence protocols prevent this problem by maintaining a uniform state for each cached block of data.

# Causes of Cache Inconsistency

- Cache inconsistency only occurs when there are multiple caches capable of storing (potentially modified) copies of the same objects.
- There are three frequent sources of this problem:
  - Sharing of writable data
  - Process migration
  - I/O activity

# Inconsistency in Data Sharing

- Suppose two processors each use (read) a data item X from a shared memory. Then each processor's cache will have a copy of X that is consistent with the shared memory copy.
- Now suppose one processor modifies X (to X'). Now that processor's cache is inconsistent with the other processor's cache and the shared memory.
- With a write-through cache, the shared memory copy will be made consistent, but the other processor still has an inconsistent value (X).
- With a write-back cache, the shared memory copy will be updated eventually, when the block containing X (actually X') is replaced or invalidated.

# Inconsistency After Process Migration

- If a process accesses variable X (resulting in it being placed in the processor cache), and is then moved to a different processor and modifies X (to X'), then the caches on the two processors are inconsistent.
- This problem exists regardless of whether write-through caches or write-back caches are used

# Inconsistency Caused by I/O

- Data movement from an I/O device to a shared primary memory usually does not cause cached copies of data to be updated.
- As a result, an input operation that writes X causes it to become inconsistent with a cached value of X.
- Likewise, writing data to an I/O device usually use the data in the shared primary memory, ignoring any potential cached data with different values.

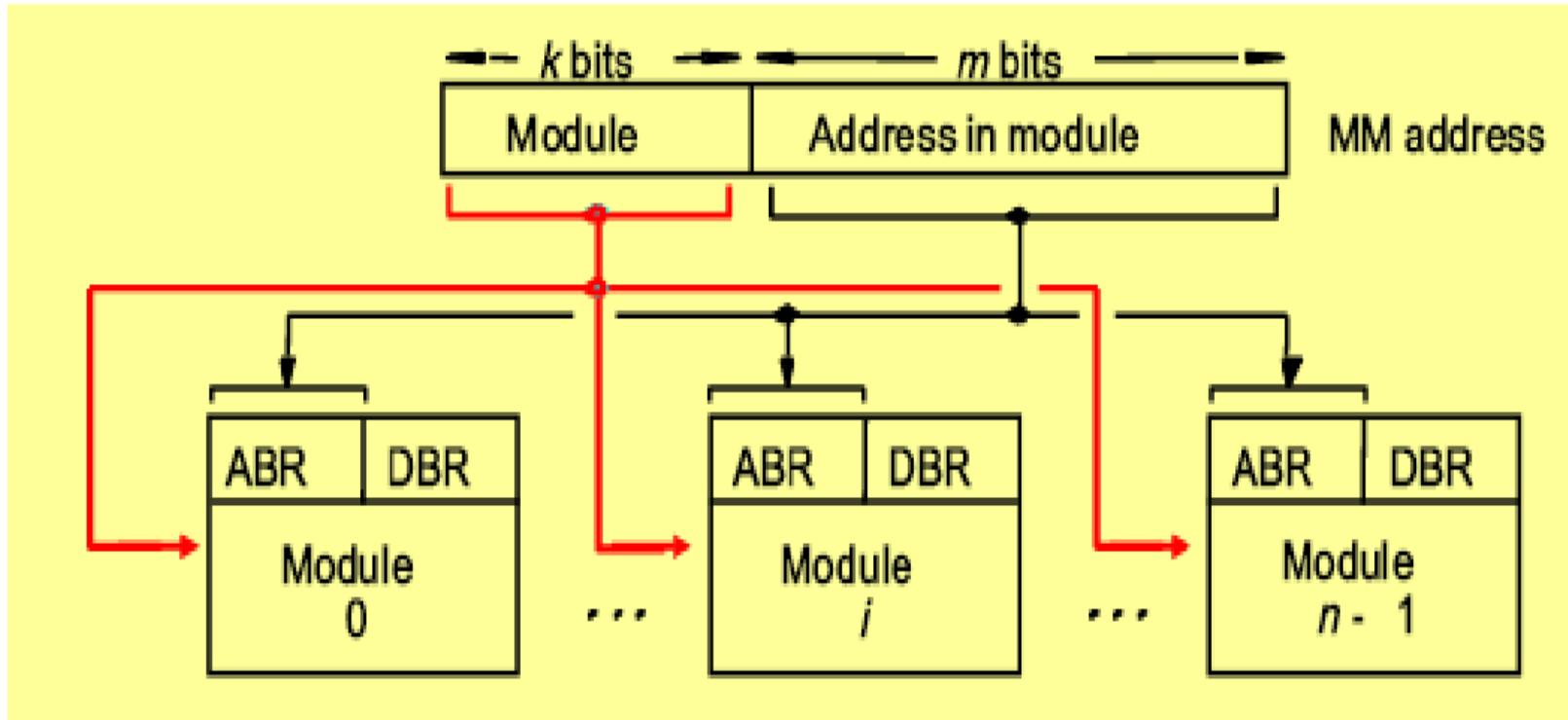
# Cache Coherence Protocols

- Snoopy protocols
- MSI Protocols
- MESI Protocols
- MOSI Protocols
- MOESI Protocols
- MERSI Protocol
- MESIF Protocol
- Write-once Protocol
- Firefly Protocol
- Dragon Protocol

# Memory Interleaving

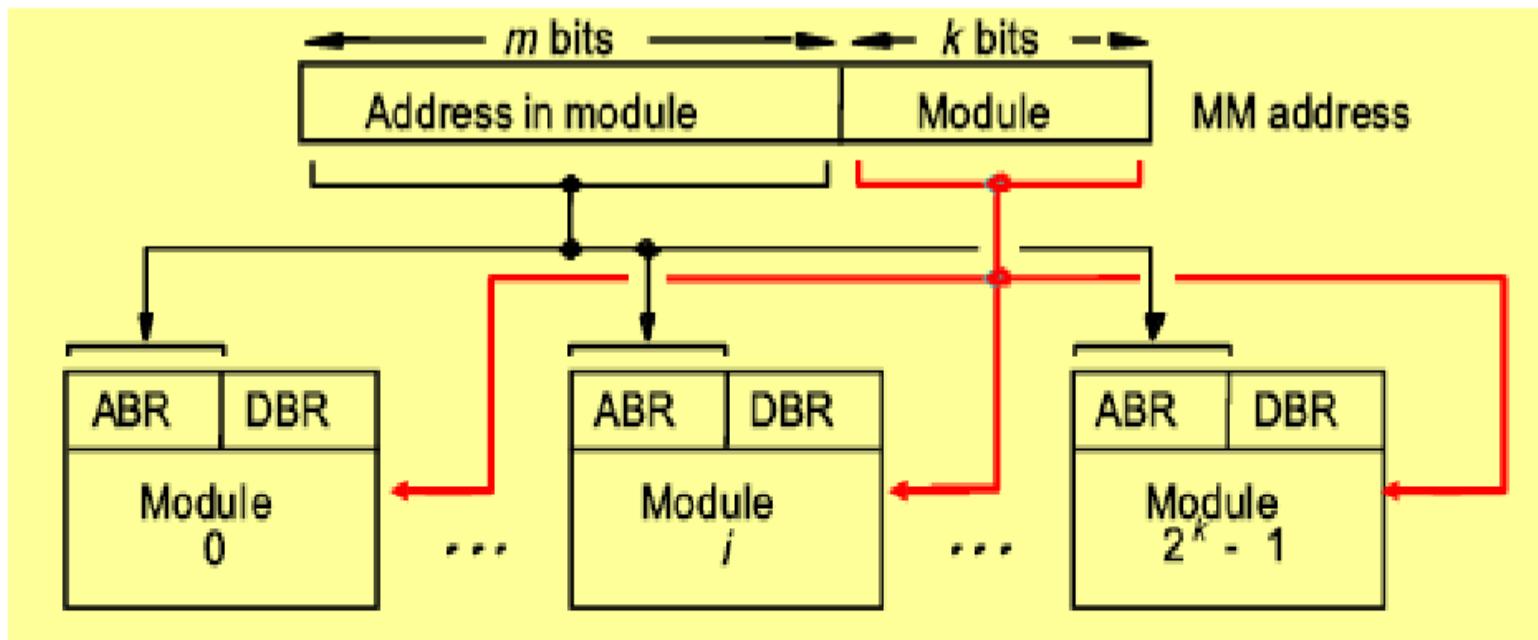
- If the main memory is structured as a collection of physically separate modules, each with its own address buffer register (ABR) and data buffer register (DBR), memory access operations may proceed in more than one module at the same time.
- Hence, aggregate rate of transmission of words to and from the memory can be increased.
- Two methods of distribution of words among modules
  - Consecutive words in a module
  - Consecutive words in consecutive modules

# Consecutive words in a module



- When consecutive locations are accessed, as happens when a block of data is transferred to a cache, only one module is involved.

# Consecutive words in consecutive modules



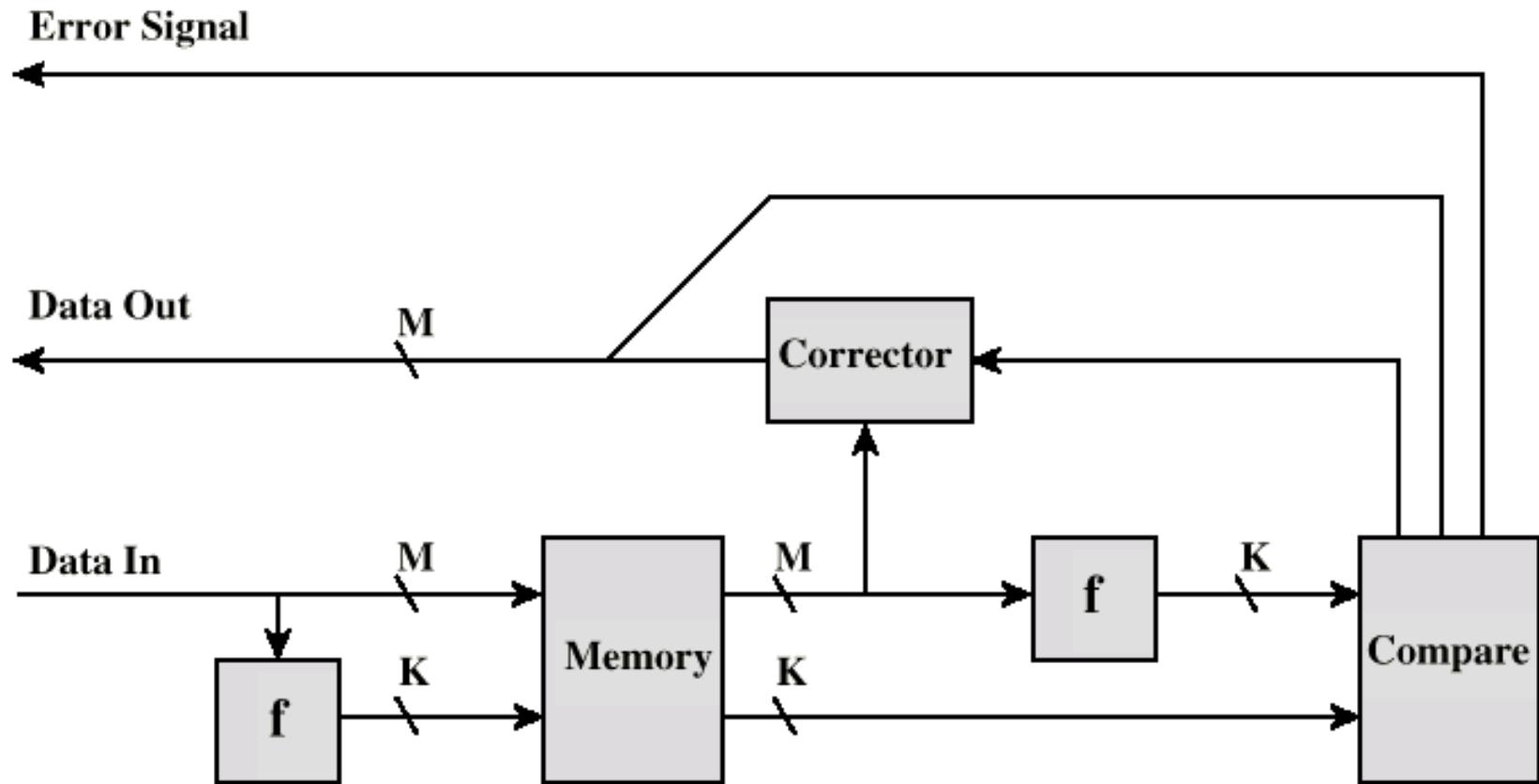
- This method is called memory interleaving
- Parallel access is possible. Hence, faster
- Higher average utilization of the memory system

# ERROR DETECTION AND CORRECTION

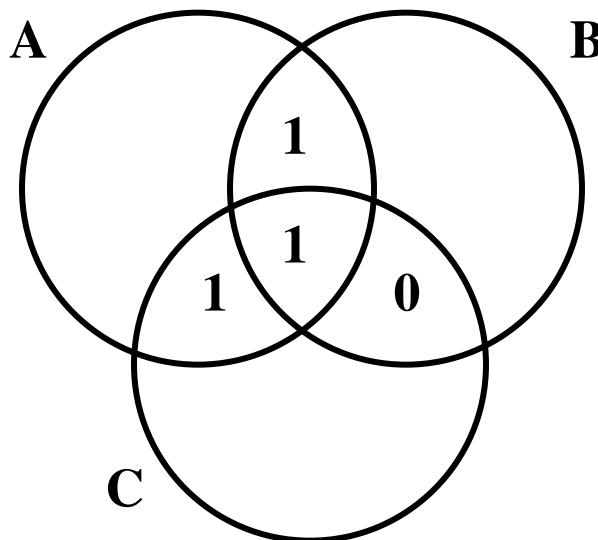
# Error Correction

- A semiconductor memory system is subject to errors.
- Hard failures – permanent physical defects  
Environmental abuse, manufacturing defects, etc.
- Soft error  
Power supply problems etc.
- Need logic for detecting and correcting errors.
- Basic technique
  - Prior to storing data a code is generated from the bits in the word.
  - Code is stored along with the word in memory.
  - Code used to identify and correct errors.
- When the word is fetched a new code is generated and compared to the stored code.
  - No error (normal case)
  - Correctable error is detected and corrected.
  - Non-fixable error is detected and reported.

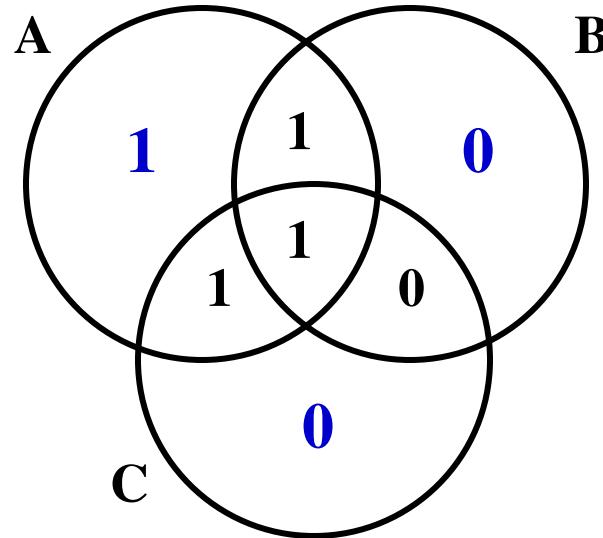
# Error Correcting Code Function



# Hamming Code



Assign data bits to the inner compartments.

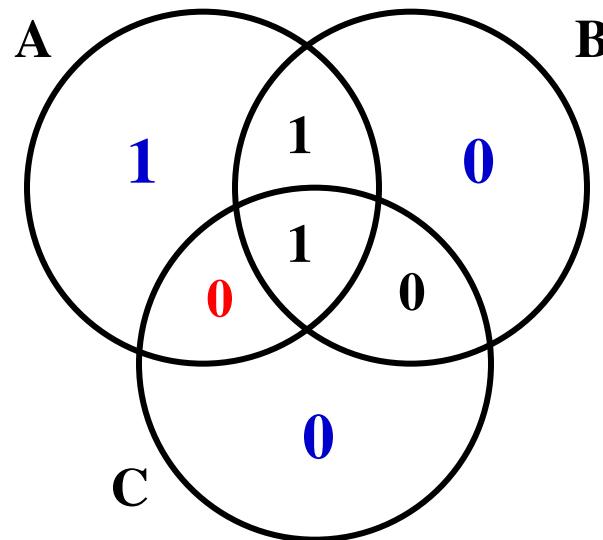
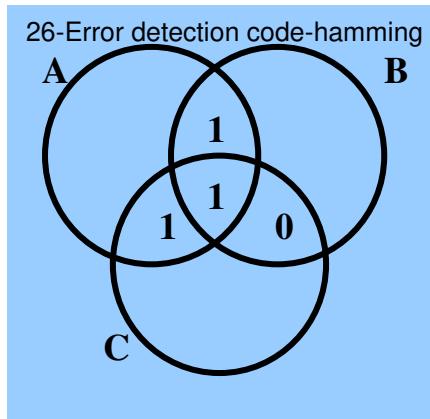


Fill the remaining compartments with parity bits.

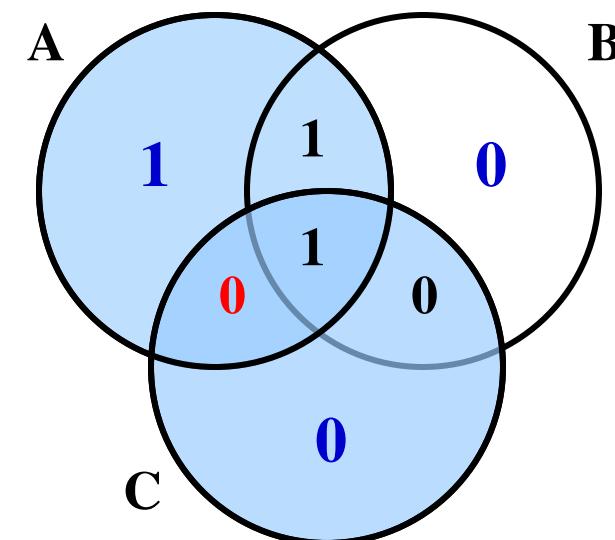
The total number of bits in a circle must be even.

For example: The data bits in A =  $1+1+1 = 3$ . This is odd – therefore add an additional 1.  
380

# Hamming Code



If a bit gets erroneously changed, the parity bits in that circle will no longer add up to 1.



Errors are found in A and C – and the shared bit in A and C is in error and can be fixed.

# Single Bit Errors in 8-bit words

- 8 data bits
- $2^K - 1 \geq M + K$  is used to find the value of K (number of check bits).
- One error bit => error occurred in one of the check bits. No action.
- More than one bit set to ‘1’ => the numerical value of the syndrome indicates the position of the data bit in error.
- $2^K - 1 \geq 8 + K \Rightarrow K = 4$

# Single Bit Errors in 8-bit words

- Data and check bits arranged into a 12-bit word.
- Bit positions numbered from 1 to 12.
- Bit positions representing position numbers that are powers of 2 are designated as check bits.
- Check bits calculated as follows:

$C_1 = D_1 \oplus D_2 \oplus$	$D_4 \oplus D_5 \oplus$	$D_7$
$C_2 = D_1 \oplus$	$D_3 \oplus D_4 \oplus$	$D_6 \oplus D_7$
$C_4 =$	$D_2 \oplus D_3 \oplus D_4 \oplus$	$D_8$
$C_8 =$	$D_5 \oplus D_6 \oplus D_7 \oplus$	$D_8$

- Data and check bits arranged into a 12 bit syndrome word:



# Calculating check bits

Bit Position	Binary	Type	
1	0001	C1	$\oplus$ All D's with a 1 in bit 1
2	0010	C2	$\oplus$ All D's with a 1 in bit 2
3	0011	D1	
4	0100	C4	$\oplus$ All D's with a 1 in bit 3
5	0101	D2	
6	0110	D3	
7	0111	D4	
8	1000	C8	$\oplus$ All D's with a 1 in bit 4
9	1001	D5	
10	1010	D6	
11	1011	D7	
12	1100	D8	

$$C1 = D1 \oplus D2 \oplus D4 \oplus D5 \oplus D7$$

Each check bit works on every data bit who shares the same bit position

# Example

- Input word: 00111001  
Data bit D1 in rightmost position

- Calculate check bits:

$$C_1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$

$$C_2 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$

$$C_3 = 0 \oplus 0 \oplus 1 \oplus 0 = 1$$

$$C_4 = 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

Stored word = 001101001111

- If data bit 3 sustains an error (001101101111)

$$C_1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$

$$C_2 = 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 0$$

$$C_3 = 0 \oplus 1 \oplus 1 \oplus 0 = 0$$

$$C_4 = 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

- Calculate syndrome word:

0110 = bit position 6.

D3 resides in bit position 6.

C	C	C	C
8	4	2	1
0	1	1	1
$\oplus$	0	0	0
0 1 1 0			

# Problems

- Suppose an 8-bit data word stored in memory is 11000010. Using the Hamming algorithm, determine what check bits would be stored in memory with the data word. Show how you got your answer.

# Problems

- Suppose an 8-bit data word stored in memory is 11000010. Using the Hamming algorithm, determine what check bits would be stored in memory with the data word. Show how you got your answer.
- $C1 = X\text{-OR}[1 \ 2 \ 4 \ 5 \ 7] = 0 \ 1 \ 0 \ 0 \ 1 = 0$
- $C2 = X\text{-OR}[1 \ 3 \ 4 \ 6 \ 7] = 0 \ 0 \ 0 \ 0 \ 1 = 1$
- $C4 = X\text{-OR}[2 \ 3 \ 4 \ 8] = 1 \ 0 \ 0 \ 1 = 0$
- $C8 = X\text{-OR}[5 \ 6 \ 7 \ 8] = 0 \ 0 \ 1 \ 1 = 0$
- 12 bit word is: 11000010010

- At the receiver side,
- If the fetched data word is,

1100 $\color{blue}{0}$ 101 $\color{red}{0}$ 010

→ extract the 8bit data

11001010

→ calculate the check bits:

- $C_1 = X\text{-OR}[1\ 2\ 4\ 5\ 7] = 0\ 1\ 1\ 0\ 1 = 1$
- $C_2 = X\text{-OR}[1\ 3\ 4\ 6\ 7] = 0\ 0\ 1\ 0\ 1 = 0$
- $C_4 = X\text{-OR}[2\ 3\ 4\ 8] = 1\ 0\ 1\ 1 = 1$
- $C_8 = X\text{-OR}[5\ 6\ 7\ 8] = 0\ 0\ 1\ 1 = 0$

- Now check the error
- Check bits at sender side is: 0010
- Check bits at sender side is: 0101
- E-OR both

0 0 1 0

0 1 0 1

---

0 1 1 1 = **7** ; if this is 0, no error else,

Error is in the seventh position of 12 bit data.

# Problems

- For the 8-bit word 00111001, the check bits stored with it would be 0111. Suppose when the word is read from memory, the check bits are calculated to be 1101. What is the data word that was read from memory?
- How many check bits are needed if the hamming error correction code is used to detect single bit errors in 1024 bit data word
- Generate the code for the data word 010100000111001. show that the code will correctly identify an error in data bit 5.

# References

- William Stallings “Computer Organization and architecture” Prentice Hall, 7th edition, 2006
- Internet: PPT slides, Author: Jane and Harold Huang

# VIRTUAL MEMORY

# Virtual memory

- Virtual memory permits the user to construct programs as though a large memory space were available.
- It gives the programmer an illusion that they have a very large memory.
- It provides a mechanism for translating program generated addresses into correct main memory locations by means of mapping table.

# Virtual Memory

- Virtual address – address used by a programmer
- Address space - Set of virtual addresses
- Physical address – address in main memory
- Memory space – set of physical addresses
- Memory mapping - process of translating virtual addresses into physical addresses

# Paged Virtual Memory

- **Block** - Organization of memory space.
- **Page** – blocks of contiguous virtual memory addresses - Organization of address space .
- **Page table**- a data structure used for translating virtual addresses (seen by application) to physical addresses (used by hardware) to process instructions.
- The hardware that handles this specific translation is called as **Memory Management Unit (MMU)**.
- Pages are moved from auxiliary memory to main memory in records equal to size of a page.

# Page Table & Address Translation

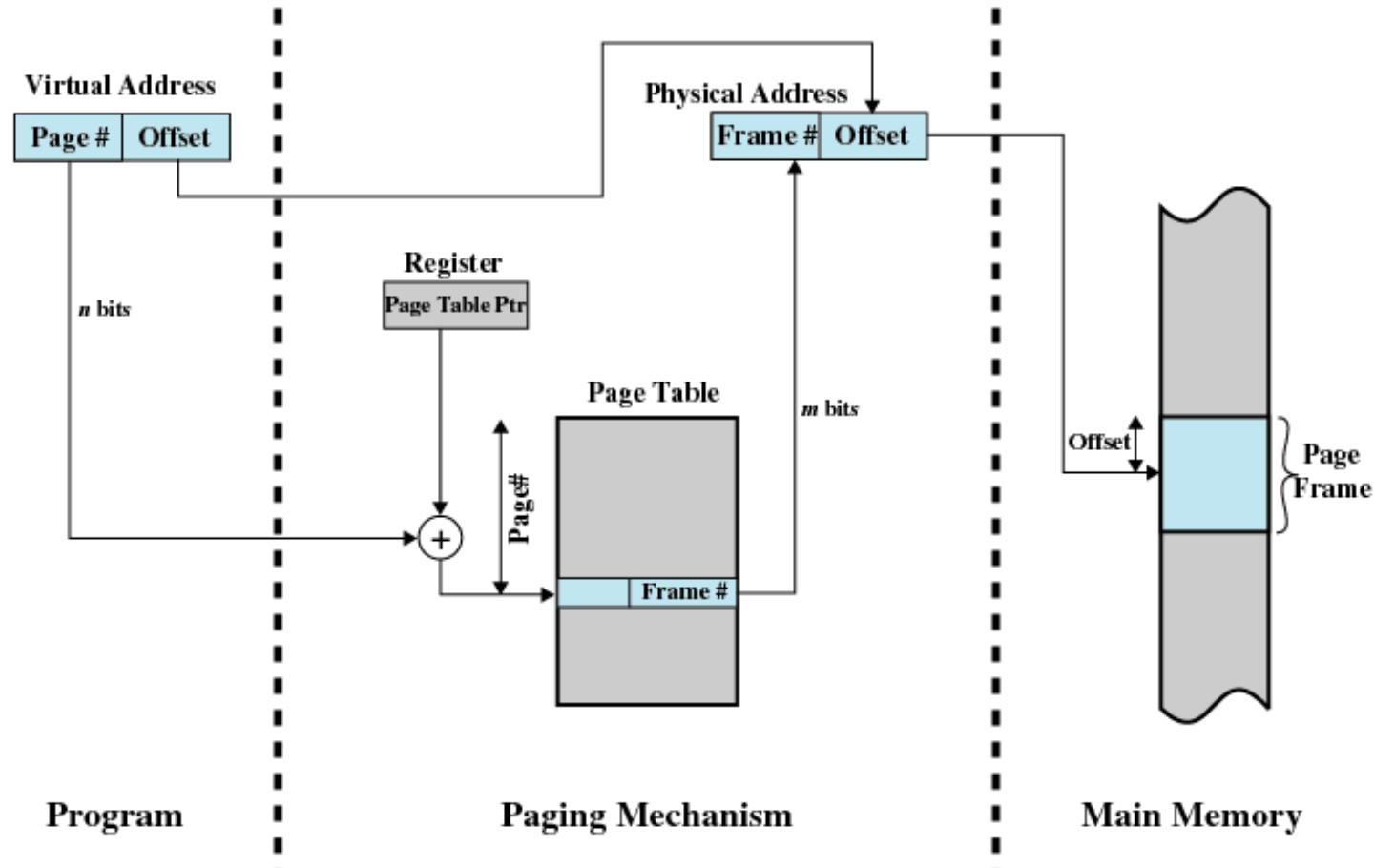
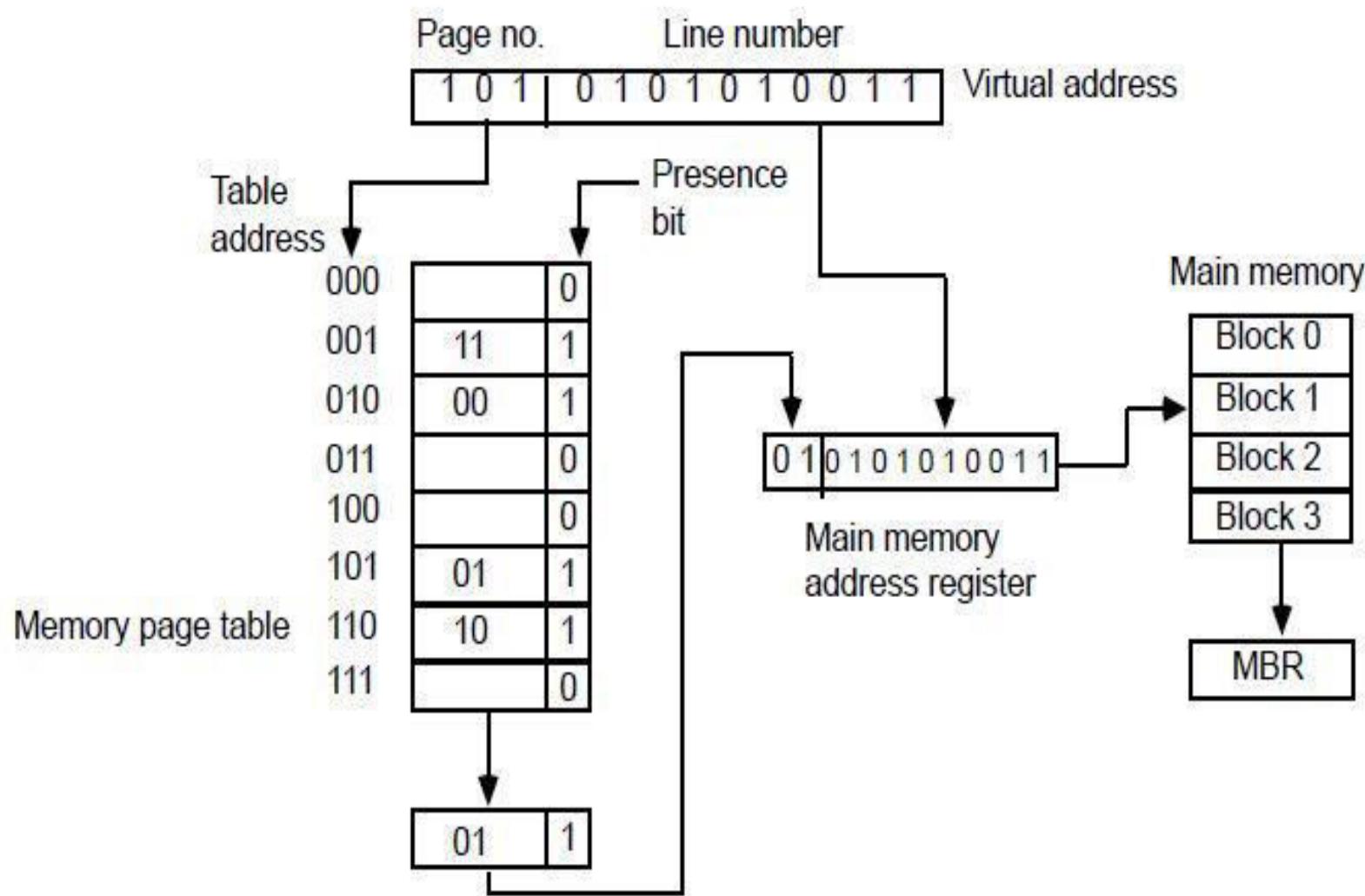


Figure 8.3 Address Translation in a Paging System

# Page table



# Page table

- Assume that
- Number of Blocks in memory = m
- Number of Pages in Virtual Address Space = n
- Page Table
  - Straight forward design -> n entry table in memory
  - Inefficient storage space utilization => n-m entries of the table is empty
- More efficient method is m-entry Page Table
- Page Table made of an Associative Memory - m words; (Page Number: Block Number)
- Page Fault - Page number cannot be found in the Page Table

# Translation Look-aside Buffer

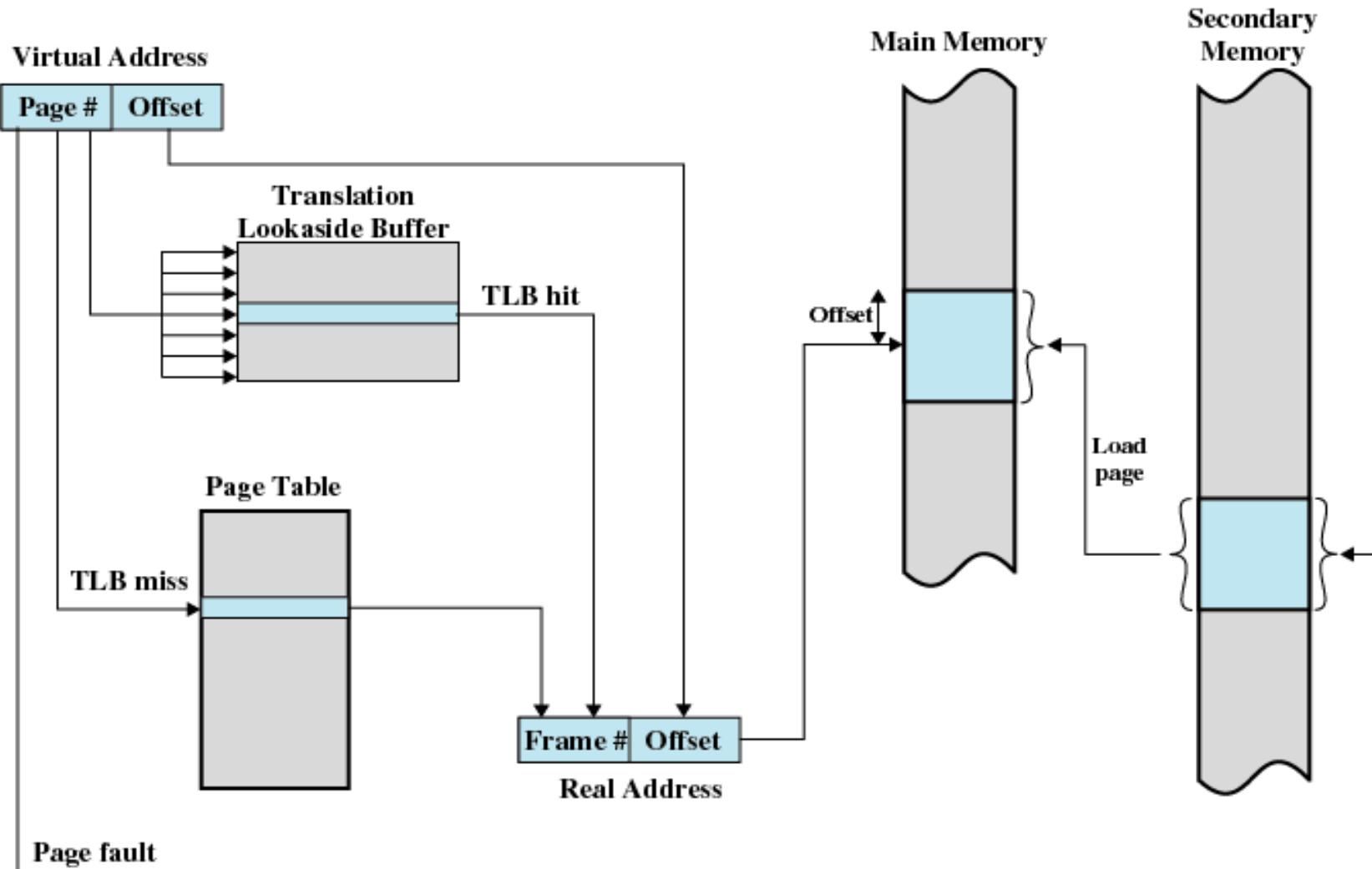
- Each virtual memory reference can cause two physical memory accesses
  - One to fetch the page table
  - One to fetch the data
- To overcome this problem a high-speed cache is set up for page table entries
  - Called a Translation Look-aside Buffer (TLB)
  - TLB Contains page table entries that have been most recently used

# Translation Look-aside Buffer

- Given a virtual address, processor examines the TLB
- If page table entry is present (TLB hit), the frame number is retrieved and the real address is formed
- If page table entry is not found in the TLB (TLB miss), the page number is used to index the process page table

# Translation Look-aside Buffer

- First checks if page is already in main memory
  - If not in main memory a page fault is issued
- The TLB is updated to include the new page entry



**Figure 8.7 Use of a Translation Lookaside Buffer**

**Return to  
Faulted Instruction**

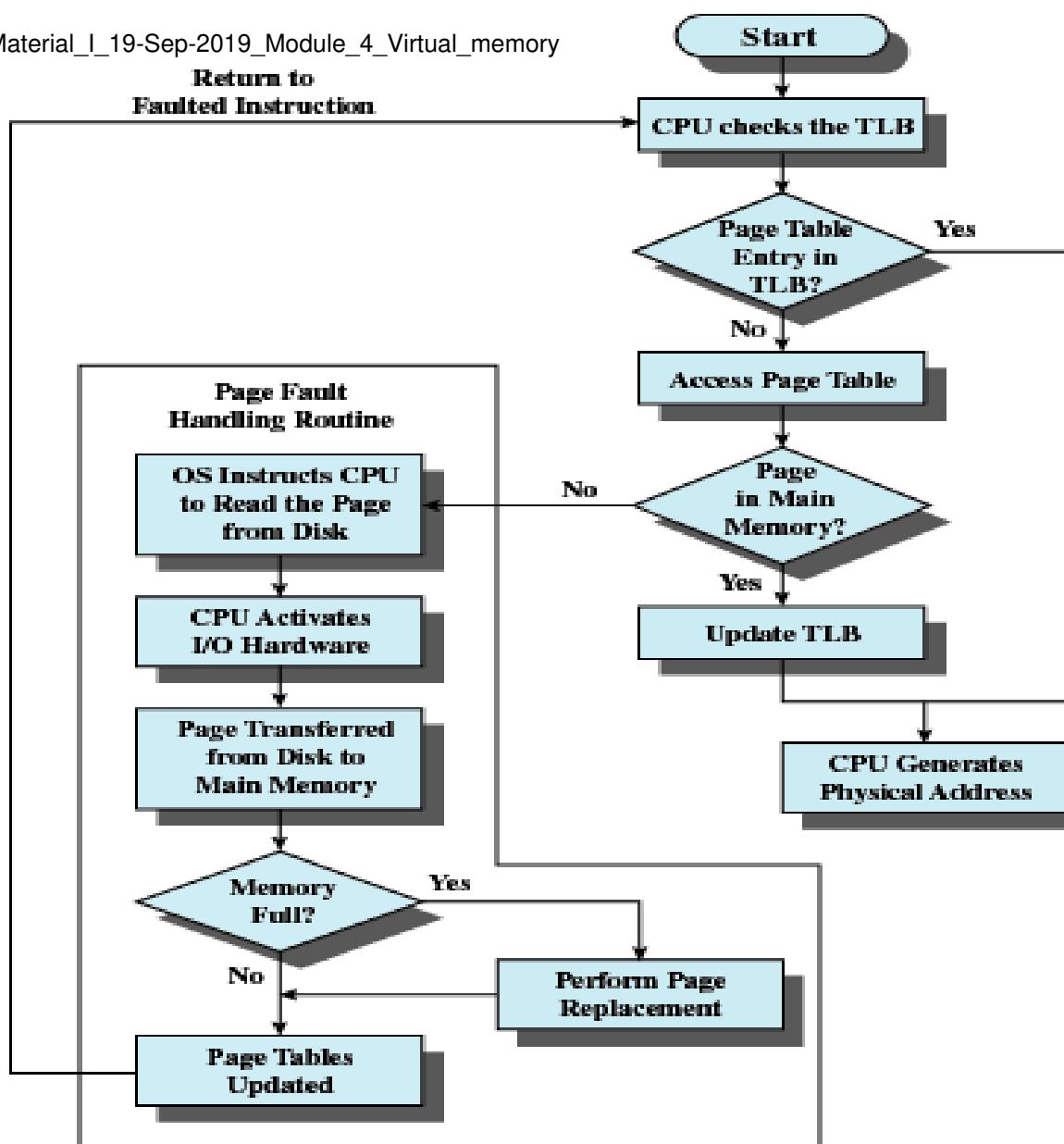


Figure 8.8 Operation of Paging and Translation Lookaside Buffer (TLB) [FURH87]

# Need for a fully- associative virtual memory

Fully associative: A virtual memory page can be placed anywhere in the physical memory.

- The design of the virtual memory focuses in reducing the miss rate
- an important source for misses are the conflicts in direct mapped and set associative caches (refer caches).
- a fully associative cache (TLB) eliminates these conflicts **but it is very expensive.**
- Hence a fully associative mapping is required through which any virtual page can be placed anywhere in the physical memory.

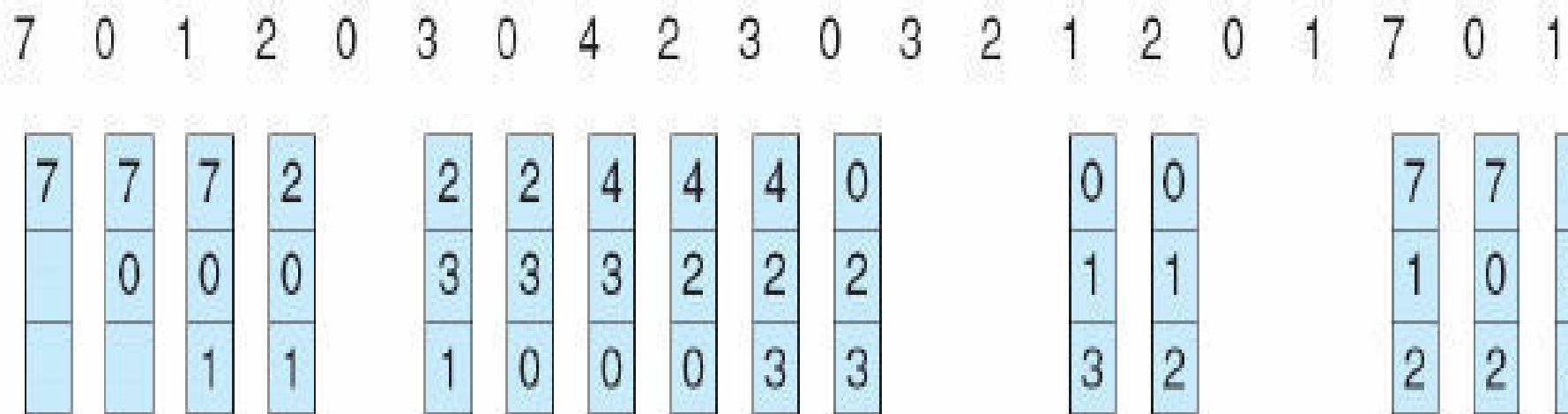
# Page replacement Algorithms

- Which page need to be replaced when page fault occurs?
- Page Replacement algorithms
  - FIFO (First-In First-Out)
  - Optimal page replacement
  - LRU (Least Recently Used)
  - LFU(Least Frequently Used)

# FIFO

- When a page must be replaced, the oldest page is chosen

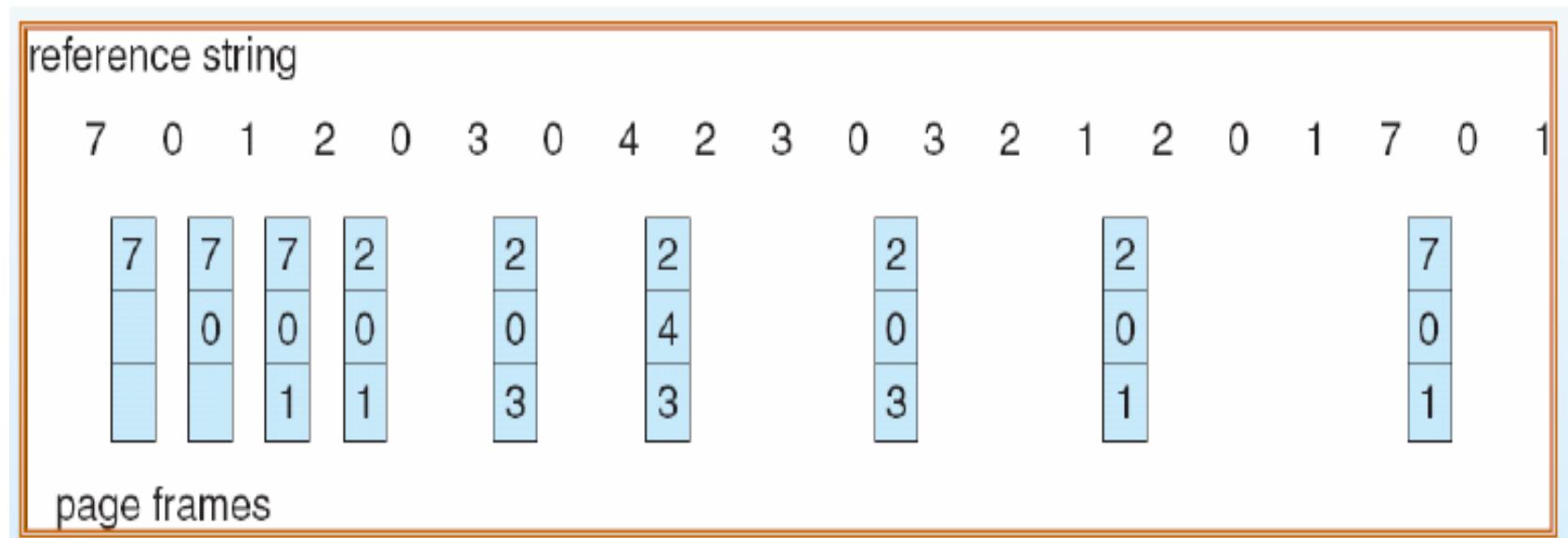
reference string



page frames

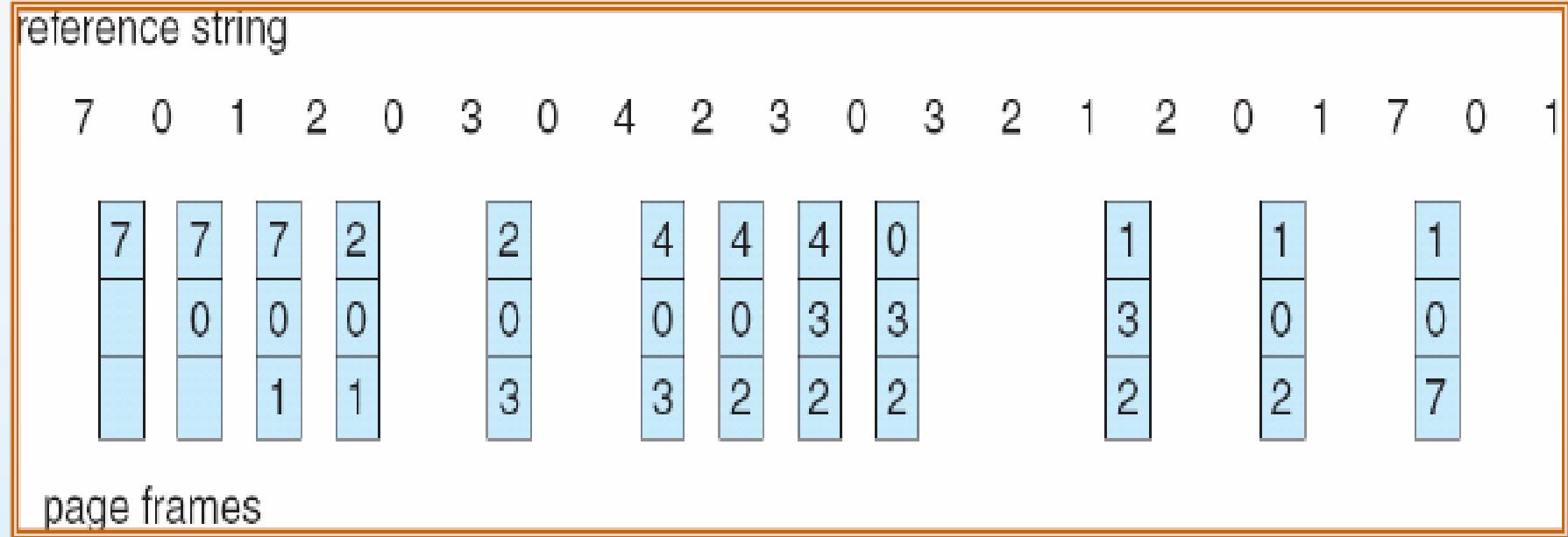
# Optimal Page-Replacement Algorithm

Replace the page that won't be needed for the longest time in the future



# Least-recently-used (LRU) algorithm

Replace the page that hasn't been referenced for the longest time



# Problem 1

A virtual memory system has an address space of 8K words, a memory space of 4K words and page and block sizes of 1K words. The following page reference changes occur during a given time interval.

Determine the four pages that are resident in main memory after each page reference change if the replacement algorithm is (a) FIFO (b) LRU (c) Optimal and (d) LFU page replacement algorithm. Compare the performance of the algorithms using

the page hit ratio.

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

## Problem 2

The following sequence of virtual page numbers is encountered in the course of execution on a computer with virtual memory:

3 4 2 6 4 7 1 3 2 6 3 5 1 2 3

Determine the four pages that are resident in main memory after each page reference. Calculate the hit ratio considering LRU as the replacement policy adopted. Identify the type of miss when occurred and total number of misses in each case.

# I/O FUNDAMENTALS

# Introduction

- The I/O subsystem provides an efficient mode of communication between CPU and outside environment
- Devices that are under the direct control of computer are said to be connected on-line
- Input or output devices attached to the computer are also referred as peripherals.
- **I/O interface or I/O Module** provides a method for transferring information between internal storage and external i/o devices.

# IO interface/IO module

Resolves the *differences* between the computer and peripheral devices

## -Design

Peripherals - Electromechanical Devices

CPU or Memory - Electronic Device

## - Data Transfer Rate

Peripherals - Usually slower

CPU or Memory - Usually faster than peripherals

Some kinds of Synchronization mechanism may be needed

## - Unit of Information

Peripherals - Byte

CPU or Memory - Word

## - Operating Modes

Peripherals - Asynchronous

CPU or Memory - Synchronous

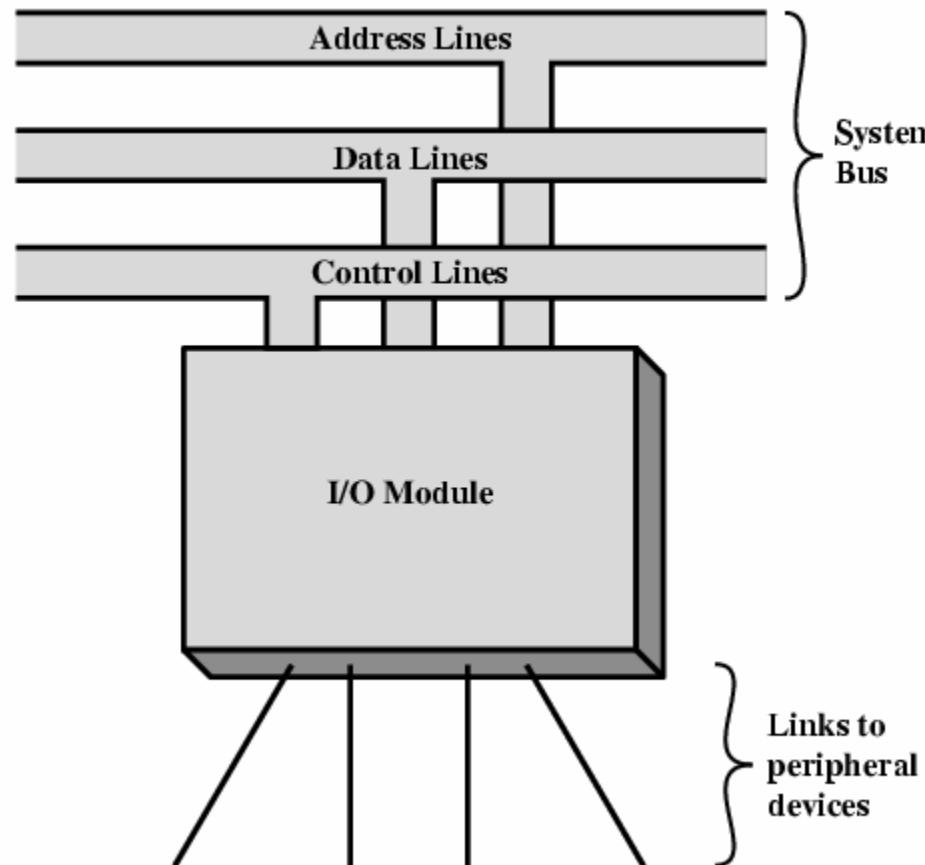
# Input/Output Problems

- Wide variety of peripherals
  - Delivering different amounts of data
  - At different speeds
  - In different formats
- All slower than CPU and RAM
- Need I/O modules

# Input/Output Module

- Interface to CPU and Memory
- Interface to one or more peripherals

# Generic Model of I/O Module



# I/O Module Function

- Control & Timing
- CPU Communication
- Device Communication:- BUSY, READY signal
- Data Buffering
- Error Detection

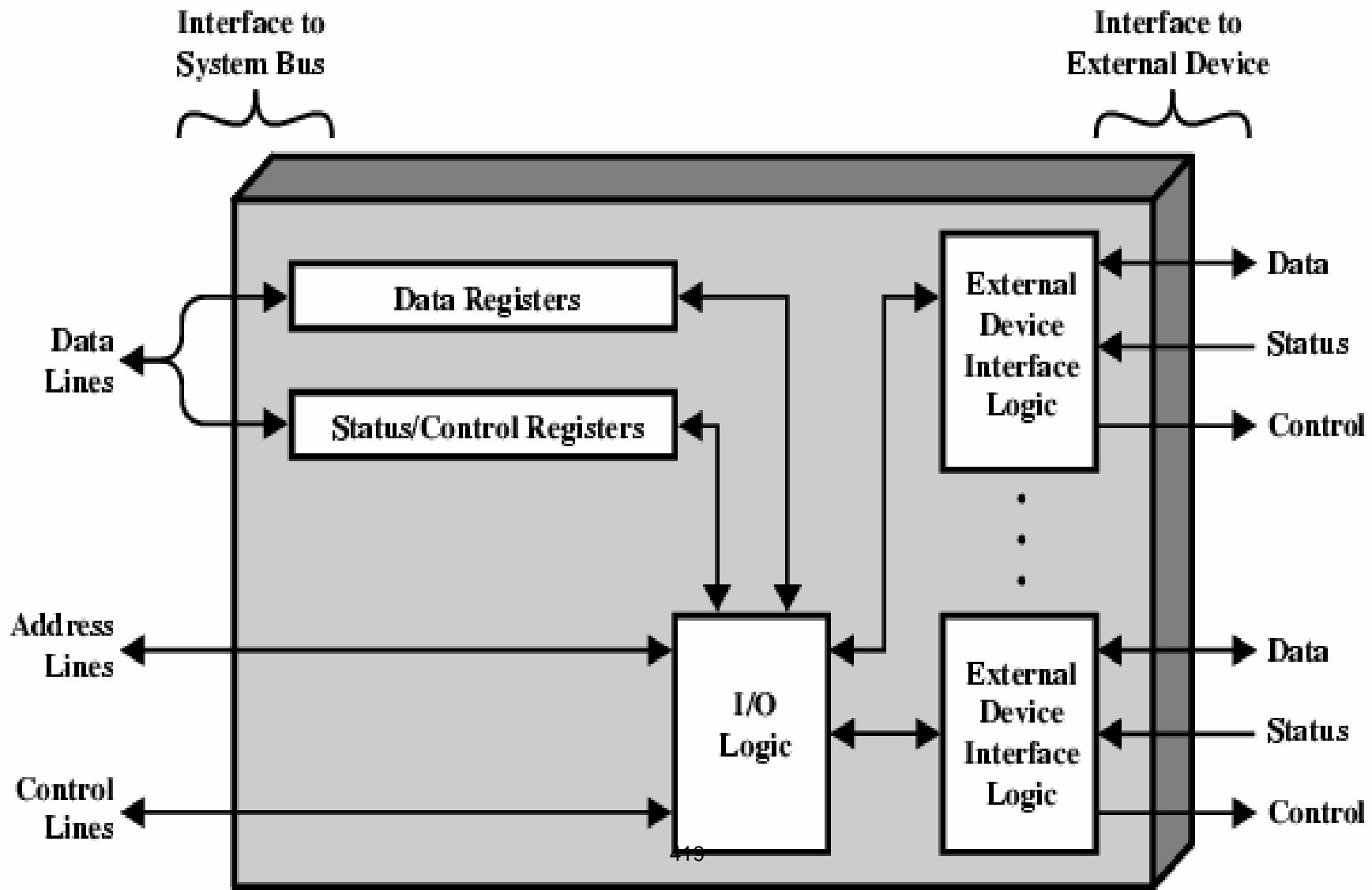
# I/O Steps

- Steps needed to transfer data to or from external device to

CPU:

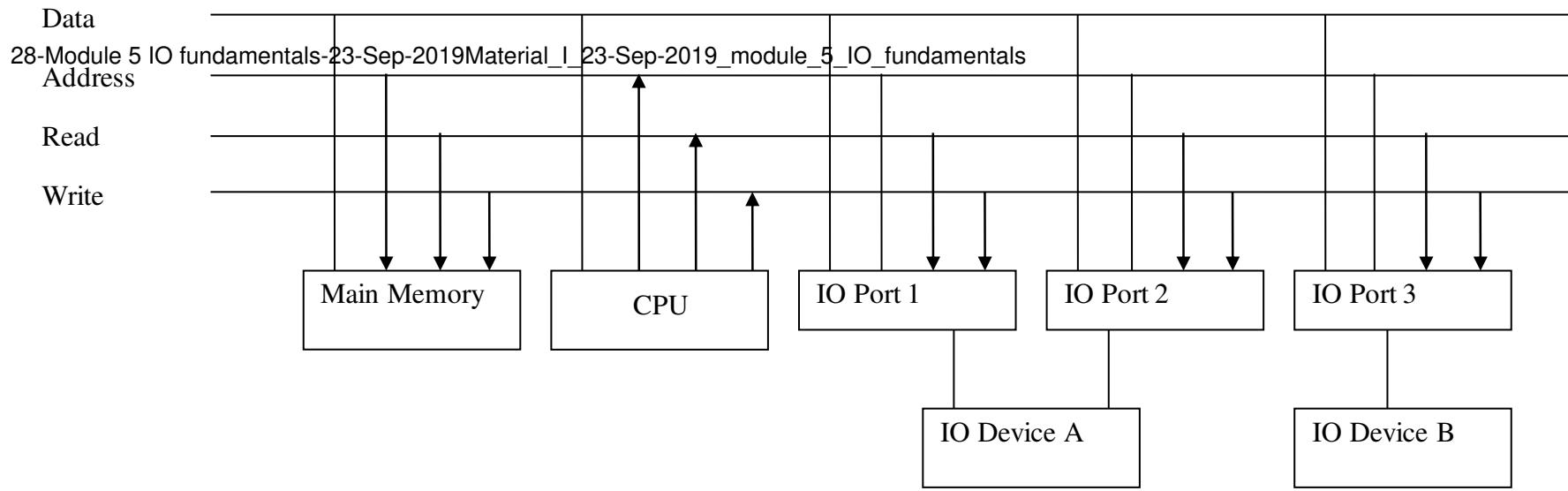
1. CPU checks I/O module device status
2. I/O module returns status
3. If ready, CPU requests data transfer
4. I/O module gets data from device
5. I/O module transfers data to CPU

# I/O Module Diagram

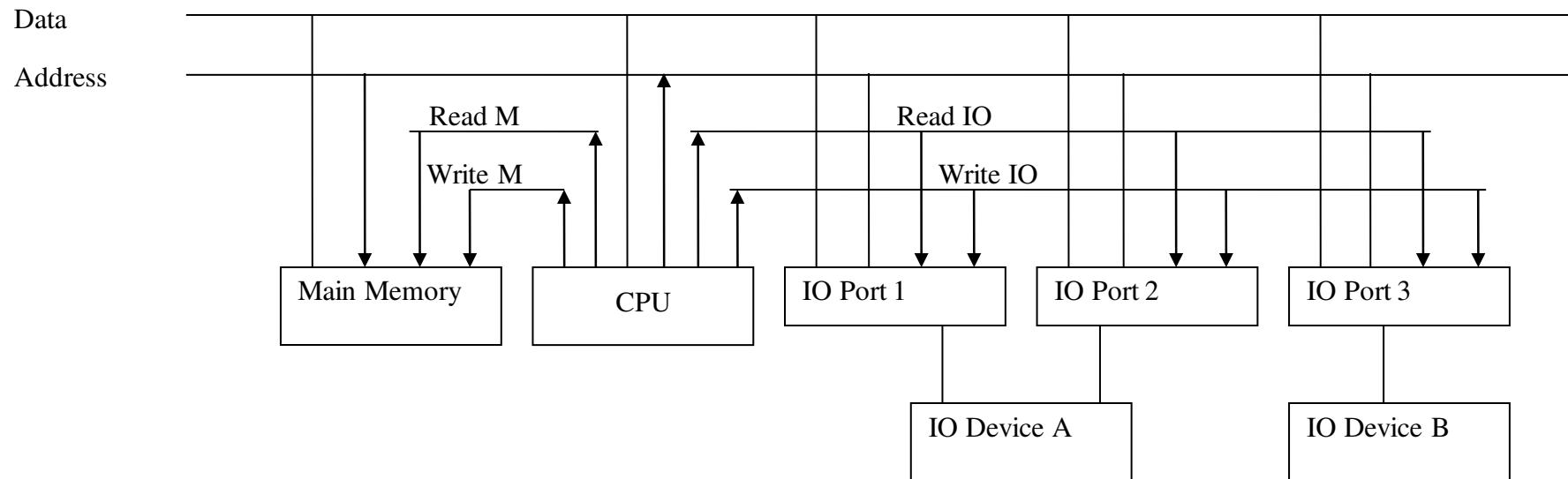


# I/O Bus and Memory Bus

- *MEMORY BUS* is for information transfers between CPU and the Main Memory.
- *I/O BUS* is for information transfers between CPU and I/O devices through their I/O interface.
- Many computers use a **common single bus** system for both memory and I/O interface units
  - Use one common bus but separate control lines for each function
    - Use one common bus with common control lines for both functions



## Memory-Mapped I/O



## I/O Mapped I/O

# Isolated I/O

## Isolated I/O

- Separate I/O read/write control lines in addition to memory read/write control lines
- Separate (isolated) memory and I/O address spaces
- Distinct input and output instructions
- When CPU fetches and decodes the opcode of an I/O instruction, it places the address into the common address lines. Also enables read/write control lines => the address in the address lines is for interface register and not for a memory word.

# Memory Mapped I/O

- A single set of read/write control lines
- Memory and I/O addresses share the common address space .
  - Reduces memory address range available.
- No specific input or output instruction
- The same memory reference instructions can be used for I/O transfers
- When the bus sees certain addresses, it knows they are not memory addresses, but are addresses for accessing I/O devices.

# Asynchronous Data Transfer

**Synchronous** - All devices derive the timing information from common clock line.

**Asynchronous** - No common clock

## Asynchronous Data Transfer:-

Asynchronous data transfer between two independent units requires that *control signals* be transmitted between the communicating units *to indicate the time at which data is being transmitted*

# Asynchronous Data Transfer

Two Asynchronous Data Transfer Methods:

Strobe pulse :-

- A strobe pulse is supplied by one unit to indicate the other unit when the transfer has to occur

Handshaking:-

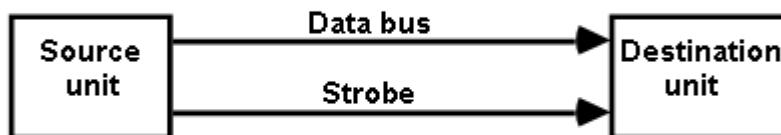
- A control signal is accompanied with each data being transmitted to indicate the presence of data
- The receiving unit responds with another control signal to acknowledge receipt of the data

# Strobe Control

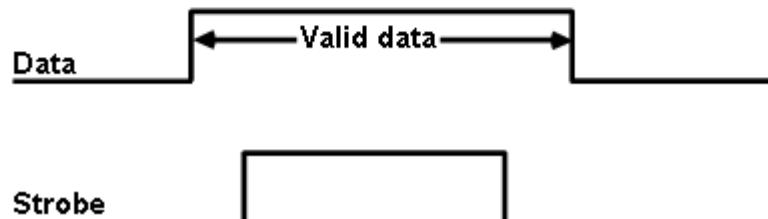
- The strobe may be activated by either the source or the destination unit
- Data bus carries the binary information from source to destination

## Source-Initiated Strobe for Data Transfer

**Block Diagram**

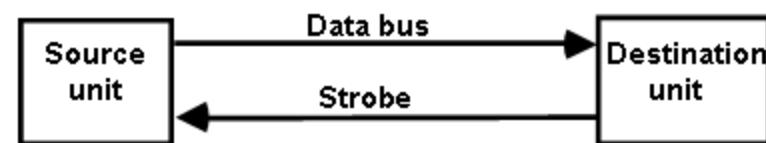


**Timing Diagram**

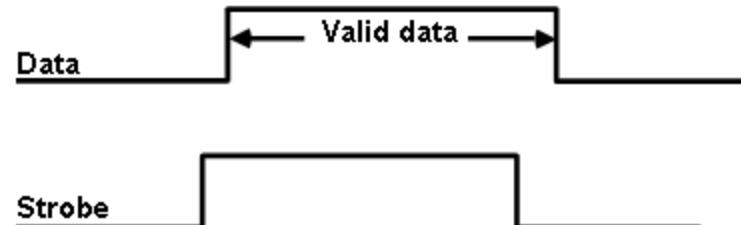


## Destination-Initiated Strobe for Data Transfer

**Block Diagram**



**Timing Diagram**



# Source initiated strobe

- It could be a memory write control signal from the CPU to a memory unit.
- Source is the CPU, places a word on the data bus and informs the memory unit which is destination, that this is a write operation.
- Disadvantage – no way of knowing whether the destination unit has actually received data.

# Destination initiated strobe

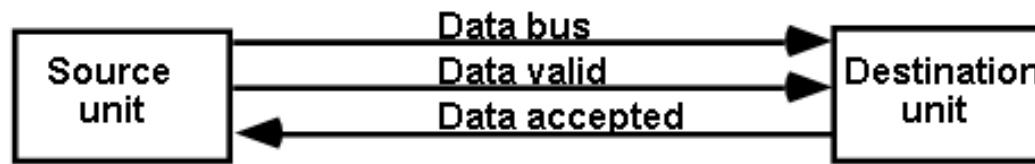
- It could be a memory read control signal.
- CPU, the destination initiates the read operation to inform the memory which is the source to place a selected word into the data bus.
- Disadvantage - no way of knowing whether the source has actually placed the data on the bus

# Hand Shaking

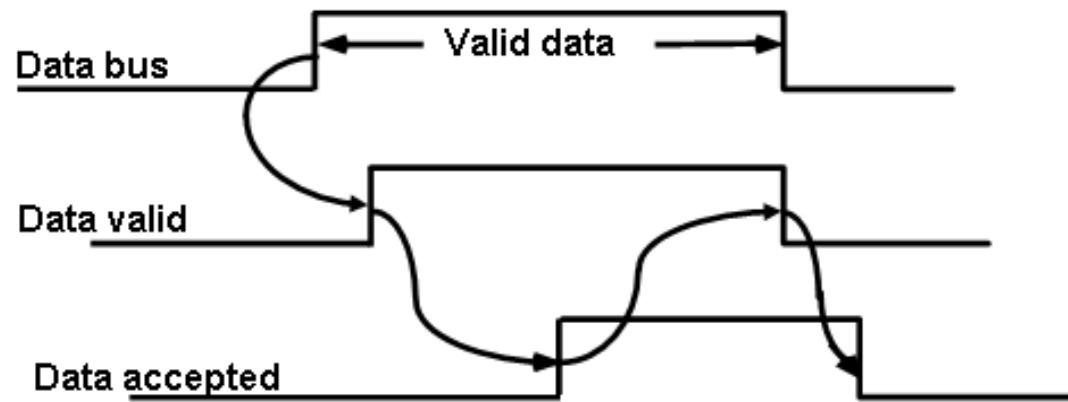
- The handshake method introduces a second control signal to provide a *reply* to the unit that initiates the transfer.
- 2 types
  - Source initiated transfer
  - Destination initiated transfer
- Provides high degree of flexibility and reliability
- Incompletion of data transfer can be detected by means of a **timeout mechanism**
- The timeout signal can be used to interrupt the processor and hence execute a service routine that takes appropriate error recovery action.

# SOURCE-INITIATED TRANSFER USING HANDSHAKE

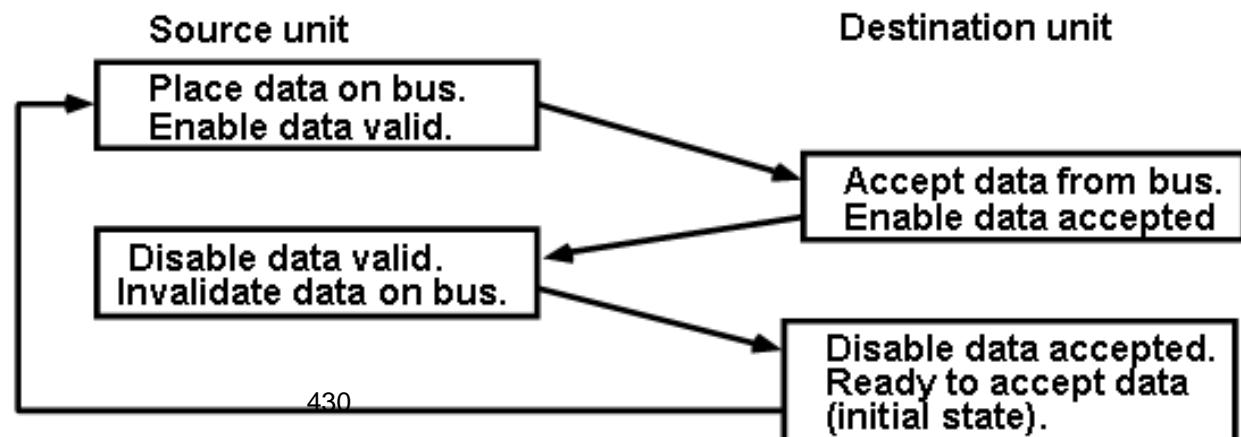
## Block Diagram



## Timing Diagram

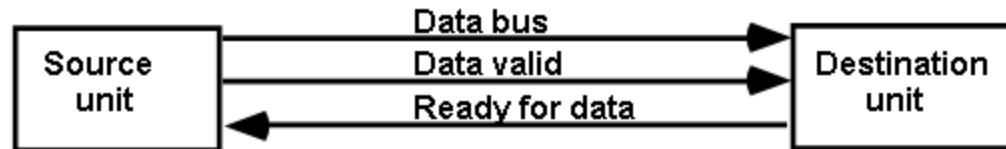


## Sequence of Events

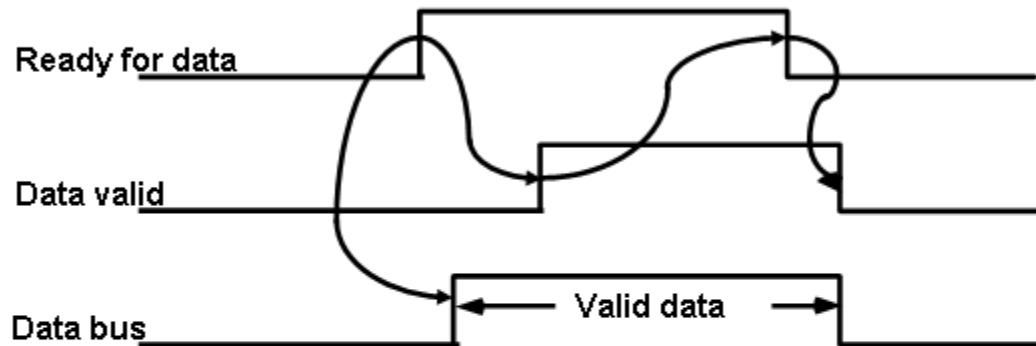


# DESTINATION-INITIATED TRANSFER USING HANDSHAKE

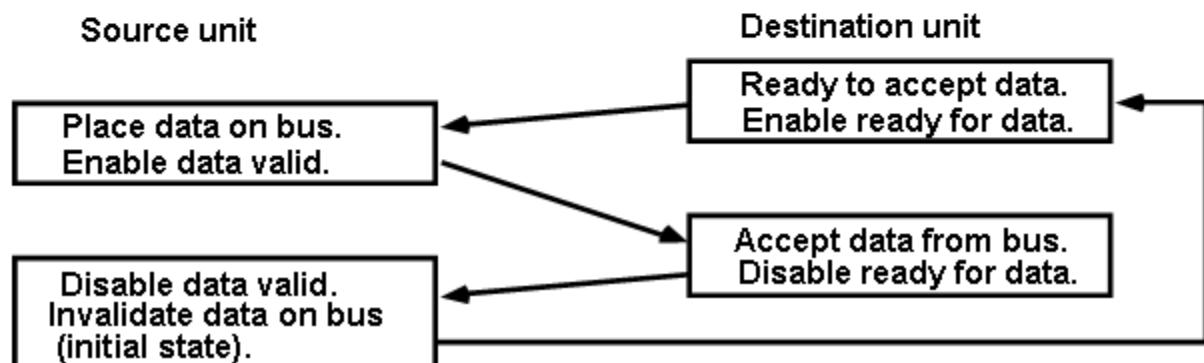
## Block Diagram



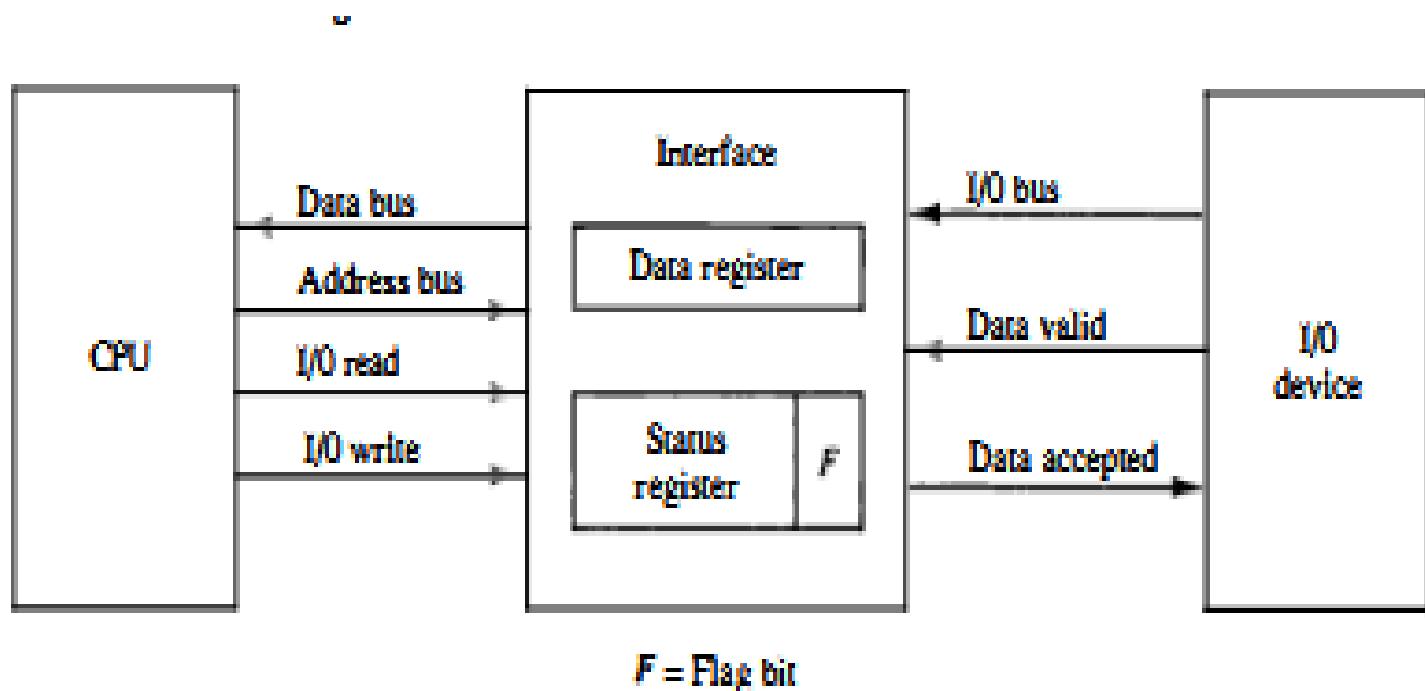
## Timing Diagram



## Sequence of Events



- **Strobe pulse:** Data transfer between CPU and Interface
- **Handshaking signals:-** Data transfer between IO interface and peripheral device.



# References

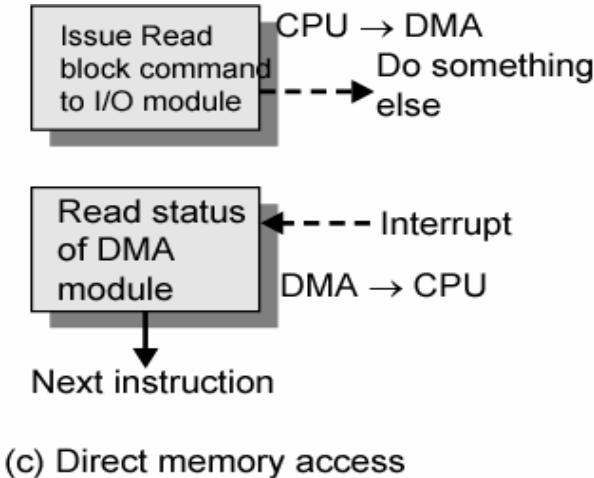
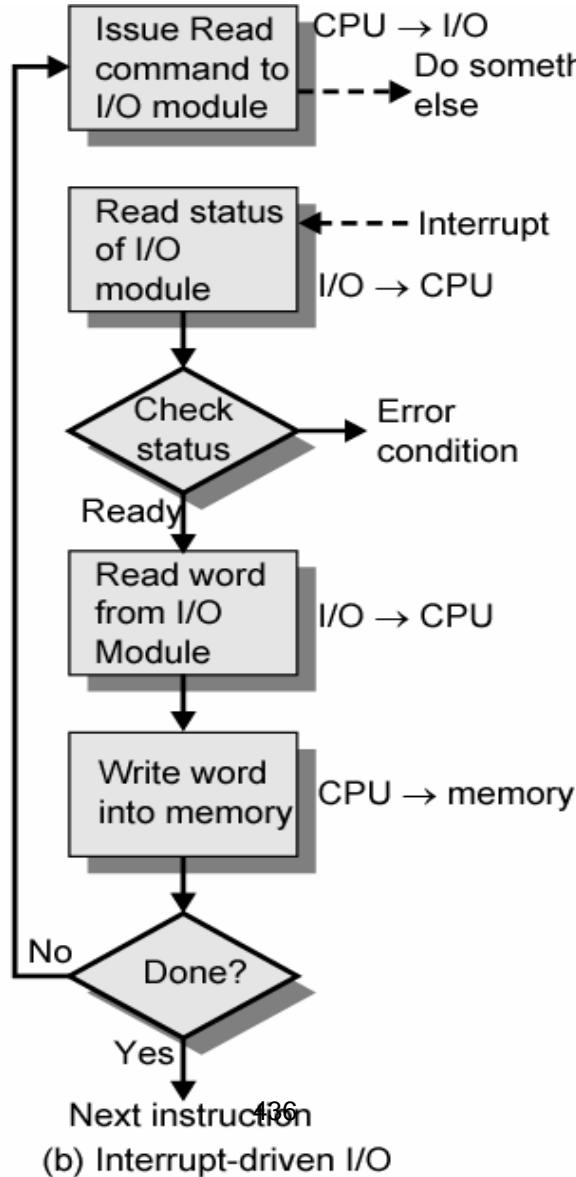
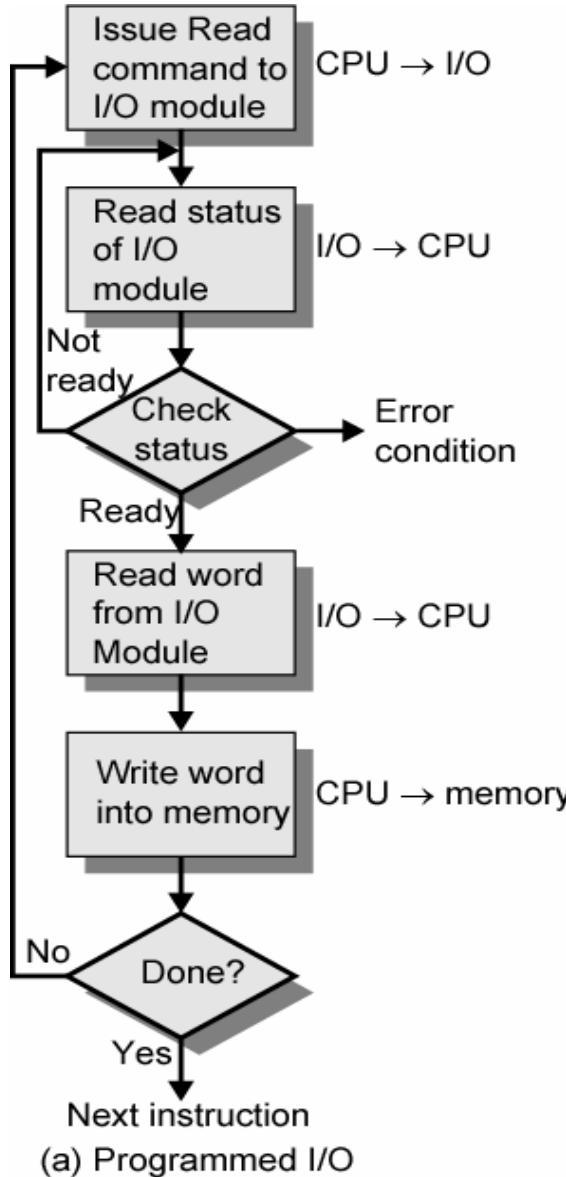
- M. M. Mano, Computer System Architecture, Prentice-Hall

# I/O Techniques

# Input Output Techniques

- Programmed I/O
- Interrupt driven I/O
- Direct Memory Access (DMA)

# Input Output Techniques



# Programmed I/O

- CPU has direct control over I/O
  - Sensing status
  - Read/write commands
  - Transferring data
- CPU waits for I/O module to complete operation
- Wastes CPU time

# Programmed I/O

- CPU requests I/O operation.
- I/O module performs operation and sets status bits after completion.
- CPU checks status bits periodically.
- I/O module does not inform CPU directly that is it does not interrupt CPU.
- CPU must wait.

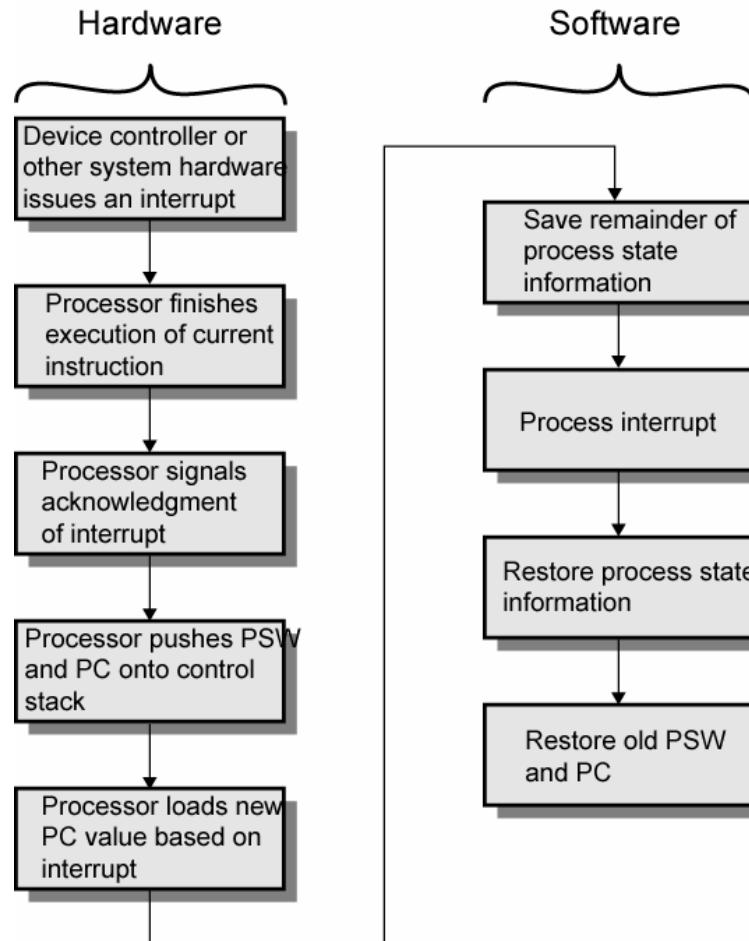
# Interrupt Driven I/O

- Overcomes CPU waiting
- No repeated CPU checking of device
- I/O module interrupts when ready

# Interrupt Driven I/O Basic Operation

- CPU issues read command
- I/O module gets data from peripheral while CPU does other work
- I/O module interrupts CPU after completion
- CPU requests data
- I/O module transfers data

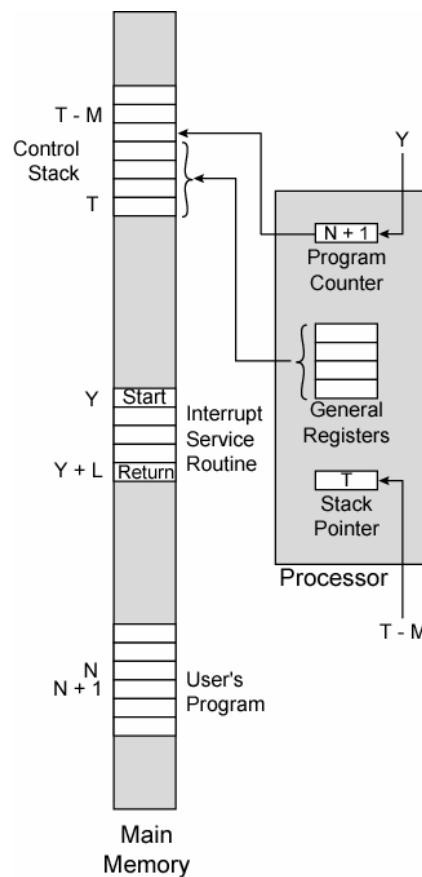
# Simple Interrupt Processing



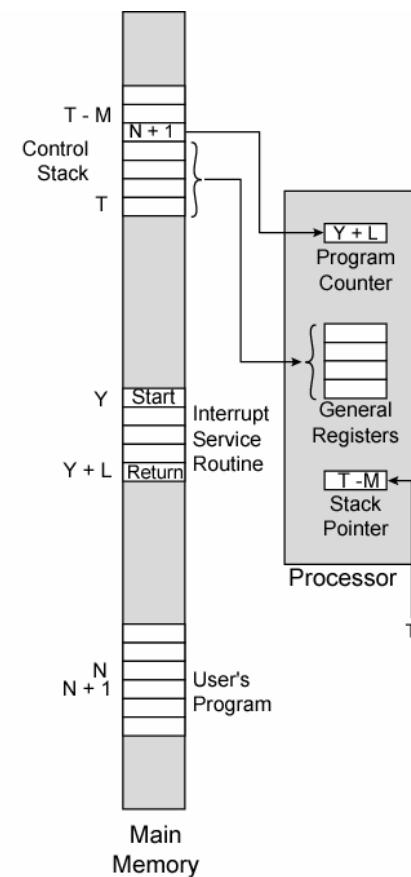
# CPU Viewpoint

- Issue read command
- Do other work
- Check for interrupt at end of each instruction cycle
- If interrupted:-
  - Save context (registers)
  - Process interrupt
- Fetch data & store

# Changes in Memory and Registers for an Interrupt



(a) Interrupt occurs after instruction 443 at location N



(b) Return from interrupt

# DMA

# DMA

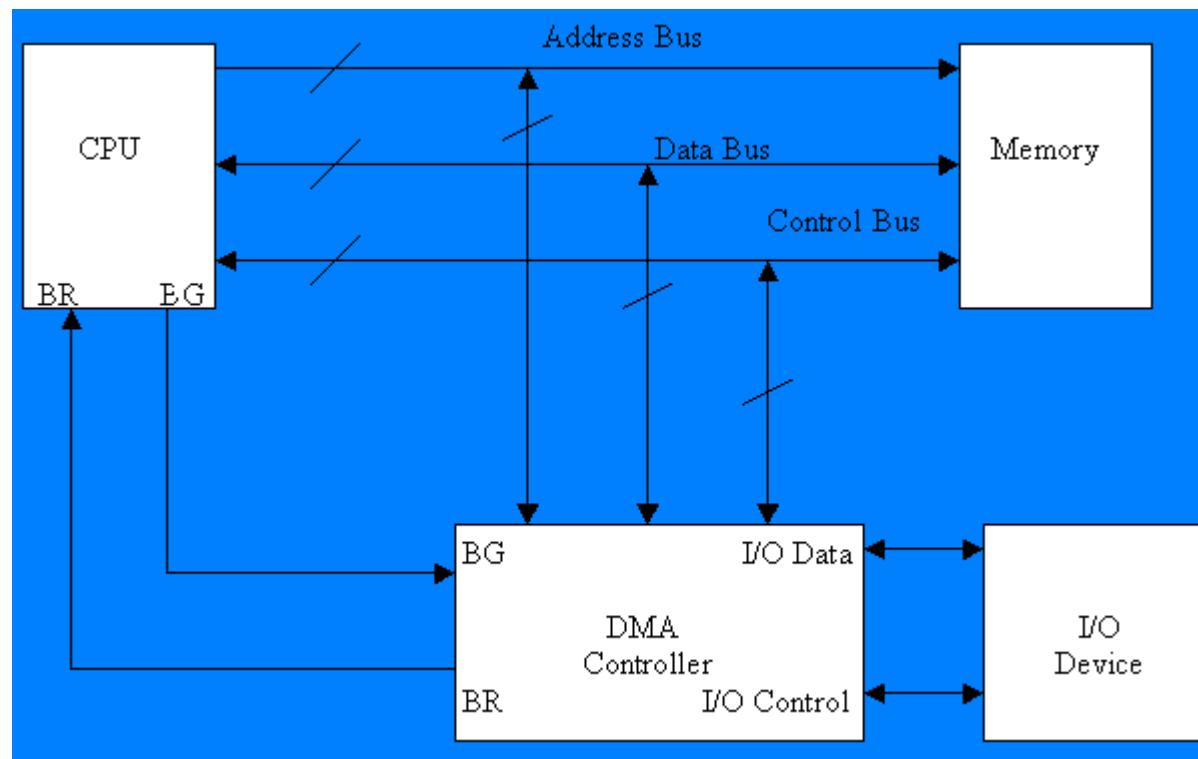
# OVERVIEW

- Introduction
- Implementing DMA in a computer system
- Data transfer using DMA controller
- Internal configuration of a DMA controller
- Process of DMA transfer
- DMA transfer modes

# Direct Memory Access

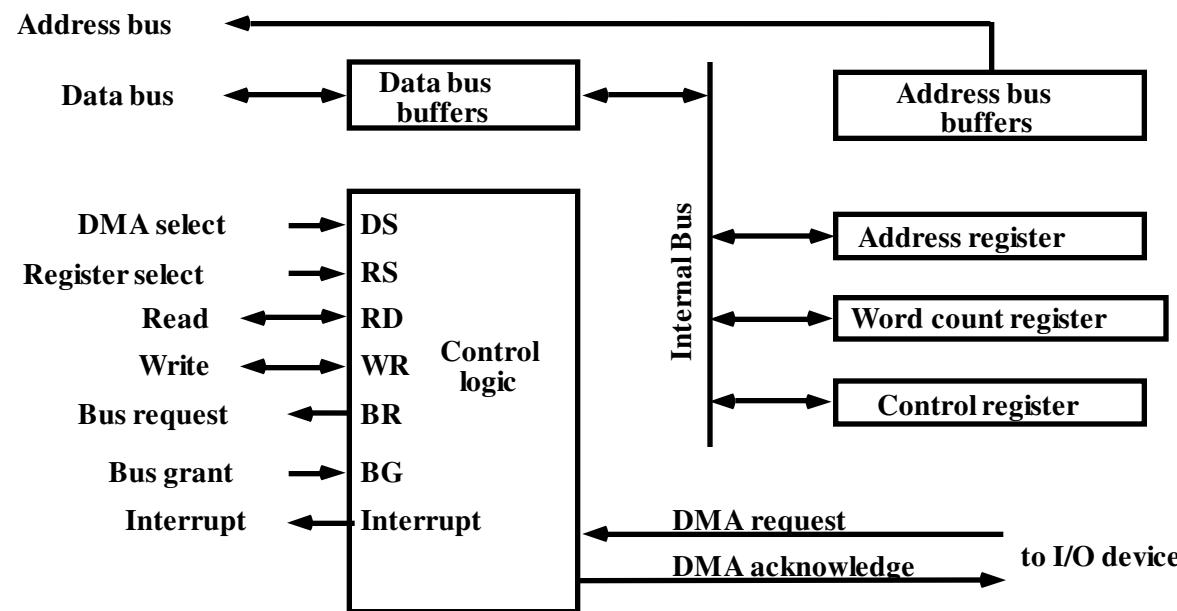
- Introduction
  - An important aspect governing the Computer System performance is the transfer of **data between memory and I/O devices**.
  - The operation involves loading **programs or data files from disk into memory, saving file on disk**, and accessing virtual memory pages on any secondary storage medium.

# Computer System with DMA



# DIRECT MEMORY ACCESS

## Block diagram of DMA controller



- Consider a typical system consisting of a CPU, memory and one or more input/output devices as shown in fig. Assume one of the I/O devices is a disk drive and that the computer must load a program from this drive into memory.
- The CPU would read the first byte of the program and then write that byte to memory. Then it would do the same for the second byte, until it had loaded the entire program into memory.

- This process proves to be **inefficient**. Loading data into, and then writing data out of the CPU significantly **slows down the transfer**. The CPU does not modify the data at all, so it only serves as an additional stop for data on the way to its final destination.
- The process would be much quicker if we could **bypass the CPU** & transfer data directly from the I/O device to memory.
- **Direct Memory Access** does exactly that.

# Implementing DMA in a Computer System

- A **DMA controller** implements direct memory access in a computer system.
- It connects directly to the I/O device at one end and to the system buses at the other end. It also interacts with the CPU, both via the system buses and two new direct connections.
- It is sometimes **referred to as a channel**. In an alternate configuration, the DMA controller may be incorporated directly into the I/O device.

# Data Transfer using DMA Controller

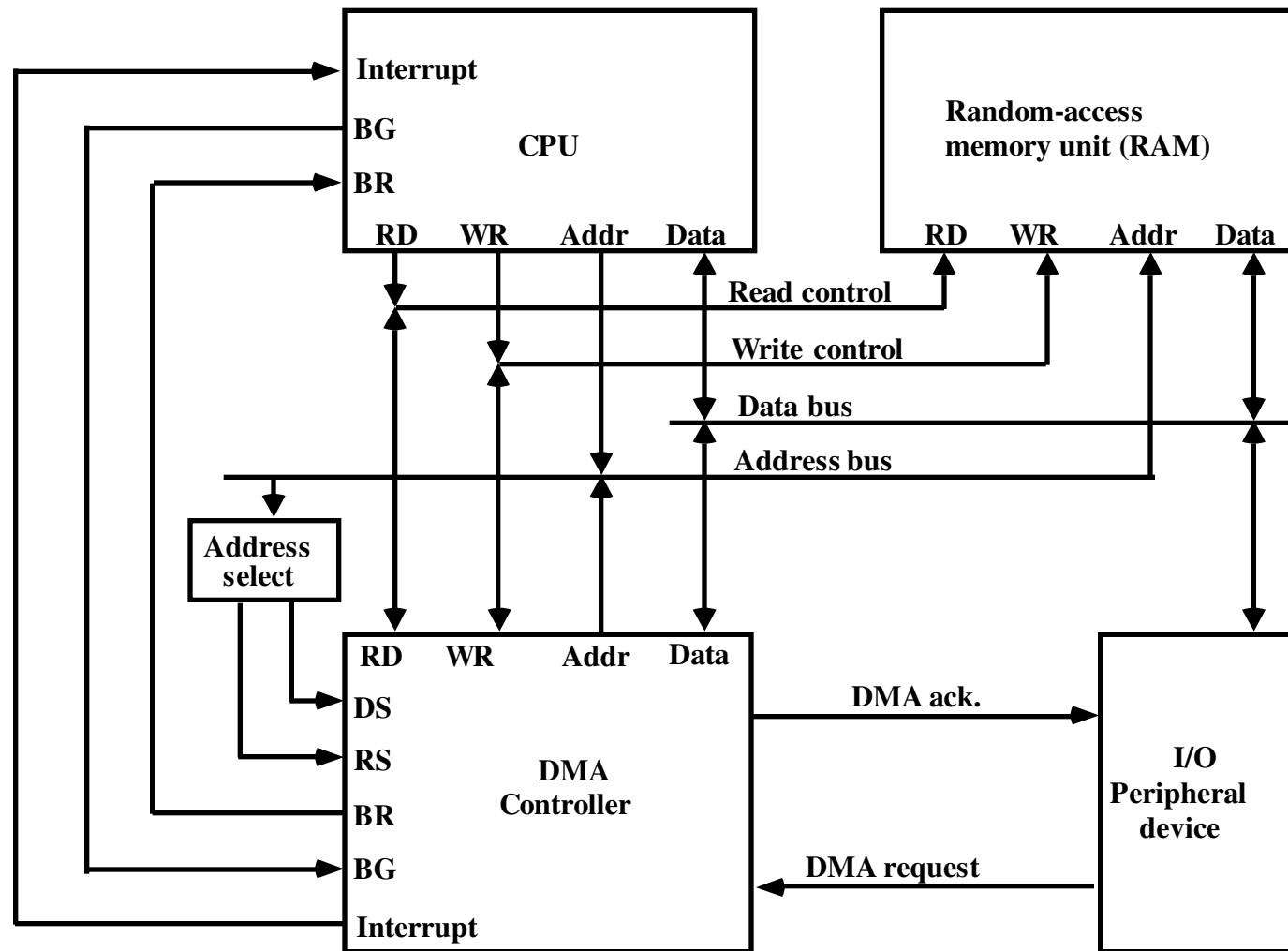
- To transfer data from an I/O device to memory, the DMA controller **first sends a Bus Request** to the CPU by setting BR to 1. When it is ready to grant this request, the CPU sets its **Bus grant signal**, BG to 1.
- The CPU also tri-states its address, data, and control lines thus truly granting control of the system buses to the DMA controller.
- The CPU will continue to tri-state its outputs as long as BR is asserted.

# Internal Configuration

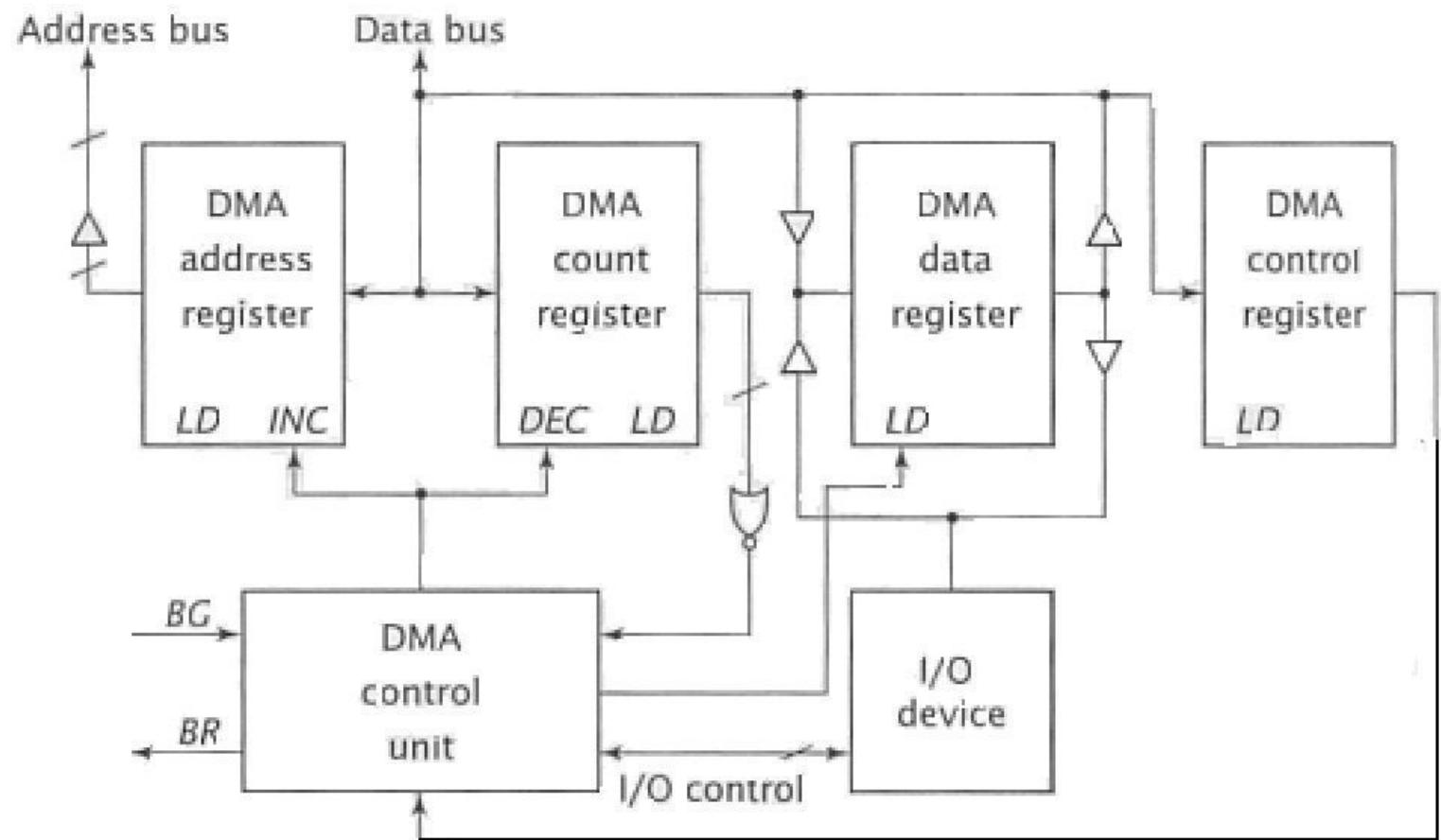
- The DMA controller includes several registers :-
  - The **DMA Address Register** contains the memory address to be used in the data transfer. The CPU treats this signal as one or more output ports.
  - The **DMA Count Register**, also called **Word Count Register**, contains the no. of bytes of data to be transferred. Like the DMA address register, it too is treated as an O/P port (with a diff. Address) by the CPU.
  - The **DMA Control Register** accepts commands from the CPU. It is also treated as an O/P port by the CPU.

- Although not shown in this fig., most DMA controllers also have a Status Register. This register supplies information to the CPU, which accesses it as an I/P port.

# DMA TRANSFER



# Internal Configuration of DMA Controller



## Process of DMA Transfer

- To initiate a DMA transfer, the CPU loads the address of the **first memory location** of the memory block (to be read or written from) into the **DMA address register**. It does this via an I/O output instruction.
- It then writes the **no. of bytes** to be transferred into the **DMA count register** in the same manner.
- Finally, it writes one or more **commands** to the **DMA control register**.

- These commands may specify transfer options such as the DMA transfer mode, but should always specify the direction of the transfer, either from I/O to memory or from memory to I/O.
- The last command causes the DMA controller to initiate the transfer. The controller then sets BR to 1 and, once BG becomes 1 , seizes control of the system buses.

# DMA Transfer Modes

Modes vary by how the DMA controller determines when to transfer data, but the actual data transfer process is the same for all the modes.

- **BURST mode/Block transfer mode/ Idle mode**
  - An entire block of data is transferred in one contiguous sequence. Once the DMA controller is granted access to the system buses by the CPU, it transfers all bytes of data in the data block before releasing control of the system buses back to the CPU.
  - This mode is useful for loading programs or data files into memory, but it does render the CPU inactive for relatively long periods of time.

- **Cycle stealing Mode/Single byte transfer mode**
  - Viable alternative for systems in which the CPU should not be disabled for the length of time needed for Burst transfer modes.
  - DMA controller obtains access to the system buses as in burst mode, using BR & BG signals. However, it transfers one byte of data and then returning control of the system buses to the CPU by BR=0.
  - DMAC and CPU are constantly stealing the bus cycles from each other.
  - It continually issues requests via BR, transferring one byte of data per request, until it has transferred its entire block of data.

- By continually obtaining and releasing control of the system buses, the DMA controller essentially interleaves instruction & data transfers. The CPU processes an instruction, then the DMA controller transfers a data value, and so on.
- The data block is not transferred as quickly as in burst mode, but the CPU is not idled for as long as in that mode.
- Useful for controllers monitoring data in real time.

- **Transparent Mode:**

- This requires the most time to transfer a block of data, yet it is also the most efficient in terms of overall system performance.
- The DMA controller only transfers data when the CPU is performing operations that do not use the system buses.
- For example, the Relatively simple CPU has several states that move or process data solely within the CPU:

NOP

MOV A, B

ADD B,C

(No operation)

data transfer between CPU registers

arithmetic operation, both the operands are  
CPU registers

- Primary advantage is that CPU never stops executing its programs and DMA transfer is free in terms of time.
- Disadvantage is that the hardware needed to determine when the CPU is not using the system buses can be quite complex and relatively expensive.

- Demand transfer mode:

- Data transfer occurred only on the demand of peripheral device.
- It is similar to the Block transfer, except that the DMAR must remain active throughout the DMA operation.
- If during the operation DMAR goes low, the DMA operation will be stopped and bus control will be back with CPU.
- Once DMAR goes high again, the DMA operation continues from where it had stopped.
- When  $dreq=1$ , data transfer will be performed
- When  $dreq=0$ , CPU will be the bus master.

# Summary

- Advantages of DMA
  - Computer system performance is improved by direct transfer of data between memory and I/O devices, bypassing the CPU.
  - CPU is free to perform operations that do not use system buses.
- Disadvantages of DMA
  - In case of Burst Mode data transfer, the CPU is rendered inactive for relatively long periods of time.



# VECTORED AND PRIORITY INTERRUPTS

# Interrupt

- An interrupt is a signal from a device attached to a computer or from a program within the computer that causes the CPU to stop its normal program execution and perform service related to the event.
- **Maskable Interrupt:** It is a hardware interrupt that may be ignored by set/reset a bit in an interrupt mask register's (IMR) bit-mask.
- **Non-maskable Interrupt:** is a hardware interrupt that does not have a bit-mask associated with it - meaning that it can never be ignored. NMIs are often used for timers.

# Interrupt Cycle

The Interrupt enable flip-flop (IEN) can be set or cleared by program instructions.

A programmer can therefore allow interrupts (clear IEN) or disallow interrupts (set IEN)

At the end of each instruction cycle the CPU checks IEN and IST. If either is equal to zero, control continues with the next instruction. If both = 1, the interrupt is handled.

Interrupt micro-operations:

$SP \leftarrow SP - 1$       (Decrement stack pointer)

$M[SP] \leftarrow PC$  Push PC onto stack

$INTACK \leftarrow 1$       Enable interrupt acknowledge

$PC \leftarrow VAD$       Transfer vector address to PC

$IEN \leftarrow 0$       Disable further interrupts

Go to fetch next instruction

# Interrupt Service Routine

An interrupt handler, also known as an interrupt service routine (ISR), is a callback subroutine in an operating system whose execution is triggered by the reception of an interrupt. Interrupt handlers have a multitude of functions, which vary based on the reason the interrupt was generated.

## Interrupt Overhead

The interrupt overhead is caused by **context switching** (storing and restoring the state of CPU )

On interrupt handler entry, the context of the **current process and its thread must be saved**. On exit, it must be restored.

On handler entry, memory locations different from the memory locations in the cache are used, and therefore **cache updates are required**.

# Vectored and Non Vectored Interrupts

- Non-vectored interrupt
  - The device has to supply the address of the subroutine to the processor
  - Ex: Call instruction, INTR
- Vectored interrupt
  - The address of the subroutine is already known to the processor
  - RST 5.5 – when this interrupt is received, the processor saves the content of PC into stack and branches to 002CH address.

# Priority Interrupt

- Establishes priority over the various sources to determine which condition is to be serviced first when two or more requests arrive simultaneously.
- The system may also determine which conditions are permitted to interrupt the computer while another interrupt is being serviced.
- Priority can be established by software or hardware

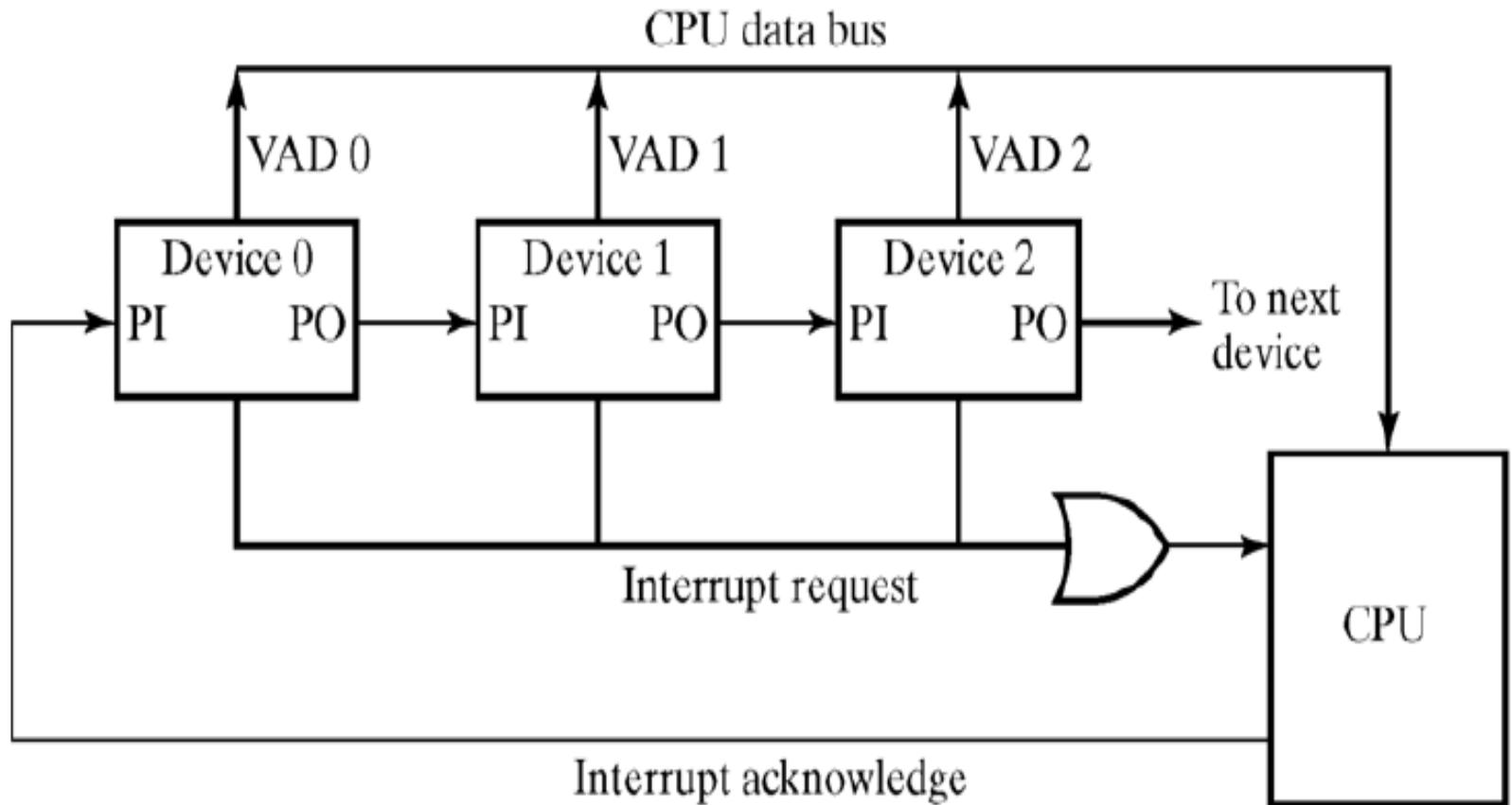
# Polling-S/W

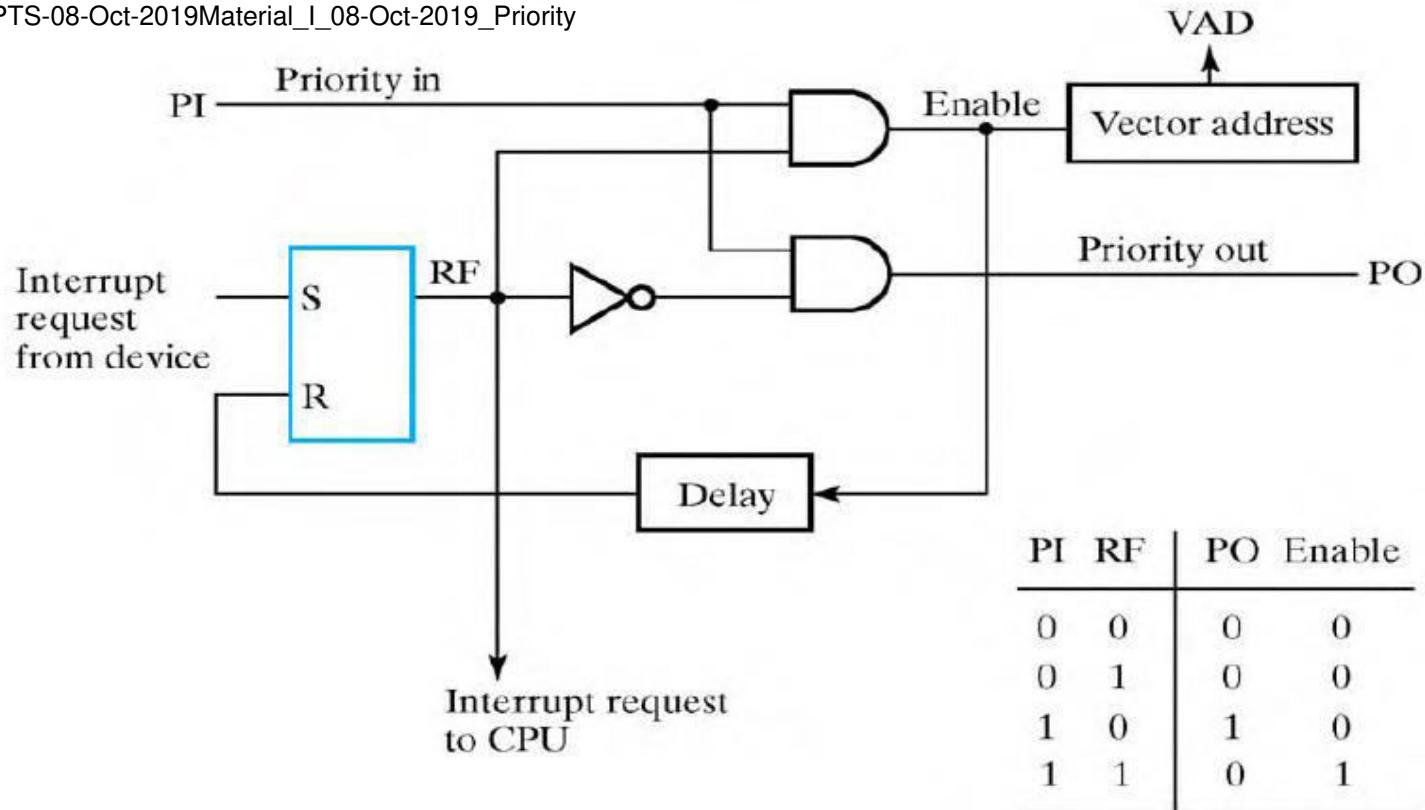
- Software Priority – **polling** – common branch address for all interrupts. Program to perform the test will be stored in this **common branch address**.
- Highest priority source is tested first and if its interrupt signal is on, control branches to a service routine for this source.
- Otherwise, the next-lower priority source is tested and so on.
- **Disadvantage – if many interrupts**, time required to poll them may exceed the time available to service the I/O device.

# Vectored Interrupts-H/W

- Then, hardware priority interrupt unit can be used to speed up the operation.
- Hardware priority interrupt handler – accepts interrupt request from many sources, determines which of the incoming requests has the highest priority and issues an interrupt request to the computer.
- Here, each interrupt source has its own interrupt vector to access its own service routine directly.
- Hardware priority interrupts – Serial or Parallel connection of interrupt lines.

# Serial - Daisy chaining





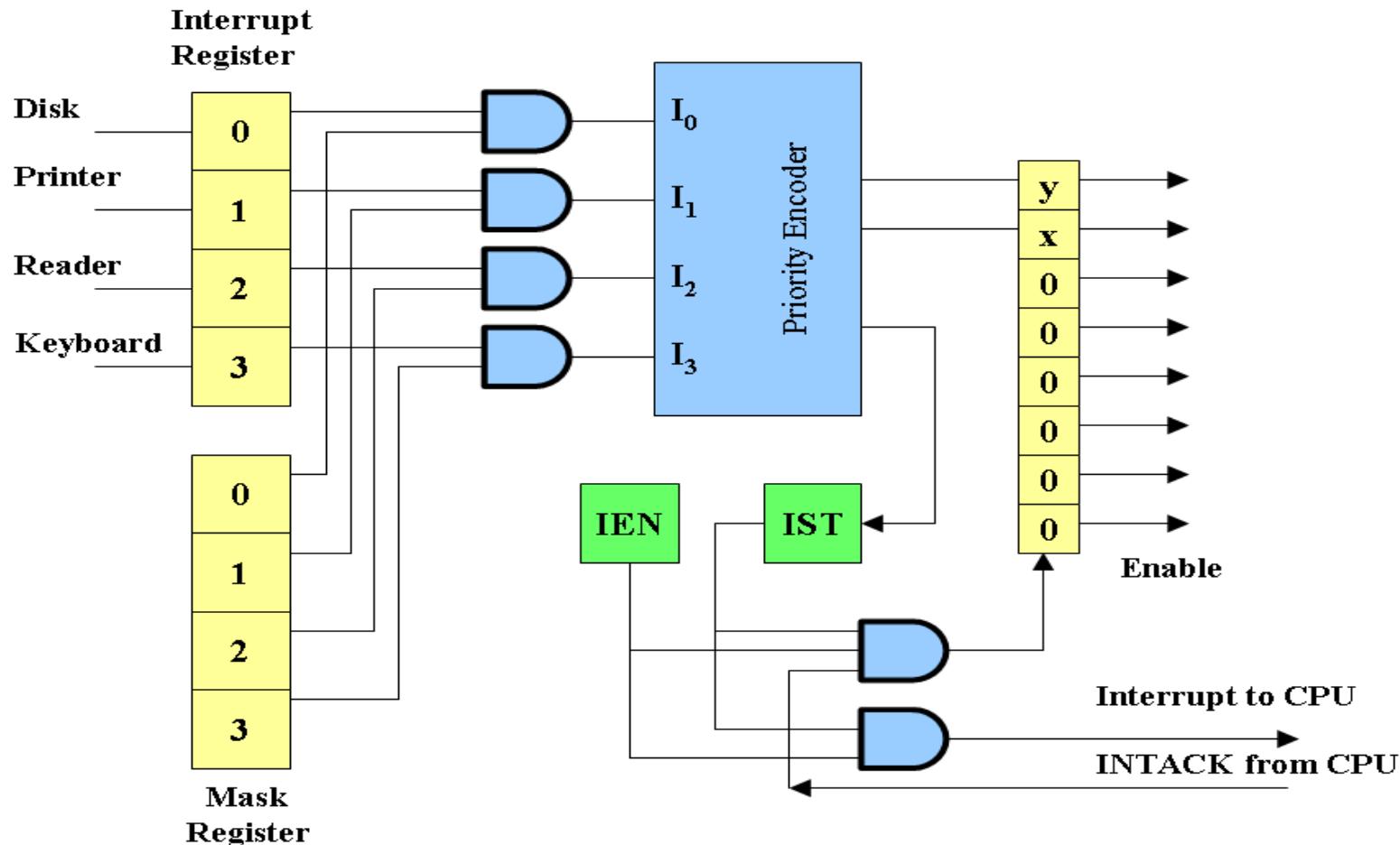
One stage of daisy chain priority interrupt scheme

If PI = 0, both PO and the enable line to VAD (vector address) are equal to 0, irrespective of RF.

- If  $PI = 1$  and  $RF = 0$ , then  $PO = 1$  and the vector address is disabled  $\Rightarrow$  passes the acknowledge signal to the next device through  $PO$ .
- The device is active when  $PI = 1$  and  $RF = 1$ . This condition places a 0 in  $PO$  and enables the vector address for the data bus.
- The RF flip-flop is reset after a sufficient delay to ensure that the CPU has received the vector address.

# Parallel Priority Interrupt

- Uses a **register** whose bits are set separately by the interrupt signal from each device.
- Priority is established according to the **position of the bits in the register**.
- Mask register is used to disable lower priority interrupts while a higher priority device is being serviced.
- It can also provide a facility that allows a high-priority device to interrupt the CPU while a lower priority device is being serviced



- **Mask register** has a same number of bits as the interrupt register.
- By means of program instructions, it is possible to set or reset any bit in the mask register.

- The priority encoder generates two bits of the vector address, which is transferred to the CPU.
- Another output from the encoder sets an interrupt status flip-flop IST when an interrupt that is not masked occurs.
- The interrupt enable flip-flop IEN can be set or cleared by the program to provide an overall control over the interrupt system.
- IST ANDed with IEN provide a common interrupt signal for the CPU.
- The INTACK signal from the CPU enables the bus buffers in the output register and a vector address VAD is placed into the data bus.

# Priority encoder

Circuit that implements the priority function.

Logic – if two or more inputs arrive at the same time, the input having the highest priority will take precedence.

Boolean functions

$$X = I'_0 I'_1$$

$$Y = I'_0 I_1 + I'_0 I'_2$$

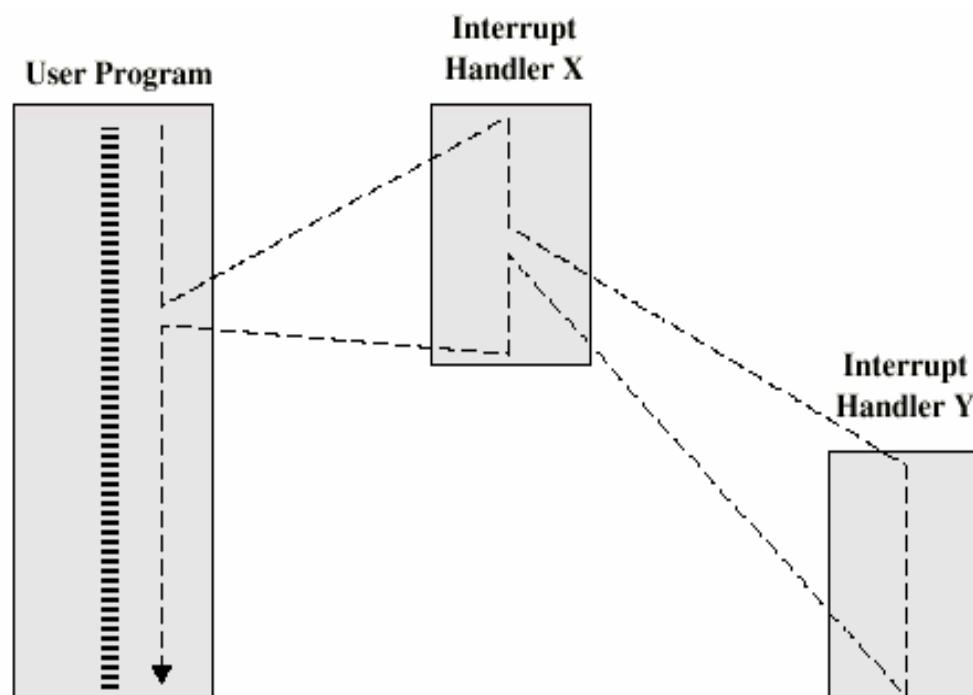
$$IST = I_0 + I_1 + I_2 + I_3$$

<b>Inputs</b>			
<b>I<sub>0</sub></b>	<b>I<sub>1</sub></b>	<b>I<sub>2</sub></b>	<b>I<sub>3</sub></b>
1	d	d	d
0	1	d	d
0	0	1	d
0	0	0	1
0	0	0	0

<b>Outputs</b>		
<b>d</b>	<b>Y</b>	<b>IST</b>
0	0	1
0	1	1
1	0	1
1	1	1
d	d	0

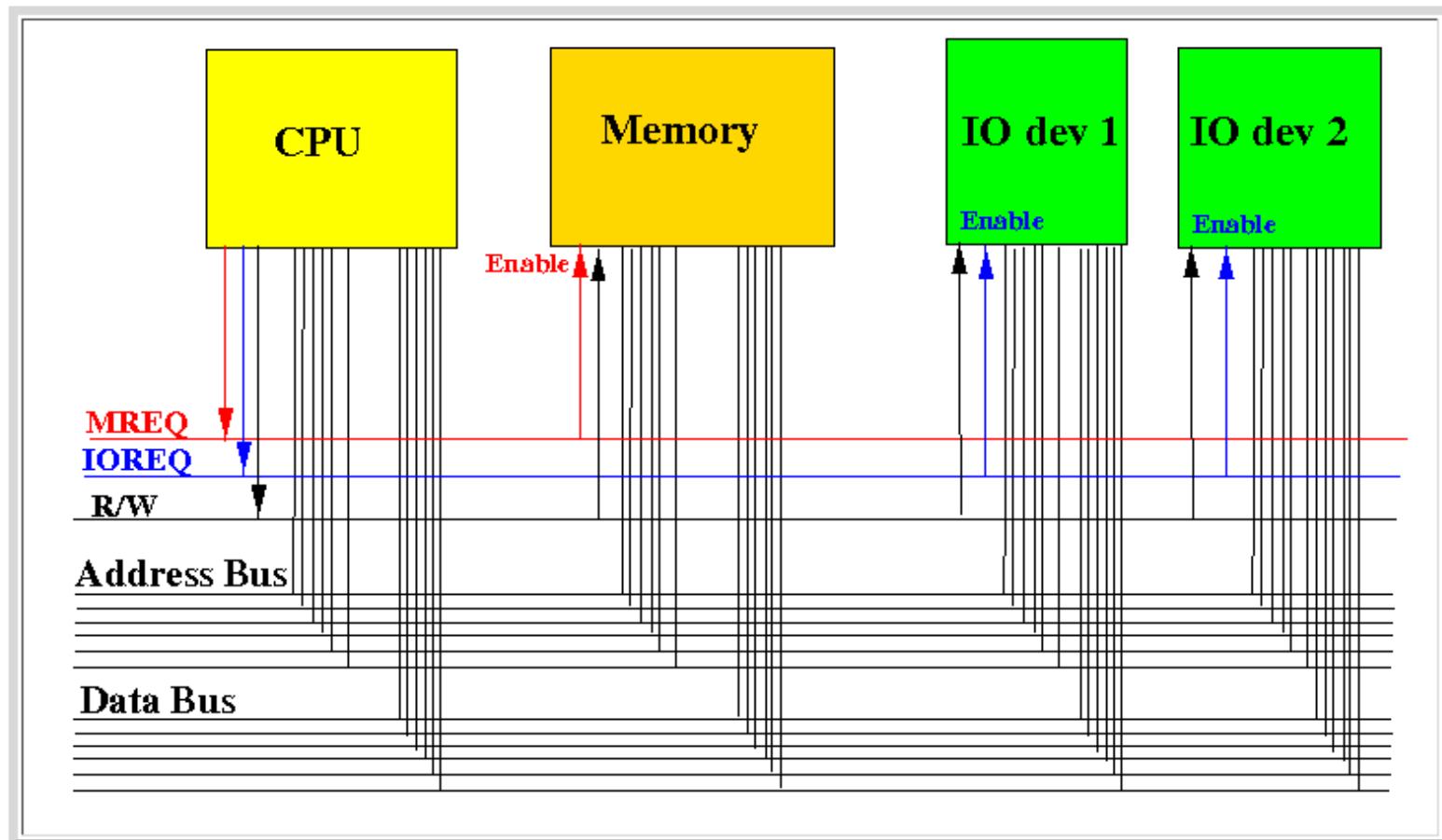
# Interrupt nesting

- An interrupt can happen while executing an ISR. This is called *interrupt nesting*.



# Bus System

- How devices are connected inside a compute:



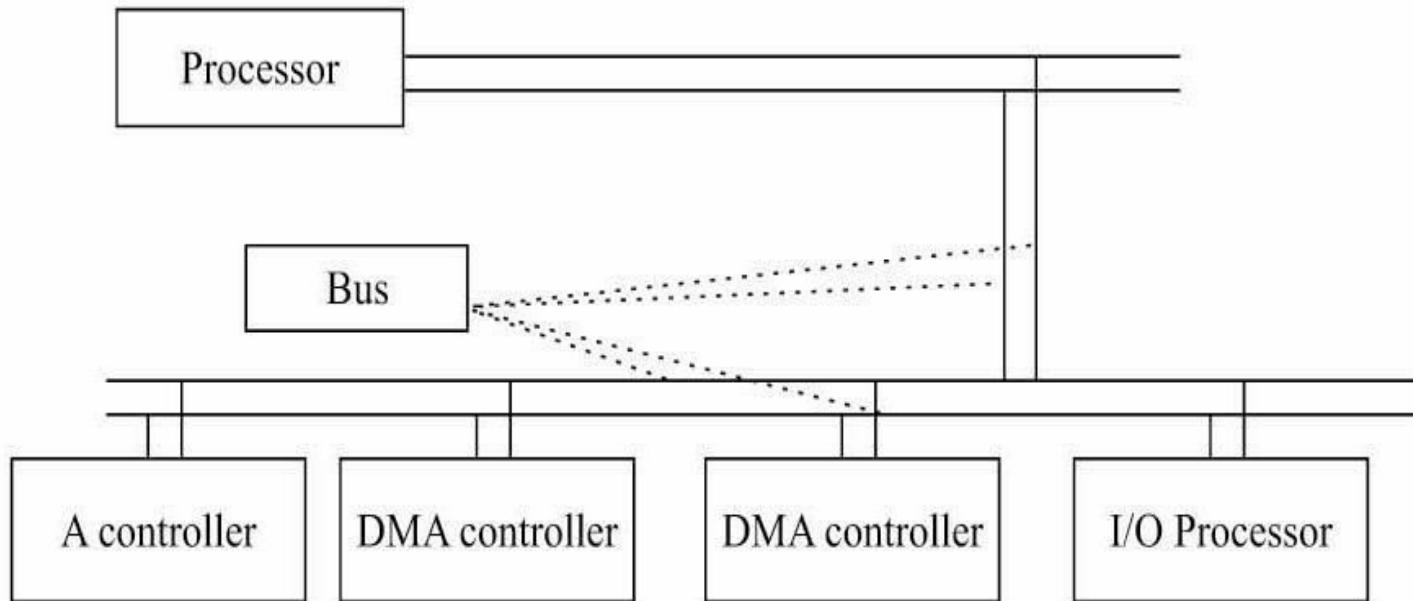
# Synchronous Bus

- Synchronous bus (e.g., processor-memory buses)
  - Includes a clock in the control lines and has a fixed protocol for communication that is **relative** to the clock
  - Advantage: involves very little logic and can run very fast
  - Disadvantages:
    - Every device communicating on the bus must use same clock rate

# Asynchronous Bus

- Asynchronous bus (e.g., I/O buses)
  - It is not clocked, so requires a handshaking protocol and additional control lines (ReadReq, Ack, DataRdy)
  - Advantages:
    - Can accommodate a wide range of devices and device speeds
  - Disadvantage: slow(er)

# Bus Arbitration



A bus (any bus) **cannot** be used by **more than one device** at one time due to electrical properties of the devices.

# Bus Arbitration

- Before any device is allowed to perform a read/write operation using the system bus, it must first **obtain permission**.
- A device that **starts a read/write operation** is called a **master device**
- The device that it "talks" to is called the **slave device**

# Bus Arbitration

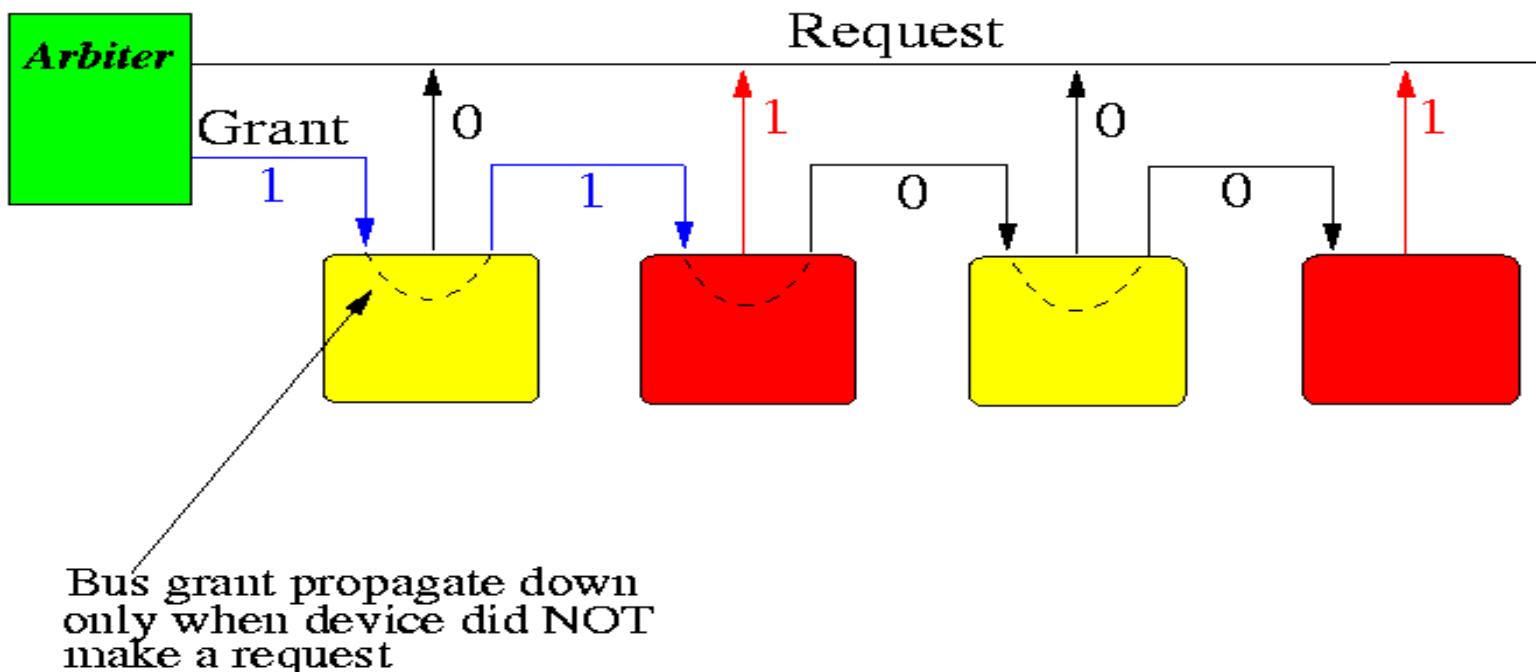
- Only one processor or controller can be **bus master**
- The bus master– the controller that has access to a bus at an instance.
- Any one controller or processor can be the bus master at the given instance (s).

# 3 BUS ARBITRATION METHODS

1. Daisy Chain
2. Independent Bus Requests and Grant
3. Polling

# Daisy Chained

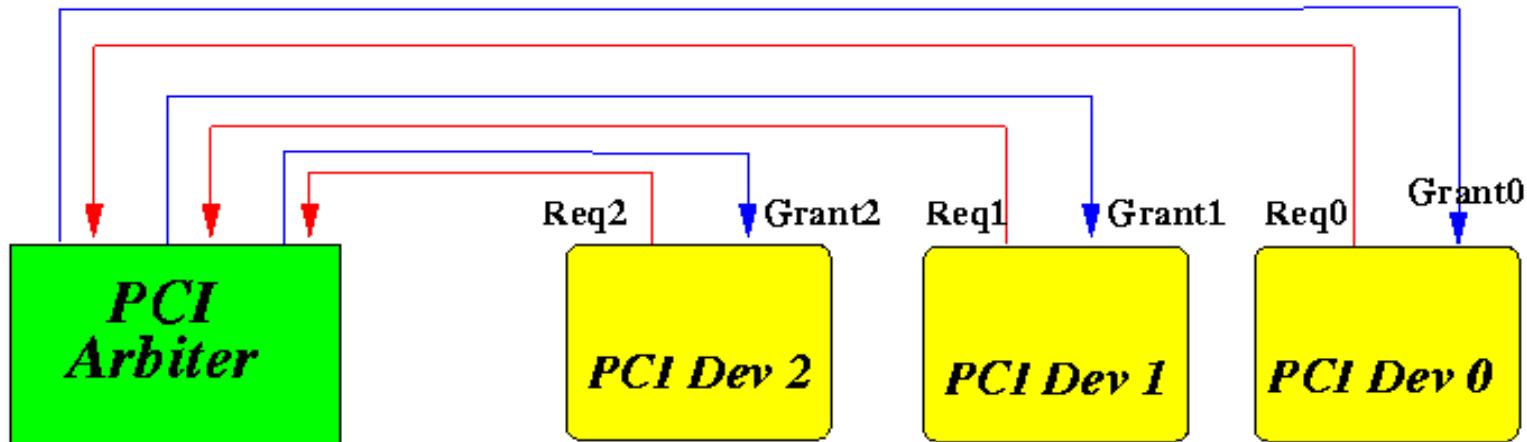
- Daisy-chained (cheap - requires no special hardware)



- The daisy chain consists of wires connecting the devices in a pre-defined order
- This ordering defines the "pecking order" of the devices.

# Independent Bus Requests and Grant

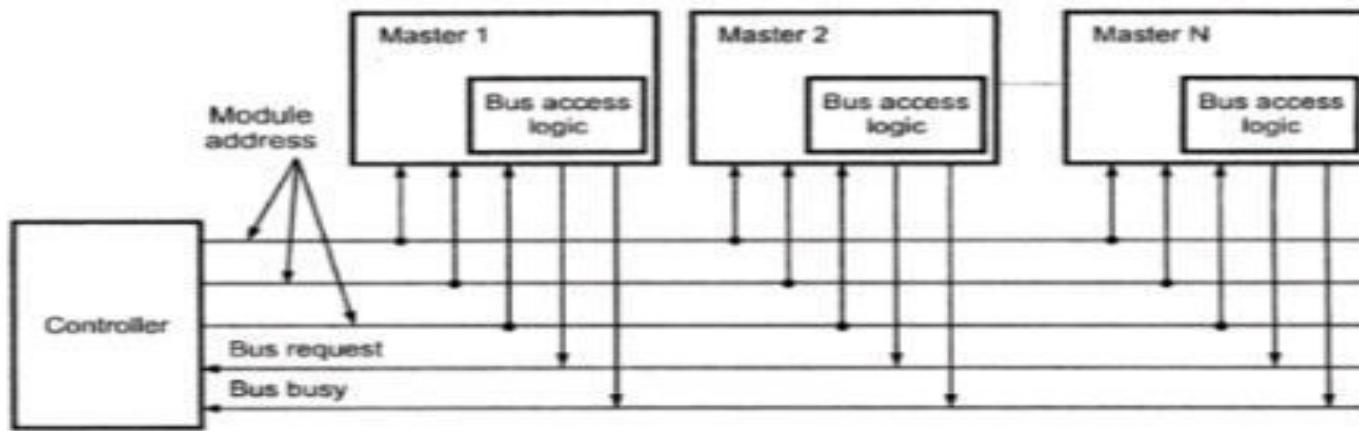
## Centralized Arbiter



- The logic of the bus arbiter is simple and depends on how the priorities are assigned.

Req2	Req1	Req0		Grant2	Grant2	Grant0
0	0	0		0	0	0
0	0	1		0	0	1
0	1	0		0	1	0
0	1	1		0	1	0
1	0	0		1	0	0
1	0	1		1	0	0
1	1	0		1	0	0
1	1	1		1	492	0

# Polling



- In this the controller is used to generate the addresses for the master. Number of address line required depends on the number of master connected in the system.
- In response to the bus request controller generates a sequence of master address. When the requesting master recognizes its address, it activates the busy line and begins to use the bus.

**Daisy chaining**, the centralized controller always sending bus control to highest priority, which passes it to next if bus access not required.

**Independent request method**, the centralized controller listens to requests of each device individually and grant access to the bus on resolving its priority.

**Polling methods**, the centralized controller does the polling of the devices and grant access to that bus which requests it on receiving the poll address.

# References

- M. M. Mano, Computer System Architecture, Prentice-Hall

**Program: B.Tech (CSE)**

**Course: Computer Architecture and  
Organization**

**Course Code: CSE 205**

## **Unit-IV-Lecture 7-**

# **Buses: bus protocols, local and geographic arbitration**

# Buses

- There are a number of possible interconnection systems
- Single and multiple BUS structures are most common
- e.g. Control/Address/Data bus (PC)
- e.g. Unibus (DEC-PDP)

# What is a Bus?

- A communication pathway connecting two or more devices
- Usually broadcast
- Often grouped
  - A number of channels in one bus
  - e.g. 32 bit data bus is 32 separate single bit channels
- Power lines may not be shown

# Data Bus

- Carries data
  - Remember that there is no difference between “data” and “instruction” at this level
- Width is a key determinant of performance
  - 8, 16, 32, 64 bit

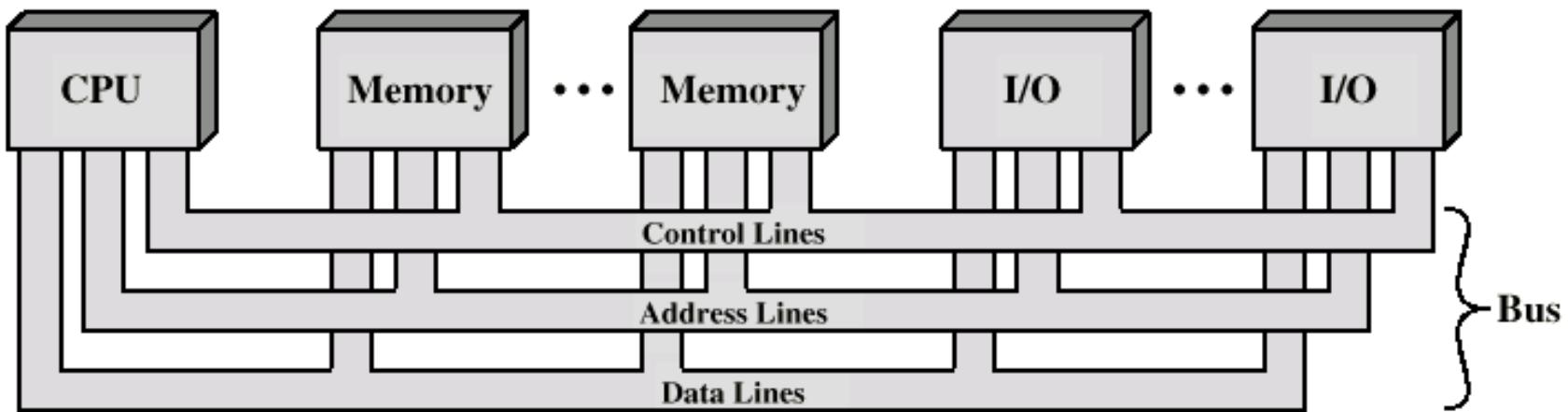
# Address bus

- Identify the source or destination of data
- e.g. CPU needs to read an instruction (data) from a given location in memory
- Bus width determines maximum memory capacity of system
  - e.g. 8080 has 16 bit address bus giving 64k address space

# Control Bus

- Control and timing information
  - Memory read/write signal
  - Interrupt request
  - Clock signals

# Bus Interconnection Scheme



# Bus Types

- Dedicated
  - Separate data & address lines
- Multiplexed
  - Shared lines
  - Address valid or data valid control line
  - Advantage - fewer lines
  - Disadvantages
    - More complex control
    - Ultimate performance

# Bus Arbitration

- More than one module controlling the bus
- e.g. CPU and DMA controller
- Only one module may control bus at one time
- Arbitration may be centralised or distributed

# Centralised or Distributed Arbitration

- Centralised
  - Single hardware device controlling bus access
    - Bus Controller
    - Arbiter
  - May be part of CPU or separate
- Distributed
  - Each module may claim the bus
  - Control logic on all modules

# References

- William Stallings “Computer Organization and architecture”, Prentice Hall, 7<sup>th</sup> edition, 2006.

# Types of External Memory

- Magnetic Disk
  - RAID
  - Removable
- Optical
  - CD-ROM
  - CD-Recordable (CD-R)
  - CD-R/W
  - DVD
- Magnetic Tape

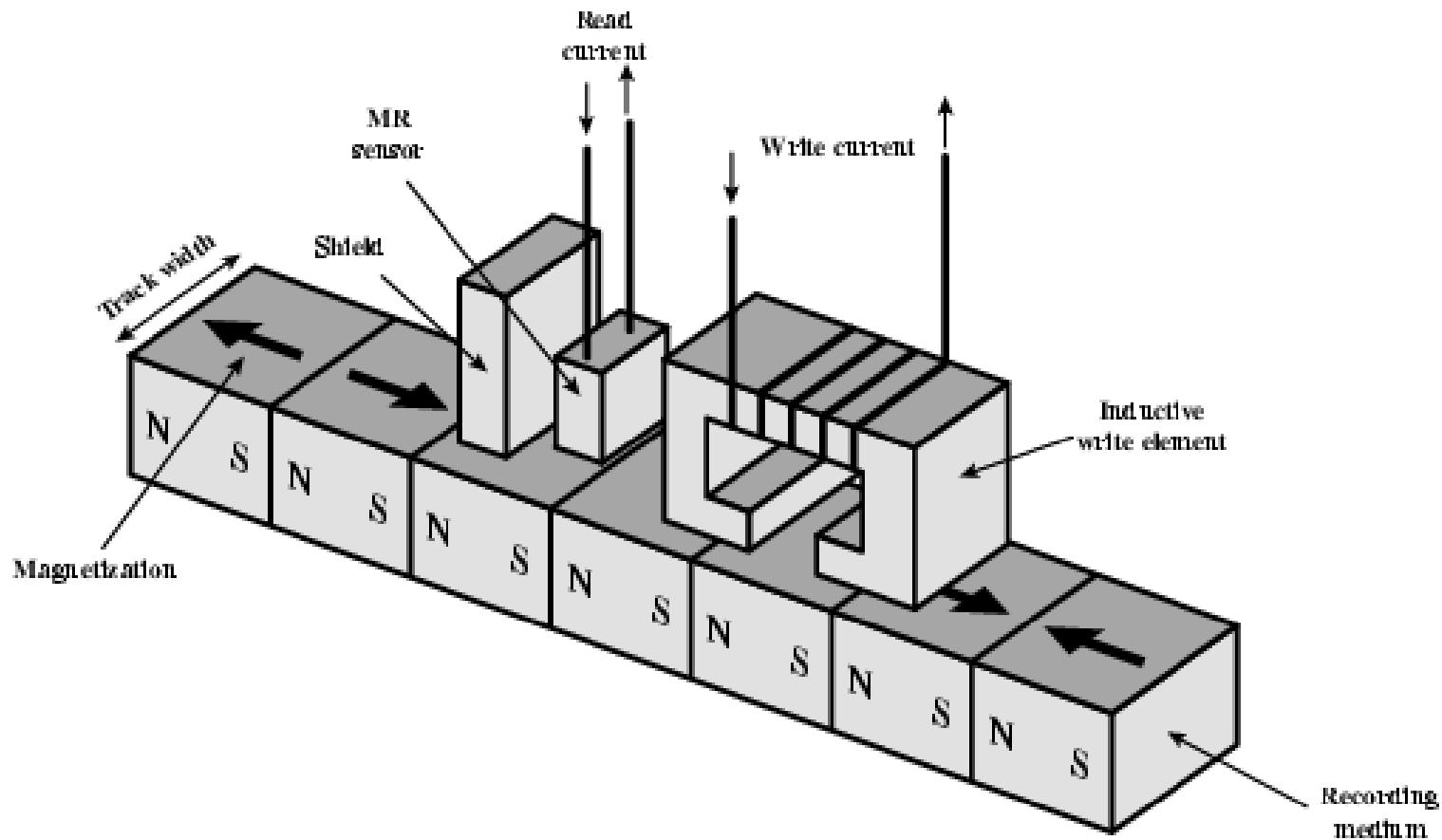
# Magnetic Disk

- A disk is a circular platter constructed of non magnetic material → **substrate**
- Disk substrate coated with magnetisable material (iron oxide...rust)
- Substrate used to be aluminium or aluminium alloy material.
- Now glass is used as **substrate**
- **Benefits of using glass as substrate**
  - Improved surface uniformity
    - Increases reliability
  - Reduction in surface defects
    - Reduced read/write errors
  - Better stiffness
  - Better shock/damage resistance

# Read and Write Mechanisms

- Recording & retrieval via conductive coil called a head
- May be single read/write head or separate ones
- During read/write, head is stationary, platter rotates
- Write
  - Current through coil produces magnetic field
  - Pulses sent to head
  - Magnetic pattern recorded on surface below
- Read (traditional)
  - Magnetic field moving relative to coil produces current
  - Coil is the same for read and write
- Read (contemporary)
  - Separate read head, close to write head
  - The head consists of partially shielded magneto resistive (MR) sensor
  - The MR material has an electrical resistance depends on direction of magnetization medium moving under it
  - MR design allows high frequency operation which equates to greater storage density and speed

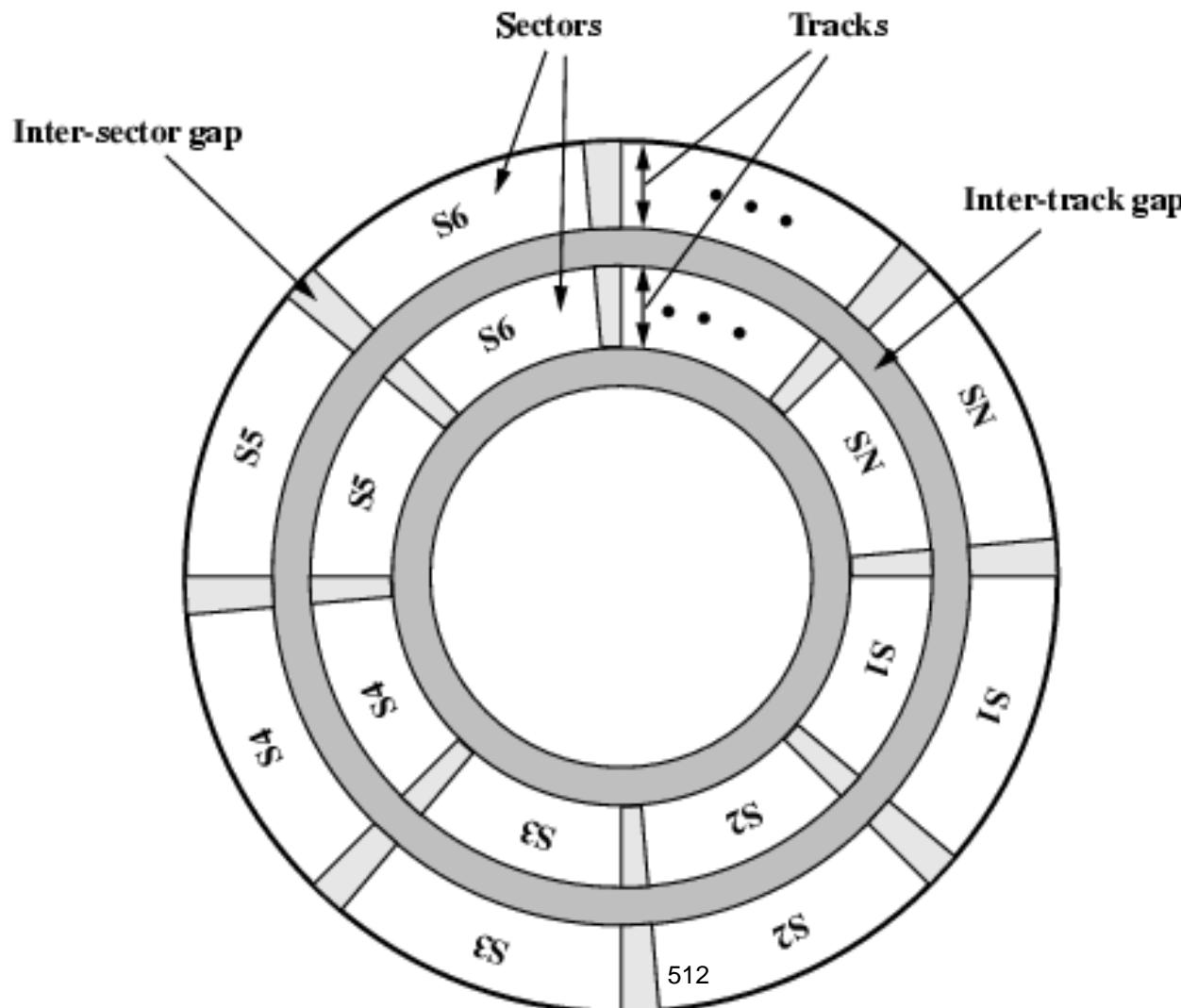
# Inductive Write MR Read



# Data Organization and Formatting

- Concentric rings or tracks
  - Gaps between tracks
  - Reduce gap to increase capacity
  - Same number of bits per track (variable packing density)
  - Constant angular velocity
- Tracks divided into sectors
- Minimum block size is one sector
- May have more than one sector per block

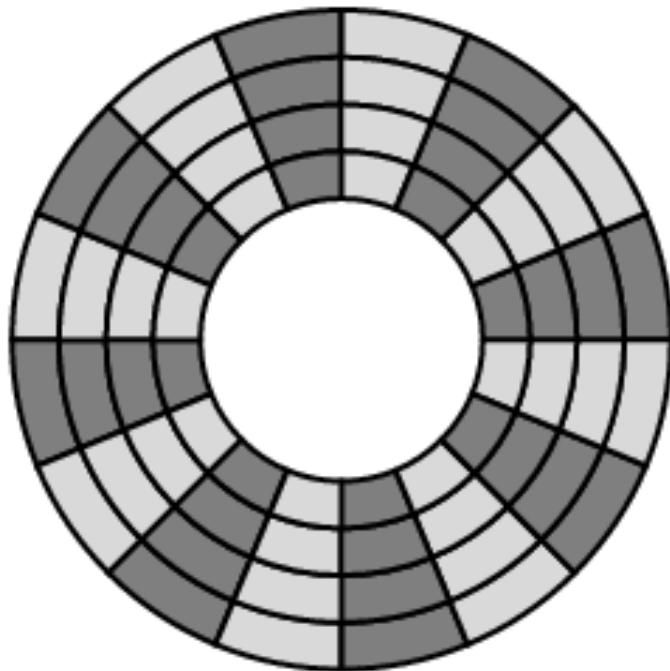
# Disk Data Layout



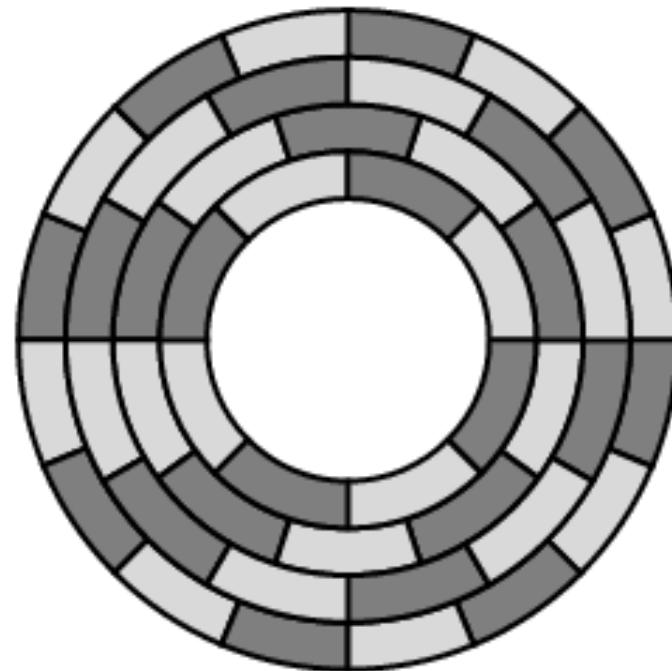
# Disk Velocity

- Bit near centre of rotating disk passes fixed point slower than bit on outside of disk
- Increase spacing between bits in different tracks
- Rotate disk at constant angular velocity (CAV)
  - Gives pie shaped sectors and concentric tracks
  - Individual tracks and sectors addressable
  - Move head to given track and wait for given sector
  - Waste of space on outer tracks
    - Lower data density
- Can use zones to increase capacity
  - Each zone has fixed bits per track
  - More complex circuitry

# Disk Layout Methods Diagram



**(a) Constant angular velocity**



**(b) Multiple zoned recording**

# Multiple Zone Recording

- To increase density, modern hard disk systems use a technique → Multiple Zone Recording
- The surface is divided into number of concentric zone .
- Within a zone a number, the number of bit /track is constant.
- Zones farther from the centre contain more sectors than zones closer to the centre.
- Greater overall storage capacity at the expense of somewhat more complex circuitry.

# Finding Sectors

- Must be able to identify start of track and sector
- Format disk
  - Additional information not available to user
  - Marks tracks and sectors

# Characteristics

- Fixed (rare) or movable head
- Removable or fixed
- Single or double (usually) sided
- Single or multiple platter
- Head mechanism
  - Contact (Floppy)
  - Fixed gap
  - Flying (Winchester)

# Fixed/Movable Head Disk

- Fixed head
  - One read write head per track
  - Heads mounted on fixed ridged arm
- Movable head
  - One read write head per side
  - Mounted on a movable arm

# Removable or Not

- Removable disk
  - Can be removed from drive and replaced with another disk
  - Provides unlimited storage capacity
  - Easy data transfer between systems
- Nonremovable disk
  - Permanently mounted in the drive

# Multiple Platter

- One head per side
- Heads are joined and aligned
- Aligned tracks on each platter form cylinders
- Data is striped by cylinder
  - reduces head movement
  - Increases speed (transfer rate)

# Physical Characteristics of Disk Systems

## **Head Motion**

- Fixed head (one per track)
- Movable head (one per surface)

## **Platters**

- Single platter
- Multiple platter

## **Disk Portability**

- Nonremovable disk
- Removable disk

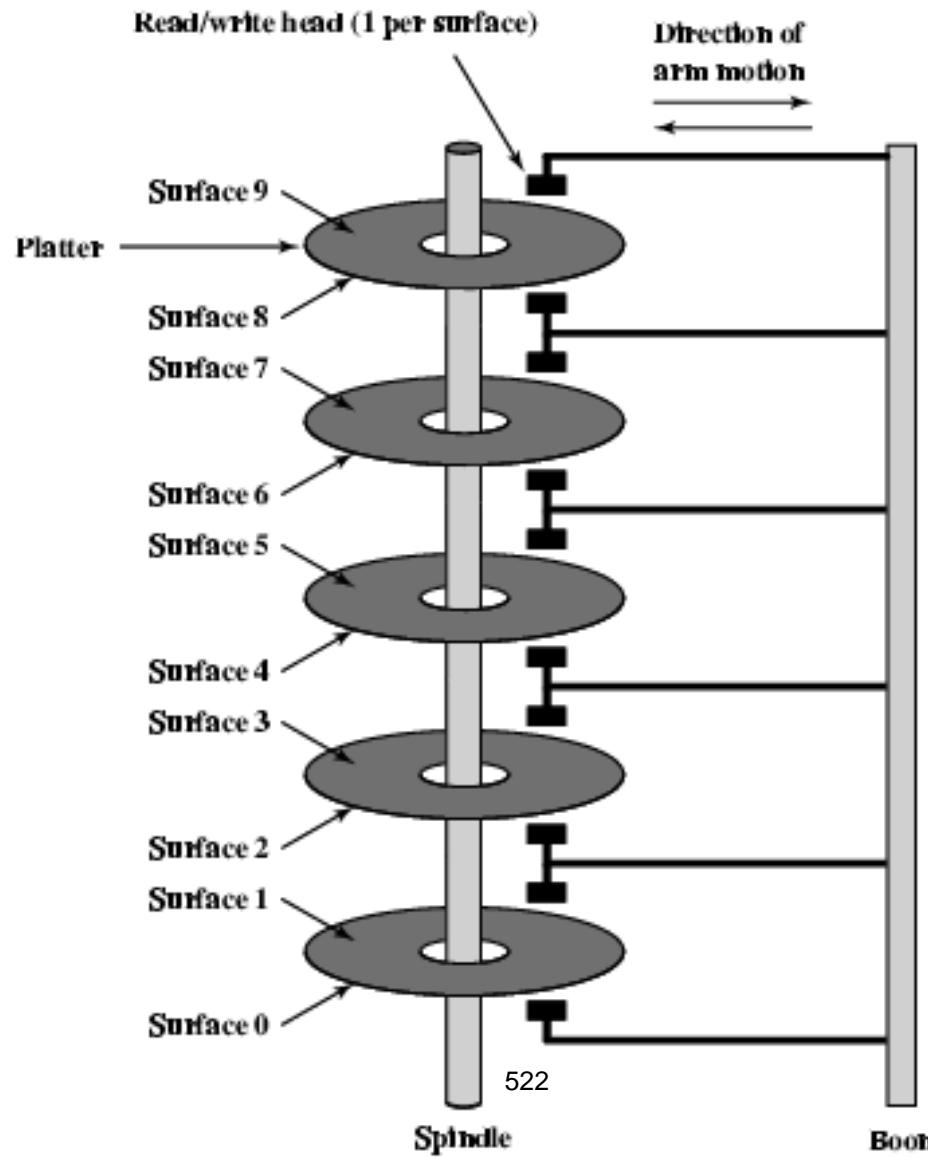
## **Head Mechanism**

- Contact (floppy)
- Fixed gap
- Aerodynamic gap (Winchester)

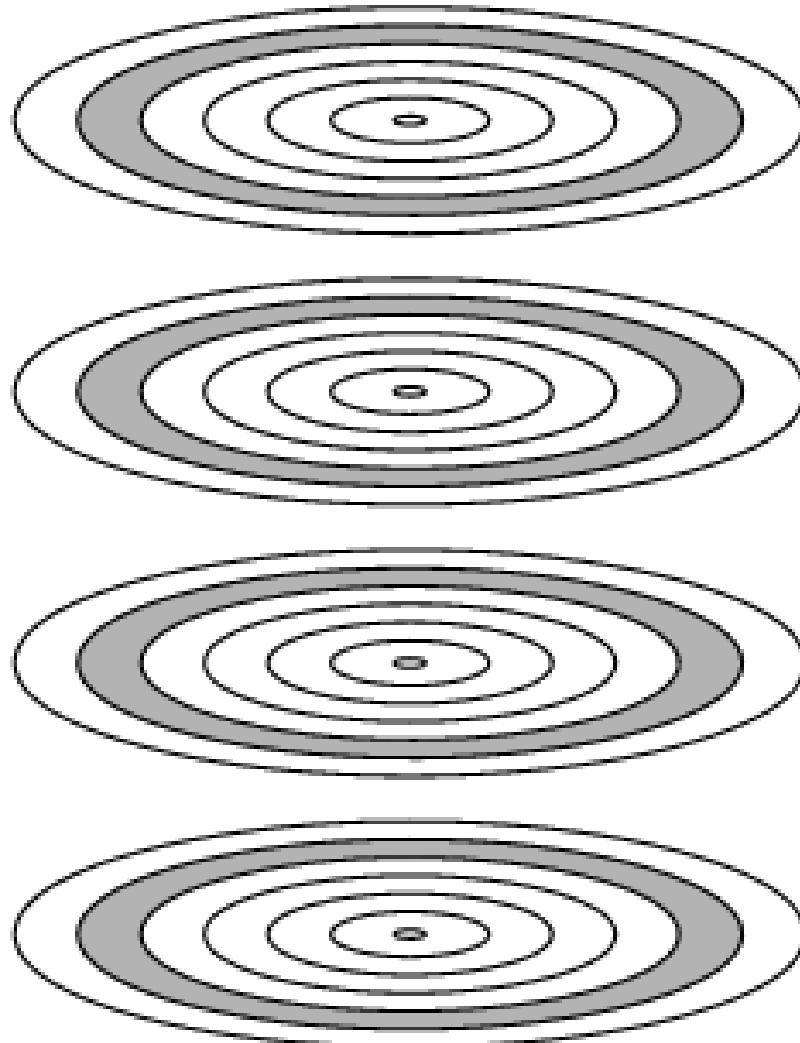
## **Sides**

- Single sided
- Double sided

# Multiple Platters



# Tracks and Cylinders



# Speed Parameters

- Seek time
  - Time to position head at track
- Latency (Rotational)
  - Time for head to rotate to beginning of sector
- Access time
  - Seek time + Latency time
- Transfer rate
  - The rate at which data can be transferred **after access**
$$T = b / N * 1/r$$

Transfer time = bytes transferred / bytes/track \* sec/revolution (track)

Note: How does organization on disk (e.g. random vs sequential) effect total time?

# Optical Products

## **CD**

Compact Disk. A nonerasable disk that stores digitized audio information. The standard system uses 12-cm disks and can record more than 60 minutes of uninterrupted playing time.

## **CD-ROM**

Compact Disk Read-Only Memory. A nonerasable disk used for storing computer data. The standard system uses 12-cm disks and can hold more than 650 Mbytes.

## **CD-R**

CD Recordable. Similar to a CD-ROM. The user can write to the disk only once.

## **CD-RW**

CD Rewritable. Similar to a CD-ROM. The user can erase and rewrite to the disk multiple times.

## **DVD**

Digital Versatile Disk. A technology for producing digitized, compressed representation of video information, as well as large volumes of other digital data. Both 8 and 12 cm diameters are used, with a double-sided capacity of up to 17 Gbytes. The basic DVD is read-only (DVD-ROM).

## **DVD-R**

DVD Recordable. Similar to a DVD-ROM. The user can write to the disk only once. Only one-sided disks can be used.

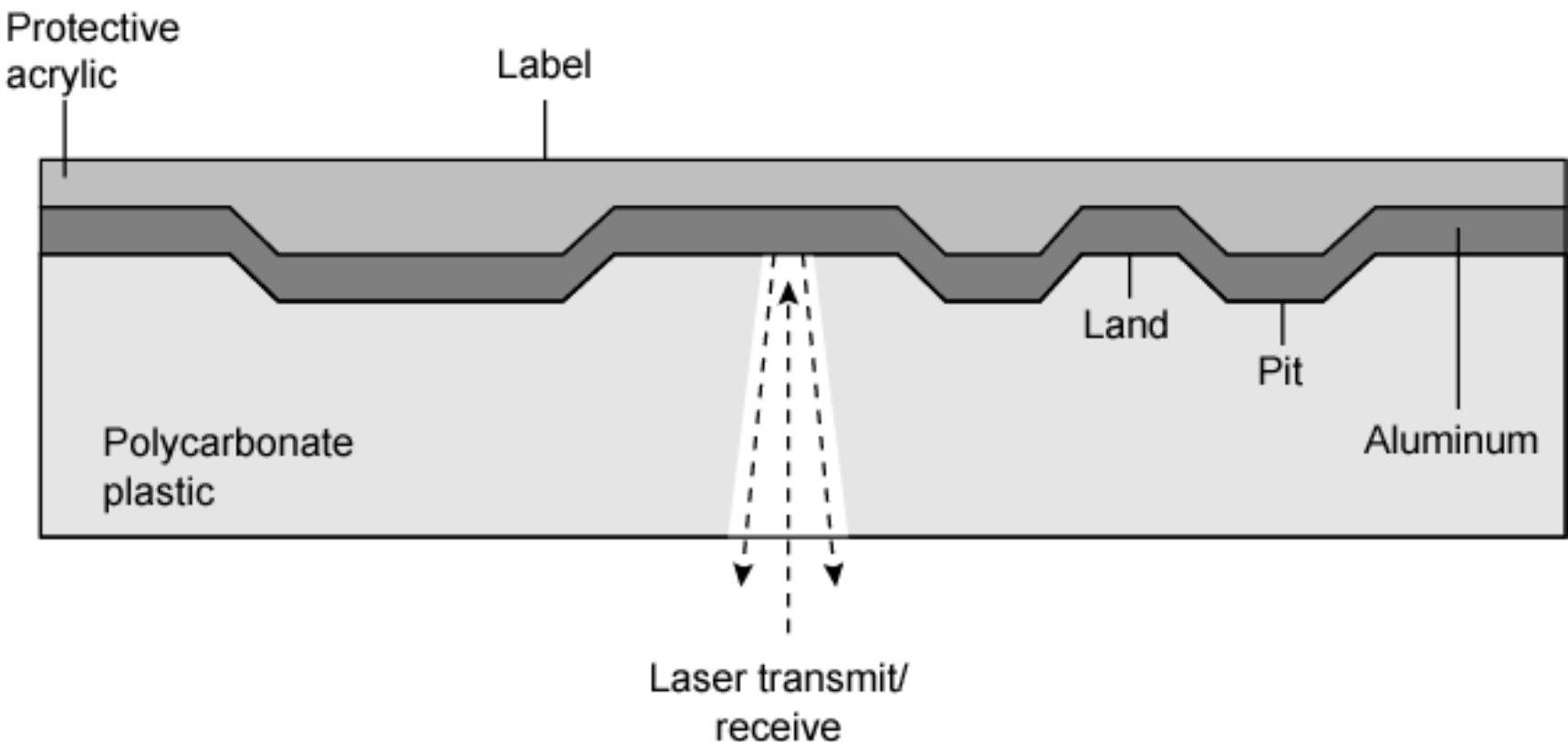
## **DVD-RW**

DVD Rewritable. Similar to a DVD-ROM. The user can erase and rewrite to the disk multiple times. Only one-sided disks can be used.

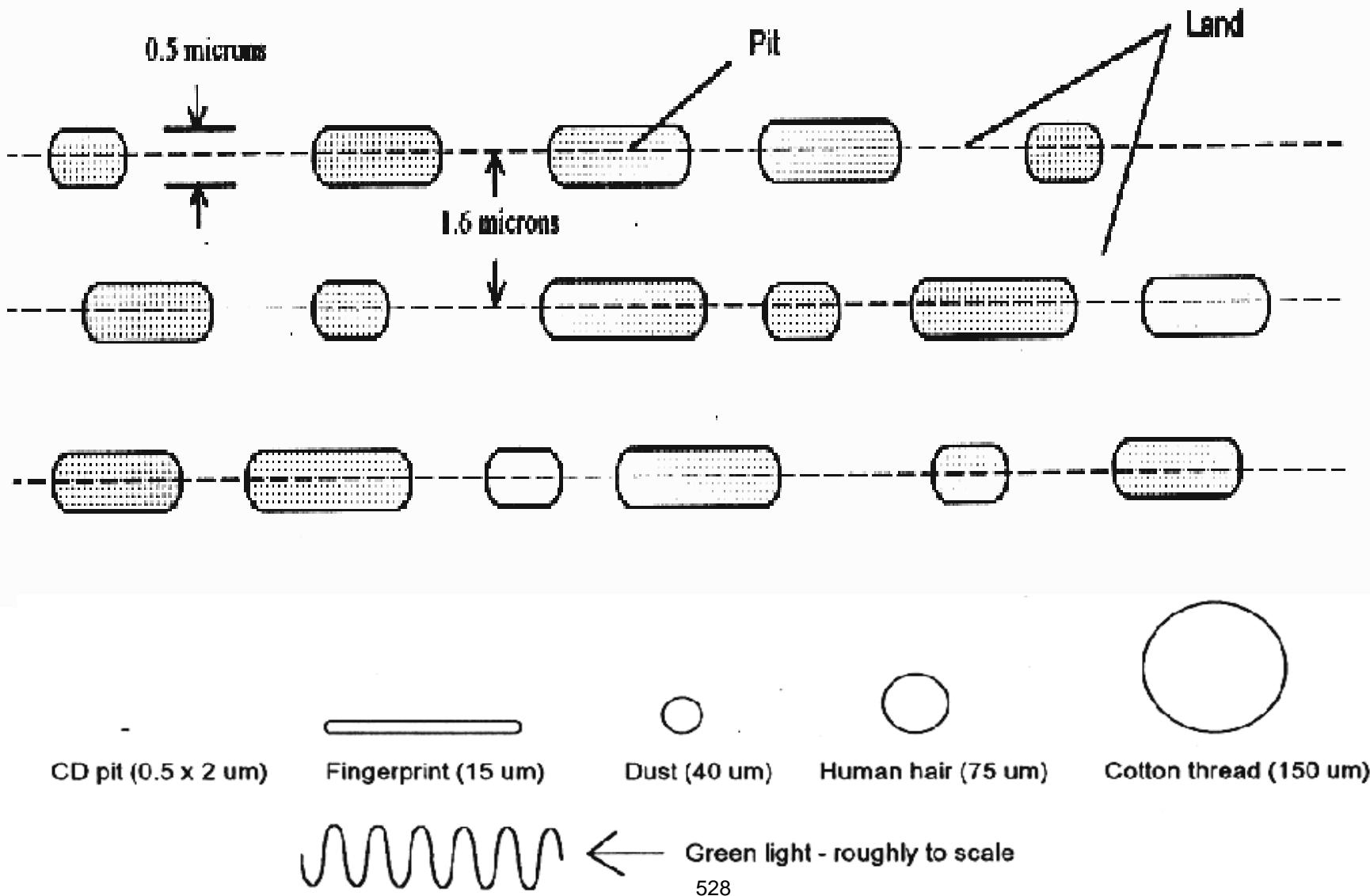
# Optical Storage CD-ROM

- Originally for audio
- 650Mbytes giving over 70 minutes audio
- Polycarbonate coated with highly reflective coat, usually aluminium
- Data stored as pits
- Read by reflecting laser
- Constant packing density
- Constant linear velocity

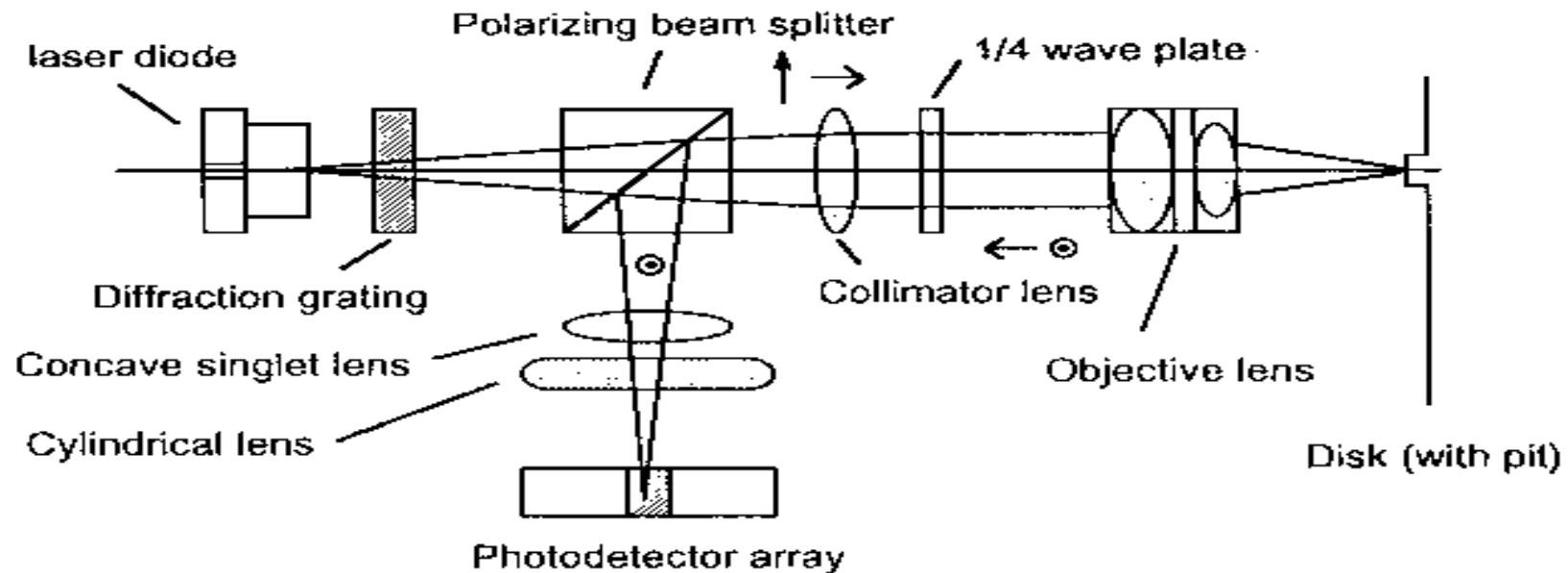
# CD Construction



# CD Layout



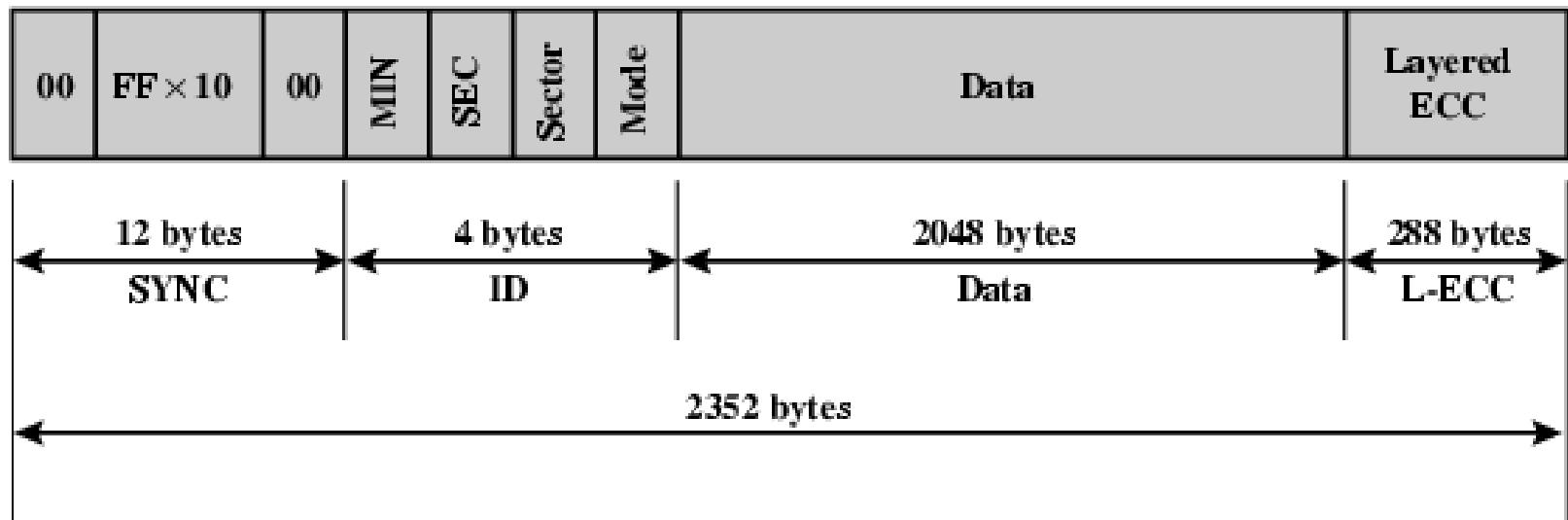
# CD reader



# CD-ROM Drive Speeds

- Audio is single speed
  - Constant linear velocity
  - 1.2 m/sec
  - Track (spiral) is 5.27km long
  - Gives 4391 seconds = 73.2 minutes
- Other speeds are quoted as multiples
  - e.g. 24x
  - Quoted figure is maximum drive can achieve
- Note: CD-ROM has option of error correction (not on CD)

# CD-ROM Format



- Mode 0 = blank data field
- Mode 1 = 2048 byte data + error correction
- Mode 2 = 2336 byte data

# Random Access on CD-ROM & CD-R

- Difficult
- Process:
  - Move head to rough position
  - Set correct speed
  - Read address
  - Adjust to required location

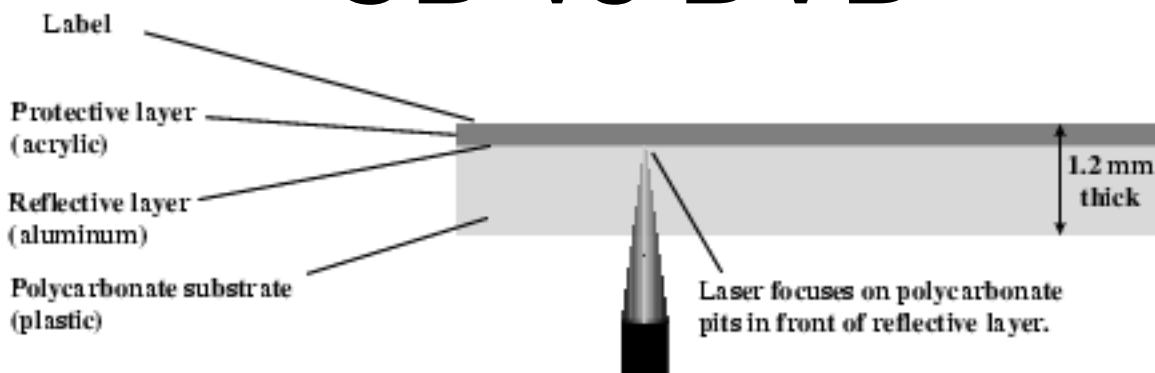
# CD-RW

- Erasable
- Getting cheaper
- Mostly CD-ROM drive compatible
- Phase change
  - Material has two different reflectivities in different phase states

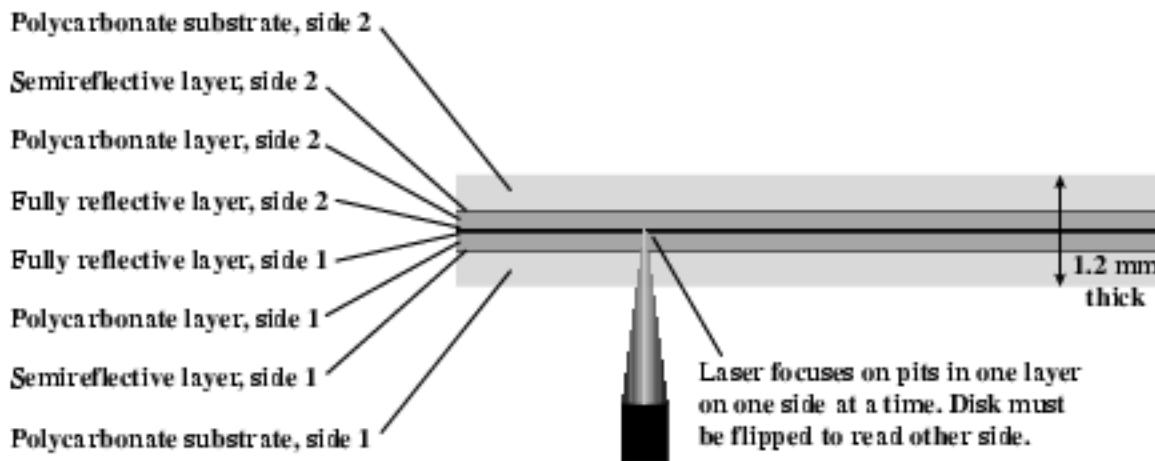
# DVD - technology

- Multi-layer
- Very high capacity (4.7G per layer)
- Full length movie on single disk
  - Using MPEG compression

# CD vs DVD



(a) CD-ROM - Capacity 682 MB



## DVD's

Two objectives had to be resolved to make the DVDs viable.

- The linear velocity of a DVD must be held constant and be able to reproduce a vertical frame rate of 29.97 frames/second
- Every DVD player had to have absolute tracking accuracy to insure the extremely narrow laser beam would scan exactly in the middle of the track where the data was recorded.

The solution:

- The disk is pressed with the track grooves accurately pre-cut and encoded with a constant bit rate frequency. Thus a blank DVD disk isn't really blank at all.

# Magnetic Tape

- Serial access
- Slow
- Very cheap
- Used for backup and archive

# References

- William Stallings “Computer Organization and architecture”, Prentice Hall, 7<sup>th</sup> edition, 2006.

# What is RAID

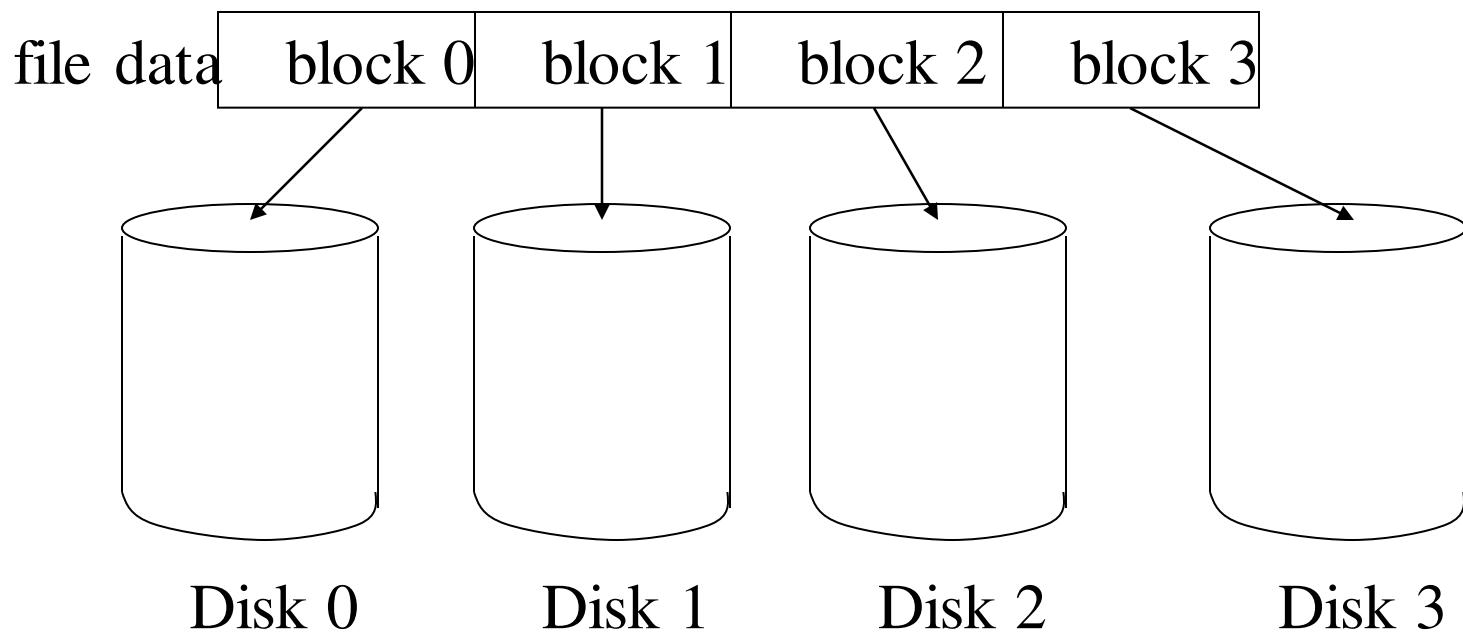
- Redundant Array of Independent (Inexpensive) Disks
- A set of disk stations treated as one logical station
- Data are distributed over the stations
- Redundant capacity is used for parity allowing for data repair

# RAID (Redundant Array of Independent Disks)

- Redundant array of inexpensive disks
- Multiple disk database design
- Set of physical disk drives viewed by the OS as a single logical drive
- Data are distributed across the physical drives of an array
- Improve access time and improve reliability
  - large storage capacity
  - redundant data
    - 7 levels (6 levels in common use)
      - differing levels of redundancy, error checking, capacity, and cost

# Striping

- Take file data and map it to different disks
- Allows for reading data in parallel



# Parity

- Way to do error checking and correction
- Add up all the bits that are 1
  - if even number, set parity bit to 0
  - if odd number, set parity bit to 1
- To actually implement this, do an exclusive OR of all the bits being considered
- Consider the following 2 bytes

<u>byte</u>	<u>parity</u>
10110011	1
01101010	0

- If a single bit is bad, it is possible to correct it

# Mirroring

- Keep two copies of data on two separate disks
- Gives good error recovery
  - if some data is lost, get it from the other source
- Expensive
  - requires twice as many disks
- Write performance can be slow
  - have to write data to two different spots
- Read performance is enhanced
  - can read data from file in parallel

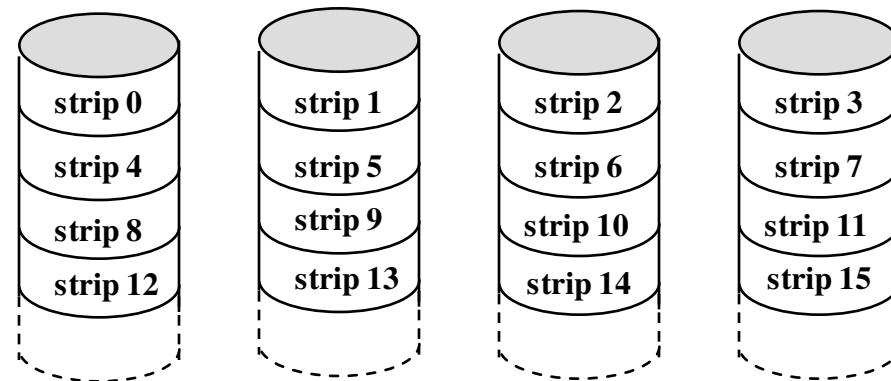
# Levels of RAID

- 6 levels of RAID (0-5) have been accepted by industry
- Other kinds have been proposed in literature
- Level 2 and 4 are not commercially available, they are included for clarity

# RAID 0

- All data (user and system) are distributed over the disks so that there is a reasonable chance for parallelism
- Disk is logically a set of strips (blocks, sectors,...). Strips are numbered and assigned consecutively to the disks (see picture.)

# Raid 0 (No redundancy)



# Data mapping Level 0

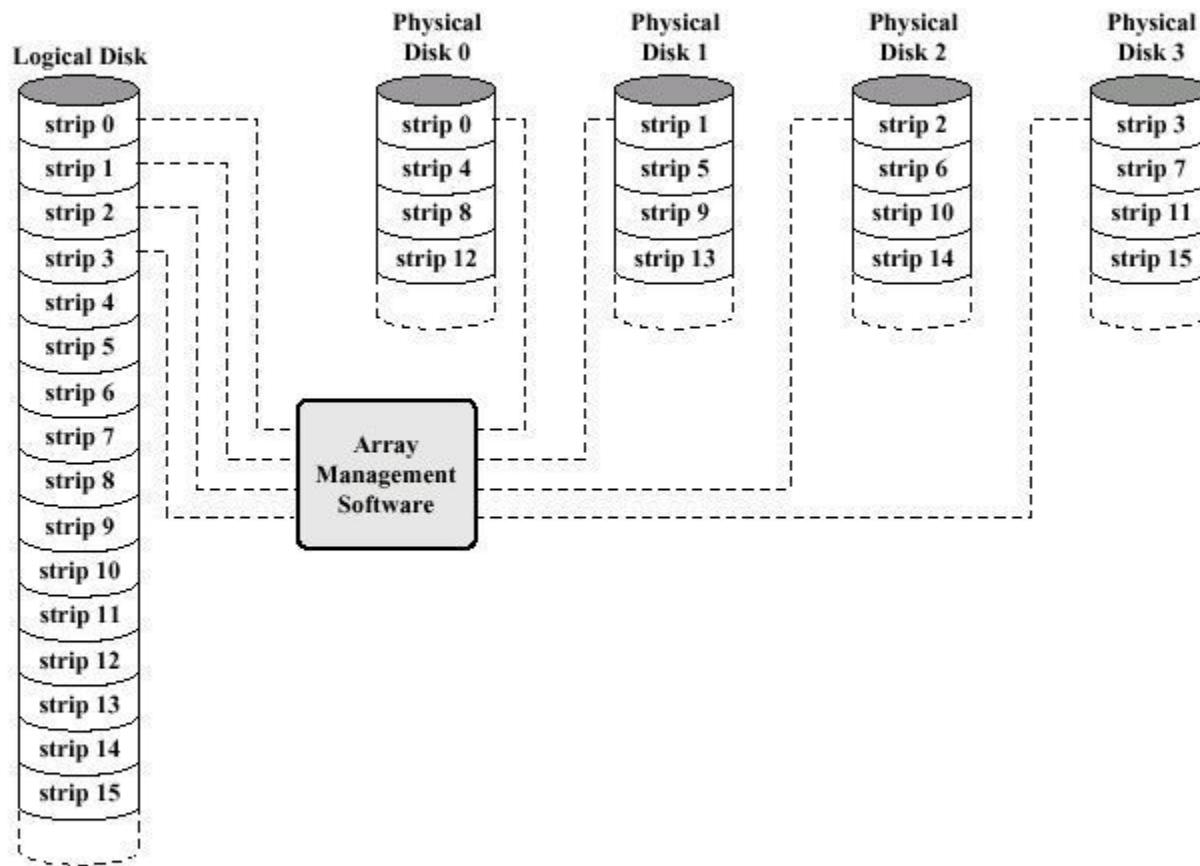


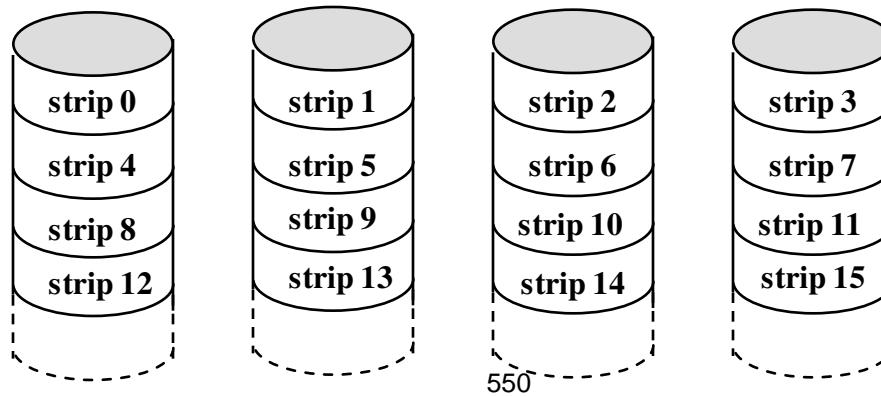
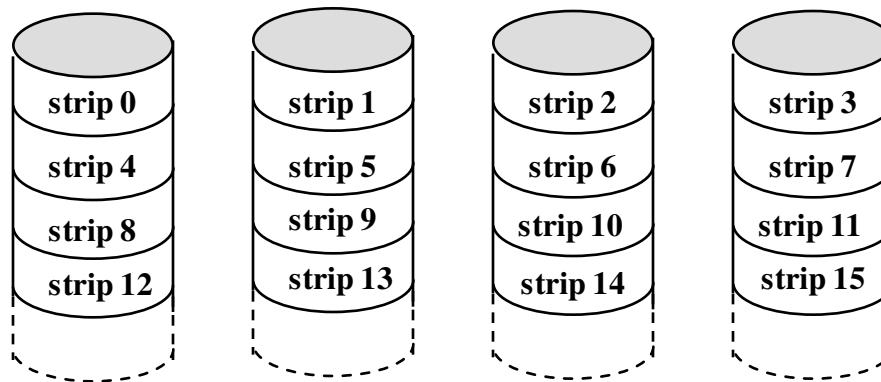
Figure 11.9 Data Mapping for a RAID Level 0 Array [MASS94]

# RAID 0:

- Performance depends highly on the request patterns
- High data transfer rates are reached if
  - Integral data path is fast (internal controllers, I/O bus of host system, I/O adapters and host memory busses)
  - Application generates efficient usage of the disk array by requests that span many consecutive strips
- If response time is important (transactions) more I/O requests can be handled in parallel

- Block level stripping
- Without parity or mirroring
- No redundancy, No backup, No fault tolerance
- Improved performance, faster, as it is uses block level stripping
- Maximum use of storage space as there is no backup
- Any drive failure destroys the array

# Raid 1 (mirrored)



# RAID 1

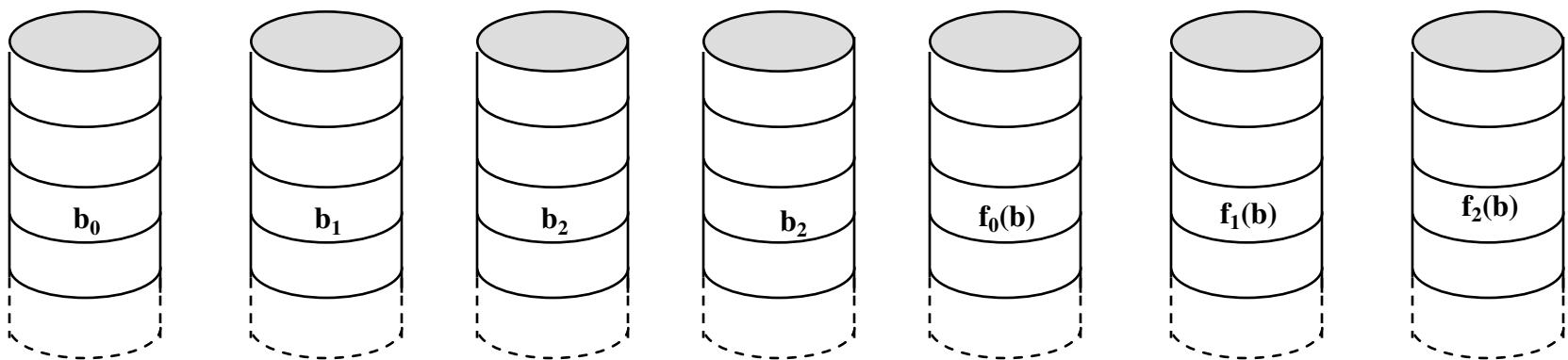
- RAID 1 does not use parity, it simply mirrors the data to obtain reliability
- Plus:
  - Reading request can be served by any of the two disks containing the requested data (minimum search time)
  - Writing request can be performed in parallel to the two disks: no “writing penalty”
  - Recovery from error is easy, just copy the data from the correct disk

# RAID 1

- Minus:
  - Price for disks is doubled
  - Will only be used for system critical data that must be available at all times
- RAID 1 can reach high transfer rates and fast response times ( $\sim 2 \times$  RAID 0) if most of the requests are reading requests. In case most requests are writing requests, RAID 1 is not much faster than RAID 0.

- Mirroring
- Without parity
- Data is written identically to two drives (parallel write)
- Not slower, not faster (Write)
- Read can be faster by parallel access
- Fault tolerance
- 50% storage space can be used

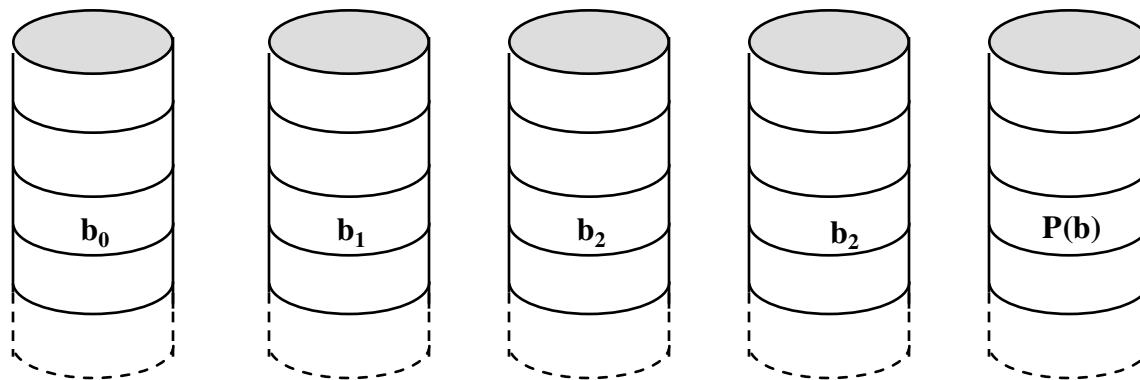
# Raid 2 (redundancy through Hamming code)



# RAID 2

- Small strips, one byte or one word
- Synchronized disks, each I/O operation is performed in a parallel way
- Error correction code (Hamming code) allows for correction of a single bit error
- Controller can correct without additional delay
- Is still expensive

# RAID 3 (bit-interleaved parity)



# RAID 3

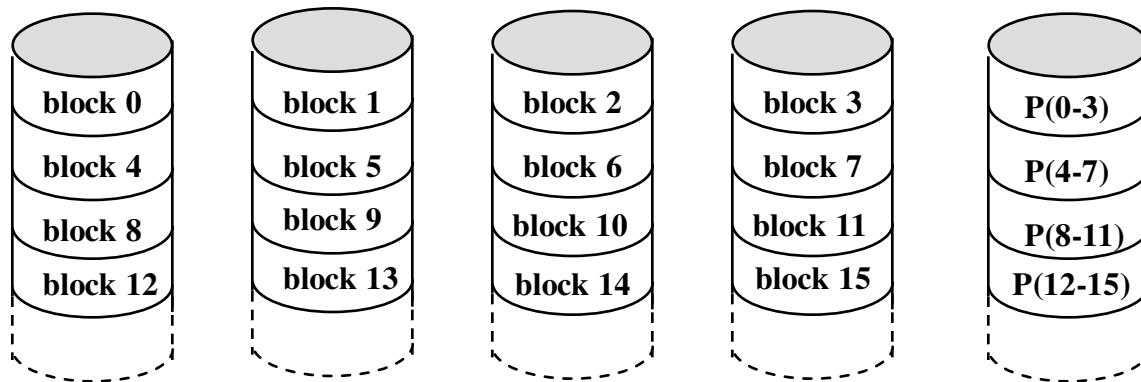
- Level 2 needs  $\log_2(\text{number of disks})$  parity disks
- Level 3 needs only one, for one parity bit
- In case one disk crashes, the data can still be reconstructed even on line (“reduced mode”) and be written (X1-4 data, P parity):

$$P = X_1 + X_2 + X_3 + X_4$$

$$X_1 = P + X_2 + X_3 + X_4$$

- RAID 2-3 have high data transfer times, but perform only one I/O at the time so that response times in transaction oriented environments are not so good

# RAID 4 (block-level parity)



# RAID 4

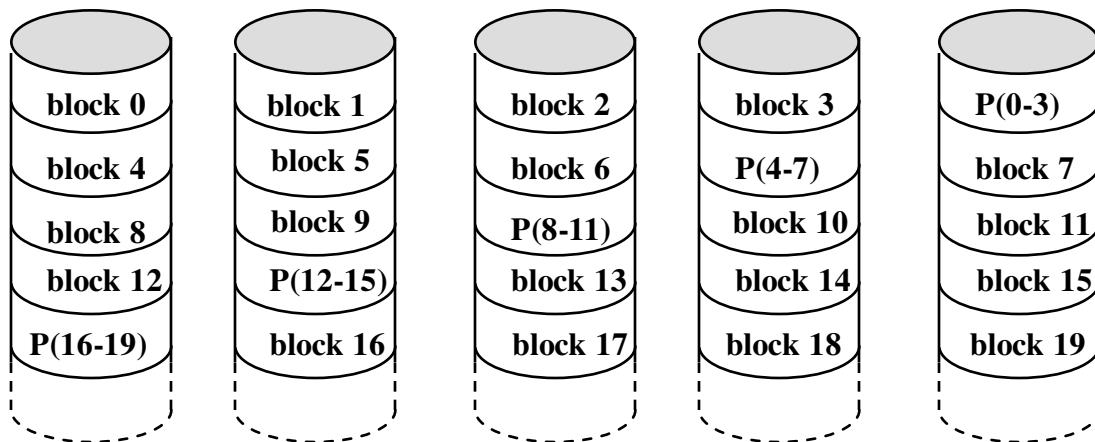
- Larger strips and one parity disk
- Blocks are kept on one disk, allowing for parallel access by multiple I/O requests
- Writing penalty: when a block is written, the parity disk must be adjusted (e.g. writing on X1):

$$P = X_4 + X_3 + X_2 + X_1$$

$$\begin{aligned} P' &= X_4 + X_3 + X_2 + X_1' \\ &= X_4 + X_3 + X_2 + X_1' + X_1 + X_1 \\ &= P + X_1 + X_1' \end{aligned}$$

- Parity disk may be a bottleneck
- Good response times, less transfer rates

# RAID 5 (block-level distributed parity)

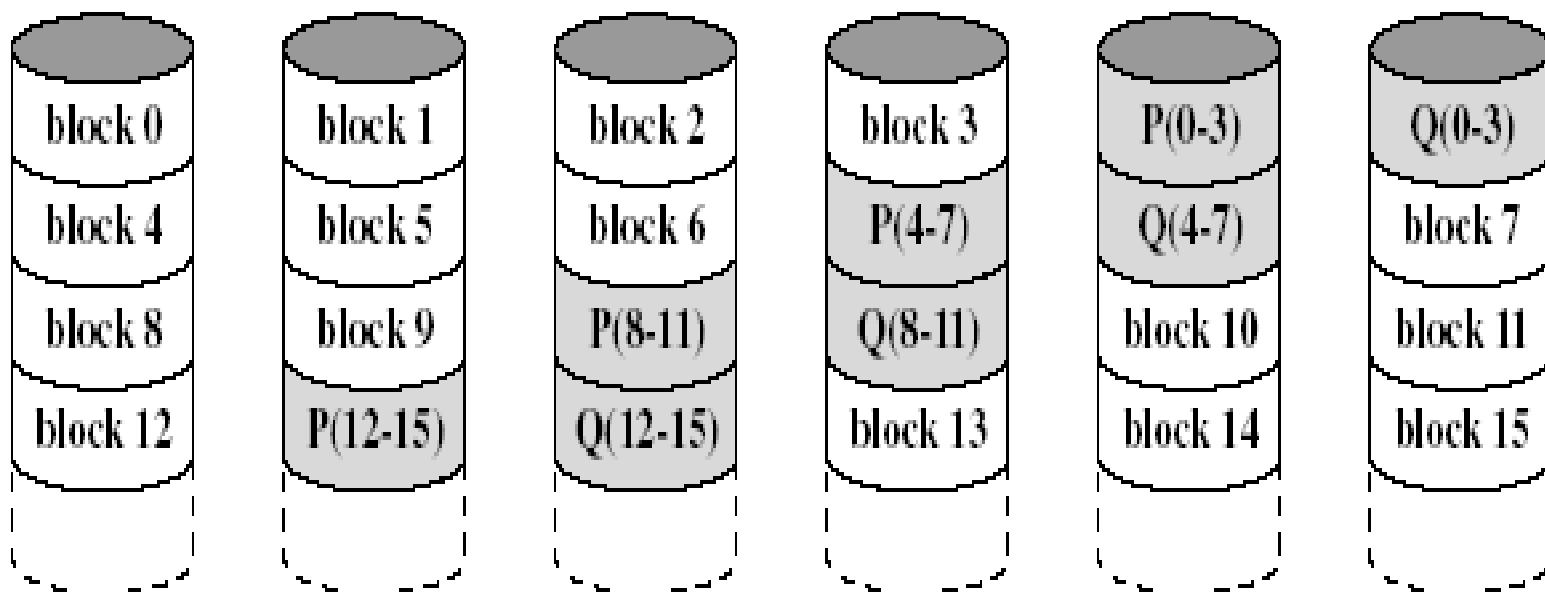


# RAID 5

- Distribution of the parity strip to avoid the bottle neck.

- Faster- Block level stripping
- Fault tolerance by Distributed parity (backup of data)
- Distribution of the parity strip to avoid the bottle neck.
- The array is not destroyed by the failure of single hard drive
- Minimum three disks
- Loose one hard disk space for parity
- 75% space

# RAID 6

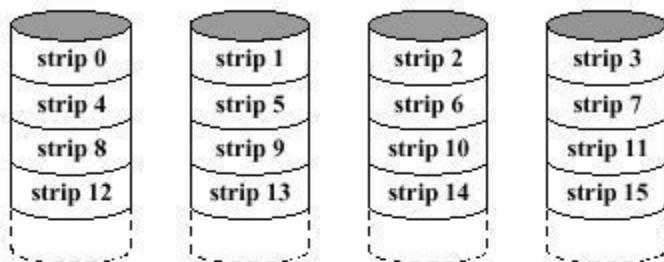


# RAID Level 6

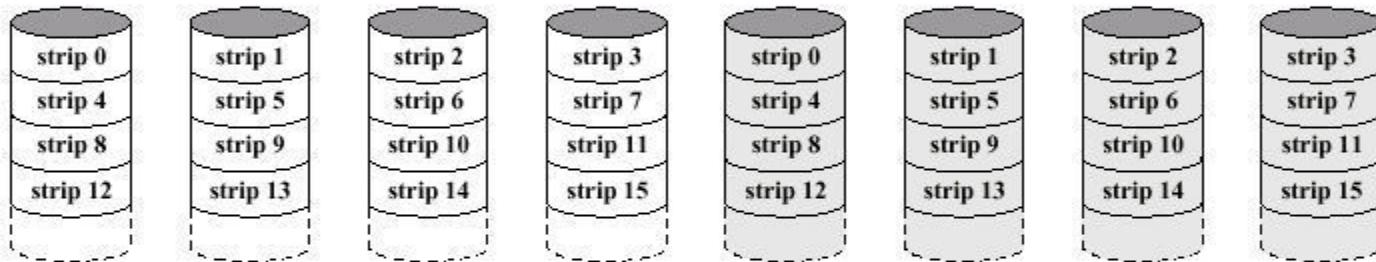
- Two different parity calculations are carried out and stored in separate blocks on different disks.
  - Example: XOR and an independent data check algorithm => makes it possible to regenerate data even if two disks containing user data fail.
- No. of disks required =  $N + 2$  (where  $N$  = number of disks required for data).
- Provides HIGH data availability.
- Incurs substantial write penalty as each write affects two parity blocks.
- Three disks would have to fail within MTTR (mean time to repair) interval to cause data to be lost

- Faster-Block level stripping
- Provides fault tolerance up to two failed drives Fault tolerance- double distributed parity (backup of backup of data)
- High availability systems
- Failure of hard drive, slow down the performance of the system
- Loose two hard disk space for parity
- Incurs substantial write penalty

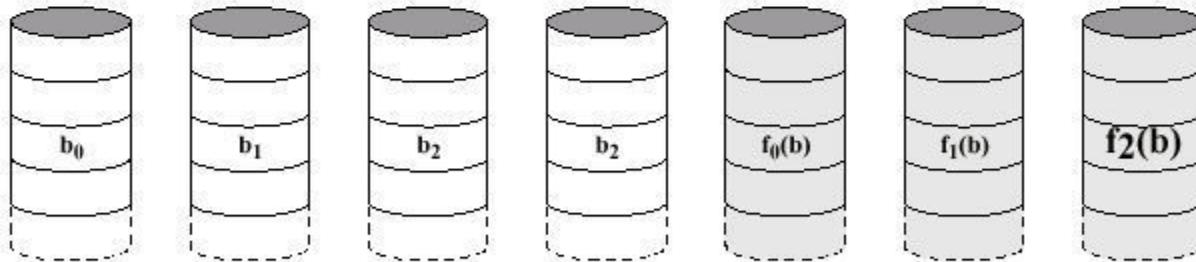
# Overview Raid 0-2



(a) RAID 0 (non-redundant)



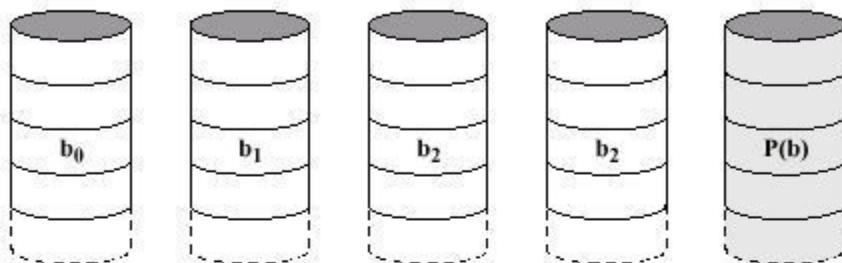
(b) RAID 1 (mirrored)



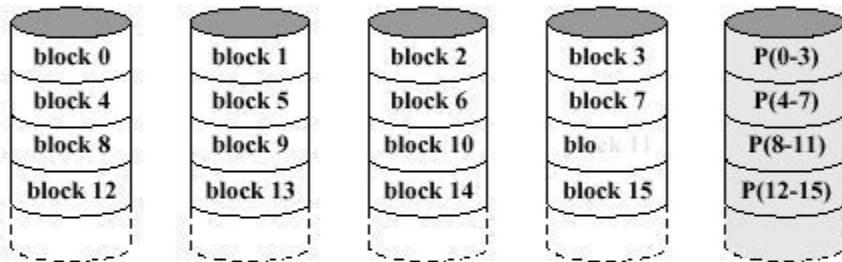
(c) RAID 2 (redundancy through Hamming code)

**Figure 11.8 RAID Levels (page 1 of 2)**

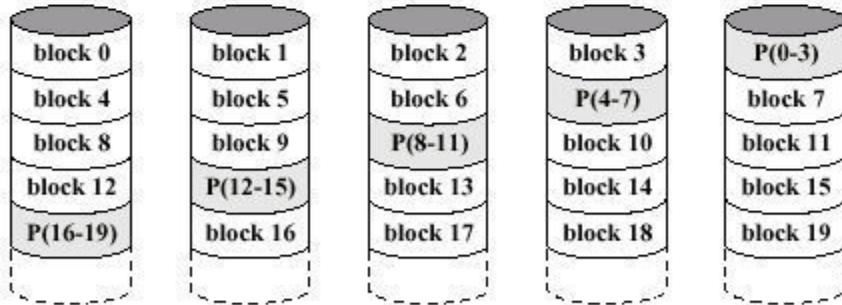
# Overview Raid 3-5



(d) RAID 3 (bit-interleaved parity)



(e) RAID 4 (block-level parity)



(f) RAID 5 (block-level distributed parity)

Figure 11.8 RAID Levels (page 2 of 2)

# Tutorial 1

# Computer Architecture and Organization

Submission date : 31-10-2019

**Submit hand written copy**

1. Consider the byte registers  $R0 = 62_H$  (H means Hexadecimal) and  $R1 = 71_H$ . What will be the contents of R0 and R1 after the following instructions are executed?  
(i) Add R1,R0 , (ii) XOR R0,R0,  
(iii) AshiftR #2, R1 (iv) RotateR #2, R0
  2. Perform multiplication using both Booth's and Modified Booth's algorithm.

$$A = 25 \text{ and } B = -35.$$

3. Explain different addressing modes with sample instructions.

## Tutorial 2

# Computer Architecture and Organization

Submission date : 31-10-2019

Submit hand written copy

1. A computer system employs a cache with the hit ratio, 80 % for write operation and 90% for read operation. Dynamic memory access time is 250ns and the cache memory access time is 28ns. Out of all memory references, 70% are read operations and 30% are write operations. Calculate the average access time for all references (both read and write), for the following two specifications of the system.
  - a. Write operation is write back and read is look through.
  - b. Write operation is write through and read is look aside.
2. Consider a 4-way set associative cache memory with total 8 cache lines and a main memory with 256 blocks. Find the memory blocks which will be present in the cache after the following sequence of memory block references with the following block replacement algorithms. Assuming that initially the cash is empty.
 

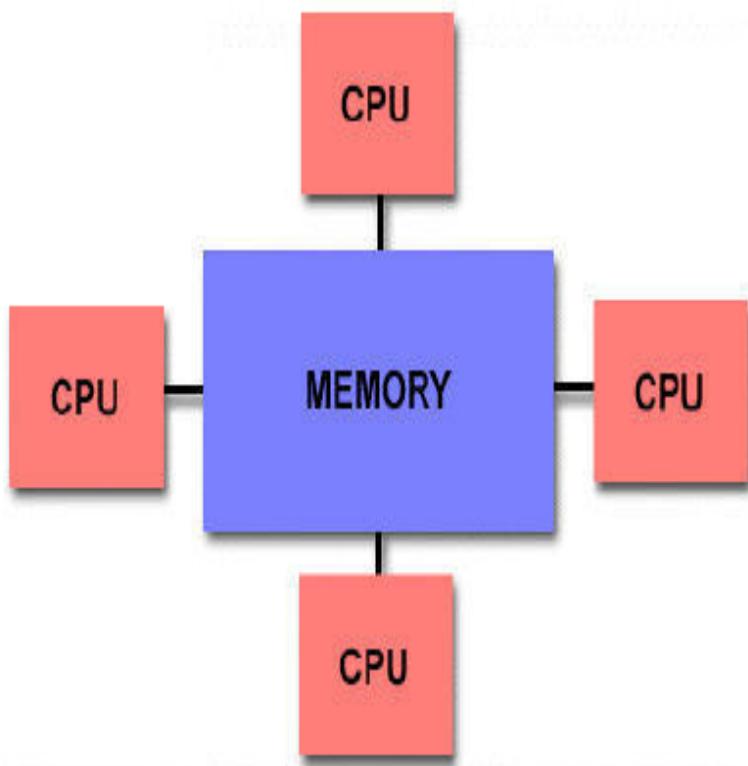
8 2 4 7 8 4 2 3 6 2 13 2 32 7 53 7 27 53 48 18 15 7 2 82

  - a. LFU policy is used for cache block replacement
  - b. LRU policy is used for cache block replacement
3. A computer employs RAM chips of  $128 \times 8$  and ROM chips of  $256 \times 8$ . The computer system needs 4K bytes of RAM, 2K x 32 of ROM, and two interface units with 256 registers each. Then, calculate the following,
  - Number of RAM chips
  - Number of ROM chips
  - Value of x, y and z
4. A computer has the main memory with  $2^{32}$  bytes size and has the word size  $2^2$  bytes. The cache memory size is  $2^{20}$  bytes and each cache line is  $2^8$  words. What is the length of tag field of its physical address, for a 4-way set-associative cache memory?

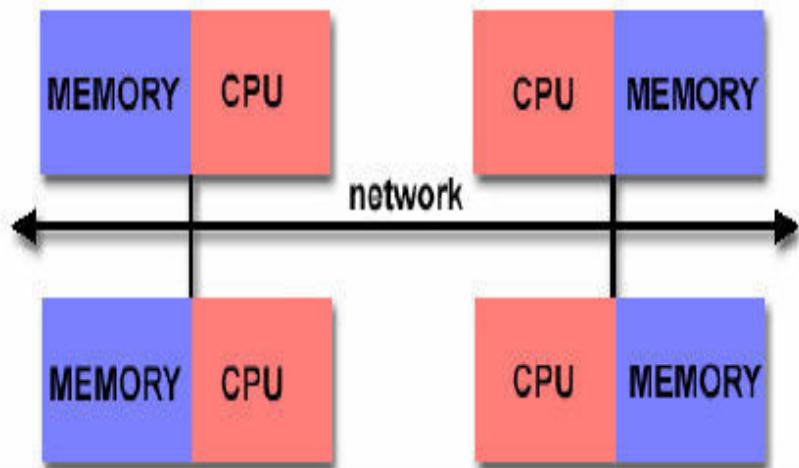
# PARALLEL PROCESSING

- Parallel computers are those that emphasize the parallel processing between the operations in some way.
- Various Varieties
  - Shared Memory
  - Distributed Memory
  - Hybrid Memory

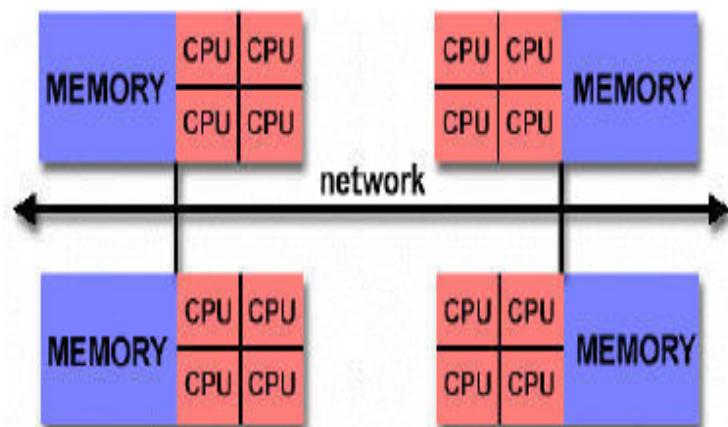
## Shared Memory



## Distributed Memory



## Hybrid Memory



# Flynn's Taxonomy

- This classification was first studied and proposed by Michael Flynn in 1972.
- Flynn did not consider the machine architecture for classification of parallel computers
- He introduced the concept of instruction and data streams for categorizing of computers.
- Instruction stream - a flow of instructions from main memory to the CPU
- Data stream - a flow of operands between processor and memory
- All the computers classified by Flynn are not parallel computers
- Let  $I_s$  and  $D_s$  are minimum number of streams flowing at any point in the execution

# Instruction Cycle

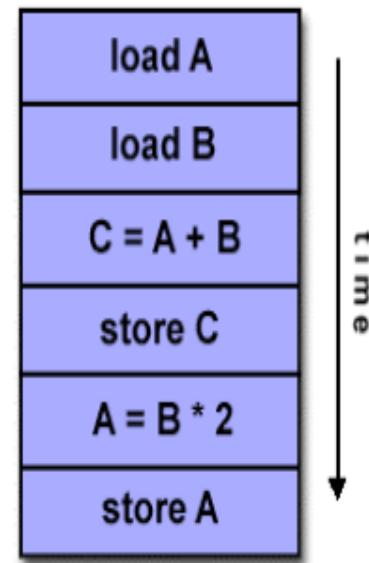
- Start
- Calculate the address of the instruction to be executed
- Fetch the instruction
- Decode the instruction
- Calculate the operand address
- Fetch the operands
- Execute the instructions
- Store the results
- If more instructions to be executed, go to step 2 else stop.

# Flynn's Classification

- SISD: Single Instruction Single Data
  - Classical Von-Neumann architecture
- SIMD: Single Instruction Multiple data
- MISD: Multiple Instructions Single Data
- MIMD: Multiple Instructions Multiple Data
  - Most common and general parallel machine

# Single Instruction Single Data stream

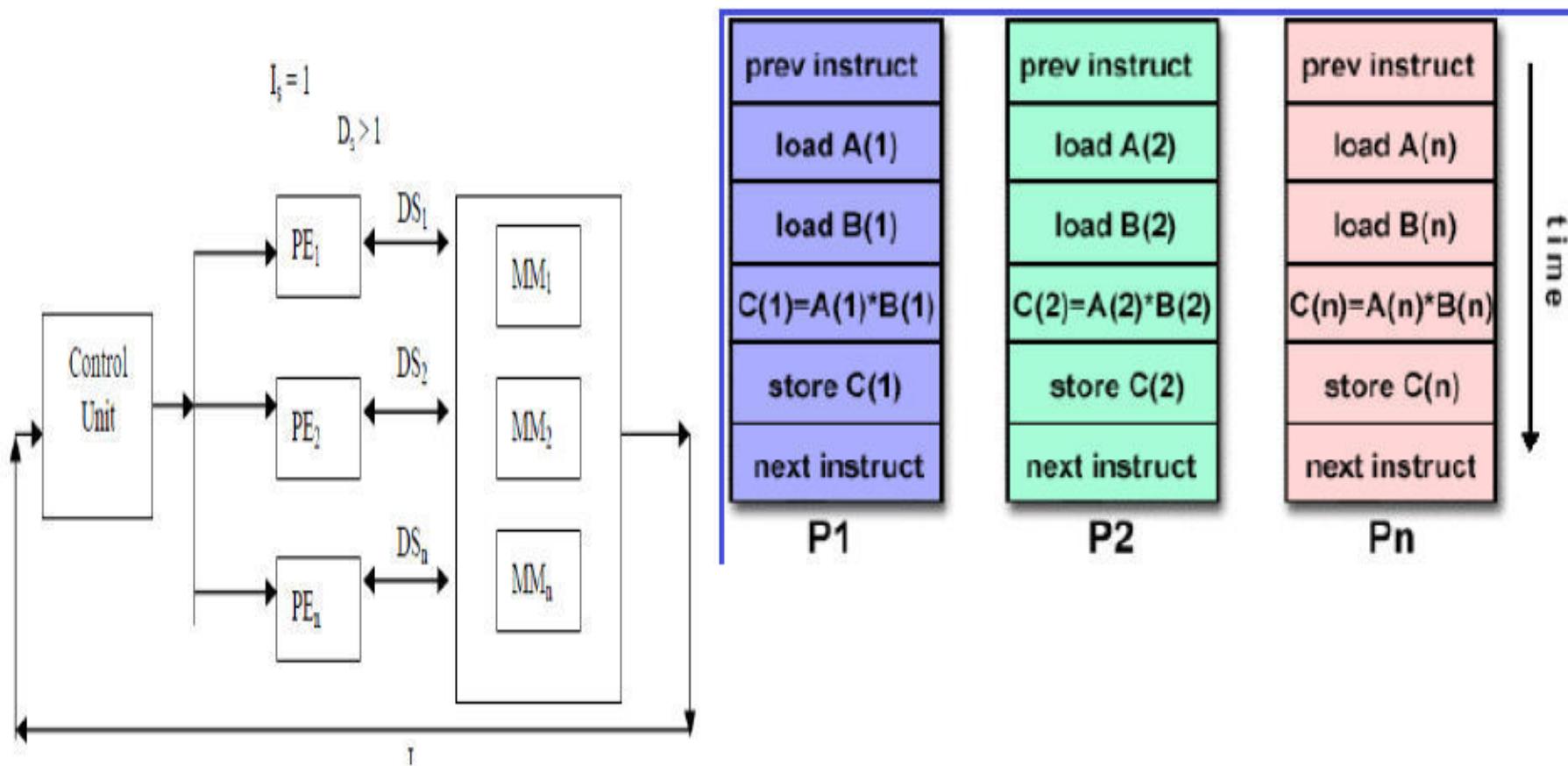
- A serial computer
- SISD machines are conventional serial computers that process only one stream of instructions and one stream of data.
- $I_s = D_s = 1$
- Examples
  - CDC 6600 which is unpipelined but has multiple functional units.
  - CDC 7600 which has a pipelined arithmetic unit.
  - Amdhal 470/6 which has pipelined instruction processing.
  - Cray-1 which supports vector processing.



# Single Instruction Multiple Data stream

- Multiple processing elements work under the control of a single control unit.
- It has one instruction and multiple data stream
- Main memory can also be divided into modules for generating multiple data streams acting as a distributed memory
- Examples of SIMD organisation are ILLIAC-IV, PEPE, BSP, STARAN, MPP, DAP and the Connection Machine (CM-1).
- A type of parallel computer
- Single instruction: All processing units execute the same instruction at any given clock cycle
- Multiple data: Each processing unit can operate on a different data element

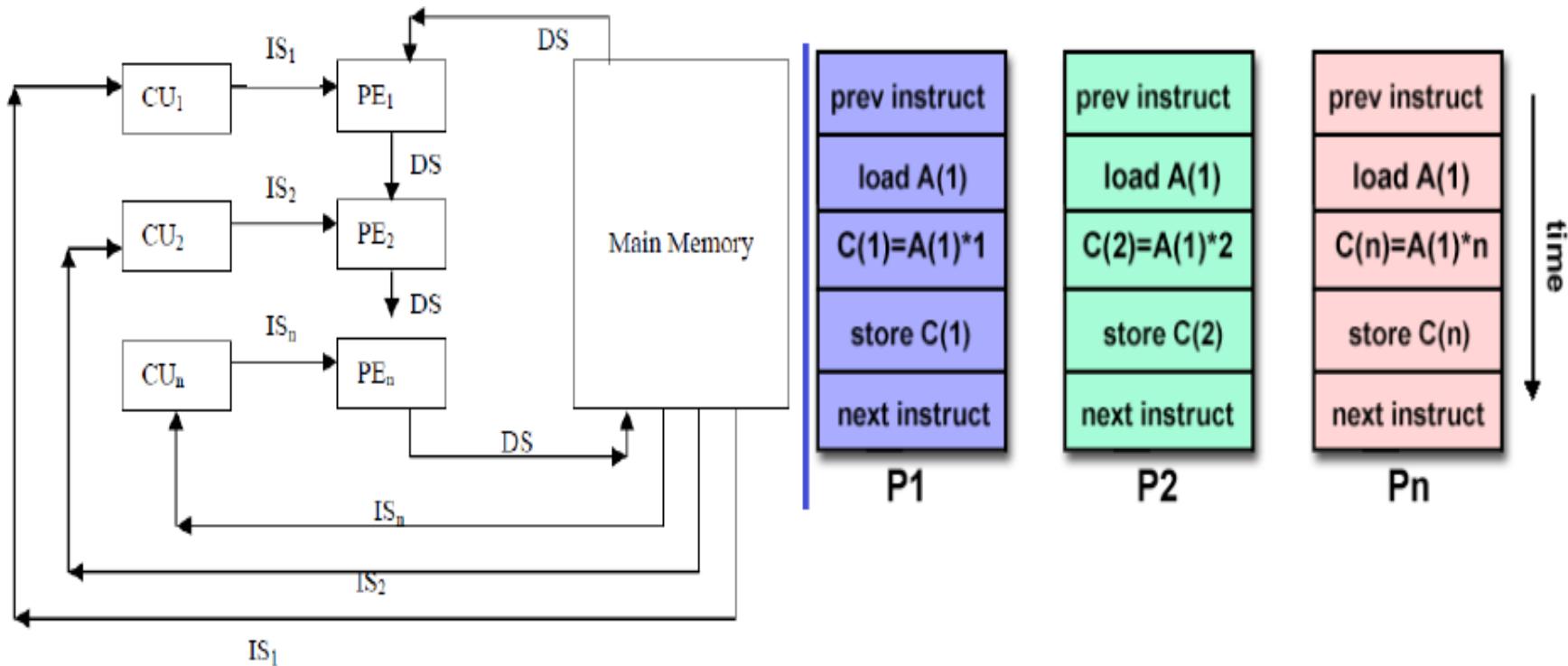
# SIMD



# Multiple Instruction Single Data stream

- Multiple processing elements are organised under the control of multiple control units.
- Each control unit is handling one instruction stream and processed through its corresponding processing element.
- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.
- But each processing element is processing only a single data stream at a time
- All processing elements are interacting with the common shared memory.

# MISD

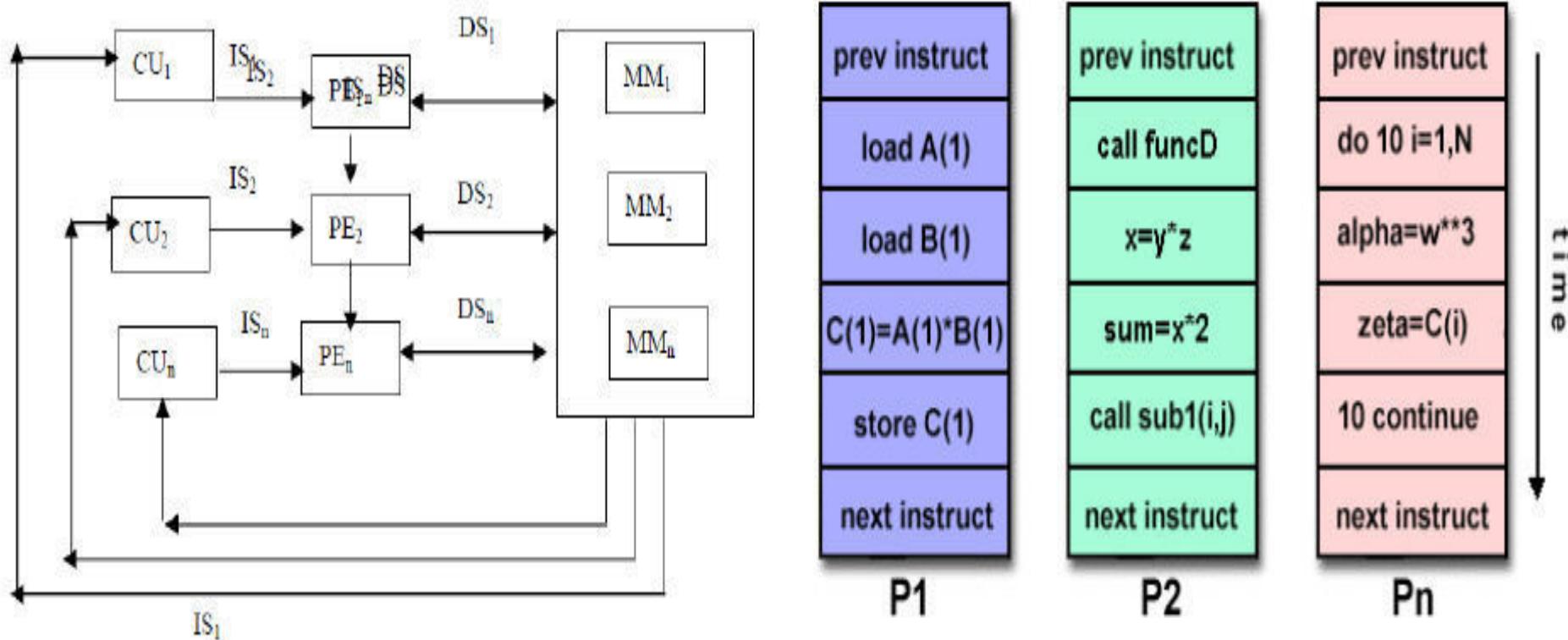


**Real time computers** need to be **fault tolerant** where several processors execute the same data for producing the redundant data. This is also known as **N-version programming**. All these redundant data are compared as results which should be same; otherwise faulty unit is replaced. Thus MISD machines can be applied to fault tolerant real time computers.

# Multiple Instruction Multiple Data stream

- Multiple instruction streams operate on multiple data streams
- The processors work on their own data with their own instructions.
- Tasks executed by different processors can start or finish at different times.
- This classification actually recognizes the parallel computer.
- Examples include; C.mmp, Burroughs D825, Cray-2, S1, Cray X-MP, HEP, Pluribus, IBM 370/168 MP, Univac 1100/80, Tandem/16, IBM 3081/3084, C.m\*, BBN Butterfly, Meiko Computing Surface (CS-1), FPS T/40000, iPSC.
- MIMD organization is the most popular for a parallel computer.
- In the real sense, parallel computers execute the instructions in MIMD mode.
- $I_s > 1, D_s > 1$

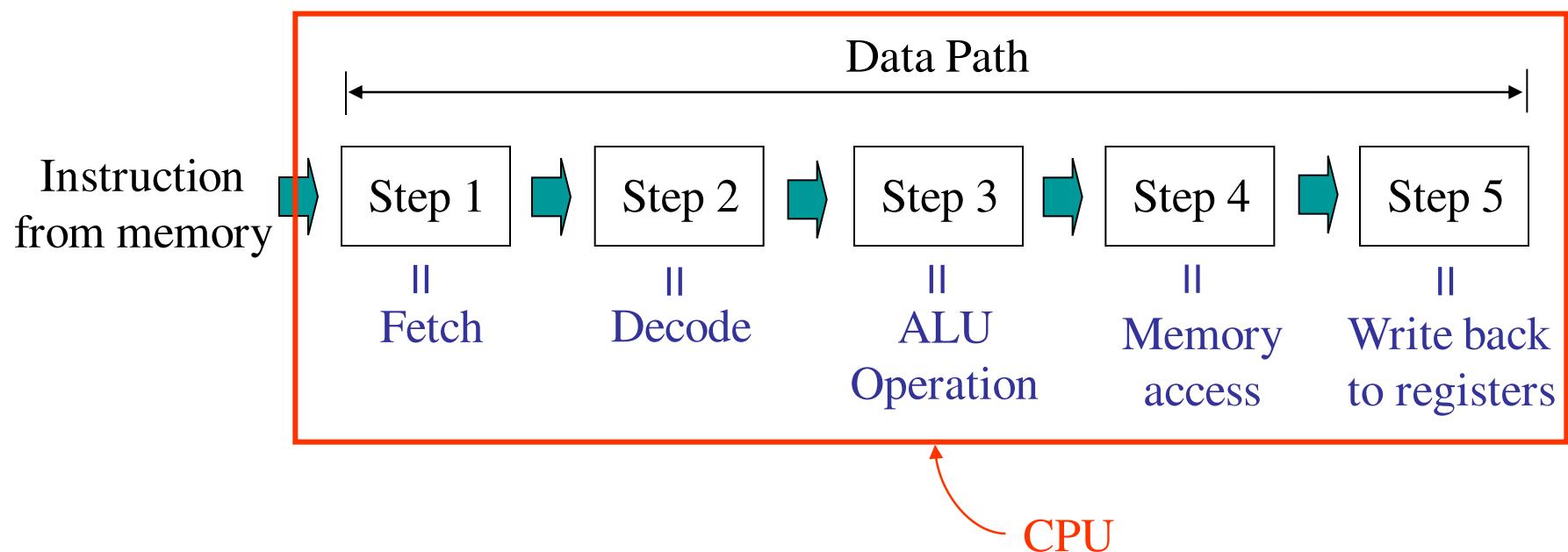
# MIMD



The way instructions are being  
executed inside a processor

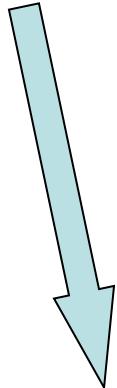
**Question:** how machine instructions are executed in a processor?

**Answer:** There are five steps to execute a machine instruction.



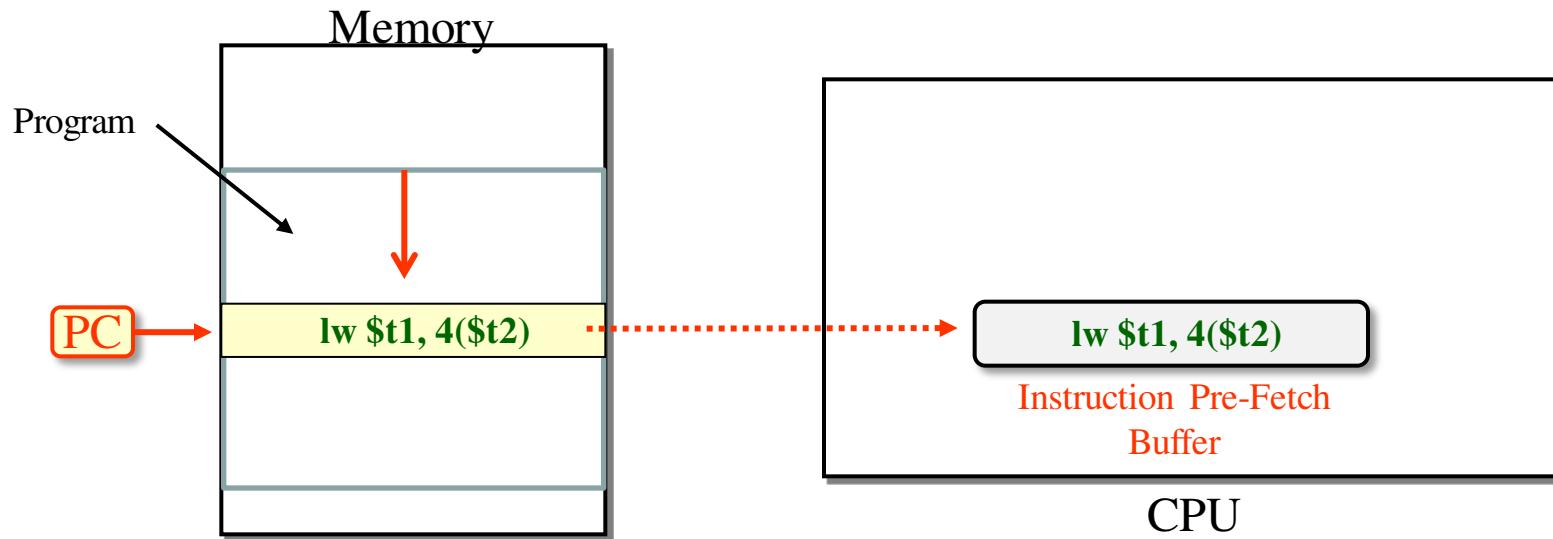
**Datapath:** a sequence of CPU circuits that execute CPU instructions step by step

**Example:** Data path for the “load” instruction



lb \$t1, 0(\$t2) # Load one byte from the memory pointed by t2 register

## Step 1: Instruction Fetch (IF)



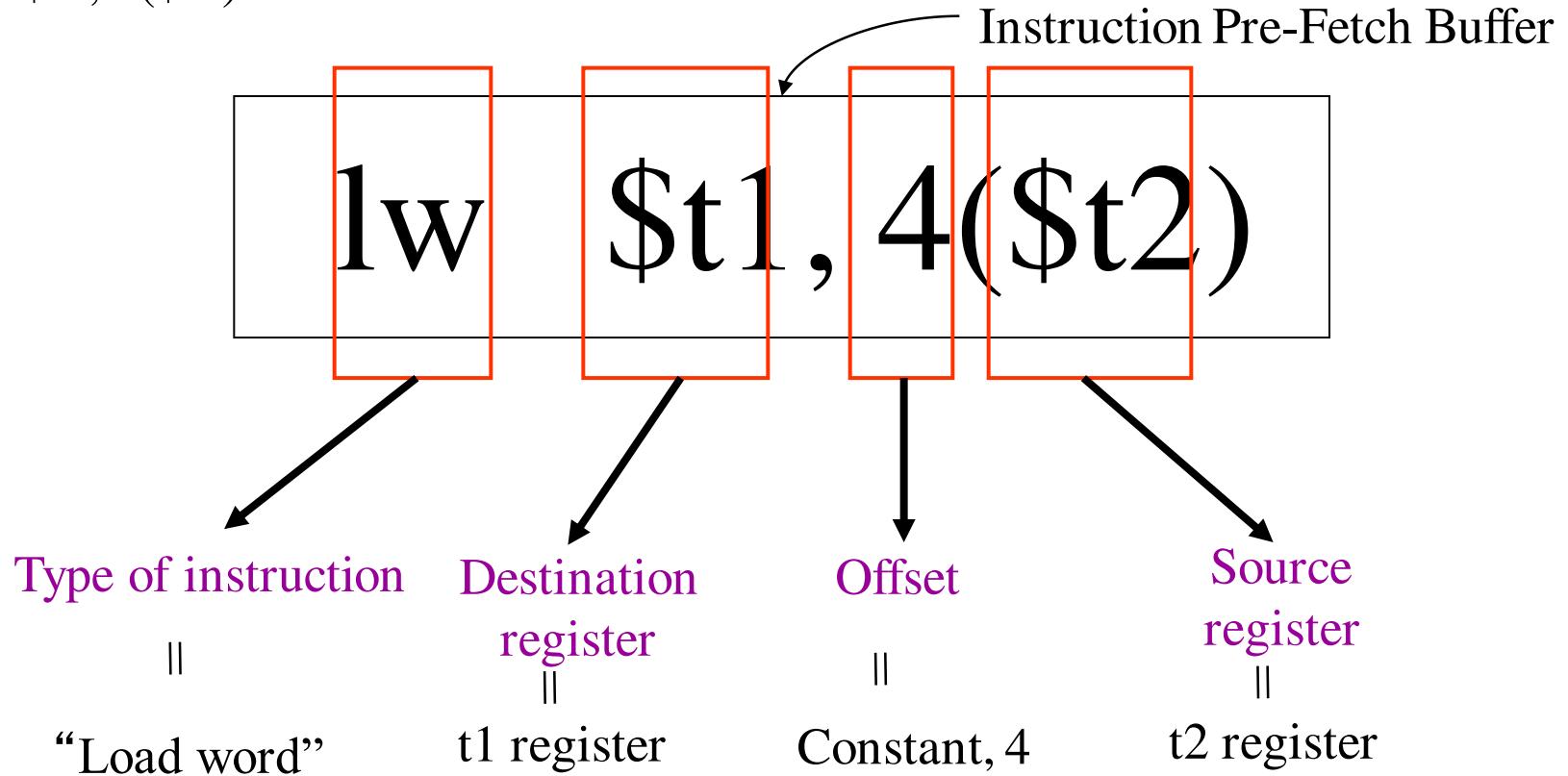
### Instruction Fetch (IF):

**Send out the PC and fetch the instruction from memory into the instruction register (IR); increment the PC (by 4) to address the next sequential instruction.**

**IR holds the instruction that will be used in the next stage.**

## Step 2: Instruction Decoding (ID)

lw \$t1, 4(\$t2):



### Instruction Decode/Register Fetch Cycle (ID):

Decode the instruction and access the register file to read the registers. The outputs of the general purpose registers are read into two temporary registers (A & B) for use in later clock cycles.

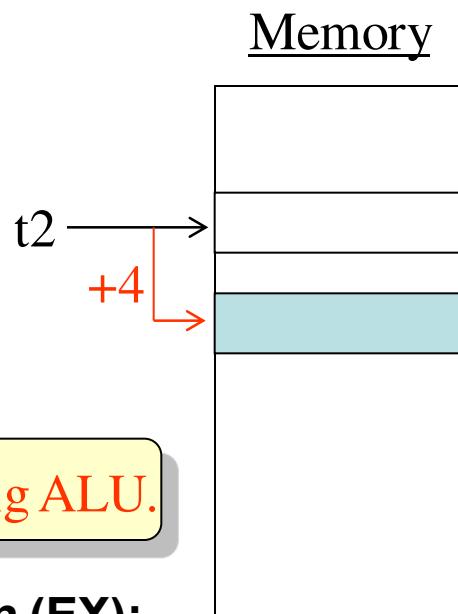
We extend the sign of the lower 16 bits of the Instruction Register.

### Step 3: ALU Operation (EX)

lw \$t1, 4(\$t2):

#### OPERATIONS

- (1) Memory pointed by  $t2$
- (2) +4 bytes offset



We need to calculate  $t2 + 4$  using ALU.

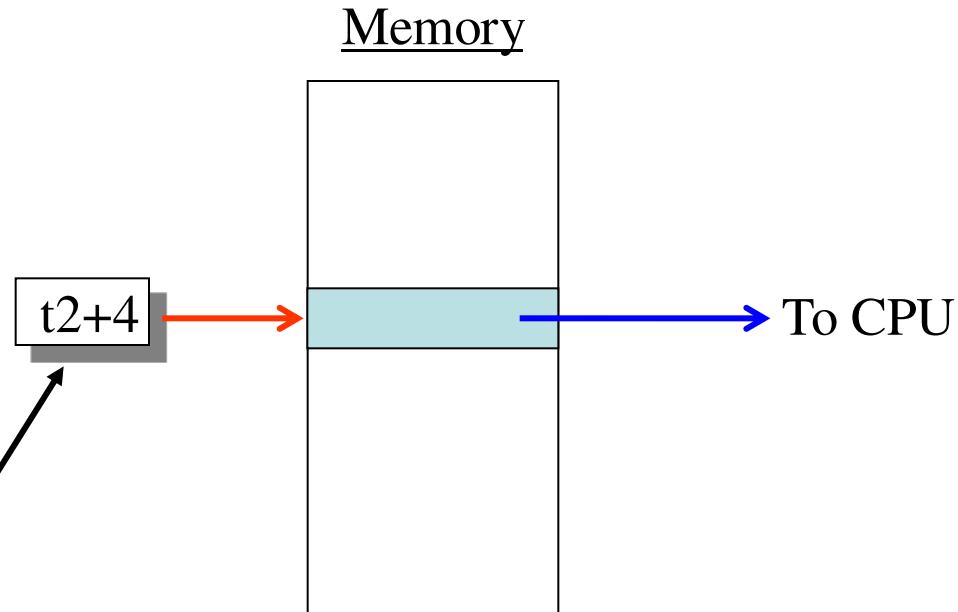
#### Execute Address Calculation (EX):

We perform an operation (for an ALU) or an address calculation (if it's a load or a Branch).

If an ALU, actually do the operation. If an address calculation, figure out how to obtain the address and stash away the location of that address for the next cycle.

## Step 4: Memory Access (ME)

lw \$t1, 4(\$t2):

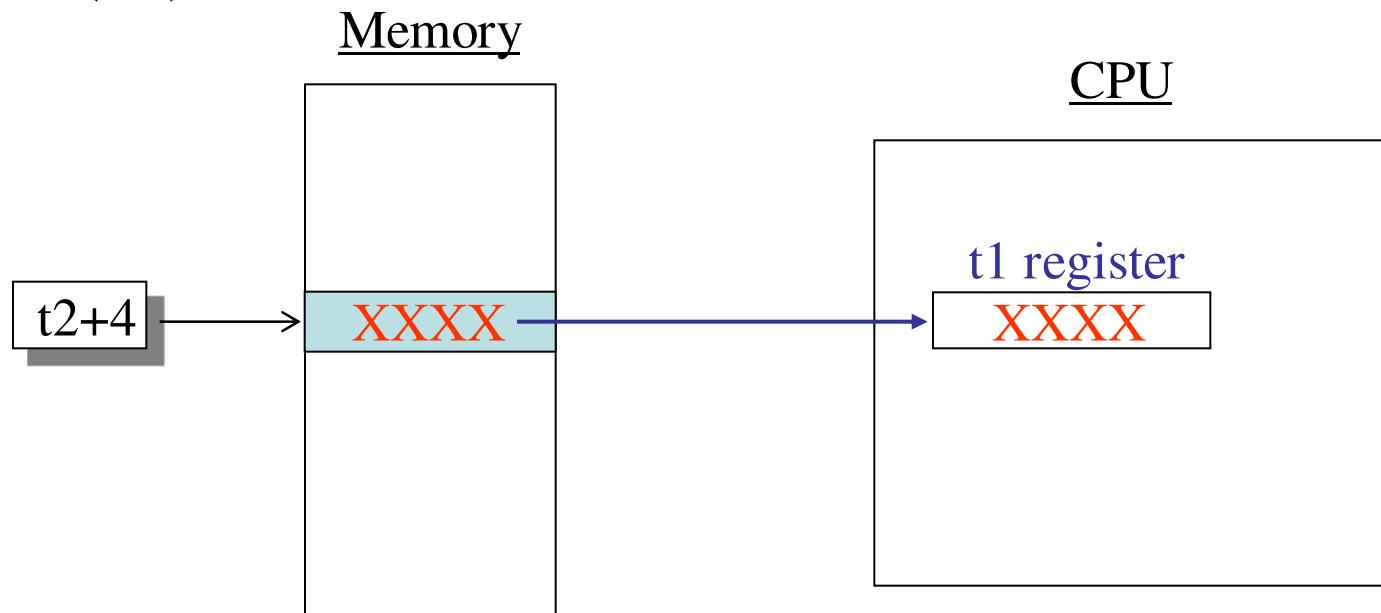


Calculated by the previous step (= Step 3)

**MEMORY ACCESS (MEM):**  
If this is an ALU, do nothing.  
If a load or store, then access memory.

## Step 5: Output to the register (WB)

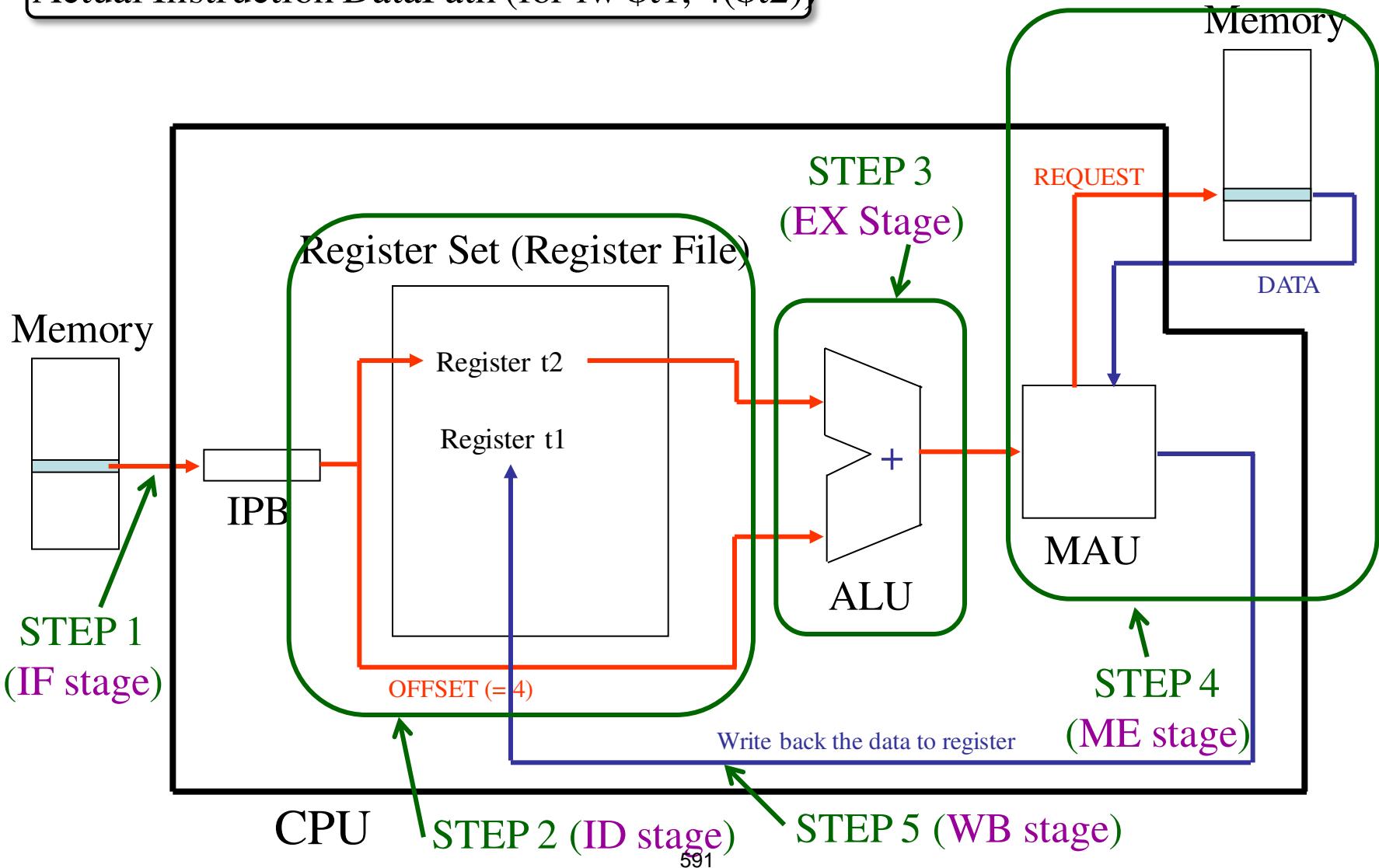
lw \$t1, 4(\$t2):



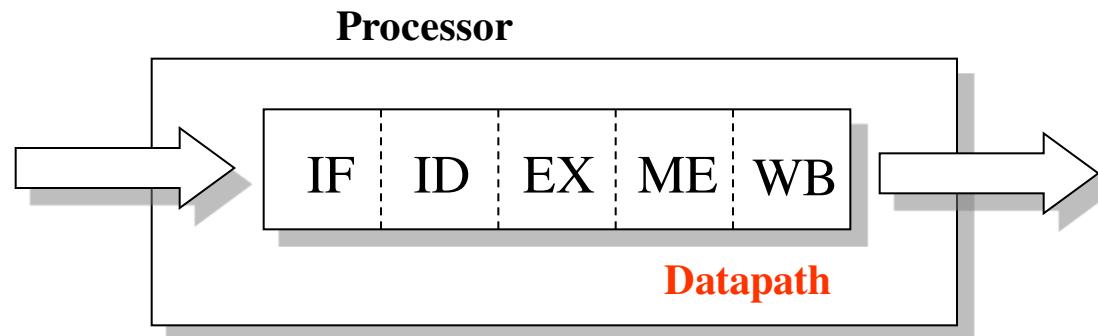
### WRITE BACK (WB):

Update the registers from either the ALU or from the data loaded.

## Actual Instruction DataPath (for lw \$t1, 4(\$t2))



- Datapath executes processor instruction
- The essential datapath phases are: IF, ID, EX, ME, and WB
- Each datapath phase requires a processor one processor clock cycle

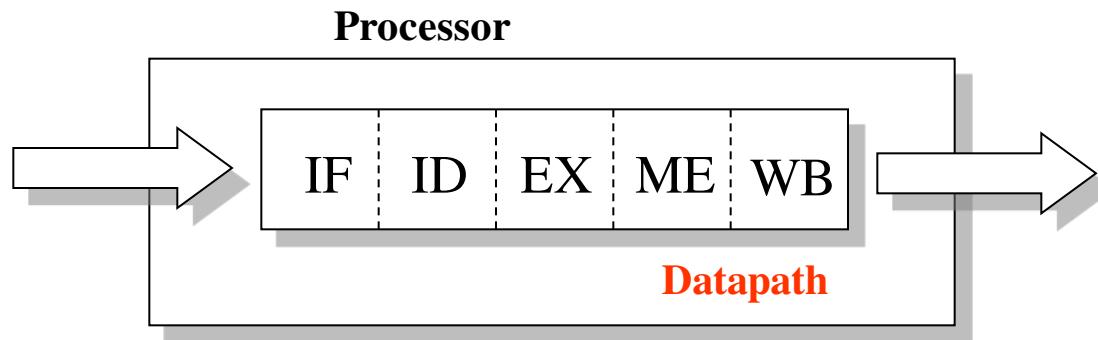


**IF:** Instruction Fetch    **ID:** Instruction Decode    **EX:** Execution

**ME:** Memory access    **WB:** Write Back to registers

## 1. Scalar Datapath Processors

- The datapath includes the five circuit units
- All five units are implemented as single monolithic unit
- When an instruction is being executed, no other instruction can enter the datapath



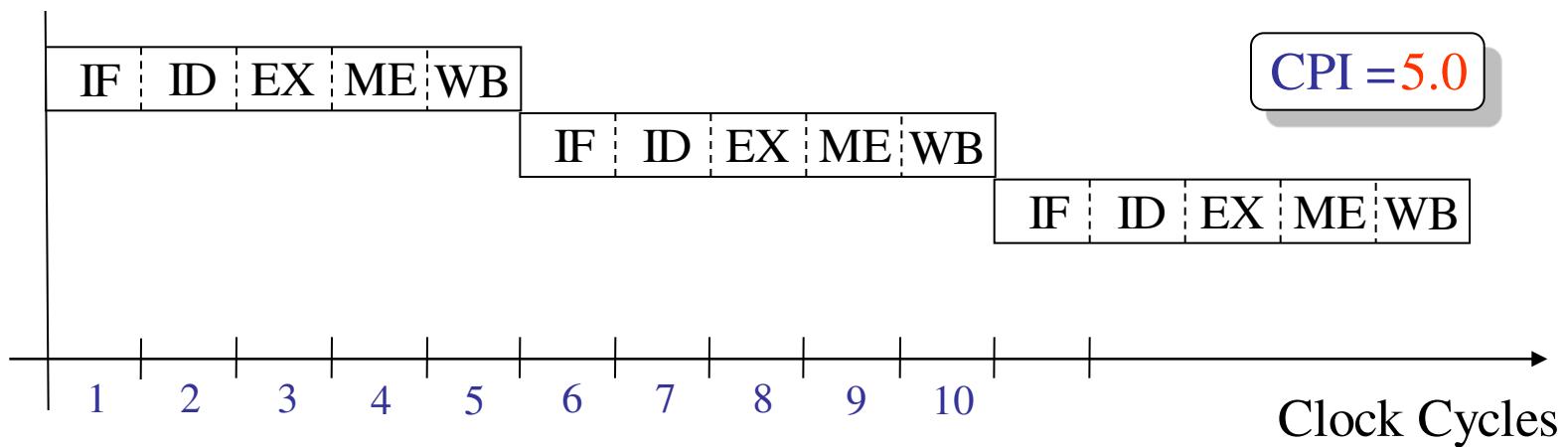
**IF:** Instruction Fetch    **ID:** Instruction Decode    **EX:** Execution

**ME:** Memory access    **WB:** Write Back to registers

## 1. Scalar Datapath Processors

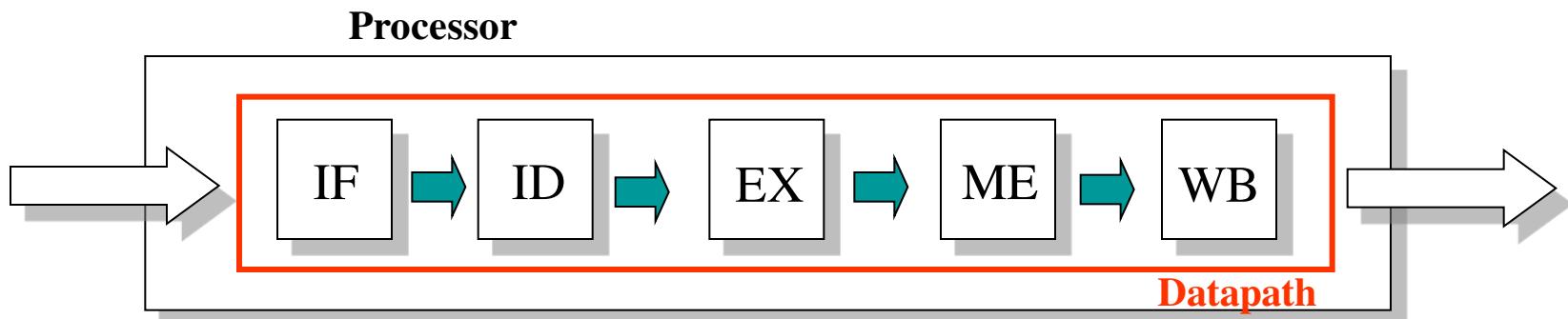
Representative processors in this generation

i4004, i8080, i8086, i80816, Z80, MC68000



## 2. Pipeline Datapath Processors

- All five units are implemented as independent units
- When an instruction is completed in a unit, the instruction can be forwarded to the next unit
- All five units can be occupied by different instructions



## 2. Pipeline Datapath Processors (continued)

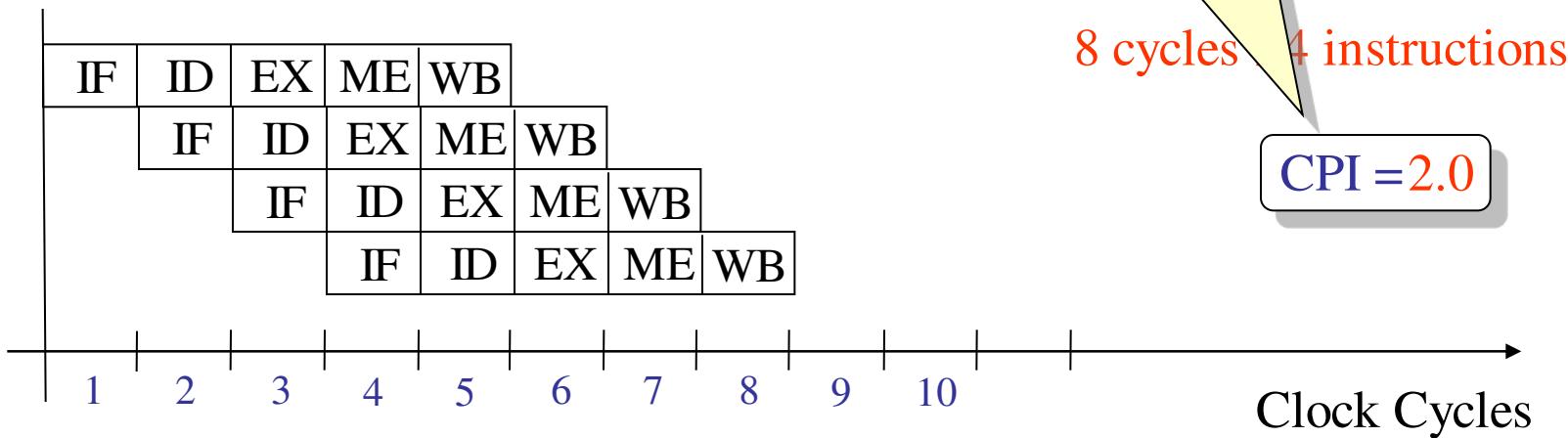
Representative processors in this generation

i80386, i40846, MC68040,

CPI for these only for  
these four instructions

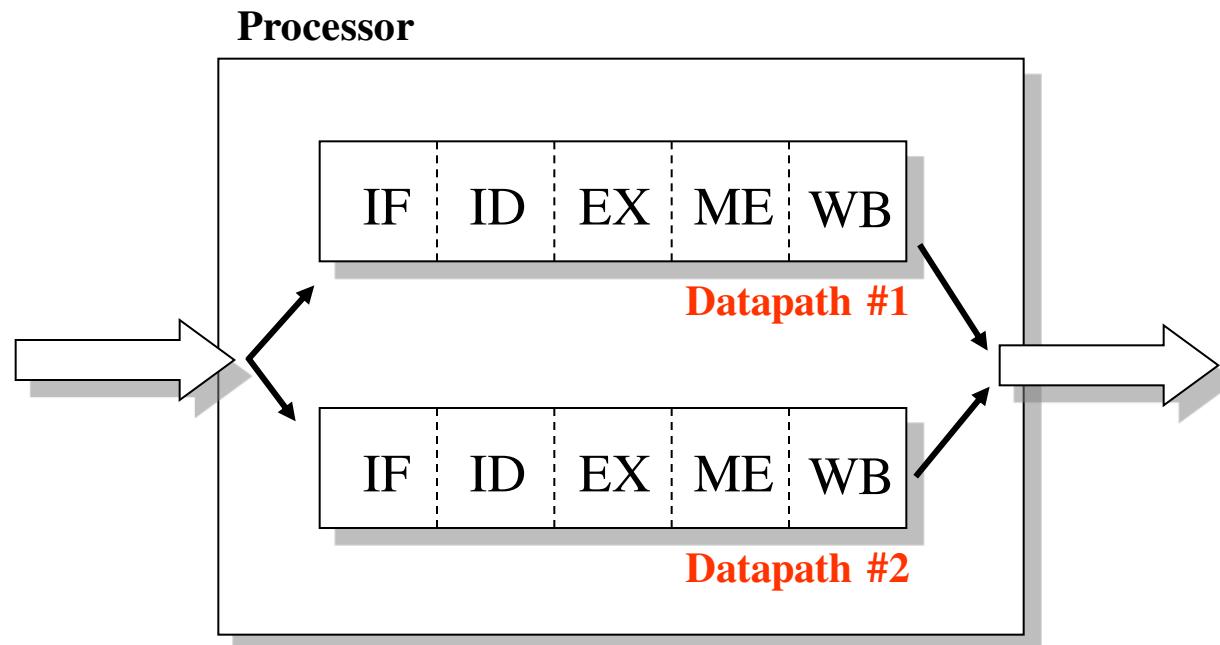
8 cycles 4 instructions

CPI = 2.0

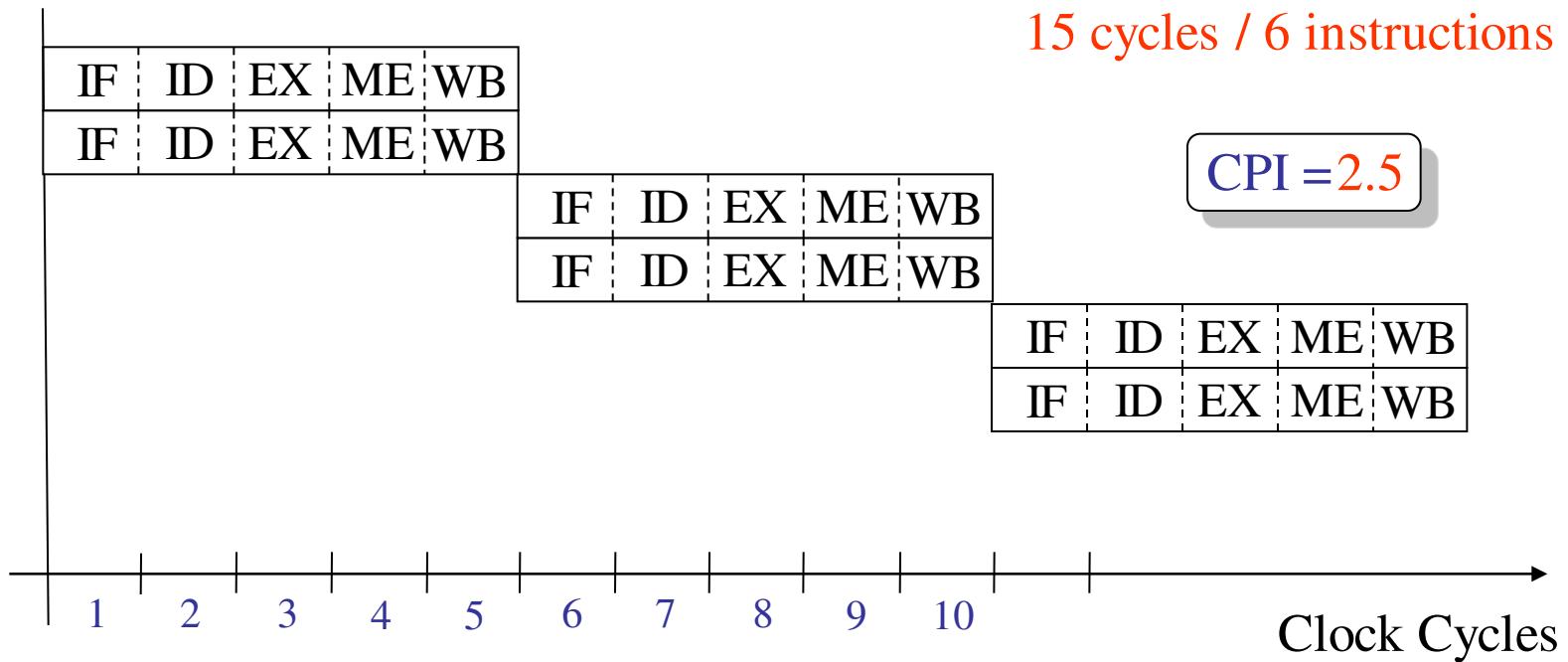


### 3. Super-Scalar Datapath Processors

- Multiple Scalar Datapath

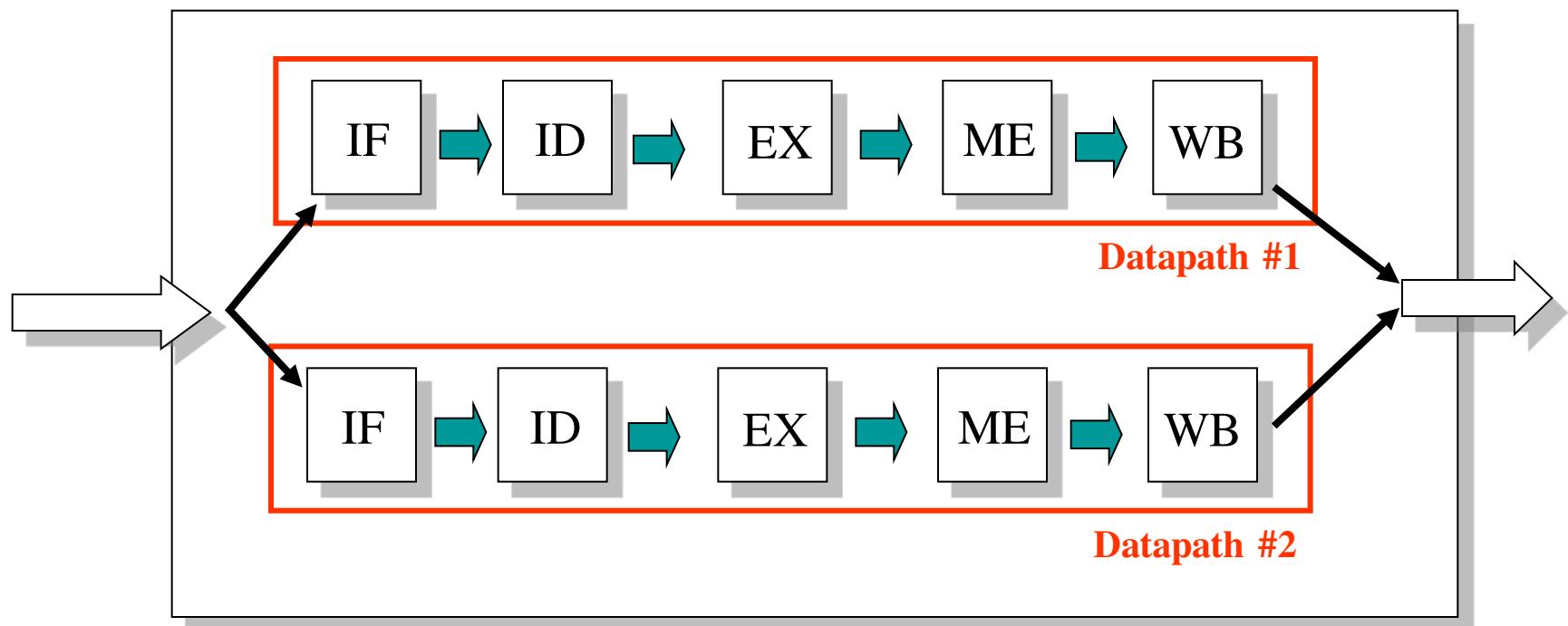


### 3. Super-Scalar Datapath Processors (continued)



## 4. Super-Pipeline Datapath Processors

- A combination of super-scalar and pipeline Processor



## 4. Super-Pipeline Datapath Processors (continued)

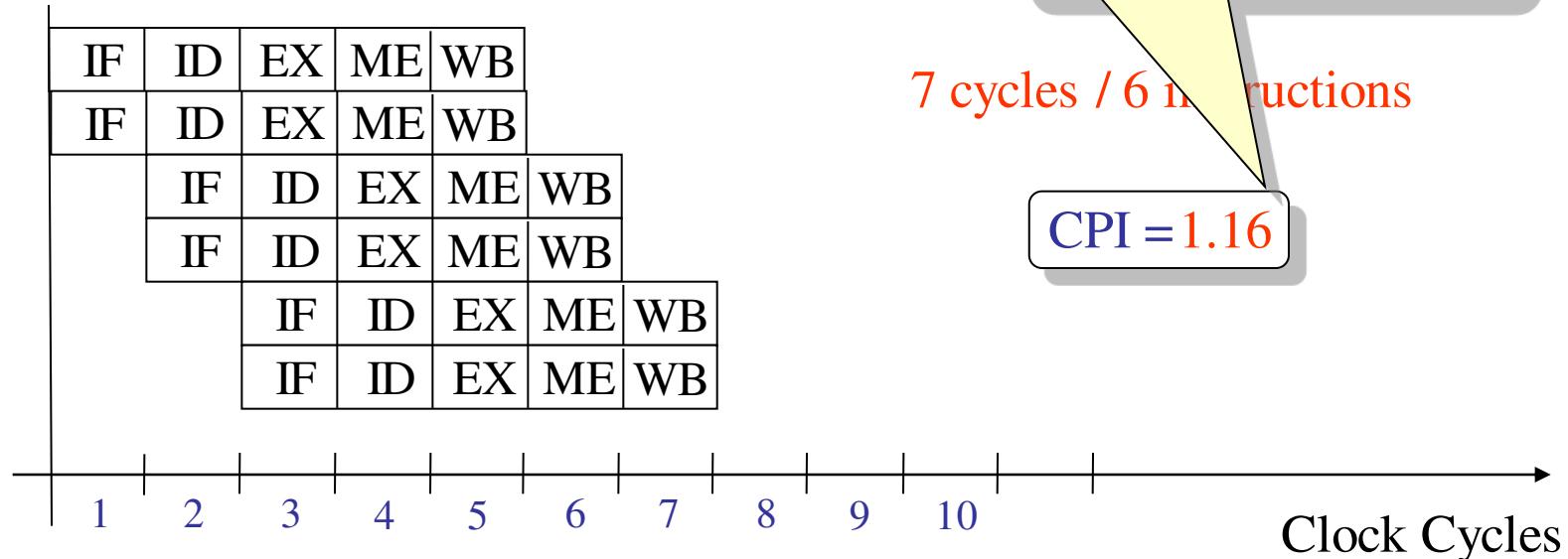
Representative processors in this generation

Pentiums (54, P55C), MIPS

CPI for these only for  
these six instructions

7 cycles / 6 instructions

CPI = 1.16



# SPEED UP IN PIPELINE

- **For a pipeline processor:**

- *k-stage* pipeline processes with a clock cycle time  $t_p$  is used to execute  $n$  tasks.
- The time required for the first task  $T_1$  to complete the operation =  $k * t_p$  (if  $k$  segments in the pipe)
- The time required to complete  $(n-1)$  tasks =  $(n-1) * t_p$
- Therefore to complete  $n$  tasks using a  $k$ -segment pipeline requires  $= k + (n-1)$  clock cycles.

- **For the non-pipelined processor :**

- Time to complete each task =  $t_n$
- Total time required to complete  $n$  tasks =  $n * t_n$
- Speed up = non pipelining processing/pipelining processing

$$S = \frac{T_1}{T_K} = \frac{n t_n}{(k + (n - 1)) t_p}$$

- As the number of tasks increases,  $n$  becomes much larger than  $k-1$ , and approaches the value of  $n$ .
- Under this condition, speed up becomes
- $S = t_n / t_p$
- Assume the time taken for pipeline and non pipeline circuits are same then  $t_n = k * t_p$
- Speed up reduces to  $= S = (k * t_p) / t_p = k$
- This shows that the theoretical maximum speedup that a pipeline can provide is  $k$ , where  $k$  is the number of segments in the pipeline.

# Speed up with pipeline

- **For a pipeline processor:**

- *k-stage* pipeline processes with a clock cycle time  $t_p$  is used to execute  $n$  tasks.
- The time required for the first task  $T_1$  to complete the operation =  $k*t_p$  (if  $k$  segments in the pipe)
- The time required to complete  $(n-1)$  tasks =  $(n-1) * t_p$
- Therefore to complete  $n$  tasks using a  $k$ -segment pipeline requires =  $k + (n-1)$  clock cycles.

- **For the non-pipelined processor :**

- Time to complete each task =  $t_n$
- Total time required to complete  $n$  tasks =  $n*t_n$
- Speed up = non pipelining processing/pipelining processing

$$S = \frac{T_1}{T_K} = \frac{nt_n}{(k + (n - 1))t_p}$$

## Pipeline Hazards

Any anomalies that prevent a pipeline datapath from functioning without pipeline stalls or flashes

## Three Major Pipeline Hazards

1. Structural Hazards
2. Data Hazards
3. Control Hazards

## 1. Structural Hazards:

= pipeline hazards due to hardware resource conflicts

**Example** Instruction fetch and data access

A processor instruction = [operator] + [parameters]

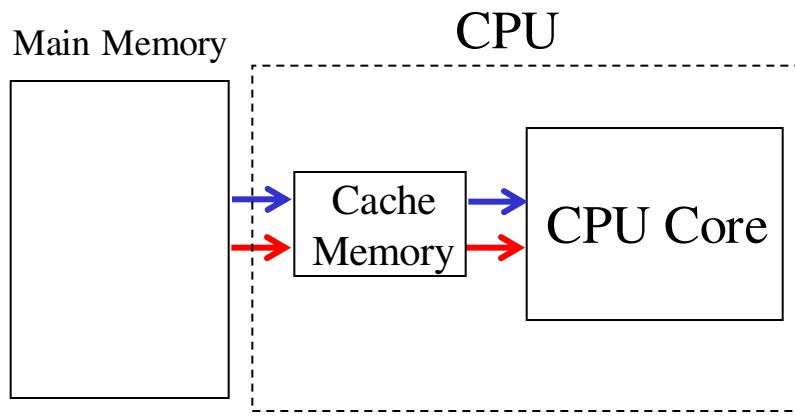
processor instruction: `sw $t1, 0($t2)` //  $\$t1 \rightarrow \text{Mem}[\$t2+0]$

① instruction fetch  
② save register to memory

Execution of this instruction requires two memory accesses

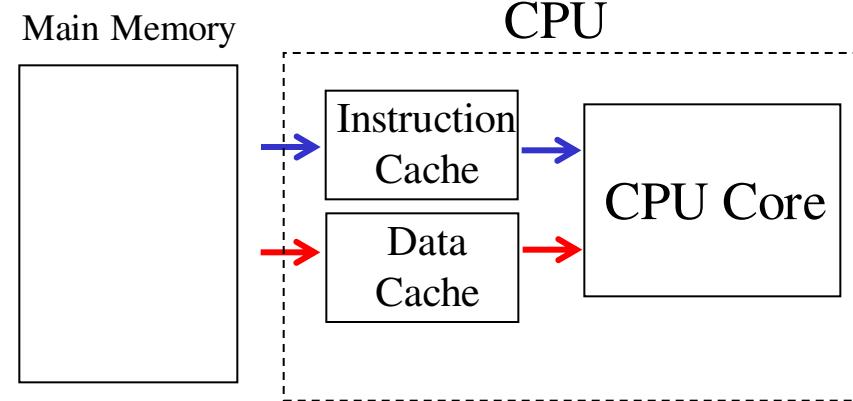
## 1. Structural Hazards (continued)

### Unified Cache Architecture

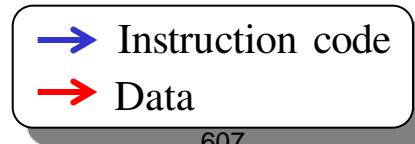


Instruction code and data  
need to take the same path  
(i486)

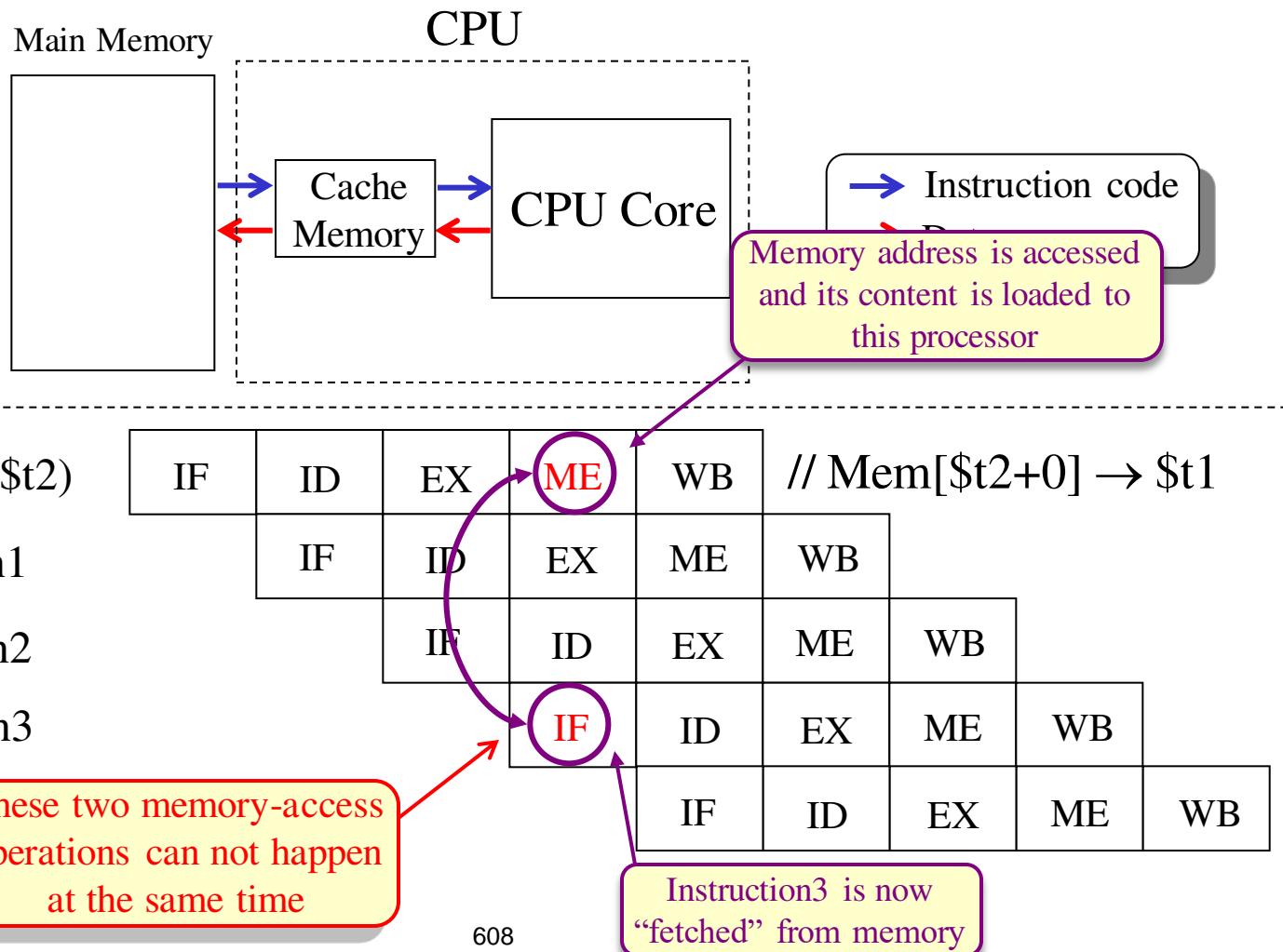
### Split Cache Architecture



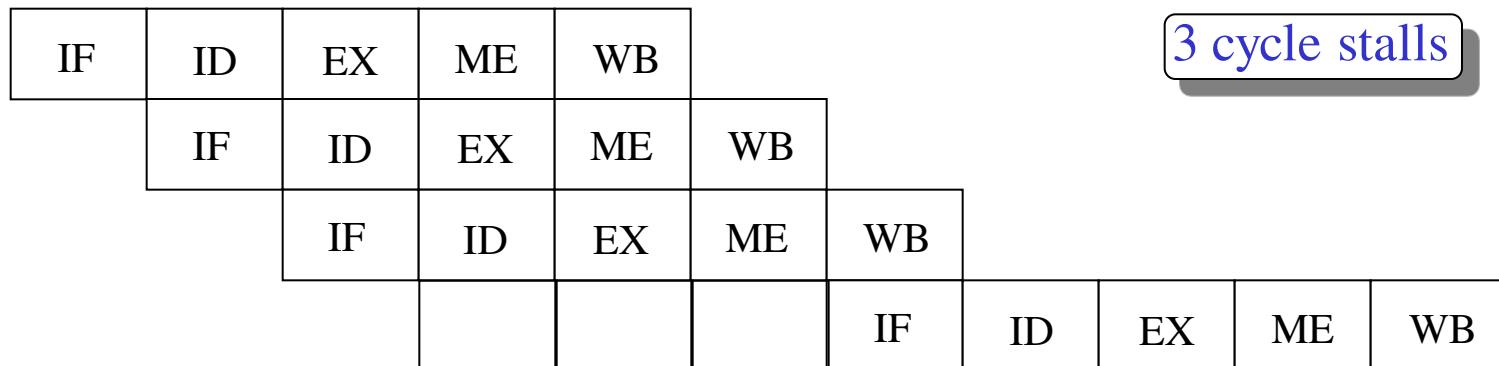
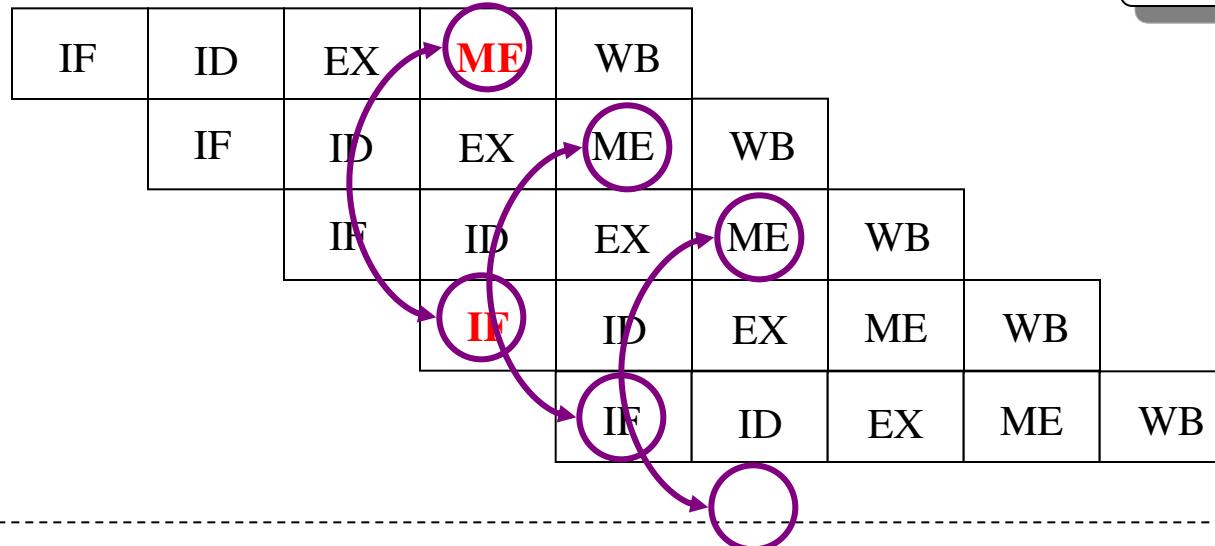
Instruction code and data  
take its own path  
(Pentium and later)



## Unified Cache Architecture

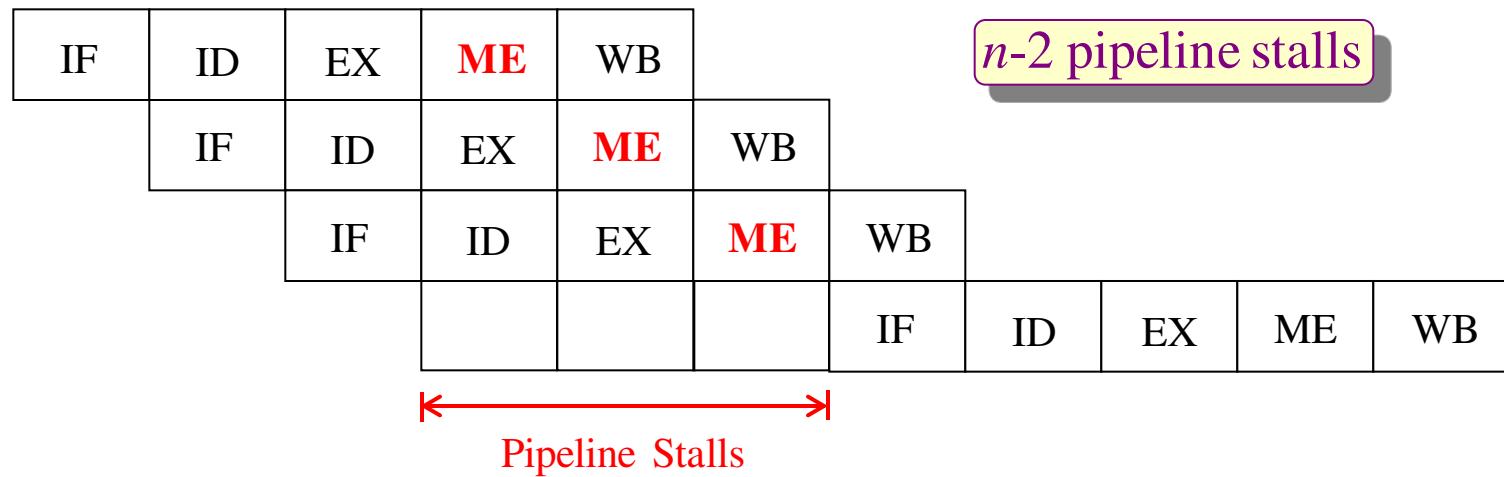


Resource Conflict



Benefits from pipeline execution is lost

## Worst case structural pipeline hazards



## 2. Data Hazards:

= pipeline hazards due to data-dependency between instructions

### Example

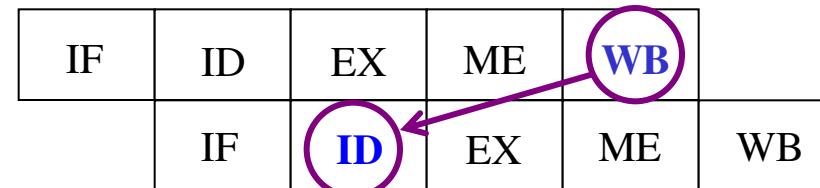
•  
 move \$t1, \$t2 //  $t1 \leftarrow t2$   
 add \$t3, \$t1, 5 //  $t3 \leftarrow t1 + 5$   
 •  
 •

The same registers are used in more than one instruction

### Data Dependency

This can not be done

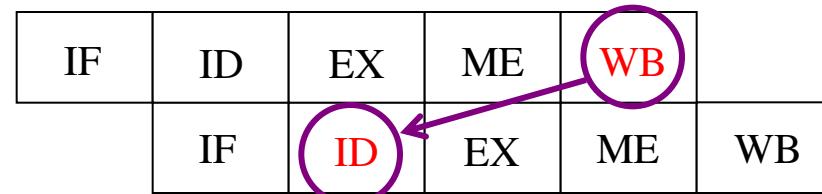
move \$t1, \$t2:



add \$t3, \$t1, 5:

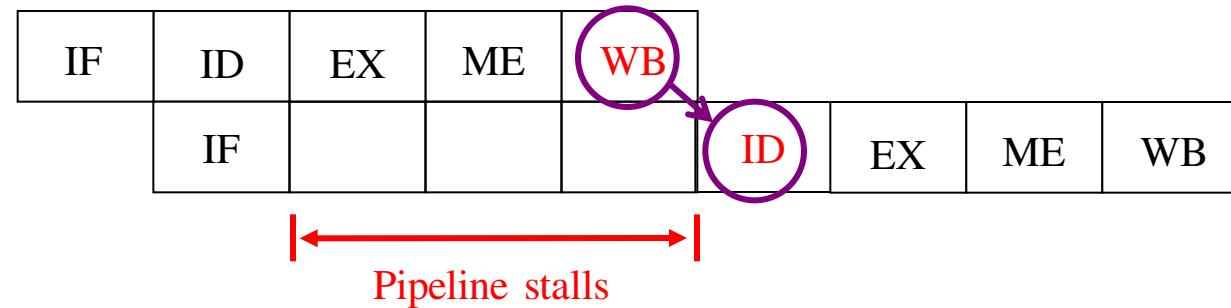
## Data Dependency

move \$t1, \$t2:  
add \$t3, \$t1, 5:



## Resolve data dependency by stalls

move \$t1, \$t2:  
add \$t3, \$t1, 5:



# Data Hazards

Execution Order is:

Instr<sub>I</sub>  
Instr<sub>J</sub>

## Read After Write (RAW)

Instr<sub>J</sub> tries to read operand before Instr<sub>I</sub> writes it

I: add r1, r2, r3  
J: sub r4, r1, r3

- Known as true dependency.

# Data Hazards

## Write After Read (WAR)

- i1.  $R4 \leftarrow R1 + R5$   
i2.  $R5 \leftarrow R1 + R2$
- In any situation with a chance that i2 may finish before i1 (i.e., with concurrent execution), it must be ensured that the result of register R5 is not stored before i1 has had a chance to fetch the operands.
  - Called an “anti-dependence”.

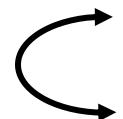
# Data Hazards

Execution Order is:

Instr<sub>I</sub>  
Instr<sub>J</sub>

## Write After Write (WAW)

Instr<sub>J</sub> tries to write operand before Instr<sub>I</sub> writes it  
– Leaves wrong result ( Instr<sub>I</sub> not Instr<sub>J</sub> )



I: sub r1, r4, r3  
J: add r1, r2, r3  
K: mul r6, r1, r7

- Called an “output dependence”.

### 3. Control Hazards:

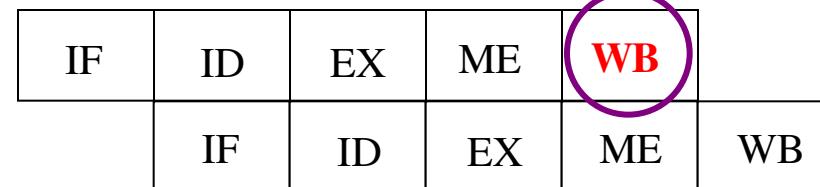
= pipeline hazards due to branch and jump instructions

Example: jump instructions

	bne \$s0, \$s1, Next instruction 1
	• • •
Next:	instruction 2

Pipeline flashes for control hazards

bne \$s0, \$s1, NEXT  
instruction 1



PC is set at the end of WB for “bne” instruction

We can not start this instruction here

*Flushing the pipeline* occurs when a branch instruction jumps to a new memory location, invalidating all prior stages in the pipeline. These prior stages are cleared, allowing the pipeline to continue at the new instruction indicated by the branch

### 3. Control Hazards:

= pipeline hazards due to branch and jump instructions

Example: jump instructions

	bne \$s0, \$s1, Next instruction 1
	• • •
Next:	instruction 2

Pipeline flashes for control hazards

bne \$s0, \$s1, Next  
instruction 2



Start fetching instruction 1

Pipeline Flushing

## Reducing pipeline hazards - three techniques

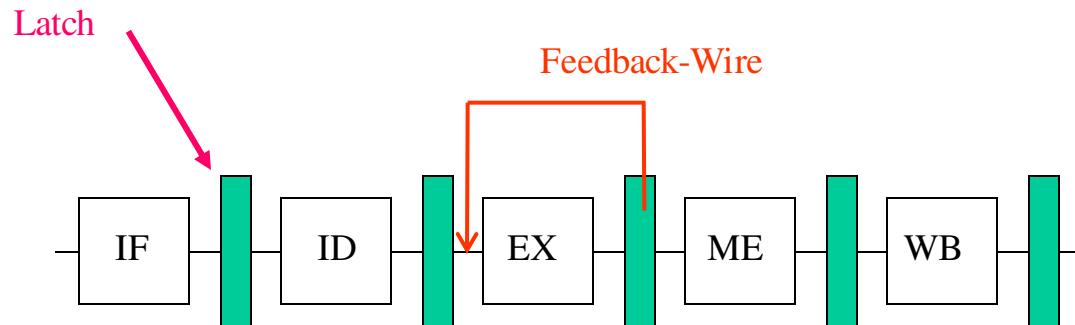
Three techniques for different types of pipeline hazards

1. **Forwarding** – for reducing RAW data dependencies
2. **Instruction Scheduling** – for reducing data hazards
3. **Delayed Branch** – for reducing control hazards

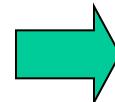
# Reducing pipeline hazards - three techniques

## Technique 1: Forwarding

= Internal pipeline circuit to feedback outputs of a stage



Outputs from a pipeline stage can be fed to the same or different stages of another instruction



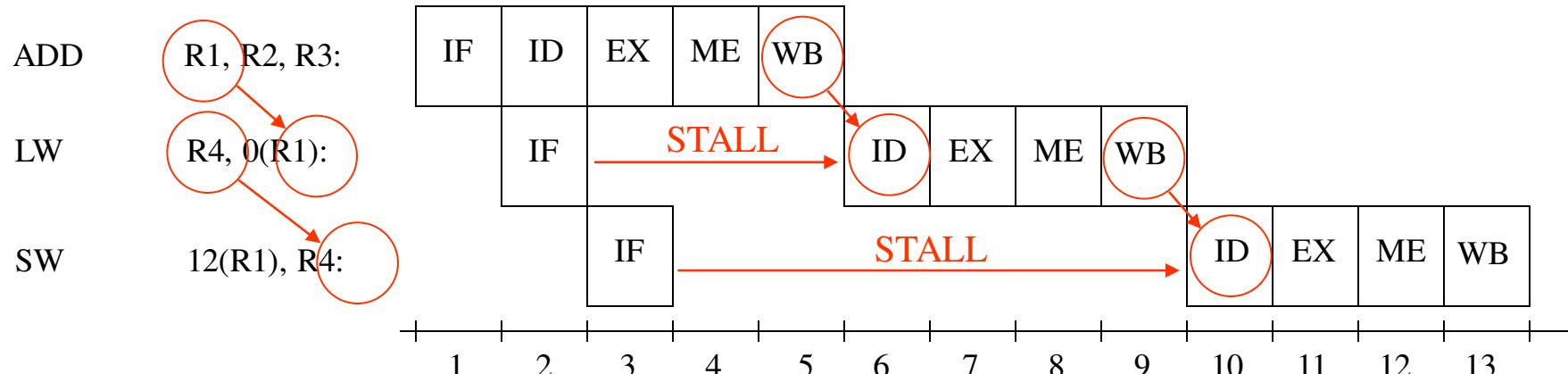
Need hardware support

# Reducing pipeline hazards - three techniques

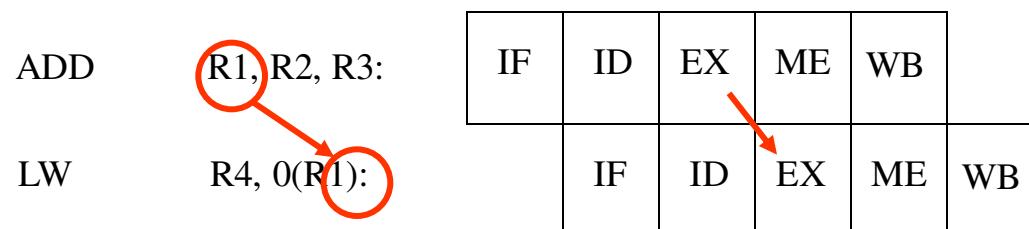
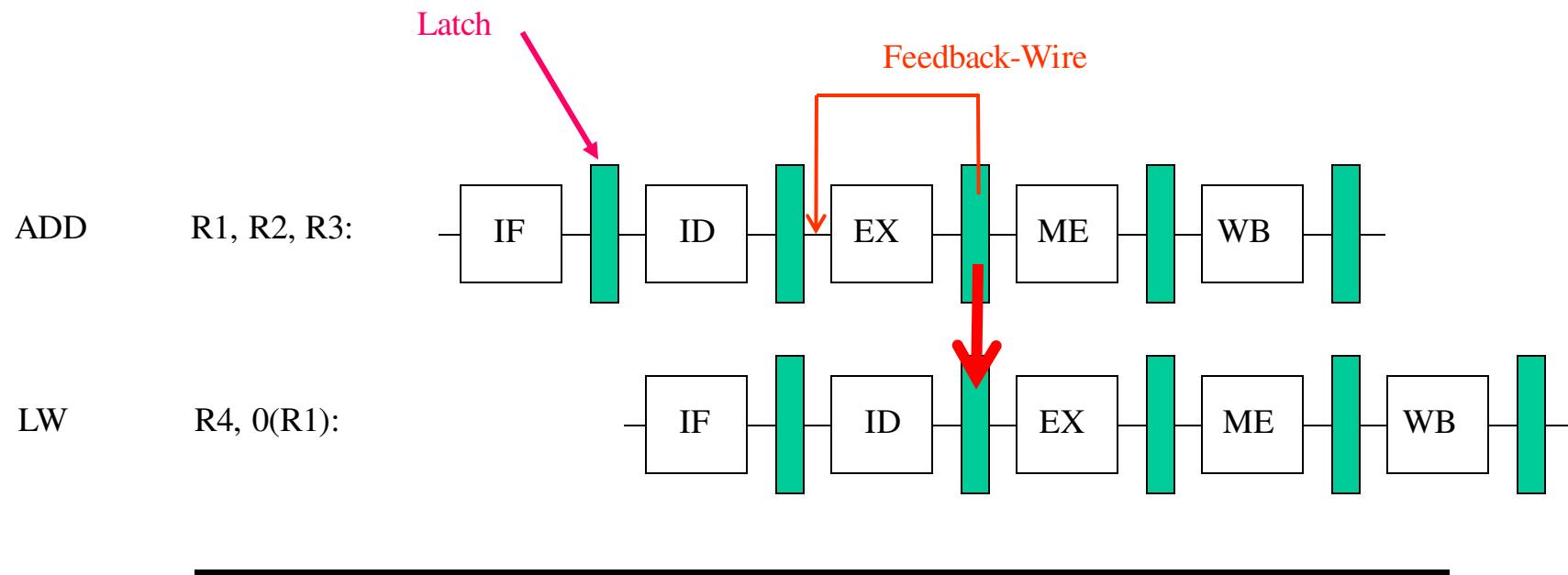
## Example

ADD	R1, R2, R3	// R1 = R2 + R3
LW	R4, 0(R1)	// R4 ← MEM [R1 + 0]
SW	12(R1), R4	// MEM [R1+12] ← R3

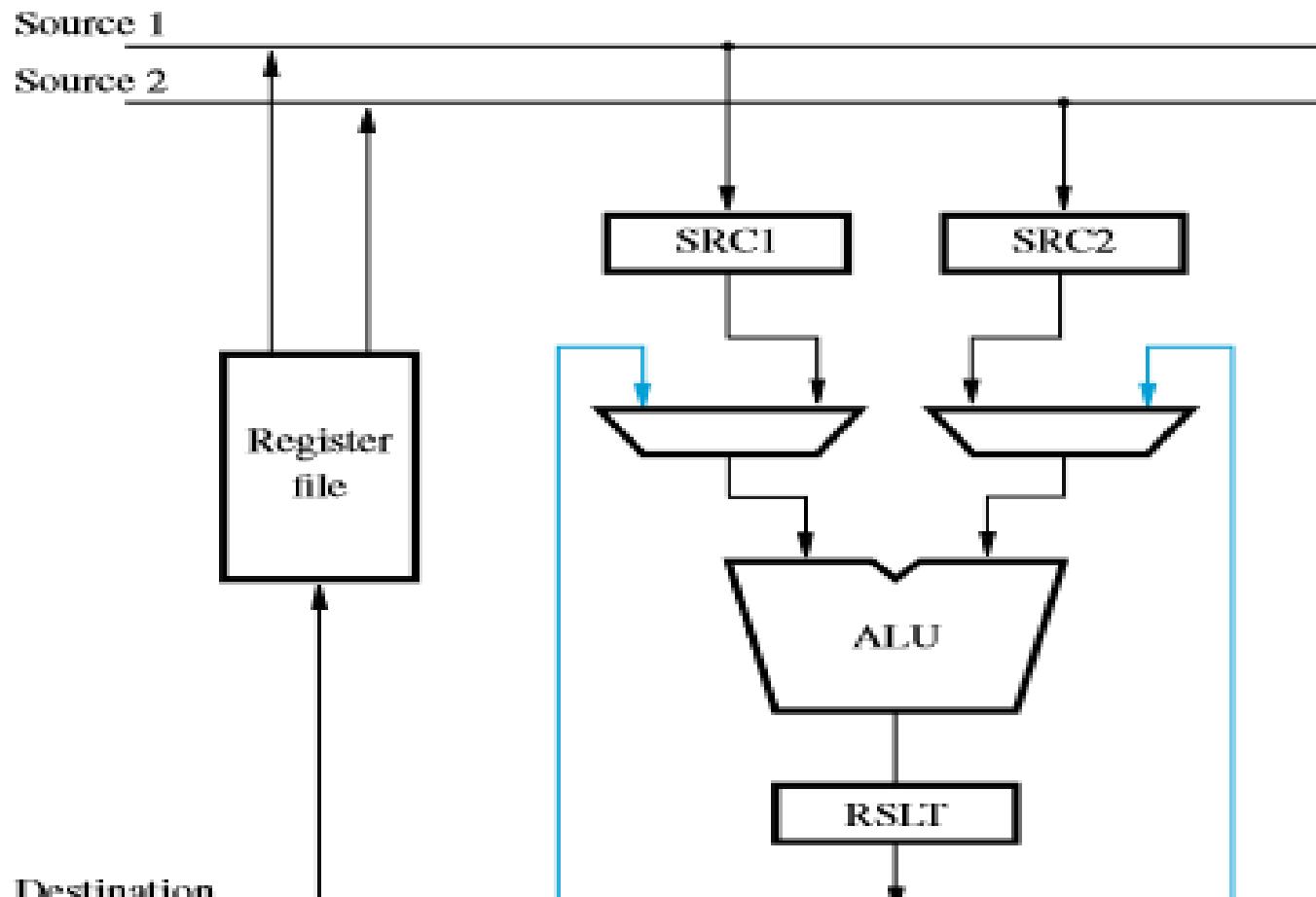
Pipeline time chart for an ordinary pipeline processor



# Reducing pipeline hazards - three techniques

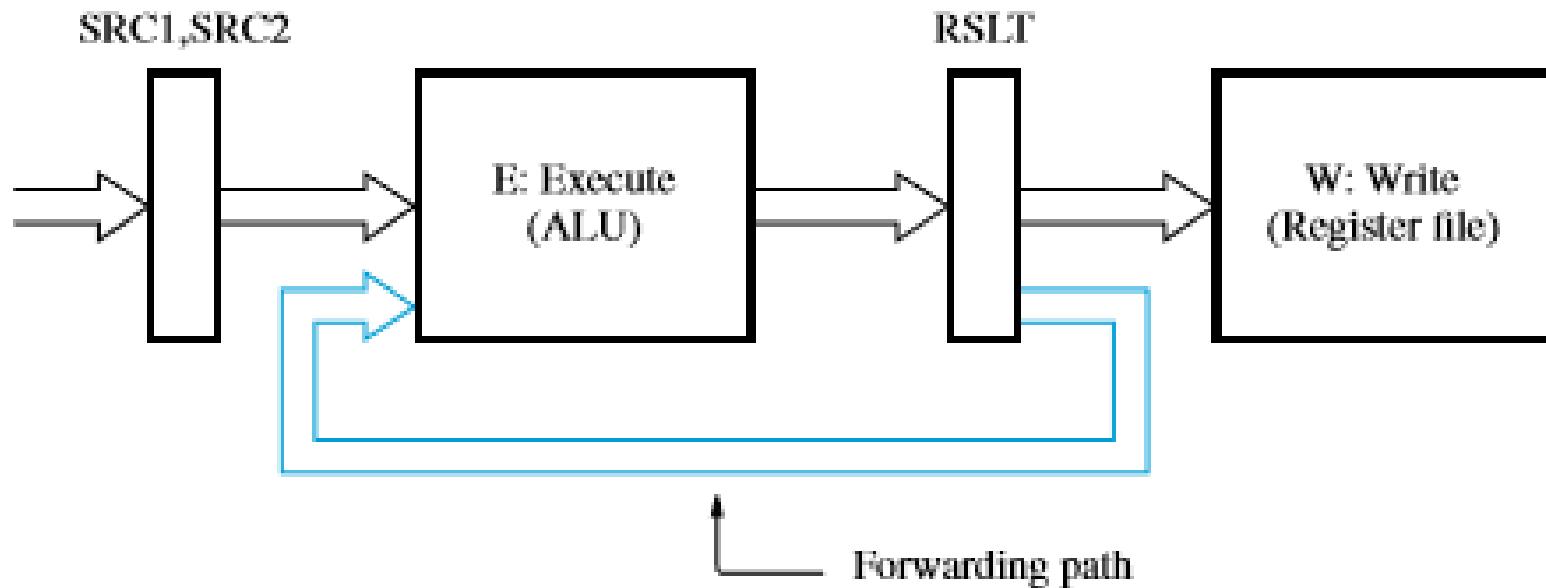


# Data Path

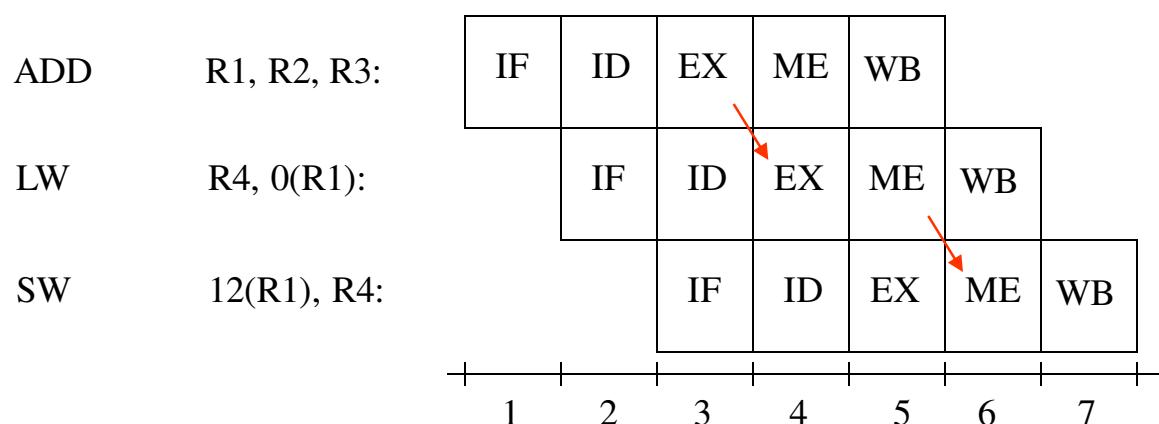
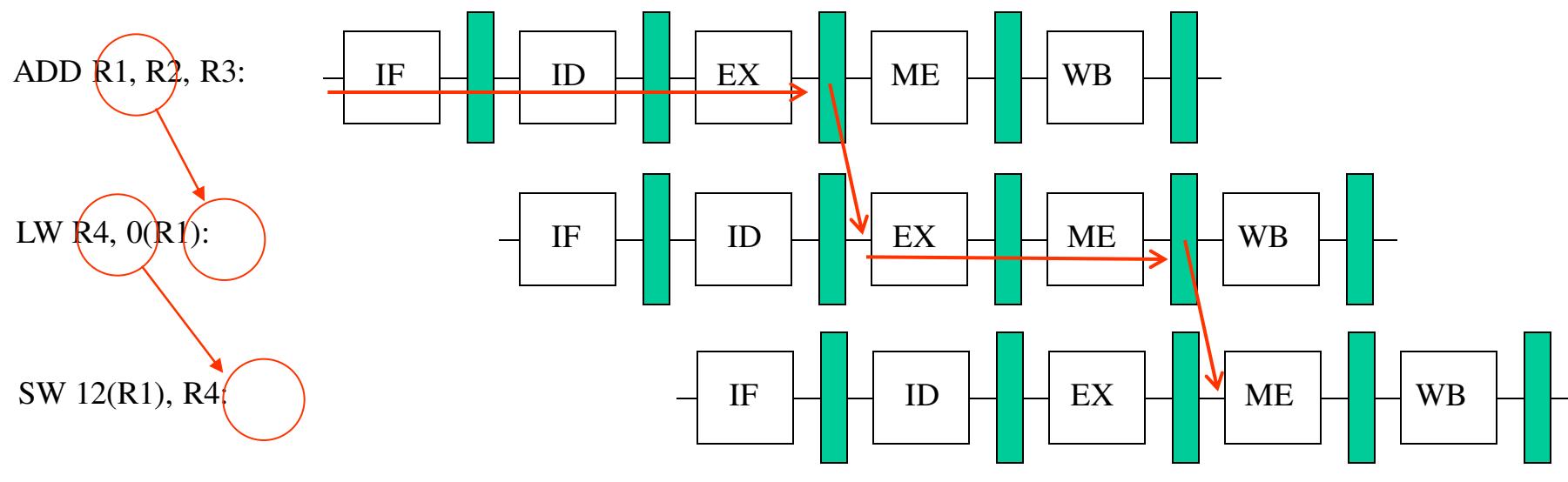


(a) Datapath

# Data forwarding



# Reducing pipeline hazards - three techniques

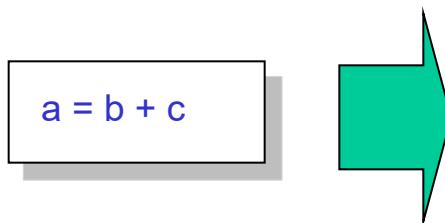


# Reducing pipeline hazards - three techniques

Technique 2: Instruction scheduling by a compiler

$$a = b + c$$

(in high-level language, such as C++)

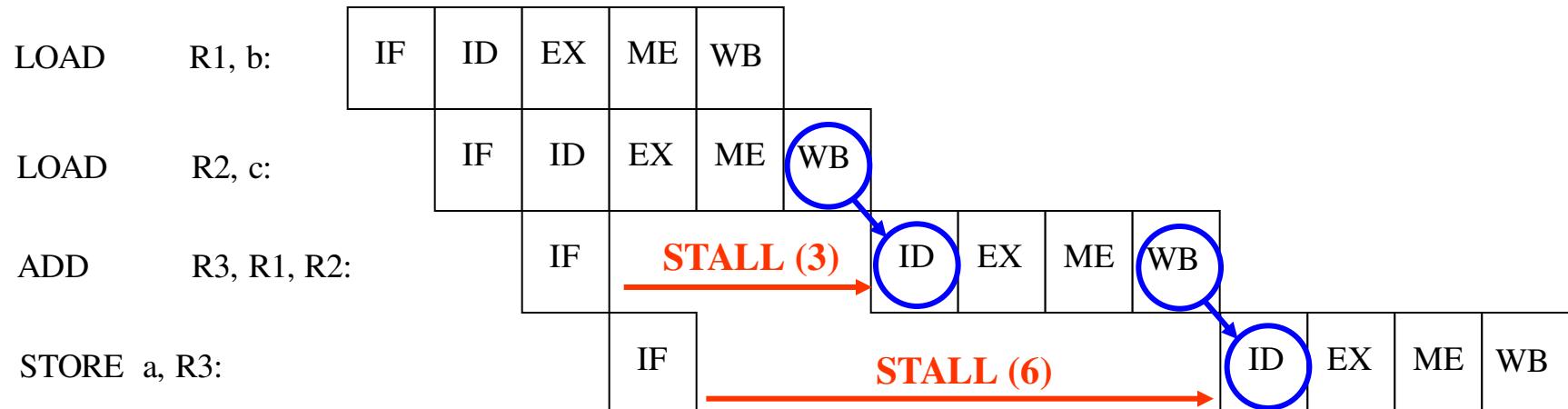


```
LOAD R1, b      // R1 ← MEM [Address of b]
LOAD R2, c      // R2 ← MEM [Address of b]
ADD R3, R1, R2  // R3 ← R1 + R2
STORE a, R3     // MEM [Address of a] ← R3
```

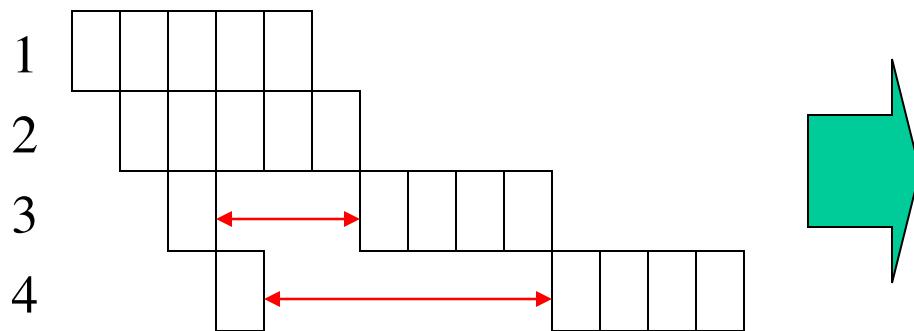
# Reducing pipeline hazards - three techniques

```

LOAD R1, b      // R1 ← MEM [Address of b]
LOAD R2, c      // R2 ← MEM [Address of c]
ADD R3, R1, R2 // R3 ← R1 + R2
STORE a, R3     // MEM [Address of a] ← R3
  
```



# Reducing pipeline hazards - three techniques



- |    |                |
|----|----------------|
| 1  | LOAD R1, b     |
| 2  | LOAD R2, c     |
| 3  | X              |
| 4  | X              |
| 5  | X              |
| 6  | ADD R3, R1, R2 |
| 7  | X              |
| 8  | X              |
| 9  | X              |
| 10 | STORE a, R3    |

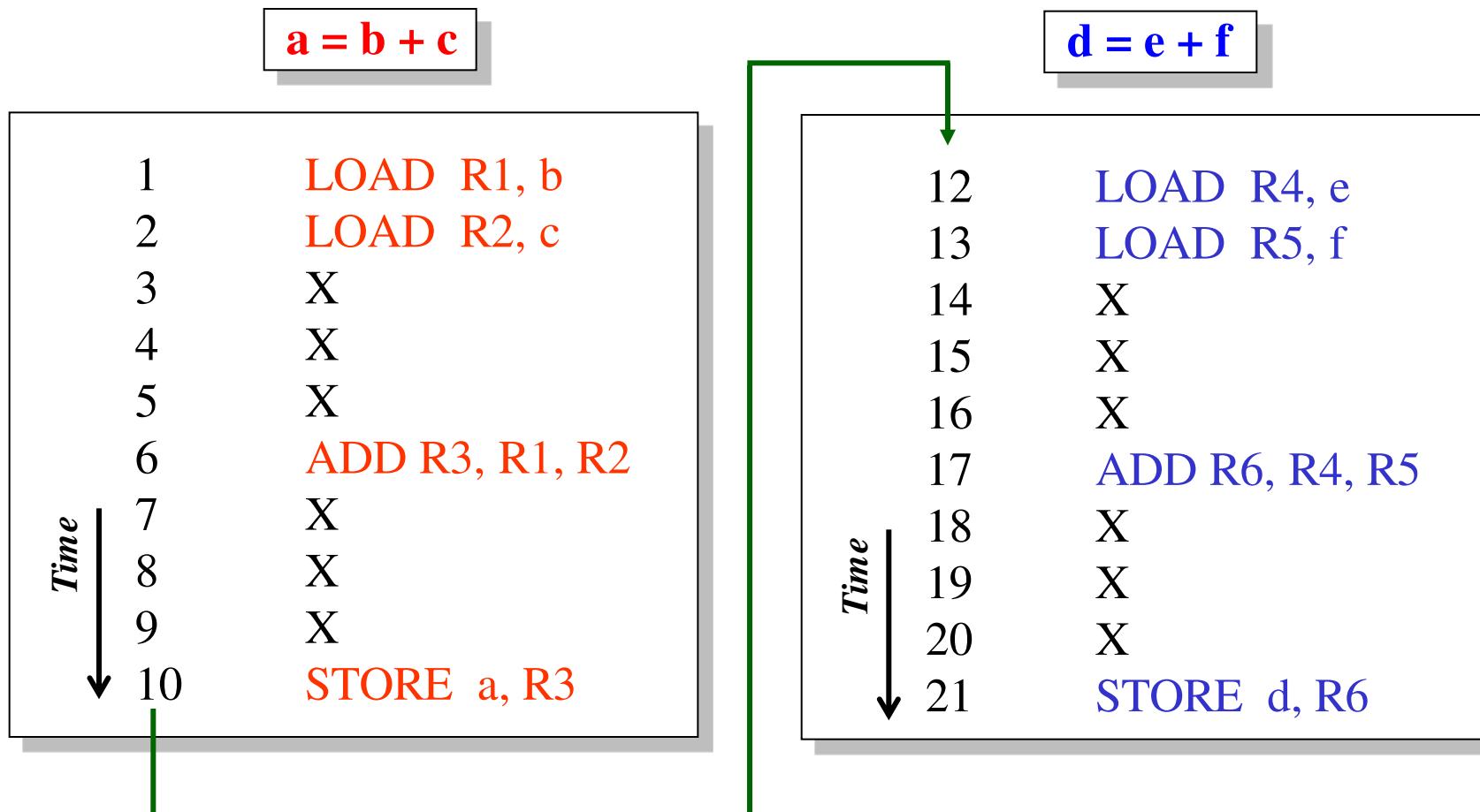
## Reducing pipeline hazards - three techniques

Now, we are going to execute two instructions

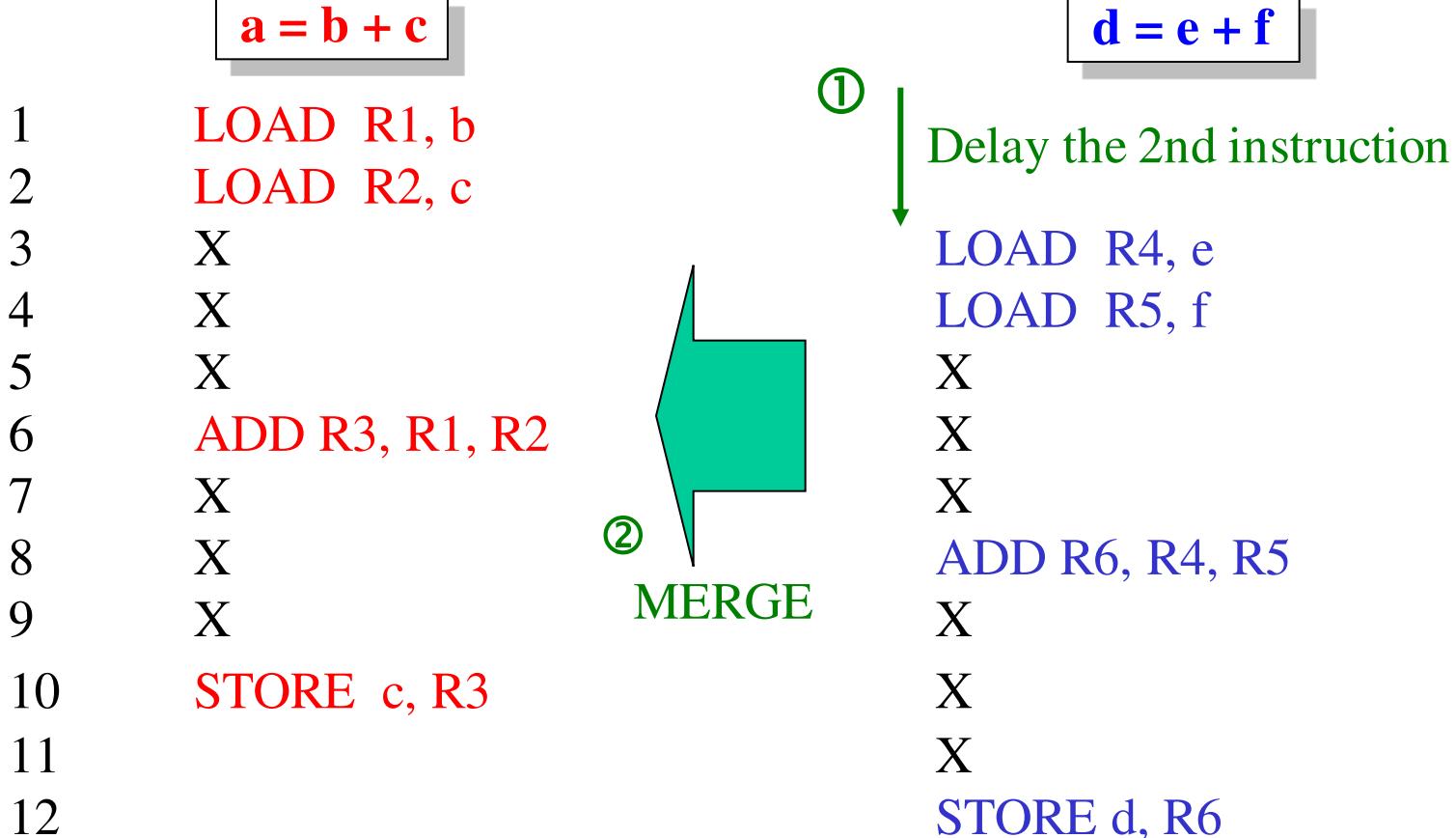
$$a = b + c$$

$$d = e + f$$

# Reducing pipeline hazards - three techniques



# Reducing pipeline hazards - three techniques



# Reducing pipeline hazards - three techniques

$$a = b + c$$

$$d = e + f$$

1	LOAD R1, b
2	LOAD R2, c
3	LOAD R4, e
4	LOAD R5, f
5	X
6	ADD R3, R1, R2
7	X
8	ADD R6, R4, R5
9	X
10	STORE c, R3
11	X
12	STORE d, R6

# Reducing pipeline hazards - three techniques

Technique 3: Delayed Branch:

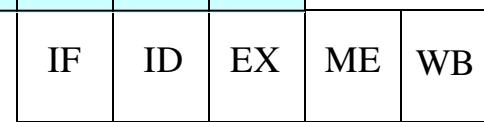
= Fill up clock cycles that will be flushed by a branch instruction

If branch NOT taken

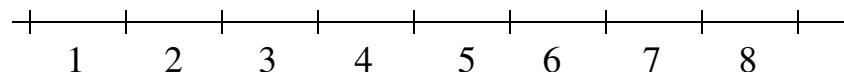
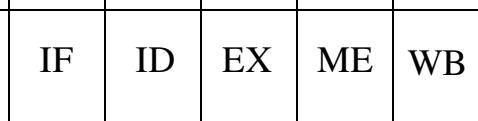
Branch Instruction<sub>(i)</sub>:



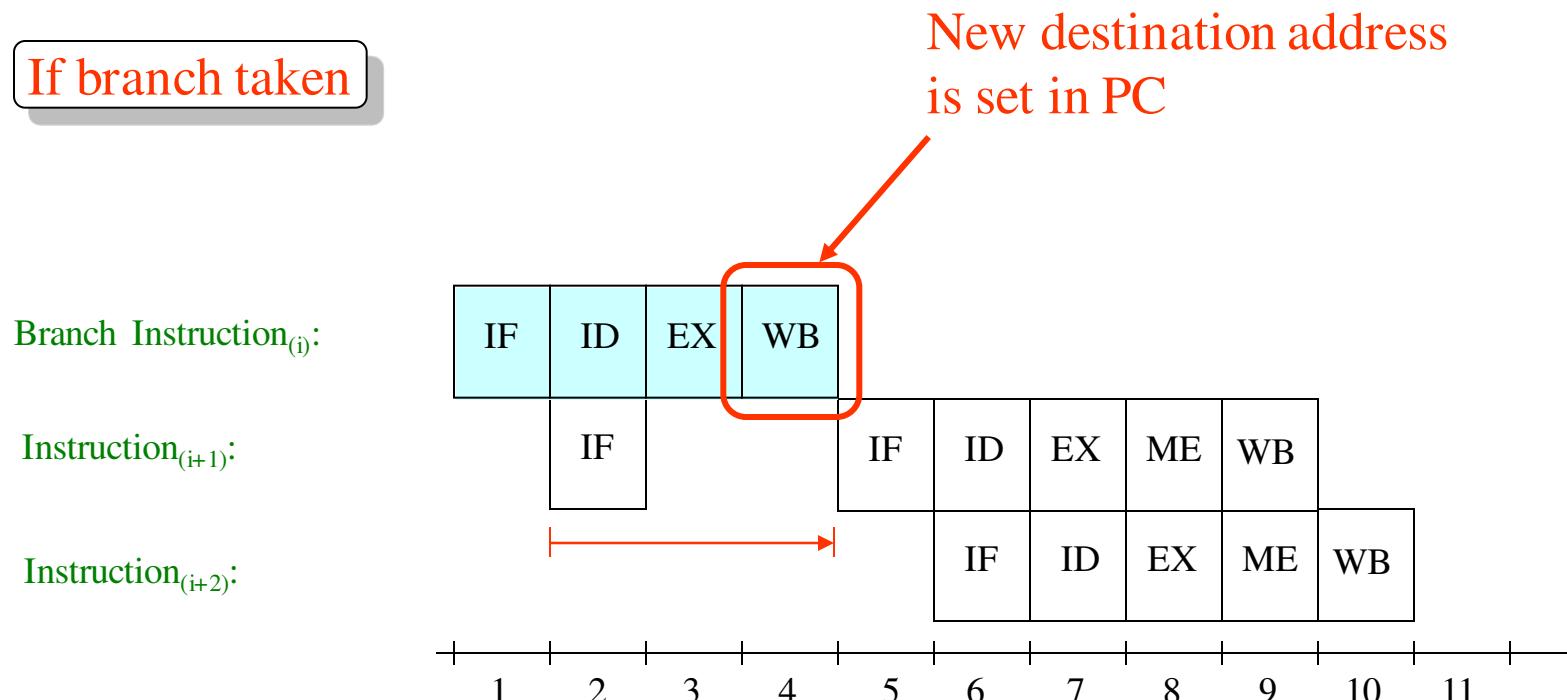
Instruction<sub>(i+1)</sub>:



Instruction<sub>(i+2)</sub>:

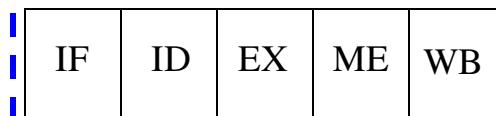
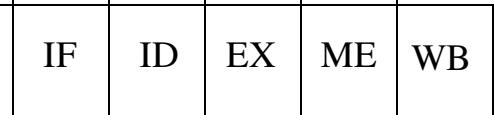
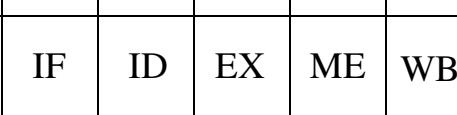
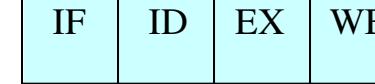
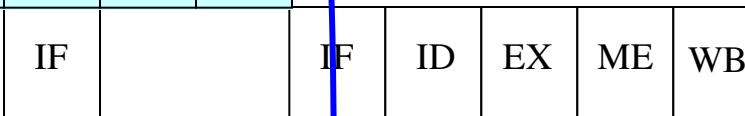
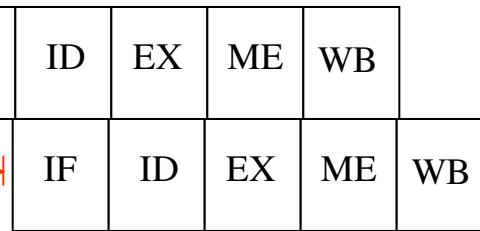


# Reducing pipeline hazards - three techniques



# Reducing pipeline hazards - three techniques

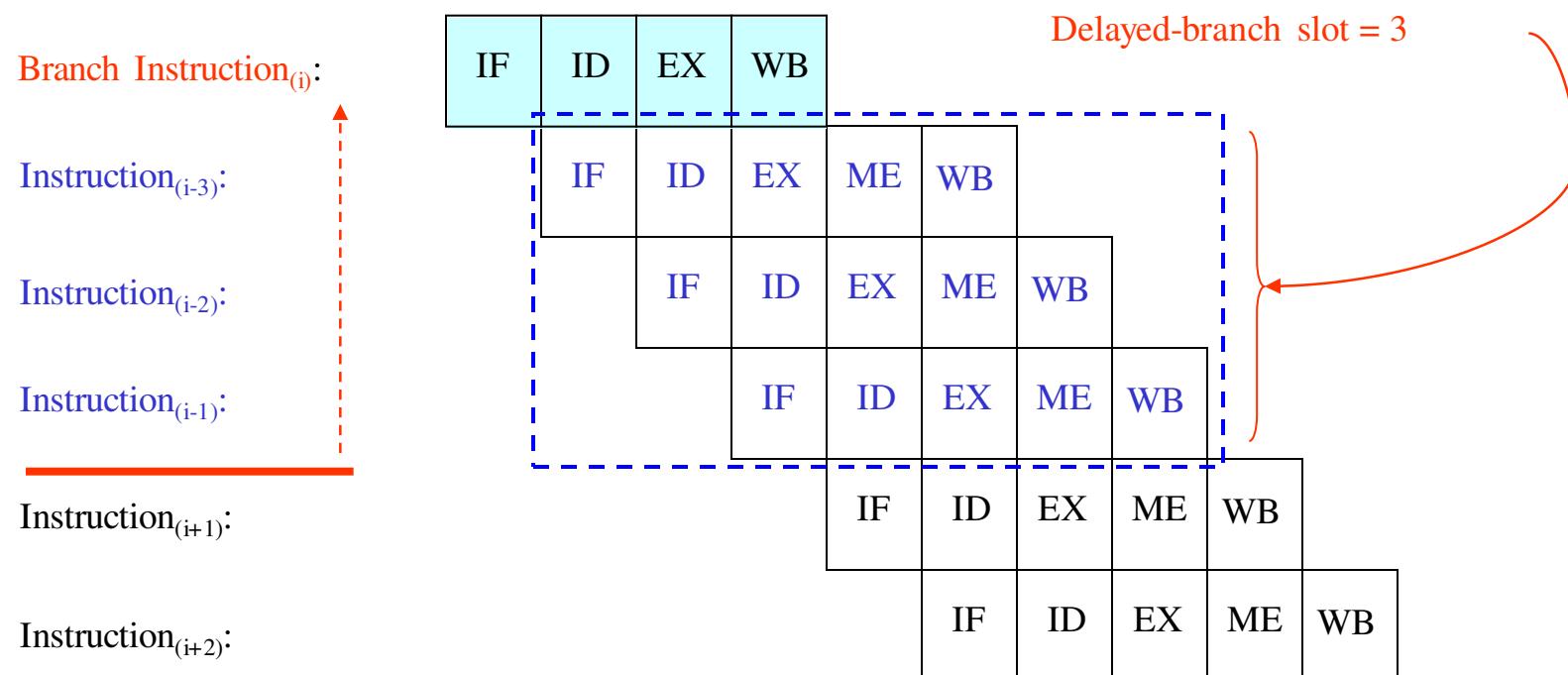
## Before Delayed Branch Applied

Instruction<sub>(i-3)</sub>:Instruction<sub>(i-2)</sub>:Instruction<sub>(i-1)</sub>:Branch Instruction<sub>(i)</sub>:Instruction<sub>(i+1)</sub>:Instruction<sub>(i+2)</sub>:

We are going to lose 3 cycles

# Reducing pipeline hazards - three techniques

After Delayed Branch Applied



# Reducing pipeline hazards - three techniques

Problem in delayed-branch: data dependency to the branch instruction

Example:

SUB R1, R2, R3  
 JPEZ R1  
 LW R8, 0(R4)

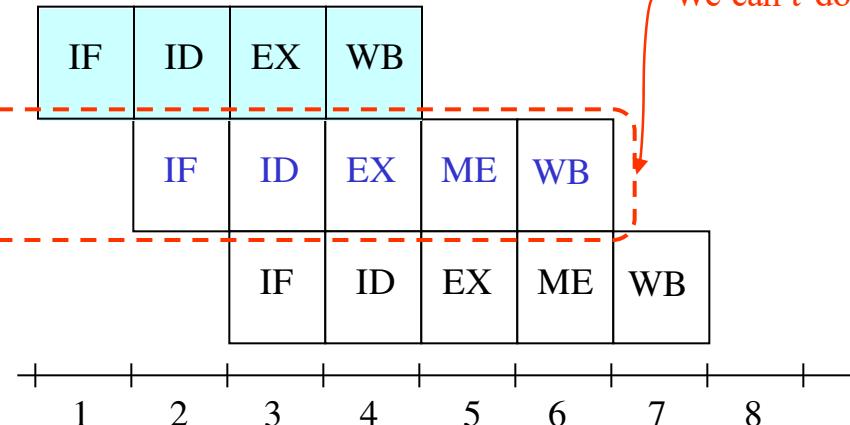
Conditional branch  
 (Jump if R1 = 0)

JPEZ R1, 0(R5):

SUB R1, R2, R3:

LW R8, 0(R4):

We can't do this!



## CHAPTER

# 8

## PIPELINING

### CHAPTER OBJECTIVES

In this chapter you will learn about:

- Pipelining as a means for executing machine instructions concurrently
- Various hazards that cause performance degradation in pipelined processors and means for mitigating their effect
- Hardware and software implications of pipelining
- Influence of pipelining on instruction set design
- Superscalar processors

The basic building blocks of a computer are introduced in preceding chapters. In this chapter, we discuss in detail the concept of pipelining, which is used in modern computers to achieve high performance. We begin by explaining the basics of pipelining and how it can lead to improved performance. Then we examine machine instruction features that facilitate pipelined execution, and we show that the choice of instructions and instruction sequencing can have a significant effect on performance. Pipelined organization requires sophisticated compilation techniques, and *optimizing compilers* have been developed for this purpose. Among other things, such compilers rearrange the sequence of operations to maximize the benefits of pipelined execution.

## 8.1 BASIC CONCEPTS

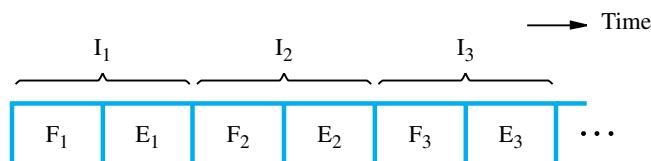
The speed of execution of programs is influenced by many factors. One way to improve performance is to use faster circuit technology to build the processor and the main memory. Another possibility is to arrange the hardware so that more than one operation can be performed at the same time. In this way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

We have encountered concurrent activities several times before. Chapter 1 introduced the concept of multiprogramming and explained how it is possible for I/O transfers and computational activities to proceed simultaneously. DMA devices make this possible because they can perform I/O transfers independently once these transfers are initiated by the processor.

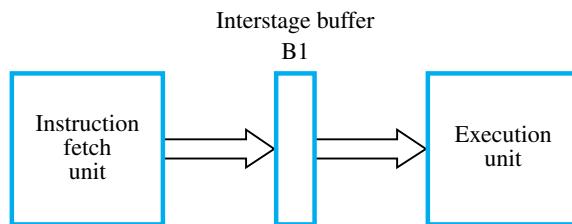
Pipelining is a particularly effective way of organizing concurrent activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as an assembly-line operation. Readers are undoubtedly familiar with the assembly line used in car manufacturing. The first station in an assembly line may prepare the chassis of a car, the next station adds the body, the next one installs the engine, and so on. While one group of workers is installing the engine on one car, another group is fitting a car body on the chassis of another car, and yet another group is preparing a new chassis for a third car. It may take days to complete work on a given car, but it is possible to have a new car rolling off the end of the assembly line every few minutes.

Consider how the idea of pipelining can be used in a computer. The processor executes a program by fetching and executing instructions, one after the other. Let  $F_i$  and  $E_i$  refer to the fetch and execute steps for instruction  $I_i$ . Execution of a program consists of a sequence of fetch and execute steps, as shown in Figure 8.1a.

Now consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in Figure 8.1b. The instruction fetched by the fetch unit is deposited in an intermediate storage buffer,  $B1$ . This buffer is needed to enable the execution unit to execute the instruction while the fetch unit is fetching the next instruction. The results of execution are deposited in the destination location specified by the instruction. For the purposes of this discussion, we assume that both the source and the destination of the data operated on by the instructions are inside the block labeled “Execution unit.”



(a) Sequential execution



(b) Hardware organization



(c) Pipelined execution

**Figure 8.1** Basic idea of instruction pipelining.

The computer is controlled by a clock whose period is such that the fetch and execute steps of any instruction can each be completed in one clock cycle. Operation of the computer proceeds as in Figure 8.1c. In the first clock cycle, the fetch unit fetches an instruction  $I_1$  (step  $F_1$ ) and stores it in buffer  $B1$  at the end of the clock cycle. In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction  $I_2$  (step  $F_2$ ). Meanwhile, the execution unit performs the operation specified by instruction  $I_1$ , which is available to it in buffer  $B1$  (step  $E_1$ ). By the end of the

second clock cycle, the execution of instruction  $I_1$  is completed and instruction  $I_2$  is available. Instruction  $I_2$  is stored in  $B1$ , replacing  $I_1$ , which is no longer needed. Step  $E_2$  is performed by the execution unit during the third clock cycle, while instruction  $I_3$  is being fetched by the fetch unit. In this manner, both the fetch and execute units are kept busy all the time. If the pattern in Figure 8.1c can be sustained for a long time, the completion rate of instruction execution will be twice that achievable by the sequential operation depicted in Figure 8.1a.

In summary, the fetch and execute units in Figure 8.1b constitute a two-stage pipeline in which each stage performs one step in processing an instruction. An inter-stage storage buffer,  $B1$ , is needed to hold the information being passed from one stage to the next. New information is loaded into this buffer at the end of each clock cycle.

The processing of an instruction need not be divided into only two steps. For example, a pipelined processor may process each instruction in four steps, as follows:

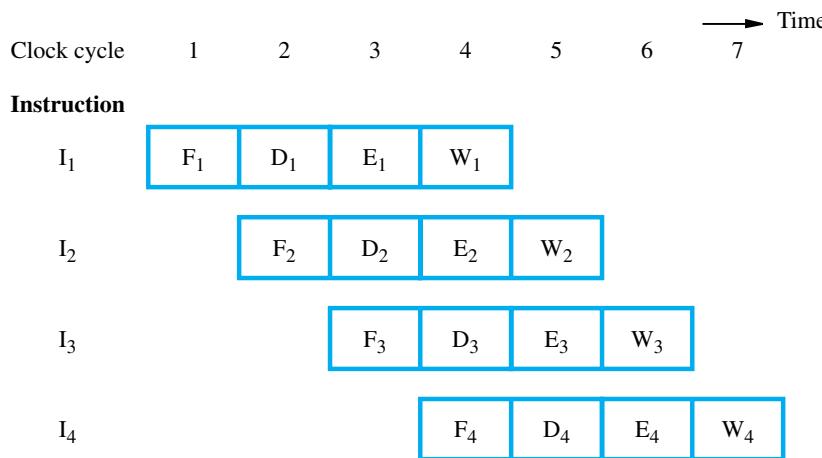
- F Fetch: read the instruction from the memory.
- D Decode: decode the instruction and fetch the source operand(s).
- E Execute: perform the operation specified by the instruction.
- W Write: store the result in the destination location.

The sequence of events for this case is shown in Figure 8.2a. Four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in Figure 8.2b. These units must be capable of performing their tasks simultaneously and without interfering with one another. Information is passed from one unit to the next through a storage buffer. As an instruction progresses through the pipeline, all the information needed by the stages downstream must be passed along. For example, during clock cycle 4, the information in the buffers is as follows:

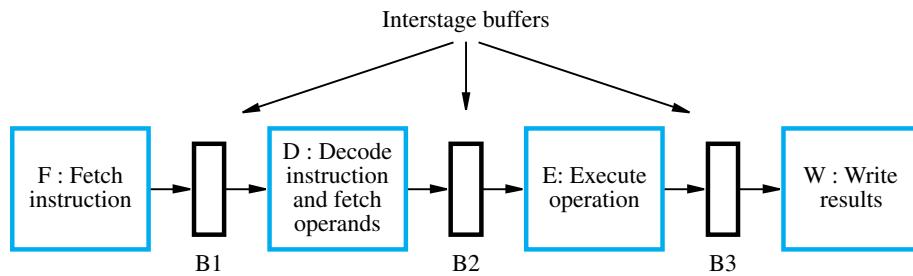
- Buffer  $B1$  holds instruction  $I_3$ , which was fetched in cycle 3 and is being decoded by the instruction-decoding unit.
- Buffer  $B2$  holds both the source operands for instruction  $I_2$  and the specification of the operation to be performed. This is the information produced by the decoding hardware in cycle 3. The buffer also holds the information needed for the write step of instruction  $I_2$  (step  $W_2$ ). Even though it is not needed by stage E, this information must be passed on to stage W in the following clock cycle to enable that stage to perform the required Write operation.
- Buffer  $B3$  holds the results produced by the execution unit and the destination information for instruction  $I_1$ .

### 8.1.1 ROLE OF CACHE MEMORY

Each stage in a pipeline is expected to complete its operation in one clock cycle. Hence, the clock period should be sufficiently long to complete the task being performed in any stage. If different units require different amounts of time, the clock period must allow the longest task to be completed. A unit that completes its task early is idle for the remainder of the clock period. Hence, pipelining is most effective in improving



(a) Instruction execution divided into four steps



(b) Hardware organization

**Figure 8.2** A 4-stage pipeline.

performance if the tasks being performed in different stages require about the same amount of time.

This consideration is particularly important for the instruction fetch step, which is assigned one clock period in Figure 8.2a. The clock cycle has to be equal to or greater than the time needed to complete a fetch operation. However, the access time of the main memory may be as much as ten times greater than the time needed to perform basic pipeline stage operations inside the processor, such as adding two numbers. Thus, if each instruction fetch required access to the main memory, pipelining would be of little value.

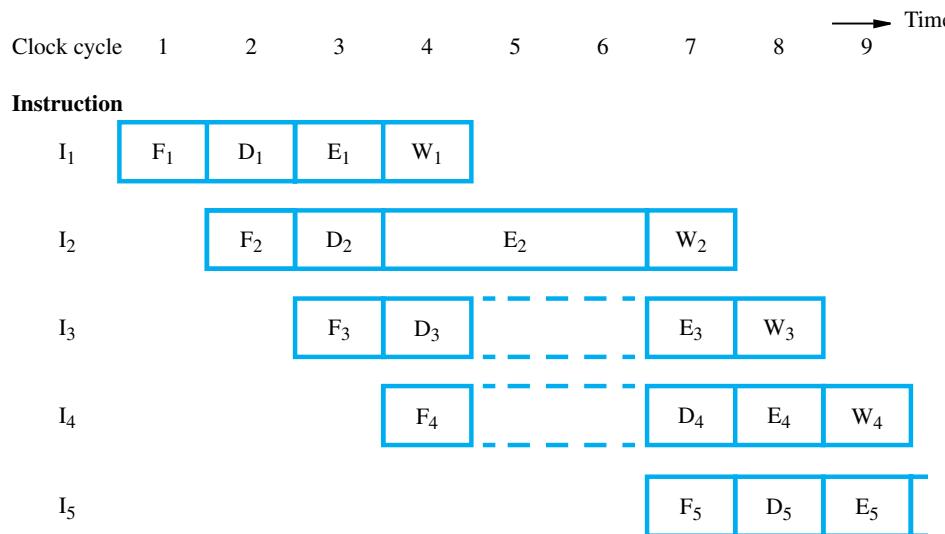
The use of cache memories solves the memory access problem. In particular, when a cache is included on the same chip as the processor, access time to the cache is usually the same as the time needed to perform other basic operations inside the processor. This

makes it possible to divide instruction fetching and processing into steps that are more or less equal in duration. Each of these steps is performed by a different pipeline stage, and the clock period is chosen to correspond to the longest one.

### 8.1.2 PIPELINE PERFORMANCE

The pipelined processor in Figure 8.2 completes the processing of one instruction in each clock cycle, which means that the rate of instruction processing is four times that of sequential operation. The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages. However, this increase would be achieved only if pipelined operation as depicted in Figure 8.2a could be sustained without interruption throughout program execution. Unfortunately, this is not the case.

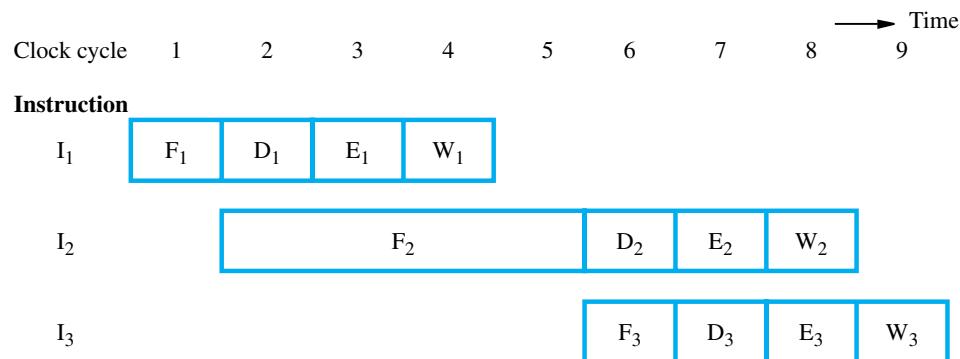
For a variety of reasons, one of the pipeline stages may not be able to complete its processing task for a given instruction in the time allotted. For example, stage E in the four-stage pipeline of Figure 8.2b is responsible for arithmetic and logic operations, and one clock cycle is assigned for this task. Although this may be sufficient for most operations, some operations, such as divide, may require more time to complete. Figure 8.3 shows an example in which the operation specified in instruction  $I_2$  requires three cycles to complete, from cycle 4 through cycle 6. Thus, in cycles 5 and 6, the Write stage must be told to do nothing, because it has no data to work with. Meanwhile, the information in buffer B2 must remain intact until the Execute stage has completed its operation. This means that stage 2 and, in turn, stage 1 are blocked from accepting new instructions because the information in B1 cannot be overwritten. Thus, steps D<sub>4</sub> and F<sub>5</sub> must be postponed as shown.



**Figure 8.3** Effect of an execution operation taking more than one clock cycle.

Pipelined operation in Figure 8.3 is said to have been *stalled* for two clock cycles. Normal pipelined operation resumes in cycle 7. Any condition that causes the pipeline to stall is called a *hazard*. We have just seen an example of a *data hazard*. A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result some operation has to be delayed, and the pipeline stalls.

The pipeline may also be stalled because of a delay in the availability of an instruction. For example, this may be a result of a miss in the cache, requiring the instruction to be fetched from the main memory. Such hazards are often called *control hazards* or *instruction hazards*. The effect of a cache miss on pipelined operation is illustrated in Figure 8.4. Instruction I<sub>1</sub> is fetched from the cache in cycle 1, and its execution proceeds normally. However, the fetch operation for instruction I<sub>2</sub>, which is started in cycle 2, results in a cache miss. The instruction fetch unit must now suspend any further fetch requests and wait for I<sub>2</sub> to arrive. We assume that instruction I<sub>2</sub> is received and loaded into buffer B1 at the end of cycle 5. The pipeline resumes its normal operation at that point.



(a) Instruction execution steps in successive clock cycles

	Time →								
Clock cycle	1	2	3	4	5	6	7	8	9
<b>Stage</b>									
F: Fetch	F <sub>1</sub>	F <sub>2</sub>	F <sub>2</sub>	F <sub>2</sub>	F <sub>2</sub>	F <sub>3</sub>			
D: Decode		D <sub>1</sub>	idle	idle	idle	D <sub>2</sub>	D <sub>3</sub>		
E: Execute			E <sub>1</sub>	idle	idle	idle	E <sub>2</sub>	E <sub>3</sub>	
W: Write				W <sub>1</sub>	idle	idle	idle	W <sub>2</sub>	W <sub>3</sub>

(b) Function performed by each processor stage in successive clock cycles

**Figure 8.4** Pipeline stall caused by a cache miss in F2.

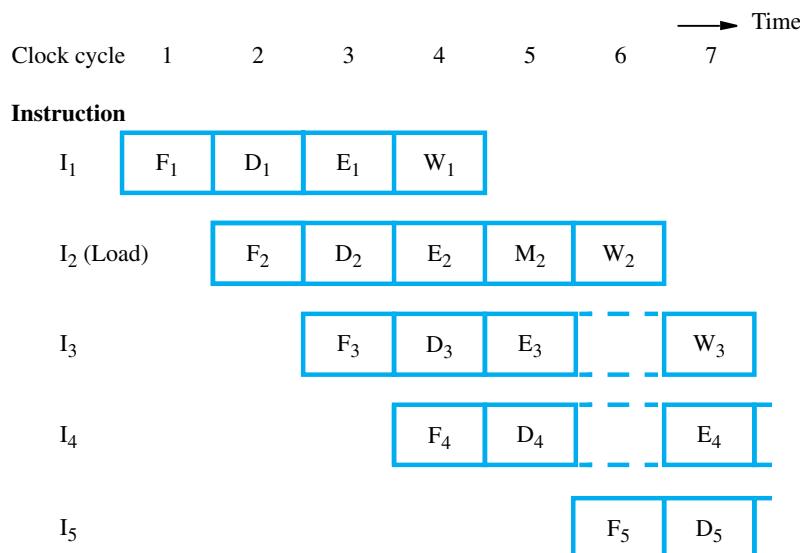
An alternative representation of the operation of a pipeline in the case of a cache miss is shown in Figure 8.4b. This figure gives the function performed by each pipeline stage in each clock cycle. Note that the Decode unit is idle in cycles 3 through 5, the Execute unit is idle in cycles 4 through 6, and the Write unit is idle in cycles 5 through 7. Such idle periods are called *stalls*. They are also often referred to as *bubbles* in the pipeline. Once created as a result of a delay in one of the pipeline stages, a bubble moves downstream until it reaches the last unit.

A third type of hazard that may be encountered in pipelined operation is known as a *structural hazard*. This is the situation when two instructions require the use of a given hardware resource at the same time. The most common case in which this hazard may arise is in access to memory. One instruction may need to access memory as part of the Execute or Write stage while another instruction is being fetched. If instructions and data reside in the same cache unit, only one instruction can proceed and the other instruction is delayed. Many processors use separate instruction and data caches to avoid this delay.

An example of a structural hazard is shown in Figure 8.5. This figure shows how the load instruction

Load X(R1),R2

can be accommodated in our example 4-stage pipeline. The memory address,  $X+[R1]$ , is computed in step  $E_2$  in cycle 4, then memory access takes place in cycle 5. The operand read from memory is written into register  $R2$  in cycle 6. This means that the execution step of this instruction takes two clock cycles (cycles 4 and 5). It causes the pipeline to stall for one cycle, because both instructions  $I_2$  and  $I_3$  require access to the register file in cycle 6. Even though the instructions and their data are all available, the pipeline is



**Figure 8.5** Effect of a Load instruction on pipeline timing.

stalled because one hardware resource, the register file, cannot handle two operations at once. If the register file had two input ports, that is, if it allowed two simultaneous write operations, the pipeline would not be stalled. In general, structural hazards are avoided by providing sufficient hardware resources on the processor chip.

It is important to understand that pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases, where throughput is measured by the rate at which instruction execution is completed. Any time one of the stages in the pipeline cannot complete its operation in one clock cycle, the pipeline stalls, and some degradation in performance occurs. Thus, the performance level of one instruction completion in each clock cycle is actually the upper limit for the throughput achievable in a pipelined processor organized as in Figure 8.2b.

An important goal in designing processors is to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact. In the following sections we discuss various hazards, starting with data hazards, followed by control hazards. In each case we present some of the techniques used to mitigate their negative effect on performance. We return to the issue of performance assessment in Section 8.8.

## 8.2 DATA HAZARDS

A data hazard is a situation in which the pipeline is stalled because the data to be operated on are delayed for some reason, as illustrated in Figure 8.3. We will now examine the issue of availability of data in some detail.

Consider a program that contains two instructions,  $I_1$  followed by  $I_2$ . When this program is executed in a pipeline, the execution of  $I_2$  can begin before the execution of  $I_1$  is completed. This means that the results generated by  $I_1$  may not be available for use by  $I_2$ . We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially. The potential for obtaining incorrect results when operations are performed concurrently can be demonstrated by a simple example. Assume that  $A = 5$ , and consider the following two operations:

$$A \leftarrow 3 + A$$

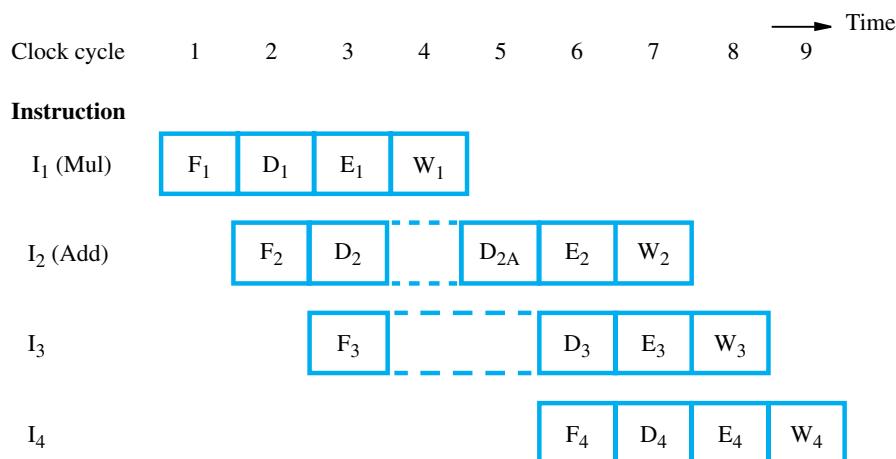
$$B \leftarrow 4 \times A$$

When these operations are performed in the order given, the result is  $B = 32$ . But if they are performed concurrently, the value of  $A$  used in computing  $B$  would be the original value, 5, leading to an incorrect result. If these two operations are performed by instructions in a program, then the instructions must be executed one after the other, because the data used in the second instruction depend on the result of the first instruction. On the other hand, the two operations

$$A \leftarrow 5 \times C$$

$$B \leftarrow 20 + C$$

can be performed concurrently, because these operations are independent.



**Figure 8.6** Pipeline stalled by data dependency between D<sub>2</sub> and W<sub>1</sub>.

This example illustrates a basic constraint that must be enforced to guarantee correct results. When two operations depend on each other, they must be performed sequentially in the correct order. This rather obvious condition has far-reaching consequences. Understanding its implications is the key to understanding the variety of design alternatives and trade-offs encountered in pipelined computers.

Consider the pipeline in Figure 8.2. The data dependency just described arises when the destination of one instruction is used as a source in the next instruction. For example, the two instructions

```
Mul R2,R3,R4
Add R5,R4,R6
```

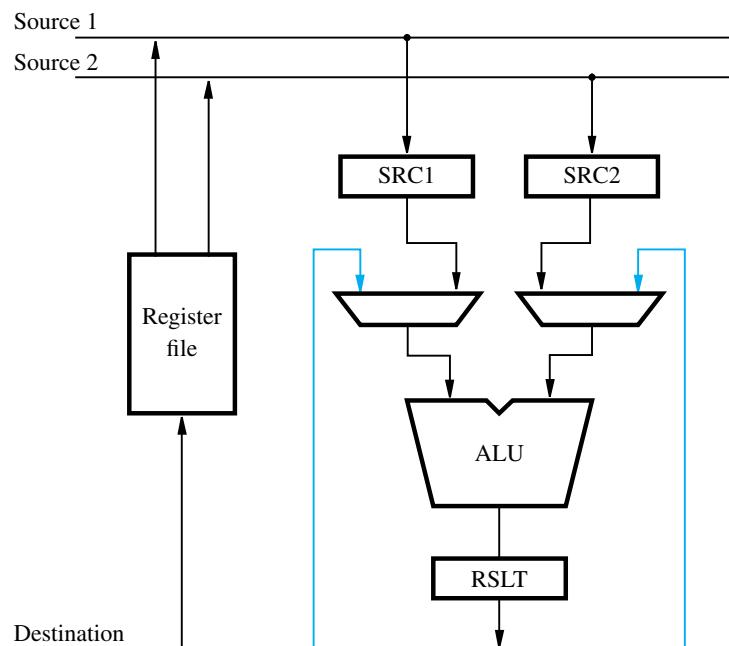
give rise to a data dependency. The result of the multiply instruction is placed into register R4, which in turn is one of the two source operands of the Add instruction. Assuming that the multiply operation takes one clock cycle to complete, execution would proceed as shown in Figure 8.6. As the Decode unit decodes the Add instruction in cycle 3, it realizes that R4 is used as a source operand. Hence, the D step of that instruction cannot be completed until the W step of the multiply instruction has been completed. Completion of step D<sub>2</sub> must be delayed to clock cycle 5, and is shown as step D<sub>2A</sub> in the figure. Instruction I<sub>3</sub> is fetched in cycle 3, but its decoding must be delayed because step D<sub>3</sub> cannot precede D<sub>2</sub>. Hence, pipelined execution is stalled for two cycles.

### 8.2.1 OPERAND FORWARDING

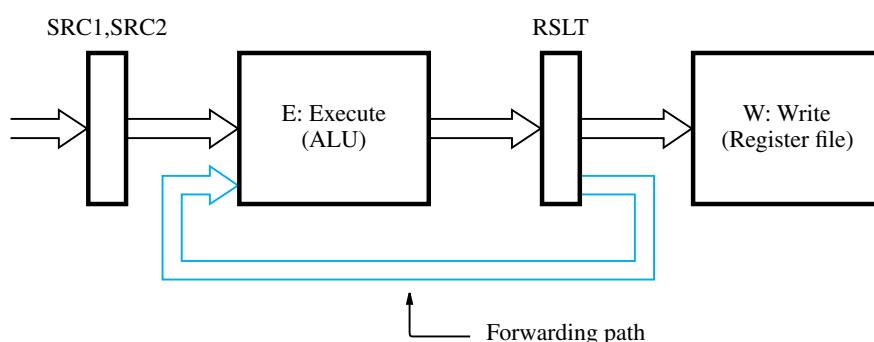
The data hazard just described arises because one instruction, instruction I<sub>2</sub> in Figure 8.6, is waiting for data to be written in the register file. However, these data are available at the output of the ALU once the Execute stage completes step E<sub>1</sub>. Hence, the delay can

be reduced, or possibly eliminated, if we arrange for the result of instruction  $I_1$  to be forwarded directly for use in step  $E_2$ .

Figure 8.7a shows a part of the processor datapath involving the ALU and the register file. This arrangement is similar to the three-bus structure in Figure 7.8, except that registers SRC1, SRC2, and RSLT have been added. These registers constitute the



(a) Datapath



(b) Position of the source and result registers in the processor pipeline

**Figure 8.7** Operand forwarding in a pipelined processor.

interstage buffers needed for pipelined operation, as illustrated in Figure 8.7b. With reference to Figure 8.2b, registers SRC1 and SRC2 are part of buffer B2 and RSLT is part of B3. The data forwarding mechanism is provided by the blue connection lines. The two multiplexers connected at the inputs to the ALU allow the data on the destination bus to be selected instead of the contents of either the SRC1 or SRC2 register.

When the instructions in Figure 8.6 are executed in the datapath of Figure 8.7, the operations performed in each clock cycle are as follows. After decoding instruction I<sub>2</sub> and detecting the data dependency, a decision is made to use data forwarding. The operand not involved in the dependency, register R2, is read and loaded in register SRC1 in clock cycle 3. In the next clock cycle, the product produced by instruction I<sub>1</sub> is available in register RSLT, and because of the forwarding connection, it can be used in step E<sub>2</sub>. Hence, execution of I<sub>2</sub> proceeds without interruption.

### 8.2.2 HANDLING DATA HAZARDS IN SOFTWARE

In Figure 8.6, we assumed the data dependency is discovered by the hardware while the instruction is being decoded. The control hardware delays reading register R4 until cycle 5, thus introducing a 2-cycle stall unless operand forwarding is used. An alternative approach is to leave the task of detecting data dependencies and dealing with them to the software. In this case, the compiler can introduce the two-cycle delay needed between instructions I<sub>1</sub> and I<sub>2</sub> by inserting NOP (No-operation) instructions, as follows:

```
I1: Mul R2,R3,R4
      NOP
      NOP
I2: Add R5,R4,R6
```

If the responsibility for detecting such dependencies is left entirely to the software, the compiler must insert the NOP instructions to obtain a correct result. This possibility illustrates the close link between the compiler and the hardware. A particular feature can be either implemented in hardware or left to the compiler. Leaving tasks such as inserting NOP instructions to the compiler leads to simpler hardware. Being aware of the need for a delay, the compiler can attempt to reorder instructions to perform useful tasks in the NOP slots, and thus achieve better performance. On the other hand, the insertion of NOP instructions leads to larger code size. Also, it is often the case that a given processor architecture has several hardware implementations, offering different features. NOP instructions inserted to satisfy the requirements of one implementation may not be needed and, hence, would lead to reduced performance on a different implementation.

### 8.2.3 SIDE EFFECTS

The data dependencies encountered in the preceding examples are explicit and easily detected because the register involved is named as the destination in instruction I<sub>1</sub> and as a source in I<sub>2</sub>. Sometimes an instruction changes the contents of a register other

than the one named as the destination. An instruction that uses an autoincrement or autodecrement addressing mode is an example. In addition to storing new data in its destination location, the instruction changes the contents of a source register used to access one of its operands. All the precautions needed to handle data dependencies involving the destination location must also be applied to the registers affected by an autoincrement or autodecrement operation. When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a *side effect*. For example, stack instructions, such as push and pop, produce similar side effects because they implicitly use the autoincrement and autodecrement addressing modes.

Another possible side effect involves the condition code flags, which are used by instructions such as conditional branches and add-with-carry. Suppose that registers R1 and R2 hold a double-precision integer number that we wish to add to another double-precision number in registers R3 and R4. This may be accomplished as follows:

Add	R1, R3
AddWithCarry	R2, R4

An implicit dependency exists between these two instructions through the carry flag. This flag is set by the first instruction and used in the second instruction, which performs the operation

$$R4 \leftarrow [R2] + [R4] + \text{carry}$$

Instructions that have side effects give rise to multiple data dependencies, which lead to a substantial increase in the complexity of the hardware or software needed to resolve them. For this reason, instructions designed for execution on pipelined hardware should have few side effects. Ideally, only the contents of the destination location, either a register or a memory location, should be affected by any given instruction. Side effects, such as setting the condition code flags or updating the contents of an address pointer, should be kept to a minimum. However, Chapter 2 showed that the autoincrement and autodecrement addressing modes are potentially useful. Condition code flags are also needed for recording such information as the generation of a carry or the occurrence of overflow in an arithmetic operation. In Section 8.4 we show how such functions can be provided by other means that are consistent with a pipelined organization and with the requirements of optimizing compilers.

### 8.3 INSTRUCTION HAZARDS

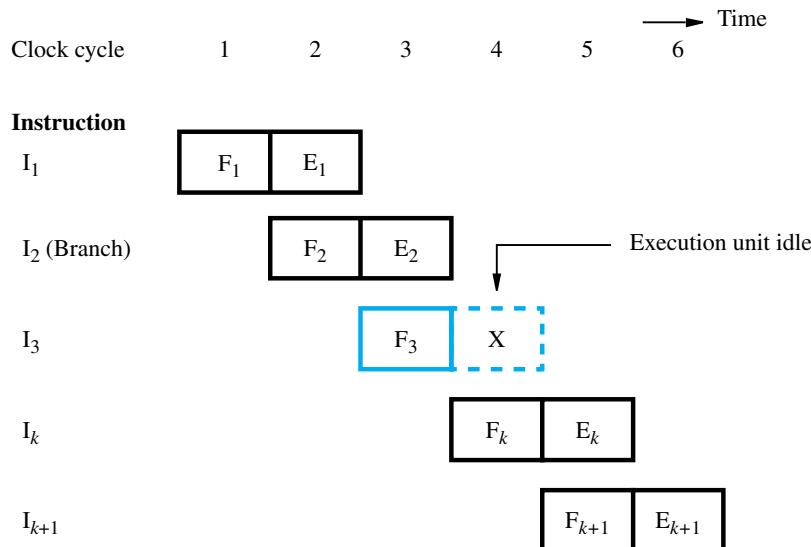
The purpose of the instruction fetch unit is to supply the execution units with a steady stream of instructions. Whenever this stream is interrupted, the pipeline stalls, as Figure 8.4 illustrates for the case of a cache miss. A branch instruction may also cause the pipeline to stall. We will now examine the effect of branch instructions and the techniques that can be used for mitigating their impact. We start with unconditional branches.

### 8.3.1 UNCONDITIONAL BRANCHES

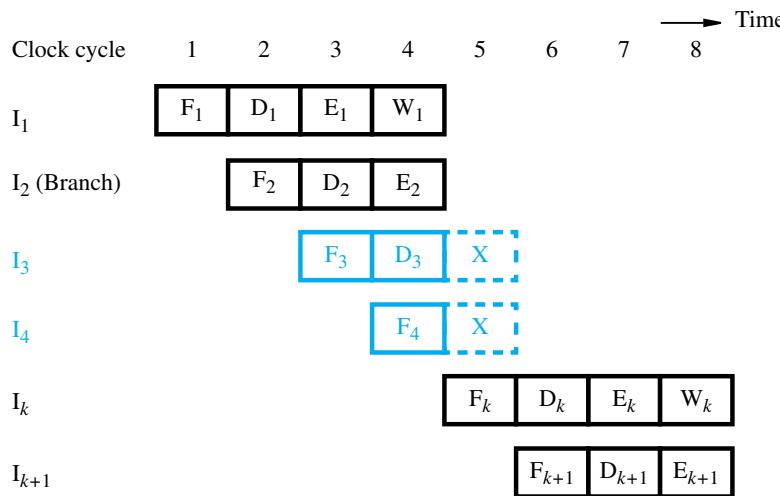
Figure 8.8 shows a sequence of instructions being executed in a two-stage pipeline. Instructions  $I_1$  to  $I_3$  are stored at successive memory addresses, and  $I_2$  is a branch instruction. Let the branch target be instruction  $I_k$ . In clock cycle 3, the fetch operation for instruction  $I_3$  is in progress at the same time that the branch instruction is being decoded and the target address computed. In clock cycle 4, the processor must discard  $I_3$ , which has been incorrectly fetched, and fetch instruction  $I_k$ . In the meantime, the hardware unit responsible for the Execute (E) step must be told to do nothing during that clock period. Thus, the pipeline is stalled for one clock cycle.

The time lost as a result of a branch instruction is often referred to as the *branch penalty*. In Figure 8.8, the branch penalty is one clock cycle. For a longer pipeline, the branch penalty may be higher. For example, Figure 8.9a shows the effect of a branch instruction on a four-stage pipeline. We have assumed that the branch address is computed in step E<sub>2</sub>. Instructions  $I_3$  and  $I_4$  must be discarded, and the target instruction,  $I_k$ , is fetched in clock cycle 5. Thus, the branch penalty is two clock cycles.

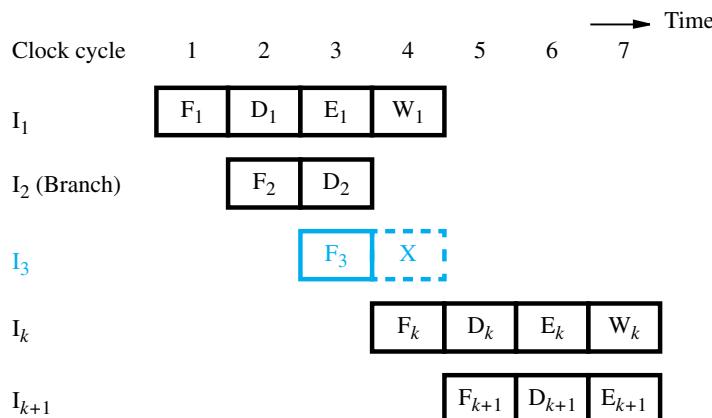
Reducing the branch penalty requires the branch address to be computed earlier in the pipeline. Typically, the instruction fetch unit has dedicated hardware to identify a branch instruction and compute the branch target address as quickly as possible after an instruction is fetched. With this additional hardware, both of these tasks can be performed in step D<sub>2</sub>, leading to the sequence of events shown in Figure 8.9b. In this case, the branch penalty is only one clock cycle.



**Figure 8.8** An idle cycle caused by a branch instruction.

**8.3 INSTRUCTION HAZARDS****467**

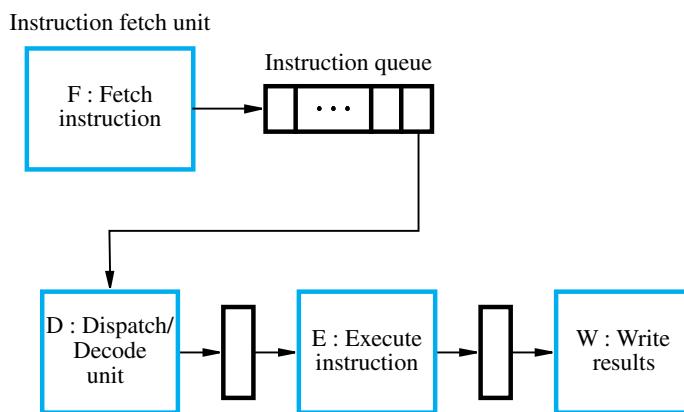
(a) Branch address computed in Execute stage



(b) Branch address computed in Decode stage

**Figure 8.9** Branch timing.**Instruction Queue and Prefetching**

Either a cache miss or a branch instruction stalls the pipeline for one or more clock cycles. To reduce the effect of these interruptions, many processors employ sophisticated fetch units that can fetch instructions before they are needed and put them in a queue. Typically, the instruction queue can store several instructions. A separate unit, which we call the *dispatch unit*, takes instructions from the front of the queue and



**Figure 8.10** Use of an instruction queue in the hardware organization of Figure 8.2b.

sends them to the execution unit. This leads to the organization shown in Figure 8.10. The dispatch unit also performs the decoding function.

To be effective, the fetch unit must have sufficient decoding and processing capability to recognize and execute branch instructions. It attempts to keep the instruction queue filled at all times to reduce the impact of occasional delays when fetching instructions. When the pipeline stalls because of a data hazard, for example, the dispatch unit is not able to issue instructions from the instruction queue. However, the fetch unit continues to fetch instructions and add them to the queue. Conversely, if there is a delay in fetching instructions because of a branch or a cache miss, the dispatch unit continues to issue instructions from the instruction queue.

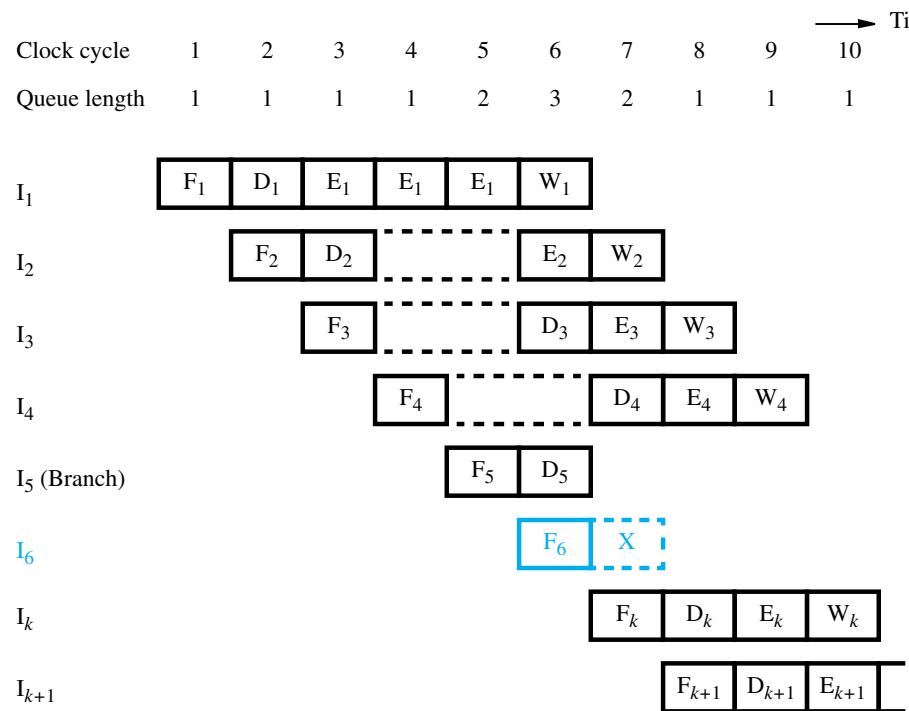
Figure 8.11 illustrates how the queue length changes and how it affects the relationship between different pipeline stages. We have assumed that initially the queue contains one instruction. Every fetch operation adds one instruction to the queue and every dispatch operation reduces the queue length by one. Hence, the queue length remains the same for the first four clock cycles. (There is both an F and a D step in each of these cycles.) Suppose that instruction  $I_1$  introduces a 2-cycle stall. Since space is available in the queue, the fetch unit continues to fetch instructions and the queue length rises to 3 in clock cycle 6.

Instruction  $I_5$  is a branch instruction. Its target instruction,  $I_k$ , is fetched in cycle 7, and instruction  $I_6$  is discarded. The branch instruction would normally cause a stall in cycle 7 as a result of discarding instruction  $I_6$ . Instead, instruction  $I_4$  is dispatched from the queue to the decoding stage. After discarding  $I_6$ , the queue length drops to 1 in cycle 8. The queue length will be at this value until another stall is encountered.

Now observe the sequence of instruction completions in Figure 8.11. Instructions  $I_1$ ,  $I_2$ ,  $I_3$ ,  $I_4$ , and  $I_k$  complete execution in successive clock cycles. Hence, the branch instruction does not increase the overall execution time. This is because the instruction fetch unit has executed the branch instruction (by computing the branch address) concurrently with the execution of other instructions. This technique is referred to as *branch folding*.

## 8.3 INSTRUCTION HAZARDS

469



**Figure 8.11** Branch timing in the presence of an instruction queue. Branch target address is computed in the D stage.

Note that branch folding occurs only if at the time a branch instruction is encountered, at least one instruction is available in the queue other than the branch instruction. If only the branch instruction is in the queue, execution would proceed as in Figure 8.9b. Therefore, it is desirable to arrange for the queue to be full most of the time, to ensure an adequate supply of instructions for processing. This can be achieved by increasing the rate at which the fetch unit reads instructions from the cache. In many processors, the width of the connection between the fetch unit and the instruction cache allows reading more than one instruction in each clock cycle. If the fetch unit replenishes the instruction queue quickly after a branch has occurred, the probability that branch folding will occur increases.

Having an instruction queue is also beneficial in dealing with cache misses. When a cache miss occurs, the dispatch unit continues to send instructions for execution as long as the instruction queue is not empty. Meanwhile, the desired cache block is read from the main memory or from a secondary cache. When fetch operations are resumed, the instruction queue is refilled. If the queue does not become empty, a cache miss will have no effect on the rate of instruction execution.

In summary, the instruction queue mitigates the impact of branch instructions on performance through the process of branch folding. It has a similar effect on stalls

caused by cache misses. The effectiveness of this technique is enhanced when the instruction fetch unit is able to read more than one instruction at a time from the instruction cache.

### 8.3.2 CONDITIONAL BRANCHES AND BRANCH PREDICTION

A conditional branch instruction introduces the added hazard caused by the dependency of the branch condition on the result of a preceding instruction. The decision to branch cannot be made until the execution of that instruction has been completed.

Branch instructions occur frequently. In fact, they represent about 20 percent of the dynamic instruction count of most programs. (The dynamic count is the number of instruction executions, taking into account the fact that some program instructions are executed many times because of loops.) Because of the branch penalty, this large percentage would reduce the gain in performance expected from pipelining. Fortunately, branch instructions can be handled in several ways to reduce their negative impact on the rate of execution of instructions.

#### Delayed Branch

In Figure 8.8, the processor fetches instruction  $I_3$  before it determines whether the current instruction,  $I_2$ , is a branch instruction. When execution of  $I_2$  is completed and a branch is to be made, the processor must discard  $I_3$  and fetch the instruction at the branch target. The location following a branch instruction is called a *branch delay slot*. There may be more than one branch delay slot, depending on the time it takes to execute a branch instruction. For example, there are two branch delay slots in Figure 8.9a and one delay slot in Figure 8.9b. The instructions in the delay slots are always fetched and at least partially executed before the branch decision is made and the branch target address is computed.

A technique called *delayed branching* can minimize the penalty incurred as a result of conditional branch instructions. The idea is simple. The instructions in the delay slots are always fetched. Therefore, we would like to arrange for them to be fully executed whether or not the branch is taken. The objective is to be able to place useful instructions in these slots. If no useful instructions can be placed in the delay slots, these slots must be filled with NOP instructions. This situation is exactly the same as in the case of data dependency discussed in Section 8.2.

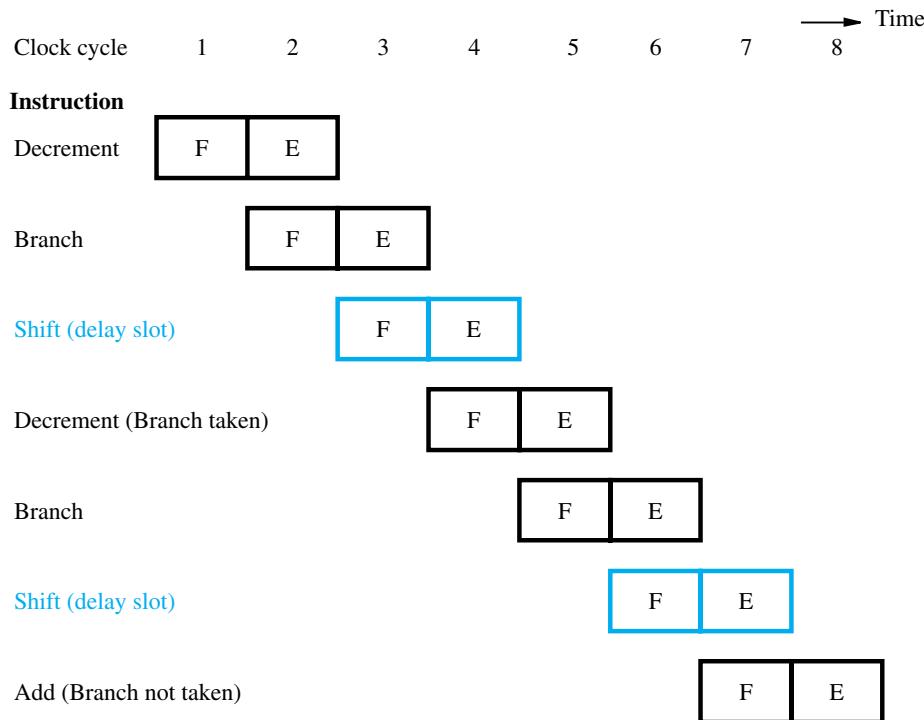
Consider the instruction sequence given in Figure 8.12a. Register R2 is used as a counter to determine the number of times the contents of register R1 are shifted left. For a processor with one delay slot, the instructions can be reordered as shown in Figure 8.12b. The shift instruction is fetched while the branch instruction is being executed. After evaluating the branch condition, the processor fetches the instruction at LOOP or at NEXT, depending on whether the branch condition is true or false, respectively. In either case, it completes execution of the shift instruction. The sequence of events during the last two passes in the loop is illustrated in Figure 8.13. Pipelined operation is not interrupted at any time, and there are no idle cycles. Logically, the program is executed as if the branch instruction were placed after the shift instruction. That is, branching takes place one instruction later than where the branch instruction appears in the instruction sequence in the memory, hence the name “delayed branch.”

LOOP	Shift_left	R1
	Decrement	R2
	Branch=0	LOOP
NEXT	Add	R1,R3

(a) Original program loop

LOOP	Decrement	R2
	Branch=0	LOOP
	Shift_left	R1
NEXT	Add	R1,R3

(b) Reordered instructions

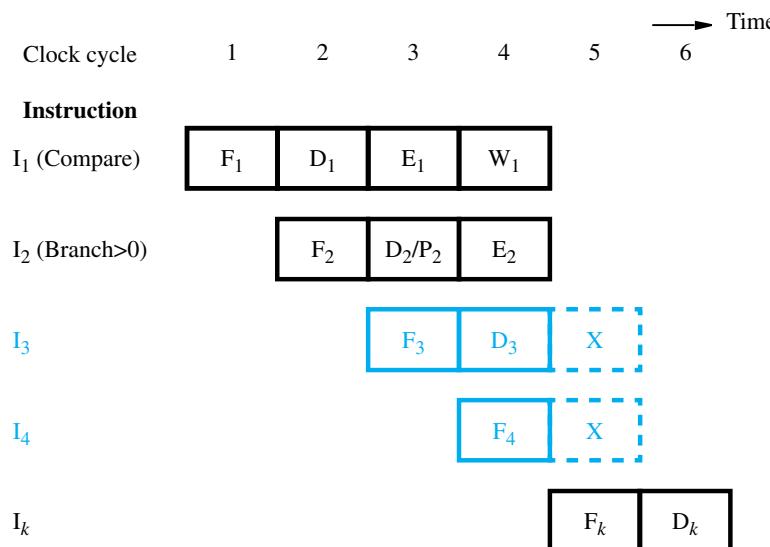
**Figure 8.12** Reordering of instructions for a delayed branch.**Figure 8.13** Execution timing showing the delay slot being filled during the last two passes through the loop in Figure 8.12b.

The effectiveness of the delayed branch approach depends on how often it is possible to reorder instructions as in Figure 8.12. Experimental data collected from many programs indicate that sophisticated compilation techniques can use one branch delay slot in as many as 85 percent of the cases. For a processor with two branch delay slots, the compiler attempts to find two instructions preceding the branch instruction that it can move into the delay slots without introducing a logical error. The chances of finding two such instructions are considerably less than the chances of finding one. Thus, if increasing the number of pipeline stages involves an increase in the number of branch delay slots, the potential gain in performance may not be fully realized.

### Branch Prediction

Another technique for reducing the branch penalty associated with conditional branches is to attempt to predict whether or not a particular branch will be taken. The simplest form of branch prediction is to assume that the branch will not take place and to continue to fetch instructions in sequential address order. Until the branch condition is evaluated, instruction execution along the predicted path must be done on a speculative basis. *Speculative execution* means that instructions are executed before the processor is certain that they are in the correct execution sequence. Hence, care must be taken that no processor registers or memory locations are updated until it is confirmed that these instructions should indeed be executed. If the branch decision indicates otherwise, the instructions and all their associated data in the execution units must be purged, and the correct instructions fetched and executed.

An incorrectly predicted branch is illustrated in Figure 8.14 for a four-stage pipeline. The figure shows a Compare instruction followed by a Branch>0 instruction. Branch



**Figure 8.14** Timing when a branch decision has been incorrectly predicted as not taken.

prediction takes place in cycle 3, while instruction  $I_3$  is being fetched. The fetch unit predicts that the branch will not be taken, and it continues to fetch instruction  $I_4$  as  $I_3$  enters the Decode stage. The results of the compare operation are available at the end of cycle 3. Assuming that they are forwarded immediately to the instruction fetch unit, the branch condition is evaluated in cycle 4. At this point, the instruction fetch unit realizes that the prediction was incorrect, and the two instructions in the execution pipe are purged. A new instruction,  $I_k$ , is fetched from the branch target address in clock cycle 5.

If branch outcomes were random, then half the branches would be taken. Then the simple approach of assuming that branches will not be taken would save the time lost to conditional branches 50 percent of the time. However, better performance can be achieved if we arrange for some branch instructions to be predicted as taken and others as not taken, depending on the expected program behavior. For example, a branch instruction at the end of a loop causes a branch to the start of the loop for every pass through the loop except the last one. Hence, it is advantageous to assume that this branch will be taken and to have the instruction fetch unit start to fetch instructions at the branch target address. On the other hand, for a branch instruction at the beginning of a program loop, it is advantageous to assume that the branch will not be taken.

A decision on which way to predict the result of the branch may be made in hardware by observing whether the target address of the branch is lower than or higher than the address of the branch instruction. A more flexible approach is to have the compiler decide whether a given branch instruction should be predicted taken or not taken. The branch instructions of some processors, such as SPARC, include a branch prediction bit, which is set to 0 or 1 by the compiler to indicate the desired behavior. The instruction fetch unit checks this bit to predict whether the branch will be taken or not taken.

With either of these schemes, the branch prediction decision is always the same every time a given instruction is executed. Any approach that has this characteristic is called *static branch prediction*. Another approach in which the prediction decision may change depending on execution history is called *dynamic branch prediction*.

### **Dynamic Branch Prediction**

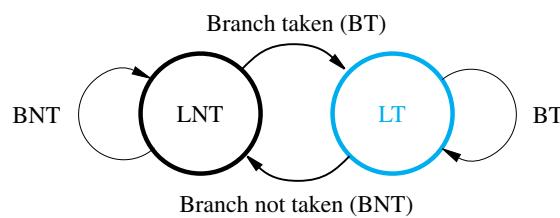
The objective of branch prediction algorithms is to reduce the probability of making a wrong decision, to avoid fetching instructions that eventually have to be discarded. In dynamic branch prediction schemes, the processor hardware assesses the likelihood of a given branch being taken by keeping track of branch decisions every time that instruction is executed.

In its simplest form, the execution history used in predicting the outcome of a given branch instruction is the result of the most recent execution of that instruction. The processor assumes that the next time the instruction is executed, the result is likely to be the same. Hence, the algorithm may be described by the two-state machine in Figure 8.15a. The two states are:

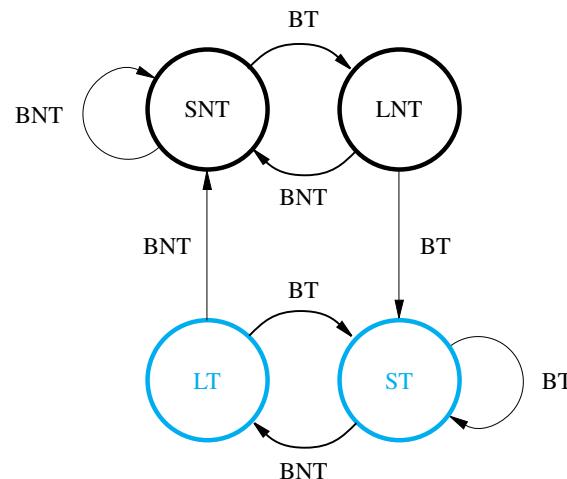
LT: Branch is likely to be taken

LNT: Branch is likely not to be taken

Suppose that the algorithm is started in state LNT. When the branch instruction is



(a) A 2-state algorithm



(b) A 4-state algorithm

**Figure 8.15** State-machine representation of branch-prediction algorithms.

executed and if the branch is taken, the machine moves to state LT. Otherwise, it remains in state LNT. The next time the same instruction is encountered, the branch is predicted as taken if the corresponding state machine is in state LT. Otherwise it is predicted as not taken.

This simple scheme, which requires one bit of history information for each branch instruction, works well inside program loops. Once a loop is entered, the branch instruction that controls looping will always yield the same result until the last pass through the loop is reached. In the last pass, the branch prediction will turn out to be incorrect, and the branch history state machine will be changed to the opposite state. Unfortunately, this means that the next time this same loop is entered, and assuming that there will be more than one pass through the loop, the machine will lead to the wrong prediction.

Better performance can be achieved by keeping more information about execution history. An algorithm that uses 4 states, thus requiring two bits of history information for each branch instruction, is shown in Figure 8.15b. The four states are:

- ST: Strongly likely to be taken
- LT: Likely to be taken
- LNT: Likely not to be taken
- SNT: Strongly likely not to be taken

Again assume that the state of the algorithm is initially set to LNT. After the branch instruction has been executed, and if the branch is actually taken, the state is changed to ST; otherwise, it is changed to SNT. As program execution progresses and the same instruction is encountered again, the state of the branch prediction algorithm continues to change as shown. When a branch instruction is encountered, the instruction fetch unit predicts that the branch will be taken if the state is either LT or ST, and it begins to fetch instructions at the branch target address. Otherwise, it continues to fetch instructions in sequential address order.

It is instructive to examine the behavior of the branch prediction algorithm in some detail. When in state SNT, the instruction fetch unit predicts that the branch will not be taken. If the branch is actually taken, that is if the prediction is incorrect, the state changes to LNT. This means that the next time the same branch instruction is encountered, the instruction fetch unit will still predict that the branch will not be taken. Only if the prediction is incorrect twice in a row will the state change to ST. After that, the branch will be predicted as taken.

Let us reconsider what happens when executing a program loop. Assume that the branch instruction is at the end of the loop and that the processor sets the initial state of the algorithm to LNT. During the first pass, the prediction will be wrong (not taken), and hence the state will be changed to ST. In all subsequent passes the prediction will be correct, except for the last pass. At that time, the state will change to LT. When the loop is entered a second time, the prediction will be correct (branch taken).

We now add one final modification to correct the mispredicted branch at the time the loop is first entered. The cause of the misprediction in this case is the initial state of the branch prediction algorithm. In the absence of additional information about the nature of the branch instruction, we assumed that the processor sets the initial state to LNT. The information needed to set the initial state correctly can be provided by any of the static prediction schemes discussed earlier. Either by comparing addresses or by checking a prediction bit in the instruction, the processor sets the initial state of the algorithm to LNT or LT. In the case of a branch at the end of a loop, the compiler would indicate that the branch should be predicted as taken, causing the initial state to be set to LT. With this modification, branch prediction will be correct all the time, except for the final pass through the loop. Misprediction in this latter case is unavoidable.

The state information used in dynamic branch prediction algorithms may be kept by the processor in a variety of ways. It may be recorded in a look-up table, which is accessed using the low-order part of the branch instruction address. In this case, it is possible for two branch instructions to share the same table entry. This may lead to a

branch being mispredicted, but it does not cause an error in execution. Misprediction only introduces a small delay in execution time. An alternative approach is to store the history bits as a tag associated with branch instructions in the instruction cache. We will see in Section 8.7 how this information is handled in the SPARC processor.

## 8.4 INFLUENCE ON INSTRUCTION SETS

We have seen that some instructions are much better suited to pipelined execution than others. For example, instruction side effects can lead to undesirable data dependencies. In this section, we examine the relationship between pipelined execution and machine instruction features. We discuss two key aspects of machine instructions — addressing modes and condition code flags.

### 8.4.1 ADDRESSING MODES

Addressing modes should provide the means for accessing a variety of data structures simply and efficiently. Useful addressing modes include index, indirect, autoincrement, and autodecrement. Many processors provide various combinations of these modes to increase the flexibility of their instruction sets. Complex addressing modes, such as those involving double indexing, are often encountered.

In choosing the addressing modes to be implemented in a pipelined processor, we must consider the effect of each addressing mode on instruction flow in the pipeline. Two important considerations in this regard are the side effects of modes such as autoincrement and autodecrement and the extent to which complex addressing modes cause the pipeline to stall. Another important factor is whether a given mode is likely to be used by compilers.

To compare various approaches, we assume a simple model for accessing operands in the memory. The load instruction Load X(R1),R2 takes five cycles to complete execution, as indicated in Figure 8.5. However, the instruction

Load (R1),R2

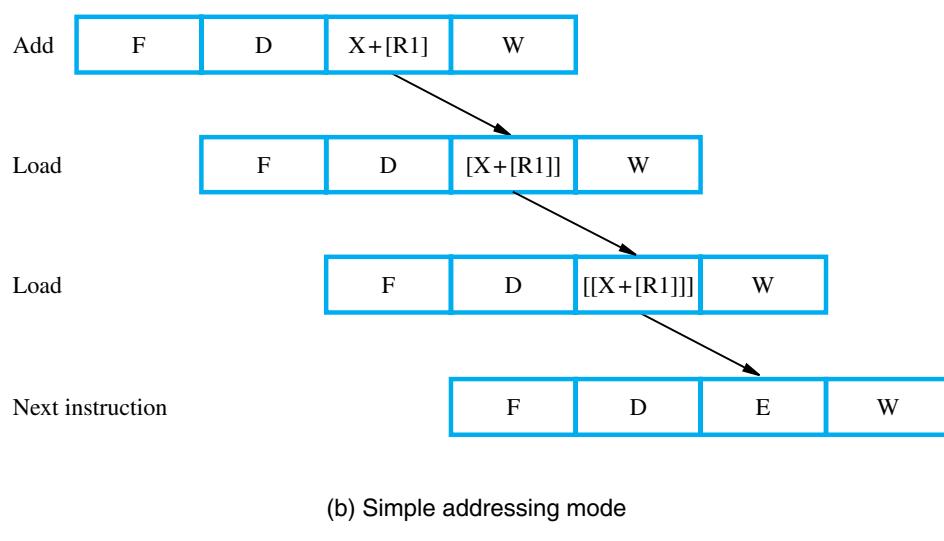
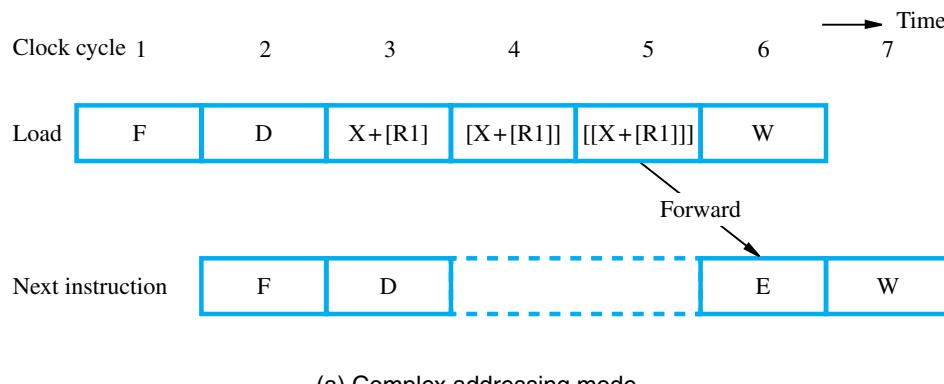
can be organized to fit a four-stage pipeline because no address computation is required. Access to memory can take place in stage E. A more complex addressing mode may require several accesses to the memory to reach the named operand. For example, the instruction

Load (X(R1)),R2

may be executed as shown in Figure 8.16a, assuming that the index offset, X, is given in the instruction word. After computing the address in cycle 3, the processor needs to access memory twice — first to read location  $X+[R1]$  in clock cycle 4 and then to read location  $[X+[R1]]$  in cycle 5. If R2 is a source operand in the next instruction, that instruction would be stalled for three cycles, which can be reduced to two cycles with operand forwarding, as shown.

## 8.4 INFLUENCE ON INSTRUCTION SETS

477



**Figure 8.16** Equivalent operations using complex and simple addressing modes.

To implement the same Load operation using only simple addressing modes requires several instructions. For example, on a computer that allows three operand addresses, we can use

```
    Add #X,R1,R2
    Load (R2),R2
    Load (R2),R2
```

The Add instruction performs the operation  $R2 \leftarrow X + [R1]$ . The two Load instructions fetch the address and then the operand from the memory. This sequence of instructions takes exactly the same number of clock cycles as the original, single Load instruction, as shown in Figure 8.16b.

This example indicates that, in a pipelined processor, complex addressing modes that involve several accesses to the memory do not necessarily lead to faster execution. The main advantage of such modes is that they reduce the number of instructions needed to perform a given task and thereby reduce the program space needed in the main memory. Their main disadvantage is that their long execution times cause the pipeline to stall, thus reducing its effectiveness. They require more complex hardware to decode and execute them. Also, they are not convenient for compilers to work with.

The instruction sets of modern processors are designed to take maximum advantage of pipelined hardware. Because complex addressing modes are not suitable for pipelined execution, they should be avoided. The addressing modes used in modern processors often have the following features:

- Access to an operand does not require more than one access to the memory.
- Only load and store instructions access memory operands.
- The addressing modes used do not have side effects.

Three basic addressing modes that have these features are register, register indirect, and index. The first two require no address computation. In the index mode, the address can be computed in one cycle, whether the index value is given in the instruction or in a register. Memory is accessed in the following cycle. None of these modes has any side effects, with one possible exception. Some architectures, such as ARM, allow the address computed in the index mode to be written back into the index register. This is a side effect that would not be allowed under the guidelines above. Note also that relative addressing can be used; this is a special case of indexed addressing in which the program counter is used as the index register.

The three features just listed were first emphasized as part of the concept of RISC processors. The SPARC processor architecture, which adheres to these guidelines, is presented in Section 8.7.

### 8.4.2 CONDITION CODES

In many processors, such as those described in Chapter 3, the condition code flags are stored in the processor status register. They are either set or cleared by many instructions, so that they can be tested by subsequent conditional branch instructions to change the flow of program execution. An optimizing compiler for a pipelined processor attempts to reorder instructions to avoid stalling the pipeline when branches or data dependencies between successive instructions occur. In doing so, the compiler must ensure that reordering does not cause a change in the outcome of a computation. The dependency introduced by the condition-code flags reduces the flexibility available for the compiler to reorder instructions.

Consider the sequence of instructions in Figure 8.17a, and assume that the execution of the Compare and Branch=0 instructions proceeds as in Figure 8.14. The branch decision takes place in step E<sub>2</sub> rather than D<sub>2</sub> because it must await the result of the Compare instruction. The execution time of the Branch instruction can be reduced

Add	R1,R2
Compare	R3,R4
Branch=0	...

(a) A program fragment

Compare	R3,R4
Add	R1,R2
Branch=0	...

(b) Instructions reordered

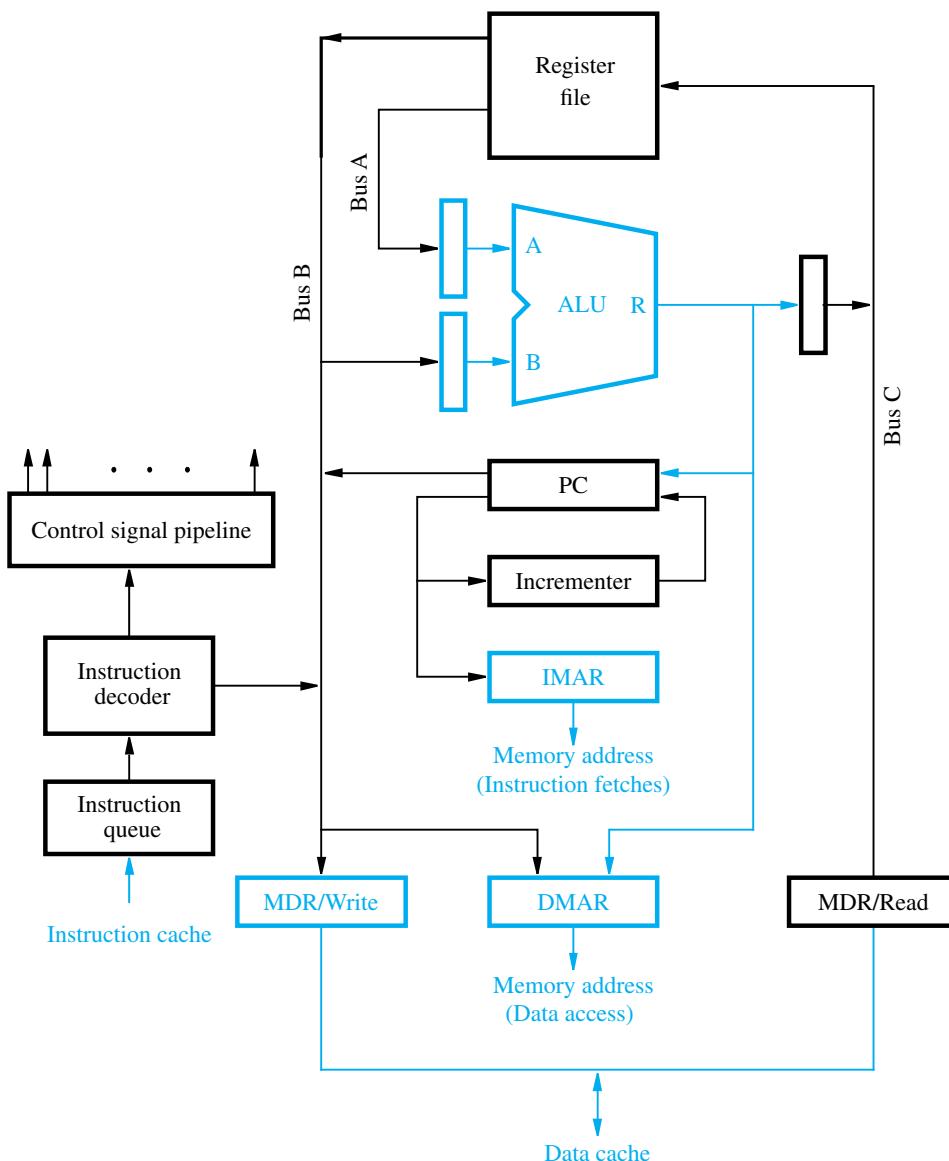
**Figure 8.17** Instruction reordering.

by interchanging the Add and Compare instructions, as shown in Figure 8.17b. This will delay the branch instruction by one cycle relative to the Compare instruction. As a result, at the time the Branch instruction is being decoded the result of the Compare instruction will be available and a correct branch decision will be made. There would be no need for branch prediction. However, interchanging the Add and Compare instructions can be done only if the Add instruction does not affect the condition codes.

These observations lead to two important conclusions about the way condition codes should be handled. First, to provide flexibility in reordering instructions, the condition-code flags should be affected by as few instructions as possible. Second, the compiler should be able to specify in which instructions of a program the condition codes are affected and in which they are not. An instruction set designed with pipelining in mind usually provides the desired flexibility. Figure 8.17b shows the instructions reordered assuming that the condition code flags are affected only when this is explicitly stated as part of the instruction OP code. The SPARC and ARM architectures provide this flexibility.

## 8.5 DATAPATH AND CONTROL CONSIDERATIONS

Organization of the internal datapath of a processor was introduced in Chapter 7. Consider the three-bus structure presented in Figure 7.8. To make it suitable for pipelined execution, it can be modified as shown in Figure 8.18 to support a 4-stage pipeline. The resources involved in stages F and E are shown in blue and those used in stages D and W in black. Operations in the data cache may happen during stage E or at a later stage, depending on the addressing mode and the implementation details. This section



**Figure 8.18** Datapath modified for pipelined execution with interstage buffers at the input and output of the ALU.

is shown in blue. Several important changes to Figure 7.8 should be noted:

1. There are separate instruction and data caches that use separate address and data connections to the processor. This requires two versions of the MAR register, IMAR for accessing the instruction cache and DMAR for accessing the data cache.
2. The PC is connected directly to the IMAR, so that the contents of the PC can be transferred to IMAR at the same time that an independent ALU operation is taking place.

3. The data address in DMAR can be obtained directly from the register file or from the ALU to support the register indirect and indexed addressing modes.

4. Separate MDR registers are provided for read and write operations. Data can be transferred directly between these registers and the register file during load and store operations without the need to pass through the ALU.

5. Buffer registers have been introduced at the inputs and output of the ALU. These are registers SRC1, SRC2, and RSLT in Figure 8.7. Forwarding connections are not included in Figure 8.18. They may be added if desired.

6. The instruction register has been replaced with an instruction queue, which is loaded from the instruction cache.

7. The output of the instruction decoder is connected to the control signal pipeline. The need for buffering control signals and passing them from one stage to the next along with the instruction is discussed in Section 8.1. This pipeline holds the control signals in buffers B2 and B3 in Figure 8.2a.

The following operations can be performed independently in the processor of Figure 8.18:

- Reading an instruction from the instruction cache
- Incrementing the PC
- Decoding an instruction
- Reading from or writing into the data cache
- Reading the contents of up to two registers from the register file
- Writing into one register in the register file
- Performing an ALU operation

Because these operations do not use any shared resources, they can be performed simultaneously in any combination. The structure provides the flexibility required to implement the four-stage pipeline in Figure 8.2. For example, let  $I_1$ ,  $I_2$ ,  $I_3$ , and  $I_4$  be a sequence of four instructions. As shown in Figure 8.2a, the following actions all happen during clock cycle 4:

- Write the result of instruction  $I_1$  into the register file
- Read the operands of instruction  $I_2$  from the register file
- Decode instruction  $I_3$
- Fetch instruction  $I_4$  and increment the PC.

## 8.6 SUPERSCALAR OPERATION

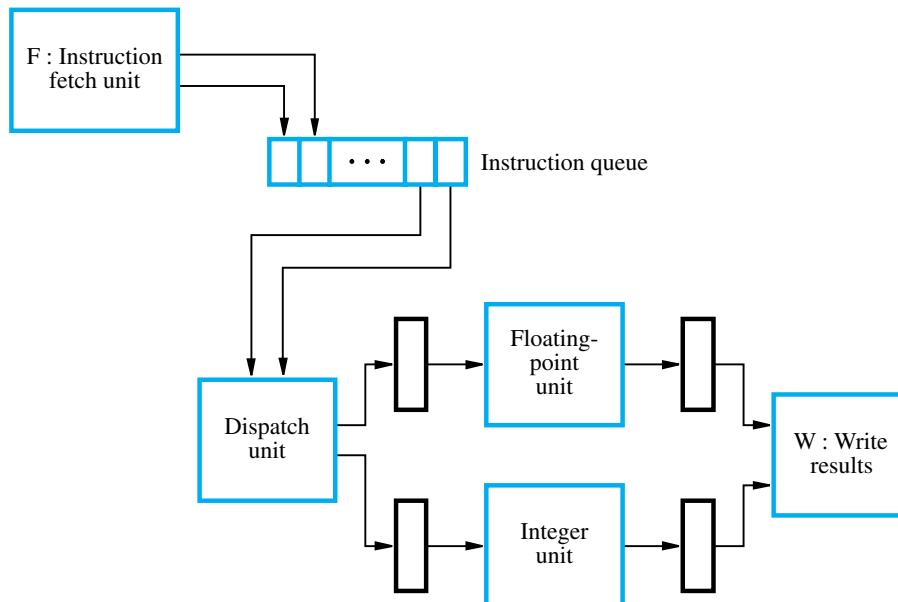
Pipelining makes it possible to execute instructions concurrently. Several instructions are present in the pipeline at the same time, but they are in different stages of their execution. While one instruction is performing an ALU operation, another instruction is being decoded and yet another is being fetched from the memory. Instructions enter the pipeline in strict program order. In the absence of hazards, one instruction enters the pipeline and one instruction completes execution in each clock cycle. This means that the maximum throughput of a pipelined processor is one instruction per clock cycle.

A more aggressive approach is to equip the processor with multiple processing units to handle several instructions in parallel in each processing stage. With this arrangement, several instructions start execution in the same clock cycle, and the processor is said to use *multiple-issue*. Such processors are capable of achieving an instruction execution throughput of more than one instruction per cycle. They are known as *superscalar* processors. Many modern high-performance processors use this approach.

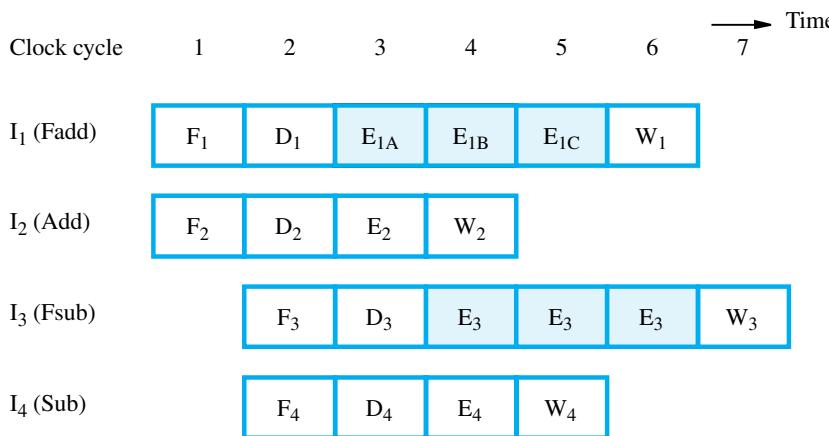
We introduced the idea of an instruction queue in Section 8.3. We pointed out that to keep the instruction queue filled, a processor should be able to fetch more than one instruction at a time from the cache. For superscalar operation, this arrangement is essential. Multiple-issue operation requires a wider path to the cache and multiple execution units. Separate execution units are provided for integer and floating-point instructions.

Figure 8.19 shows an example of a processor with two execution units, one for integer and one for floating-point operations. The Instruction fetch unit is capable of reading two instructions at a time and storing them in the instruction queue. In each clock cycle, the Dispatch unit retrieves and decodes up to two instructions from the front of the queue. If there is one integer, one floating-point instruction, and no hazards, both instructions are dispatched in the same clock cycle.

In a superscalar processor, the detrimental effect on performance of various hazards becomes even more pronounced. The compiler can avoid many hazards through judicious selection and ordering of instructions. For example, for the processor in Figure 8.19, the compiler should strive to interleave floating-point and integer instructions. This would enable the dispatch unit to keep both the integer and floating-point



**Figure 8.19** A processor with two execution units.



**Figure 8.20** An example of instruction execution flow in the processor of Figure 8.19, assuming no hazards are encountered.

units busy most of the time. In general, high performance is achieved if the compiler is able to arrange program instructions to take maximum advantage of the available hardware units.

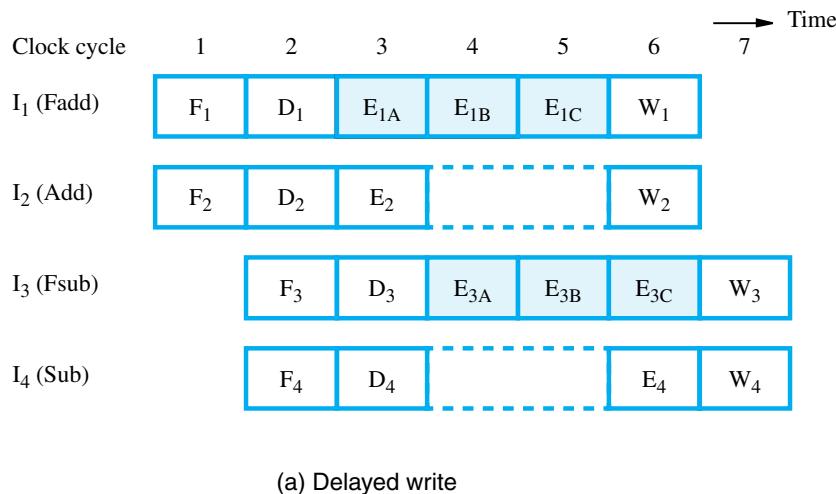
Pipeline timing is shown in Figure 8.20. The blue shading indicates operations in the floating-point unit. The floating-point unit takes three clock cycles to complete the floating-point operation specified in I<sub>1</sub>. The integer unit completes execution of I<sub>2</sub> in one clock cycle. We have also assumed that the floating-point unit is organized internally as a three-stage pipeline. Thus, it can still accept a new instruction in each clock cycle. Hence, instructions I<sub>3</sub> and I<sub>4</sub> enter the dispatch unit in cycle 3, and both are dispatched in cycle 4. The integer unit can receive a new instruction because instruction I<sub>2</sub> has proceeded to the Write stage. Instruction I<sub>1</sub> is still in the execution phase, but it has moved to the second stage of the internal pipeline in the floating-point unit. Therefore, instruction I<sub>3</sub> can enter the first stage. Assuming that no hazards are encountered, the instructions complete execution as shown.

### 8.6.1 OUT-OF-ORDER EXECUTION

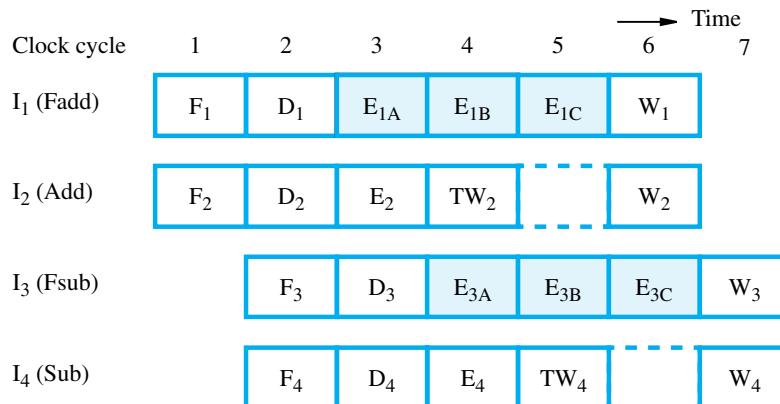
In Figure 8.20, instructions are dispatched in the same order as they appear in the program. However, their execution is completed out of order. Does this lead to any problems? We have already discussed the issues arising from dependencies among instructions. For example, if instruction I<sub>2</sub> depends on the result of I<sub>1</sub>, the execution of I<sub>2</sub> will be delayed. As long as such dependencies are handled correctly, there is no reason to delay the execution of an instruction. However, a new complication arises when we consider the possibility of an instruction causing an exception. Exceptions may be caused by a bus error during an operand fetch or by an illegal operation, such as an attempt to divide by zero. The results of I<sub>2</sub> are written back into the register file in

cycle 4. If instruction  $I_1$  causes an exception, program execution is in an inconsistent state. The program counter points to the instruction in which the exception occurred. However, one or more of the succeeding instructions have been executed to completion. If such a situation is permitted, the processor is said to have *imprecise exceptions*.

To guarantee a consistent state when exceptions occur, the results of the execution of instructions must be written into the destination locations strictly in program order. This means we must delay step  $W_2$  in Figure 8.20 until cycle 6. In turn, the integer execution unit must retain the result of instruction  $I_2$ , and hence it cannot accept instruction  $I_4$  until cycle 6, as shown in Figure 8.21a. If an exception occurs during an instruction,



(a) Delayed write



(b) Using temporary registers

**Figure 8.21** Instruction completion in program order.

all subsequent instructions that may have been partially executed are discarded. This is called a *precise exception*.

It is easier to provide precise exceptions in the case of external interrupts. When an external interrupt is received, the Dispatch unit stops reading new instructions from the instruction queue, and the instructions remaining in the queue are discarded. All instructions whose execution is pending continue to completion. At this point, the processor and all its registers are in a consistent state, and interrupt processing can begin.

### 8.6.2 EXECUTION COMPLETION

It is desirable to use out-of-order execution, so that an execution unit is freed to execute other instructions as soon as possible. At the same time, instructions must be completed in program order to allow precise exceptions. These seemingly conflicting requirements are readily resolved if execution is allowed to proceed as shown in Figure 8.20, but the results are written into temporary registers. The contents of these registers are later transferred to the permanent registers in correct program order. This approach is illustrated in Figure 8.21b. Step TW is a write into a temporary register. Step W is the final step in which the contents of the temporary register are transferred into the appropriate permanent register. This step is often called the *commitment* step because the effect of the instruction cannot be reversed after that point. If an instruction causes an exception, the results of any subsequent instruction that has been executed would still be in temporary registers and can be safely discarded.

A temporary register assumes the role of the permanent register whose data it is holding and is given the same name. For example, if the destination register of  $I_2$  is R5, the temporary register used in step  $TW_2$  is treated as R5 during clock cycles 6 and 7. Its contents would be forwarded to any subsequent instruction that refers to R5 during that period. Because of this feature, this technique is called *register renaming*. Note that the temporary register is used only for instructions that follow  $I_2$  in program order. If an instruction that precedes  $I_2$  needs to read R5 in cycle 6 or 7, it would access the actual register R5, which still contains data that have not been modified by instruction  $I_2$ .

When out-of-order execution is allowed, a special control unit is needed to guarantee in-order commitment. This is called the *commitment unit*. It uses a queue called the *reorder buffer* to determine which instruction(s) should be committed next. Instructions are entered in the queue strictly in program order as they are dispatched for execution. When an instruction reaches the head of that queue and the execution of that instruction has been completed, the corresponding results are transferred from the temporary registers to the permanent registers and the instruction is removed from the queue. All resources that were assigned to the instruction, including the temporary registers, are released. The instruction is said to have been *retired* at this point. Because an instruction is retired only when it is at the head of the queue, all instructions that were dispatched before it must also have been retired. Hence, instructions may complete execution out of order, but they are retired in program order.

### 8.6.3 DISPATCH OPERATION

We now return to the dispatch operation. When dispatching decisions are made, the dispatch unit must ensure that all the resources needed for the execution of an instruction are available. For example, since the results of an instruction may have to be written in a temporary register, the required register must be free, and it is reserved for use by that instruction as a part of the dispatch operation. A location in the reorder buffer must also be available for the instruction. When all the resources needed are assigned, including an appropriate execution unit, the instruction is dispatched.

Should instructions be dispatched out of order? For example, if instruction  $I_2$  in Figure 8.20b is delayed because of a cache miss for a source operand, the integer unit will be busy in cycle 4, and  $I_4$  cannot be dispatched. Should  $I_5$  be dispatched instead? In principle this is possible, provided that a place is reserved in the reorder buffer for instruction  $I_4$  to ensure that all instructions are retired in the correct order. Dispatching instructions out of order requires considerable care. If  $I_5$  is dispatched while  $I_4$  is still waiting for some resource, we must ensure that there is no possibility of a deadlock occurring.

A *deadlock* is a situation that can arise when two units, A and B, use a shared resource. Suppose that unit B cannot complete its task until unit A completes its task. At the same time, unit B has been assigned a resource that unit A needs. If this happens, neither unit can complete its task. Unit A is waiting for the resource it needs, which is being held by unit B. At the same time, unit B is waiting for unit A to finish before it can release that resource.

If instructions are dispatched out of order, a deadlock can arise as follows. Suppose that the processor has only one temporary register, and that when  $I_5$  is dispatched, that register is reserved for it. Instruction  $I_4$  cannot be dispatched because it is waiting for the temporary register, which, in turn, will not become free until instruction  $I_5$  is retired. Since instruction  $I_5$  cannot be retired before  $I_4$ , we have a deadlock.

To prevent deadlocks, the dispatcher must take many factors into account. Hence, issuing instructions out of order is likely to increase the complexity of the Dispatch unit significantly. It may also mean that more time is required to make dispatching decisions. For these reasons, most processors use only in-order dispatching. Thus, the program order of instructions is enforced at the time instructions are dispatched and again at the time instructions are retired. Between these two events, the execution of several instructions can proceed at their own speed, subject only to any interdependencies that may exist among instructions.

In the next section, we present the UltraSPARC II as a case study of a commercially successful, superscalar, highly pipelined processor. The way in which the various issues raised in this chapter have been handled in this processor and the choices made are highly instructive.

### 8.7 UltraSPARC II EXAMPLE

Processor design has advanced greatly in recent years. The classification of processors as either purely RISC or CISC is no longer appropriate because modern high-performance processors contain elements of both design styles.

The early RISC processors showed how certain features can contribute to high performance. The following two observations proved to be particularly important:

- Pipelining, which enables a processor to execute several instructions at the same time, can lead to significant performance enhancements provided that the pipeline is not stalled frequently.
- A close synergy between the hardware and compiler design enables the compiler to take maximum advantage of the pipelined structure by reducing the events that lead to pipeline stalls.

It is these factors, rather than simply a reduced instruction set, that have contributed to the success of RISC processors. Of particular importance in this regard is the close co-ordination between the design of the hardware, particularly the structure of the pipeline, and the compiler. Much of the credit for today's high levels of performance goes to developments in compiler technology, which in turn have led to new hardware features that would have been of little use a few years ago.

The SPARC architecture, which is the basis for the processors used in Sun workstations, is an excellent case in point. One of Sun's implementations of the SPARC architecture is called UltraSPARC II. This is the processor we will discuss. We have chosen it instead of one of the processors presented in Chapter 3 because it illustrates very well superscalar operation as well as most of the pipeline design options and trade-offs discussed in this chapter. We will start with a brief introduction to the SPARC architecture. For a complete description, the reader should consult the SPARC Architecture Manual [1].

### 8.7.1 SPARC Architecture

SPARC stands for Scalable Processor ARChitecture. It is a specification of the instruction set architecture of a processor, that is, it is a specification of the processor's instruction set and register organization, regardless of how these may be implemented in hardware. Furthermore, SPARC is an "open architecture," which means that computer companies other than Sun Microsystems can develop their own hardware to implement the same instruction set.

The SPARC architecture was first announced in 1987, based on ideas developed at the University of California at Berkeley in the early eighties, in a project that coined the name reduced instruction set computer and its corresponding acronym RISC. The Sun Corporation and several other processor chip manufacturers have designed and built many processors based on this architecture, covering a wide range of performance. The SPARC architecture specifications are controlled by an international consortium, which introduces new enhanced versions every few years. The most recent version is SPARC-V9.

The instruction set of the SPARC architecture has a distinct RISC style. The architecture specifications describe a processor in which data and memory addresses are 64 bits long. Instructions are of equal length, and they are all 32 bits long. Both integer and floating-point instructions are provided.

There are two register files, one for integer data and one for floating-point data. Integer registers are 64 bits long. Their number is implementation dependent and can vary from 64 to 528. SPARC uses a scheme known as *register windows*. At any given time, an application program sees only 32 registers, called R0 to R31. Of these, the first eight are global registers that are always accessible. The remaining 24 registers are local to the current context.

Floating-point registers are only 32 bits long because this is the length of single-precision floating-point numbers according to the IEEE Standard described in Chapter 6. The instruction set includes floating-point instructions for double- and quad-precision operations. Two sequentially numbered floating-point registers are used to hold a double-precision operand and four are used for quad precision. There is a total of 64 registers, F0 to F63. Single precision operands can be stored in F0 to F31, double precision operands in F0, F2, F4, ..., F62, and quad-precision in F0, F4, F8, ..., F60.

### Load and Store Instructions

Only load and store instructions access the memory, where an operand may be an 8-bit byte, a 16-bit half word, or a 32-bit word. Load and store instructions also handle 64-bit quantities, which come in two varieties: extended word or doubleword. An LDX (Load extended) instruction loads a 64-bit quantity, called an *extended word*, into one of the processor's integer registers. A *doubleword* consists of two 32-bit words. The two words are loaded into two sequentially numbered processor registers using a single LDD (Load double) instruction. They are loaded into the low-order 32 bits of each register, and the high order bits are filled with 0s. The first of the two registers, which is the register named in the instruction, must be even numbered. Load and store instructions that handle doublewords are useful for moving multiple-precision floating-point operands between the memory and floating-point registers.

Load and store instructions use one of two indexed addressing modes, as follows:

1. The effective address is the sum of the contents of two registers:

$$EA = [Radr1] + [Radr2]$$

2. The effective address is the sum of the contents of one register plus an immediate operand that is included in the instruction

$$EA = [Radr1] + \text{Immediate}$$

For most instructions, the immediate operand is a signed 13-bit value. It is sign-extended to 64 bits and then added to the contents of Radr1.

A load instruction that uses the first addressing mode is written as

Load [Radr1+Radr2], Rdst

It generates the effective address [Radr1] + [Radr2] and loads the contents of that location into register Rdst. For an immediate displacement, Radr2 is replaced with the

**8.7 UltraSPARC II EXAMPLE****489**

immediate operand value, which yields

Load [Radr1+Imm], Rdst

Store instructions use a similar syntax, with the first operand specifying the source register from which data will be stored in the memory, as follows:

Store Rsrc, [Radr1+Radr2]

Store Rsrc, [Radr1+Imm]

In the recommended syntax for SPARC instructions, a register is specified by a % sign followed by the register number. Either %r2 or %2 refers to register number 2. However, for better readability and consistency with earlier chapters, we will use R0, R1, and so on, to refer to integer registers and F0, F1, . . . for floating-point registers.

As an example, consider the Load unsigned byte instruction

LDUB [R2+R3], R4

This instruction loads one byte from memory location  $[R2] + [R3]$  into the low-order 8 bits of register R4, and fills the high-order 56 bits with 0s. The Load signed word instruction:

LDSW [R2+2500], R4

reads a 32-bit word from location  $[R2] + 2500$ , sign extends it to 64 bits, and then stores it in register R4.

### **Arithmetic and Logic Instructions**

The usual set of arithmetic and logic instructions is provided. A few examples are shown in Table 8.1. We pointed out in Section 8.4.2 that an instruction should set the condition code flags only when these flags are going to be tested by a subsequent conditional branch instruction. This maximizes the flexibility the compiler has in re-ordering instructions to avoid stalling the pipeline. The SPARC instruction set has been designed with this feature in mind. Arithmetic and logic instructions are available in two versions, one sets the condition code flags and the other does not. The suffix cc in an OP code is used to indicate that the flags should be set. For example, the instructions ADD, SUB, SMUL (signed multiply), OR, and XOR do not affect the flags, while ADDcc and SUBcc do.

Register R0 always contains the value 0. When it is used as the destination operand, the result of the instruction is discarded. For example, the instruction

SUBcc R2, R3, R0

subtracts the contents of R3 from R2, sets the condition code flags, and discards the result of the subtraction operation. In effect, this is a compare instruction, and it has the alternative syntax

CMP R2, R3

In the SPARC nomenclature, CMP is called a synthetic instruction. It is not a real

**Table 8.1** Examples of SPARC instructions

Instruction	Description
ADD R5, R6, R7	Integer add: $R7 \leftarrow [R5] + [R6]$
ADDcc R2, R3, R5	$R5 \leftarrow [R2] + [R3]$ , set condition code flags
SUB R5, Imm, R7	Integer subtract: $R7 \leftarrow [R5] - \text{Imm(sign-extended)}$
AND R3, Imm, R5	Bitwise AND: $R5 \leftarrow [R3] \text{ AND Imm(sign-extended)}$
XOR R3, R4, R5	Bitwise Exclusive OR: $R5 \leftarrow [R3] \text{ XOR } [R4]$
FADDq F4, F12, F16	Floating-point add, quad precision: $F12 \leftarrow [F4] + [F12]$
FSUBs F2, F5, F7	Floating-point subtract, single precision: $F7 \leftarrow [F2] - [F5]$
FDIVs F5, F10, F18	Floating-point divide, single precision, $F18 \leftarrow [F5]/[F10]$
LDSW R3, R5, R7	$R7 \leftarrow$ 32-bit word at $[R3] + [R5]$ sign extended to a 64-bit value
LDX R3, R5, R7	$R7 \leftarrow$ 64-bit extended word at $[R3] + [R5]$
LDUB R4, Imm, R5	Load unsigned byte from memory location $[R4] + \text{Imm}$ , the byte is loaded into the least significant 8 bits of register R5, and all higher-order bits are filled with 0s
STW R3, R6, R12	Store word from register R3 into memory location $[R6] + [R12]$
LDF R5, R6, F3	Load a 32-bit word at address $[R5] + [R6]$ into floating-point register F3
LDDF R5, R6, F8	Load doubleword (two 32-bit words) at address $[R5] + [R6]$ into floating-point registers F8 and F9
STF F14, R6, Imm	Store word from floating-register F14 into memory location $[R6] + \text{Imm}$
BLE icc, Label	Test the <i>icc</i> flags and branch to Label if less than or equal to zero
BZ,pn xcc, Label	Test the <i>xcc</i> flags and branch to Label if equal to zero, branch is predicted not taken
BGT,a,pt icc, Label	Test the 32-bit integer condition codes and branch to Label if greater than zero, set annul bit, branch is predicted taken
FBNE,pn Label	Test floating-point status flags and branch if not equal, the annul bit is set to zero, and the branch is predicted not taken

instruction recognized by the hardware. It is provided only for the convenience of the programmer. The assembler replaces it with a SUBcc instruction.

A condition code register, CCR, is provided, which contains two sets of condition code flags, *icc* and *xcc*, for integer and extended condition codes, respectively. Each set consists of four flags N, Z, V, and C. Instructions that set the condition code flags, such as ADDcc, will set both the *icc* and *xcc* bits; the *xcc* flags are set based on the 64-bit result of the instruction, and the *icc* flags are set based on the low-order 32 bits only.

The condition codes for floating-point operations are held in a 64-bit register called the floating-point state register, FSR.

### Branch Instructions

The way in which branches are handled is an important factor in determining performance. Branch instructions in the SPARC instruction set contain several features that are intended to enhance performance of a pipelined processor and to help the compiler in optimizing the code it emits.

A SPARC processor uses delayed branching with one delay slot (see Section 8.3.2). Branch instructions include a branch prediction bit, which the compiler can use to give the hardware a hint about the expected behavior of the branch. Branch instructions also contain an Annul bit, which is intended to increase flexibility in handling the instruction in the delay slot. This instruction is always executed, but its results are not committed until after the branch decision is known. If the branch is taken, execution of the instruction in the delay slot is completed and the results are committed. If the branch is not taken, this instruction is annulled if the Annul bit is equal to 1. Otherwise, execution of the instruction is completed.

The compiler may be able to place in the delay slot an instruction that is needed whether or not the branch is taken. This may be an instruction that logically belongs before the branch instruction but can be moved into the delay slot. The Annul bit should be set to 0 in this case. Otherwise, the delay slot should be filled with an instruction that is to be executed only if the branch is taken, in which case the Annul bit should be set to 1.

Conditional branch instructions can test the *icc*, *xcc*, or FSR flags. For example, the instruction

```
BGT,a,pt  icc, Label
```

will cause a branch to location Label if the previous instruction that set the flags in *icc* produced a greater-than-zero result. The instruction will have both the Annul bit and the branch prediction bit set to 1. The instruction

```
FBGT,a,pt  Label
```

is exactly the same, except that it will test the FSR flags. If neither *pt* (predicted taken) nor *pn* (predicted not taken) is specified, the assembler will default to *pt*.

An example that illustrates the prediction and annul facilities in branch instructions is given in Figure 8.22, which shows a program loop that adds a list of  $n$  64-bit integers. We have assumed that the number of items in the list is stored at address LIST as a 64-bit integer, followed by the numbers to be added in successive 64-bit locations. We have also assumed that there is at least one item in the list and that the address LIST has been loaded into register R3 earlier in the program.

Figure 8.22a shows the desired loop as it would be written for execution on a nonpipelined processor. For execution on a SPARC processor, we should first reorganize the instructions to make effective use of the branch delay slot. Observe that the ADD instruction following LOOPSTART is executed during every pass through the loop.

	LDX	R3, 0, R6	Load number of items in the list.
	OR	R0, R0, R4	R4 to be used as offset in the list
	OR	R0, R0, R7	Clear R7 to be used as accumulator.
LOOPSTART	LDX	R3, R4, R5	Load list item into R5.
	ADD	R5, R7, R7	Add number to accumulator.
	ADD	R4, 8, R4	Point to the next entry.
	SUBcc	R6, 1, R6	Decrement R6 and set condition flags.
	BG	xcc, LOOPSTART	Loop if more items in the list.
NEXT	...		

(a) Desired program loop

	LDX	R3, 0, R6	
	OR	R0, R0, R4	
	OR	R0, R0, R7	
LOOPSTART	LDX	R3, R4, R5	
	ADD	R4, 8, R4	
	SUBcc	R6, 1, R6	
	BG,pt	xcc, LOOPSTART	Predicted taken, Annul bit = 0
	ADD	R5, R7, R7	
NEXT	...		

(b) Instructions reorganized to use the delay slot

**Figure 8.22** An addition loop showing the use of the branch delay slot and branch prediction.

Also, none of the instructions following it depends on its result. Hence, this instruction may be moved into the delay slot following the branch at the end of the loop, as shown in Figure 8.22b. Since it is to be executed regardless of the branch outcome, the Annul bit in the branch instruction is set to 0 (this is the default condition).

As for branch prediction, observe that the number of times the loop will be executed is equal to the number of items in the list. This means that, except for the trivial case of  $n = 1$ , the branch will be taken a number of times before exiting the loop. Hence, we have set the branch prediction bit in the BG instruction to indicate that the branch is expected to be taken.

Conditional branch instructions are not the only instructions that check the condition code flags. For example, there is a conditional move instruction, MOVcc, which copies data from one register into another only if the condition codes satisfy the condition specified in the instruction suffix, cc. Consider the two instructions

```
CMP    R5, R6
MOVle  icc, R5, R6
```

The MOVle instruction copies the contents of R5 into R6 if the condition code flags in *icc* indicate a less-than-or-equal-to condition ( $Z + (N \oplus V) = 1$ ). The net result is

to place the smaller of the two values in register R6. In the absence of a conditional move instruction, the same task would require a branch instruction, as in the following sequence

```
CMP    R5, R6
BG     icc, GREATER
MOVA  icc, R5, R6
GREATER ...
```

where MOVA is the move-always instruction. The MOVle instruction not only reduces the number of instructions needed, but more importantly, it avoids the performance degradation caused by branch instructions in pipelined execution.

The instruction set has many other features that are intended to maximize performance in a highly pipelined superscalar processor. We will discuss some of these features in the context of the UltraSPARC II processor. The ideas behind these features have already been introduced earlier in the chapter.

### 8.7.2 UltraSPARC II

The main building blocks of the UltraSPARC II processor are shown in Figure 8.23. The processor uses two levels of cache: an external cache (E-cache) and two internal caches, one for instructions (I-cache) and one for data (D-cache). The external cache controller is on the processor chip, as is the control hardware for memory management. The memory management unit uses two translation lookaside buffers, one for instructions, iTLB, and one for data, dTLB. The processor communicates with the memory and the I/O subsystem over the system interconnection bus.

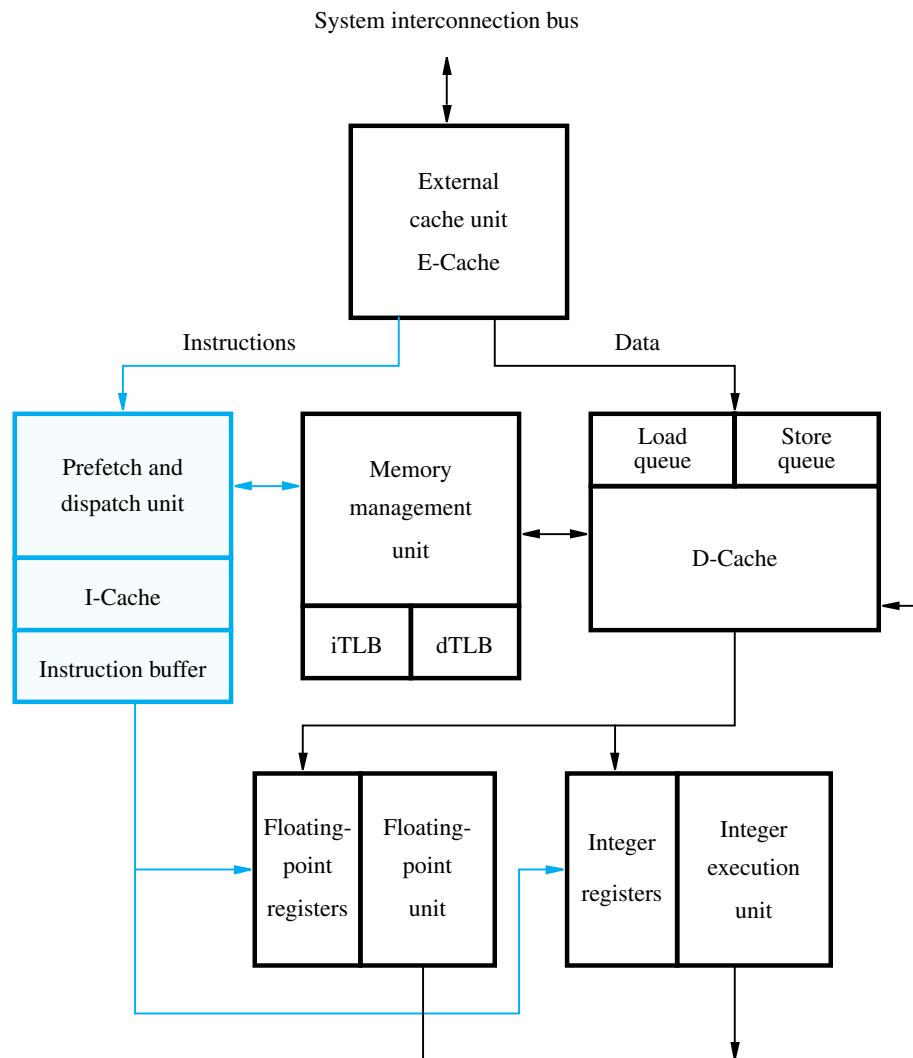
There are two execution units, one for integer and one for floating-point operations. Each of these units contains a register set and two independent pipelines for instruction execution. Thus, the processor can simultaneously start the execution of up to four instructions, two integer and two floating-point. These four instructions proceed in parallel, each through its own pipeline. If instructions are available and none of the four pipelines is stalled, four new instructions can enter the execution phase every clock cycle.

The Prefetch and Dispatch Unit (PDU) of the processor is responsible for maintaining a continuous supply of instructions for the execution units. It does so by prefetching instructions before they are needed and placing them in a temporary storage buffer called the instruction buffer, which performs the role of the instruction queue in Figure 8.19.

### 8.7.3 PIPELINE STRUCTURE

The UltraSPARC II has a nine-stage instruction execution pipeline, shown in Figure 8.24. The function of each stage is completed in one processor clock cycle. We will give an overview of the operation of the pipeline, then discuss each stage in detail.

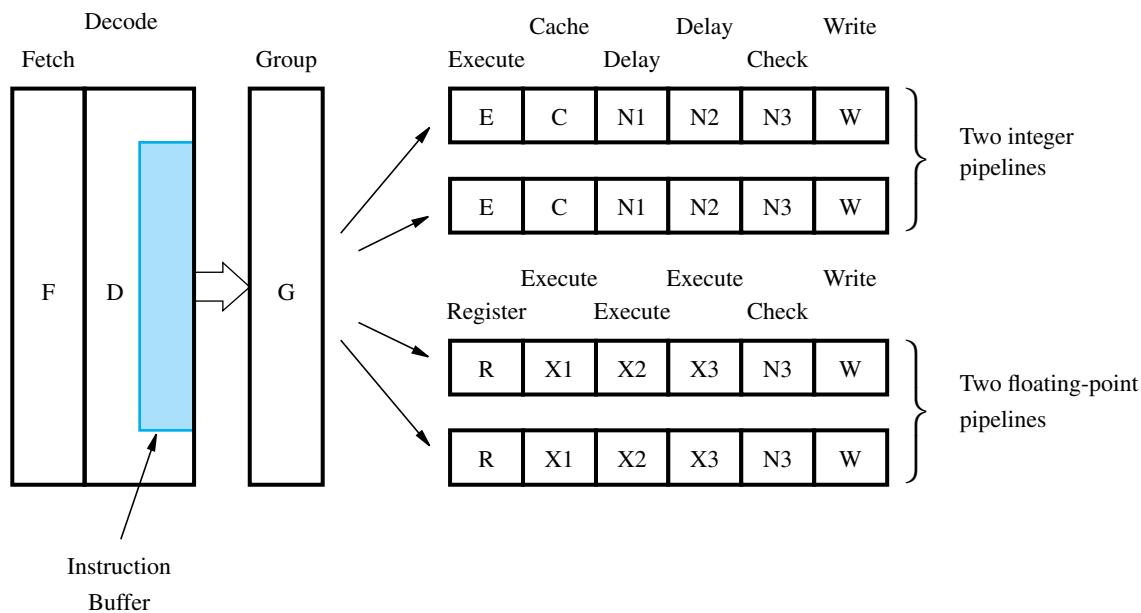
The first three stages of the pipeline are common to all instructions. Instructions



**Figure 8.23** Main building blocks of the UltraSPARC II processor.

are fetched from the instruction cache in the first stage (F) and partially decoded in the second stage (D). Then, in the third stage (G), a group of up to four instructions is selected for execution in parallel. The instructions are then dispatched to the integer and floating-point execution units.

Each of the two execution units consists of two parallel pipelines with six stages each. The first four stages are available to perform the operation specified by the instruction, and the last two are used to check for exceptions and store the result of the instruction.

**8.7 UltraSPARC II EXAMPLE****495****Figure 8.24** Pipeline organization of the UltraSPARC II processor.**Instruction Fetch and Decode**

The PDU fetches up to four instructions from the instruction cache, partially decodes them, and stores the results in the instruction buffer, which can hold up to 12 instructions. The decoding that takes place in this stage enables the PDU to determine whether the instruction is a branch instruction. It also detects salient features that can be used to speed up the decisions to be made later in the pipeline.

A cache block in the instruction cache consists of 32 bytes. It contains eight instructions. As instructions are loaded into the cache they are stored based on their virtual addresses, so that they can be fetched quickly by the PDU without requiring address translation. The PDU can maintain the rate of four instructions per cycle as long as each group does not cross cache block boundaries. If there are fewer than four instructions left in a cache block, the unit will read only the remaining instructions in the current block.

The PDU uses a four-state branch prediction algorithm similar to that described in Figure 8.15. It uses the branch prediction bit in the branch instruction to set the initial state to either LT or LNT. For every two instructions in the instruction cache, the PDU uses two bits to record the state of the branch prediction algorithm. These bits are stored in the cache, in a tag associated with the instructions.

For each four instructions in the instruction cache, a tag field is provided called Next Address. The PDU computes the target address of a branch instruction when the instruction is first fetched for execution, and it records this address in the Next Address field. This field makes it possible to continue prefetching instructions in subsequent passes, without having to recompute the target address each time. Since there is only

one Next Address field for each half of a cache line, its benefit can be fully realized only if there is at most one branch instruction in each group of four instructions.

### Grouping

In the third stage of the pipeline, stage G, the Grouping Logic selects a group of up to four instructions to be executed in parallel and dispatches them to the integer and floating-point execution units. Figure 8.25 shows a short instruction sequence and the way these instructions would be dispatched. Parts *b* and *c* of the figure show the instruction grouping when the PDU predicts that the branch will be taken and not taken, respectively. Note that the instruction in the delay slot, FCMP, is included in the selected group in both cases. It will be executed, but not committed until the branch decision is made. Its results will be annulled if the branch is not taken, because the Annul bit in the branch instruction is set to 1. The first two instructions in each group are dispatched to the integer unit and the next two to the floating-point unit.

ADDcc	R3, R4, R7	$R7 \leftarrow [R3] + [R4]$ , Set condition codes
BRZ,a	Label	Branch if zero, set Annul bit to 1
FCMP	F1, F5	FP: Compare [F2] and [F5]
FADD	F2, F3, F6	FP: $F6 \leftarrow [F2] + [F3]$
FMOVs	F3, F4	Move single precision operand from F3 to F4
:		
Label	FSUB	FP: $F6 \leftarrow [F2] - [F3]$
	LDSW	Load single word at location $[R3] + [R4]$ into R7
:		

(a) Program fragment

ADDcc	R3, R4, R7
BRZ,a	Label
FCMP	F1, F5
FSUB	F2, F3, F6

(b) Instruction grouping, branch taken

ADDcc	R3, R4, R7
BRZ,a	Label
FCMP	R1, R5
FADD	R2, R3, R6

(c) Instruction grouping, branch not taken

**Figure 8.25** Example of instruction grouping.

**8.7 UltraSPARC II EXAMPLE****497**

The grouping logic circuit is responsible for ensuring that the instructions it dispatches are ready for execution. For example, all the operands referenced by the instruction in a group must be available. No two instructions can be included in the same group if one of them depends on the result of the other. Branch instructions are exempted from this condition, as will be explained shortly.

Instructions are dispatched in program order. Recall that if a group includes a branch instruction, that instruction will have already been tentatively executed as a result of branch prediction in the prefetch and decode unit. Hence, the instructions in the instruction buffer will be in correct order based on this prediction. The grouping logic simply examines the instructions in the instruction buffer in order, with the objective of selecting the largest number at the head of the queue that satisfy the grouping constraints.

Some of the constraints that the grouping logic takes into account in selecting instructions to include in a group are:

1. Instructions can only be dispatched in sequence. If one instruction cannot be included in a group, no later instruction can be selected.

2. The source operand of an instruction cannot depend on the destination operand of any other instruction in the same group. There are two exceptions to this rule:

- A store instruction, which stores the contents of a register in the memory, may be grouped with an earlier instruction that has that register as a destination. This is allowed because, as we will see shortly, the store instruction does not require the data until a later stage in the pipeline.
- A branch instruction may be grouped with an earlier instruction that sets the condition codes.

3. No two instructions in a group can have the same destination operand, unless the destination is register R0. For example, the LDSW instruction in Figure 8.26a cannot be grouped with the ADD instruction and must be delayed to the next group as shown.

4. In some cases, certain instructions must be delayed two or three clock cycles relative to other instructions. For example, the conditional instruction

MOVRZ R1, R6, R7

(Move on register condition) moves the contents of R6 into R7 if the contents of R1 are equal to zero. This instruction requires an additional clock cycle to check if the contents

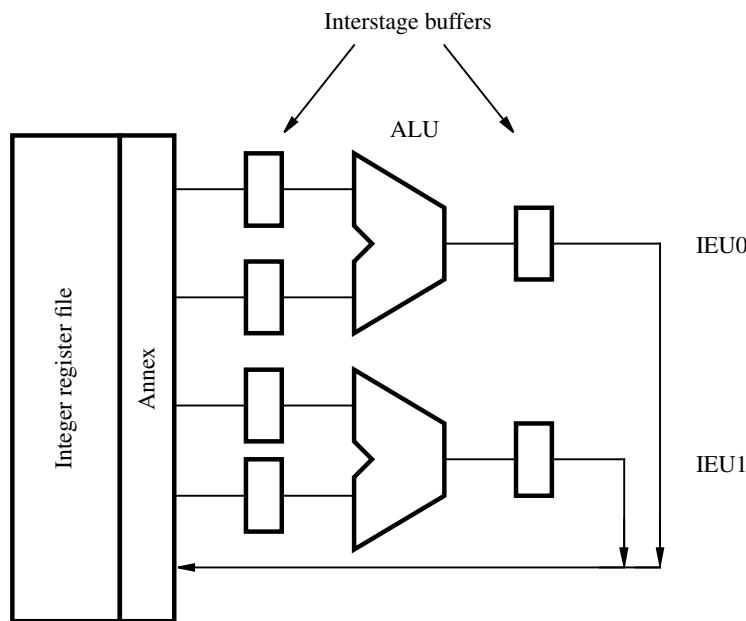
ADD	R3, R5, R6	G	E	C	N1	N2	N3	W
LDSW	R4, R7, R6	G	E	C	N1	N2	N3	W

(a) Instructions with common destination

MOVRZ	R1, R6, R7	G	E	C	N1	N2	N3	W
OR	R7, R8, R9	G	E	C	N1	N2	N3	W

(b) Delay caused by MOVR instruction

**Figure 8.26** Dispatch delays due to hazards.



**Figure 8.27** Integer execution unit.

of R1 are equal to zero. Hence, an instruction that reads register R7 cannot be in the same group or in the following group. The earliest dispatch for such an instruction is as shown in Figure 8.26b.

When the grouping logic dispatches an instruction to the integer unit, it also fetches the source operands of that instruction from the integer register file. The information needed to access the register file is available in the decoded bits that were entered into the instruction buffer by the prefetch and decode unit. Thus, by the end of the clock cycle of stage G, one or two integer instructions will be ready to enter the execution phase. The data read from the register file are stored in interstage buffers, as shown in Figure 8.27. Access to operands in the floating-point register file takes place in stage R, after the instruction has been forwarded to the floating-point unit.

### Execution Units

The Integer execution unit consists of two similar but not identical units, IEU0 and IEU1. Only unit IEU0 is equipped to handle shift instructions, while only IEU1 can generate condition codes. Instructions that do not involve these operations can be executed in either unit.

The ALU operation for most integer instructions is completed in one clock cycle. This is stage E in the pipeline. At the end of this clock cycle, the result is stored in the buffer shown at the output of the ALU in Figure 8.27. In the next clock cycle, stage C, the contents of this buffer are transferred to a part of the register file called the Annex. The Annex contains the temporary registers used in register renaming, as explained in

Section 8.6. The contents of a temporary register are transferred to the corresponding permanent register in stage W of the pipeline.

Another action that takes place during stage C is the generation of condition codes. Of course, this is done only for instructions such as ADDcc, which specify that the condition code flags are to be set. Such instructions must be executed in unit IEU1.

Consider an instruction Icc that sets the condition code flags and a subsequent conditional branch instruction, BRcc, that checks these flags. When BRcc is encountered by the prefetch and dispatch unit, the results of execution of Icc may not yet be available. The PDU predicts the outcome of the branch and continues prefetching instructions on that basis. Later, the condition codes are generated when Icc reaches stage C of the pipeline, and they are sent to the PDU during the same clock cycle. The PDU checks whether its branch prediction was correct. If it was, execution continues without interruption. Otherwise, the contents of the pipeline and the instruction buffer are flushed, and the PDU begins to fetch the correct instructions. Aborting instructions at this point is possible because these instructions will not have reached stage W of the pipeline.

When a branch is incorrectly predicted, many instructions may be incorrectly prefetched and partially executed. The situation is illustrated in Figure 8.28. We have assumed that the grouping logic has been able to dispatch four instructions in three successive clock cycles. Instruction Icc at the beginning of the first group sets the condition codes, which are tested by the following instruction, BRcc. The test is performed when the first group reaches stage C. At this time, the third group, I<sub>9</sub> to I<sub>12</sub>, is entering stage G of the pipeline. If the branch prediction was incorrect, the nine instructions I<sub>4</sub> to I<sub>12</sub> will be aborted (recall that instruction I<sub>3</sub> in the delay slot is always executed). In addition, any instructions that may have been prefetched and loaded into the instruction buffer will also be discarded. Hence, in the extreme case, up to 21 instructions may be discarded.

No operation is performed in pipeline stages N1 and N2. These stages introduce a delay of two clock cycles, to make the total length of the integer pipeline the same

I <sub>1</sub> (Icc)	G	E	C
I <sub>2</sub> (BRcc)	G	E	C
I <sub>3</sub>	G	E	C
I <sub>4</sub>	G	E	C
I <sub>5</sub>	G	E	
I <sub>6</sub>	G	E	
I <sub>7</sub>	G	E	
I <sub>8</sub>	G	E	
I <sub>9</sub>		G	
I <sub>10</sub>		G	
I <sub>11</sub>		G	
I <sub>12</sub>		G	

↑ Abort

**Figure 8.28** Worst-case timing for an incorrectly predicted branch.

as that of the floating-point pipeline. For integer instructions that do not complete their execution in stage C, such as divide instructions, execution continues through stages N1 and N2. If more time is needed, additional clock cycles are inserted between N1 and N2. The instruction enters N2 only in the last clock cycle of its execution. For example, if the operation performed by an instruction requires 16 clock cycles, 12 clock cycles are inserted after stage N1.

The Floating-point execution unit also has two independent pipelines. Register operands are fetched in stage R, and the operation is performed in up to three pipeline stages (X1 to X3). Here also, if additional clock cycles are needed, such as for the square-root instruction, additional clock cycles are inserted between X2 and X3.

In stage N3, the processor examines various exception conditions to determine whether a trap (interrupt) should be taken. Finally, the result of an instruction is stored in the destination location, either in a register or in the data cache, during the Write stage (W). An instruction may be aborted and all its effects annulled at any time up to this stage. Once the Write stage is entered, the execution of the instruction cannot be stopped.

### Load and Store Unit

The instruction

**LDUW R5, R6, R7**

loads an unsigned 32-bit word from location  $[R5] + [R6]$  in the memory into register R7. As for other integer instructions, the contents of registers R5 and R6 are fetched during stage G of the pipeline. However, instead of this data being sent to one of the integer execution units, the instruction and its operands are forwarded to the Load and Store Unit, shown in Figure 8.29. The unit begins by adding the contents of registers R5 and R6 during stage E to generate the effective address of the memory location to be accessed. The result is a virtual address value, which is sent to the data cache. At the same time, it is sent to the data lookaside buffer, dTLB, to be translated into a physical address.

Data are stored in the cache according to their virtual address, so that they can be accessed quickly without waiting for address translation to be completed. Both the data and the corresponding tag information are read from the D-cache in stage C, and the physical address is read from the dTLB. The tag used in the D-cache is a part of the physical address of the data. During stage N1, the tag read from the D-cache is checked against the physical address obtained from the dTLB. In the case of a hit, the data are loaded into an Annex register, to be transferred to the destination register in stage W. If the tags do not match, the instruction enters the Load/store queue, where it waits for a cache block to be loaded from the external cache into the D-cache.

Once an instruction enters the Load/store queue it is no longer considered to be in the execution pipeline. Other instructions may proceed to completion while a load instruction is waiting in the queue, unless one of these instructions references the register awaiting data from the memory (R7 in the example above). Thus, the Load/store queue decouples the operation of the pipeline from external data access operations so that the two can proceed independently.

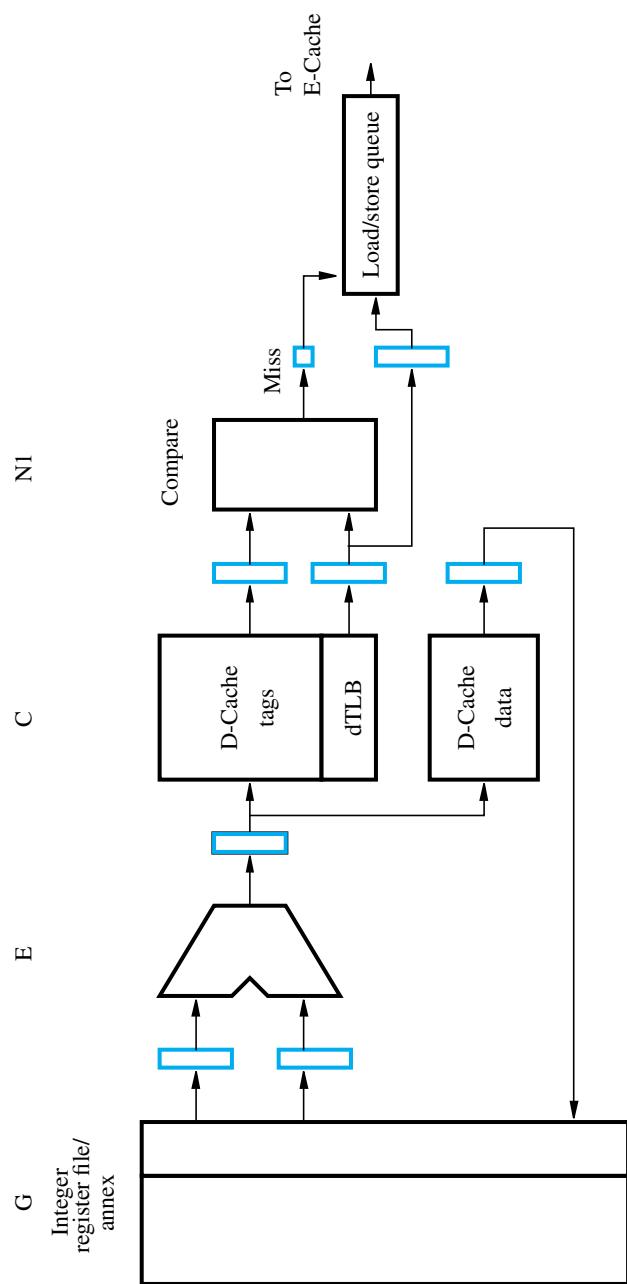


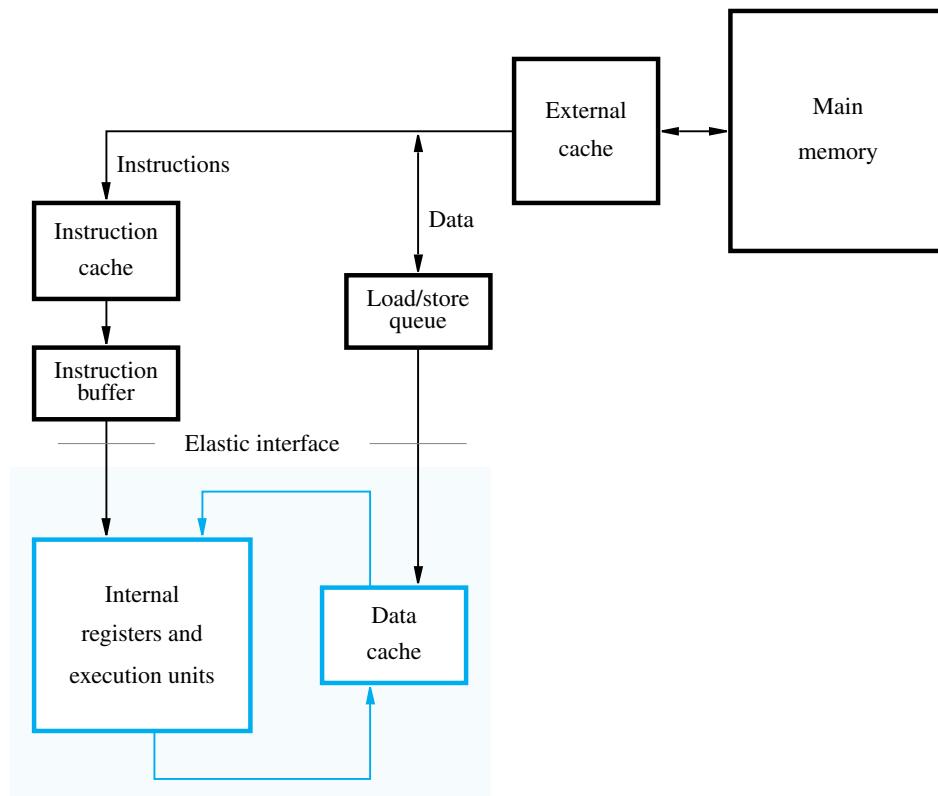
Figure 8.29 Load and store unit.

### Execution Flow

It is instructive to examine the flow of instructions and data in the UltraSPARC II processor and between it and the external cache and the memory. Figure 8.30 shows the main functional units of Figure 8.23 reorganized to illustrate the flow of instructions and data and the role that the instruction and data queues play.

Instructions are fetched from the I-cache and loaded into the instruction buffer, which can store up to 12 instructions. From there, instructions are forwarded, up to four at a time, to the block labeled “Internal registers and execution units,” where they are executed. On average, the speed with which the PDU can fill the instruction buffer is higher than the speed with which the grouping logic dispatches instructions. Hence, the instruction buffer tends to be full most of the time. In the absence of cache misses and mispredicted branches, the internal execution units are never starved for instructions. Similarly, the memory operands of load and store instructions are likely to be found in the data cache most of the time, where they are accessed in one clock cycle. Hence execution proceeds without delay.

When a miss occurs in the instruction cache, there is a delay of a few clock cycles while the appropriate block is loaded from the external cache. During that time, the



**Figure 8.30** Execution flow.

grouping logic continues to dispatch instructions from the instruction buffer until the buffer becomes empty. It takes three or four clock cycles to load a cache block (eight instructions) from the external cache, depending on the processor model. This is about the same length of time it takes the grouping logic to dispatch the instructions in a full instruction buffer. (Recall that it is not always possible to dispatch four instructions in every clock cycle.) Hence, if the instruction buffer is full at the time a cache miss occurs, operation of the execution pipeline may not be interrupted at all. If a miss also occurs in the external cache, considerably more time will be needed to access the memory. In this case, it is inevitable that the pipeline will be stalled.

A load operation that causes a cache miss enters the Load/store queue and waits for a transfer from the external cache or the memory. However, as long as the destination register of the load operation is not referenced by later instructions, internal instruction execution continues. Thus, the instruction buffer and the Load/store queue isolate the internal processor pipeline from external data transfers. They act as elastic interfaces that allow the internal high-speed pipeline to continue to run while slow external data transfers are taking place.

## 8.8 PERFORMANCE CONSIDERATIONS

We pointed out in Section 1.6 that the execution time,  $T$ , of a program that has a dynamic instruction count  $N$  is given by

$$T = \frac{N \times S}{R}$$

where  $S$  is the average number of clock cycles it takes to fetch and execute one instruction, and  $R$  is the clock rate. This simple model assumes that instructions are executed one after the other, with no overlap. A useful performance indicator is the *instruction throughput*, which is the number of instructions executed per second. For sequential execution, the throughput,  $P_s$  is given by

$$P_s = R/S$$

In this section, we examine the extent to which pipelining increases instruction throughput. However, we should reemphasize the point made in Chapter 1 regarding performance measures. The only real measure of performance is the total execution time of a program. Higher instruction throughput will not necessarily lead to higher performance if a larger number of instructions is needed to implement the desired task. For this reason, the SPEC ratings described in Chapter 1 provide a much better indicator when comparing two processors.

Figure 8.2 shows that a four-stage pipeline may increase instruction throughput by a factor of four. In general, an  $n$ -stage pipeline has the potential to increase throughput  $n$  times. Thus, it would appear that the higher the value of  $n$ , the larger the performance gain. This leads to two questions:

- How much of this potential increase in instruction throughput can be realized in practice?
- What is a good value for  $n$ ?

Any time a pipeline is stalled, the instruction throughput is reduced. Hence, the performance of a pipeline is highly influenced by factors such as branch and cache miss penalties. First, we discuss the effect of these factors on performance, and then we return to the question of how many pipeline stages should be used.

### 8.8.1 EFFECT OF INSTRUCTION HAZARDS

The effects of various hazards have been examined qualitatively in the previous sections. We now assess the impact of cache misses and branch penalties in quantitative terms.

Consider a processor that uses the four-stage pipeline of Figure 8.2. The clock rate, hence the time allocated to each step in the pipeline, is determined by the longest step. Let the delay through the ALU be the critical parameter. This is the time needed to add two integers. Thus, if the ALU delay is 2 ns, a clock of 500 MHz can be used. The on-chip instruction and data caches for this processor should also be designed to have an access time of 2 ns. Under ideal conditions, this pipelined processor will have an instruction throughput,  $P_p$ , given by

$$P_p = R = 500 \text{ MIPS} \text{ (million instructions per second)}$$

To evaluate the effect of cache misses, we use the same parameters as in Section 5.6.2. The cache miss penalty,  $M_p$ , in that system is computed to be 17 clock cycles. Let  $T_I$  be the time between two successive instruction completions. For sequential execution,  $T_I = S$ . However, in the absence of hazards, a pipelined processor completes the execution of one instruction each clock cycle, thus,  $T_I = 1$  cycle. A cache miss stalls the pipeline by an amount equal to the cache miss penalty. This means that the value of  $T_I$  increases by an amount equal to the cache miss penalty for the instruction in which the miss occurs. A cache miss can occur for either instructions or data. Consider a computer that has a shared cache for both instructions and data, and let  $d$  be the percentage of instructions that refer to data operands in the memory. The average increase in the value of  $T_I$  as a result of cache misses is given by

$$\delta_{\text{miss}} = ((1 - h_i) + d(1 - h_d)) \times M_p$$

where  $h_i$  and  $h_d$  are the hit ratios for instructions and data, respectively. Assume that 30 percent of the instructions access data in memory. With a 95-percent instruction hit rate and a 90-percent data hit rate,  $\delta_{\text{miss}}$  is given by

$$\delta_{\text{miss}} = (0.05 + 0.3 \times 0.1) \times 17 = 1.36 \text{ cycles}$$

Taking this delay into account, the processor's throughput would be

$$P_p = \frac{R}{T_I} = \frac{R}{1 + \delta_{\text{miss}}} = 0.42R$$

Note that with  $R$  expressed in MHz, the throughput is obtained directly in millions of instructions per second. For  $R = 500$  MHz,  $P_p = 210$  MIPS.

Let us compare this value to the throughput obtainable without pipelining. A processor that uses sequential execution requires four cycles per instruction. Its throughput

would be

$$P_s = \frac{R}{4 + \delta_{miss}} = 0.19R$$

For  $R = 500$  MHz,  $P_s = 95$  MIPS. Clearly, pipelining leads to significantly higher throughput. But the performance gain of  $0.42/0.19 = 2.2$  is only slightly better than one-half the ideal case.

Reducing the cache miss penalty is particularly worthwhile in a pipelined processor. As Chapter 5 explains, this can be achieved by introducing a secondary cache between the primary, on-chip cache and the memory. Assume that the time needed to transfer an 8-word block from the secondary cache is 10 ns. Hence, a miss in the primary cache for which the required block is found in the secondary cache introduces a penalty,  $M_s$ , of 5 cycles. In the case of a miss in the secondary cache, the full 17-cycle penalty ( $M_p$ ) is still incurred. Hence, assuming a hit rate  $h_s$  of 94 percent in the secondary cache, the average increase in  $T_I$  is

$$\delta_{miss} = ((1 - h_i) + d(1 - h_d)) \times (h_s \times M_s + (1 - h_s) \times M_p) = 0.46 \text{ cycle}$$

The instruction throughput in this case is  $0.68R$ , or 340 MIPS. An equivalent non-pipelined processor would have a throughput of  $0.22R$ , or 110 MIPS. Thus, pipelining provides a performance gain of  $0.68/0.22 = 3.1$ .

The values of 1.36 and 0.46 are, in fact, somewhat pessimistic, because we have assumed that every time a data miss occurs, the entire miss penalty is incurred. This is the case only if the instruction immediately following the instruction that references memory is delayed while the processor waits for the memory access to be completed. However, an optimizing compiler attempts to increase the distance between two instructions that create a dependency by placing other instructions between them whenever possible. Also, in a processor that uses an instruction queue, the cache miss penalty during instruction fetches may have a much reduced effect as the processor is able to dispatch instructions from the queue.

### 8.8.2 NUMBER OF PIPELINE STAGES

The fact that an  $n$ -stage pipeline may increase instruction throughput by a factor of  $n$  suggests that we should use a large number of stages. However, as the number of pipeline stages increases, so does the probability of the pipeline being stalled, because more instructions are being executed concurrently. Thus, dependencies between instructions that are far apart may still cause the pipeline to stall. Also, branch penalties may become more significant, as Figure 8.9 shows. For these reasons, the gain from increasing the value of  $n$  begins to diminish, and the associated cost is not justified.

Another important factor is the inherent delay in the basic operations performed by the processor. The most important among these is the ALU delay. In many processors, the cycle time of the processor clock is chosen such that one ALU operation can be completed in one cycle. Other operations are divided into steps that take about the same time as an add operation. It is also possible to use a pipelined ALU. For example, the ALU of the Compaq Alpha 21064 processor consists of a two-stage pipeline, in which each stage completes its operation in 5 ns.

Many pipelined processors use four to six stages. Others divide instruction execution into smaller steps and use more pipeline stages and a faster clock. For example, the UltraSPARC II uses a 9-stage pipeline and Intel's Pentium Pro uses a 12-stage pipeline. The latest Intel processor, Pentium 4, has a 20-stage pipeline and uses a clock speed in the range 1.3 to 1.5 GHz. For fast operations, there are two pipeline stages in one clock cycle.

## 8.9 CONCLUDING REMARKS

Two important features have been introduced in this chapter, pipelining and multiple issue. Pipelining enables us to build processors with instruction throughput approaching one instruction per clock cycle. Multiple issue makes possible superscalar operation, with instruction throughput of several instructions per clock cycle.

The potential gain in performance can only be realized by careful attention to three aspects:

- The instruction set of the processor
- The design of the pipeline hardware
- The design of the associated compiler

It is important to appreciate that there are strong interactions among all three. High performance is critically dependent on the extent to which these interactions are taken into account in the design of a processor. Instruction sets that are particularly well-suited for pipelined execution are key features of modern processors.

## PROBLEMS

- 8.1** Consider the following sequence of instructions

```

Add #20,R0,R1
Mul #3,R2,R3
And #$3A,R2,R4
Add R0,R2,R5

```

In all instructions, the destination operand is given last. Initially, registers R0 and R2 contain 2000 and 50, respectively. These instructions are executed in a computer that has a four-stage pipeline similar to that shown in Figure 8.2. Assume that the first instruction is fetched in clock cycle 1, and that instruction fetch requires only one clock cycle.

- Draw a diagram similar to Figure 8.2a. Describe the operation being performed by each pipeline stage during each of clock cycles 1 through 4.
- Give the contents of the interstage buffers, B1, B2, and B3, during clock cycles 2 to 5.

- 8.2** Repeat Problem 8.1 for the following program:

```

Add #20,R0,R1
Mul #3,R2,R3
And #$3A,R1,R4
Add R0,R2,R5

```

- 8.3** Instruction  $I_2$  in Figure 8.6 is delayed because it depends on the results of  $I_1$ . By occupying the Decode stage, instruction  $I_2$  blocks  $I_3$ , which, in turn, blocks  $I_4$ . Assuming that  $I_3$  and  $I_4$  do not depend on either  $I_1$  or  $I_2$  and that the register file allows two Write steps to proceed in parallel, how would you use additional storage buffers to make it possible for  $I_3$  and  $I_4$  to proceed earlier than in Figure 8.6? Redraw the figure, showing the new order of steps.
- 8.4** The delay bubble in Figure 8.6 arises because instruction  $I_2$  is delayed in the Decode stage. As a result, instructions  $I_3$  and  $I_4$  are delayed even if they do not depend on either  $I_1$  or  $I_2$ . Assume that the Decode stage allows two Decode steps to proceed in parallel. Show that the delay bubble can be completely eliminated if the register file also allows two Write steps to proceed in parallel.
- 8.5** Figure 8.4 shows an instruction being delayed as a result of a cache miss. Redraw this figure for the hardware organization of Figure 8.10. Assume that the instruction queue can hold up to four instructions and that the instruction fetch unit reads two instructions at a time from the cache.
- 8.6** A program loop ends with a conditional branch to the beginning of the loop. How would you implement this loop on a pipelined computer that uses delayed branching with one delay slot? Under what conditions would you be able to put a useful instruction in the delay slot?
- 8.7** The branch instruction of the UltraSPARC II processor has an Annul bit. When set by the compiler, the instruction in the delay slot is discarded if the branch is not taken. An alternative choice is to have the instruction discarded if the branch is taken. When is each of these choices advantageous?
- 8.8** A computer has one delay slot. The instruction in this slot is always executed, but only on a speculative basis. If a branch does not take place, the results of that instruction are discarded. Suggest a way to implement program loops efficiently on this computer.
- 8.9** Rewrite the sort routine shown in Figure 2.34 for the SPARC processor. Recall that the SPARC architecture has one delay slot with an associated Annul bit and uses branch prediction. Attempt to fill the delay slots with useful instructions wherever possible.
- 8.10** Consider a statement of the form

IF  $A > B$  THEN action 1 ELSE action 2

Write a sequence of assembly language instructions, first using branch instructions only, then using conditional instructions such as those available on the ARM processor.

Assume a simple two-stage pipeline, and draw a diagram similar to that in Figure 8.8 to compare execution times for the two approaches.

- 8.11** The feed-forward path in Figure 8.7 (blue lines) allows the content of the RSLT register to be used directly in an ALU operation. The result of that operation is stored back in the RSLT register, replacing its previous contents. What type of register is needed to make such an operation possible?

Consider the two instructions

$I_1:$	Add	R1,R2,R3
$I_2:$	Shift_left	R3

Assume that before instruction  $I_1$  is executed, R1, R2, R3, and RSLT contain the values 30, 100, 45, and 198, respectively. Draw a timing diagram for a 4-stage pipeline, showing the clock signal and the contents of the RSLT register during each cycle. Use your diagram to show that correct results will be obtained during the forwarding operation.

- 8.12** Write the program in Figure 2.37 for a processor in which only load and store instructions access memory. Identify all dependencies in the program and show how you would optimize it for execution on a pipelined processor.
- 8.13** Assume that 20 percent of the dynamic count of the instructions executed on a computer are branch instructions. Delayed branching is used, with one delay slot. Estimate the gain in performance if the compiler is able to use 85 percent of the delay slots.
- 8.14** A pipelined processor has two branch delay slots. An optimizing compiler can fill one of these slots 85 percent of the time and can fill the second slot only 20 percent of the time. What is the percentage improvement in performance achieved by this optimization, assuming that 20 percent of the instructions executed are branch instructions?
- 8.15** A pipelined processor uses the delayed branch technique. You are asked to recommend one of two possibilities for the design of this processor. In the first possibility, the processor has a 4-stage pipeline and one delay slot, and in the second possibility, it has a 6-stage pipeline with two delay slots. Compare the performance of these two alternatives, taking only the branch penalty into account. Assume that 20 percent of the instructions are branch instructions and that an optimizing compiler has an 80 percent success rate in filling the single delay slot. For the second alternative, the compiler is able to fill the second slot 25 percent of the time.
- 8.16** Consider a processor that uses the branch prediction mechanism represented in Figure 8.15b. The initial state is either LT or LNT, depending on information provided in the branch instruction. Discuss how the compiler should handle the branch instructions used to control “do while” and “do until” loops, and discuss the suitability of the branch prediction mechanism in each case.
- 8.17** Assume that the instruction queue in Figure 8.10 can hold up to six instructions. Redraw Figure 8.11 assuming that the queue is full in clock cycle 1 and that the fetch unit can read up to two instructions at a time from the cache. When will the queue become full again after instruction  $I_k$  is fetched?

- 8.18** Redraw Figure 8.11 for the case of the mispredicted branch in Figure 8.14.
- 8.19** Figure 8.16 shows that one instruction that uses a complex addressing mode takes the same time to execute as an equivalent sequence of instructions that use simpler addressing modes. Yet, the use of simple addressing modes is one of the tenets of the RISC philosophy. How would you design a pipeline to handle complex addressing modes? Discuss the pros and cons of this approach.

## REFERENCE

1. The SPARC Architecture Manual, Version 9, D. Weaver and T. Germond, ed., PTR Prentice Hall, Englewood Cliffs, New Jersey, 1994.



# Principles of Parallel Computing

---

- Finding enough parallelism (Amdahl's Law)
- Granularity
- Locality
- Load balance
- Coordination and synchronization
- Performance modeling



All of these things makes parallel programming even harder than sequential programming.  
696

# What Is Parallelization ?

**"Something" is parallel if there is a certain level of independence in the order of operations**

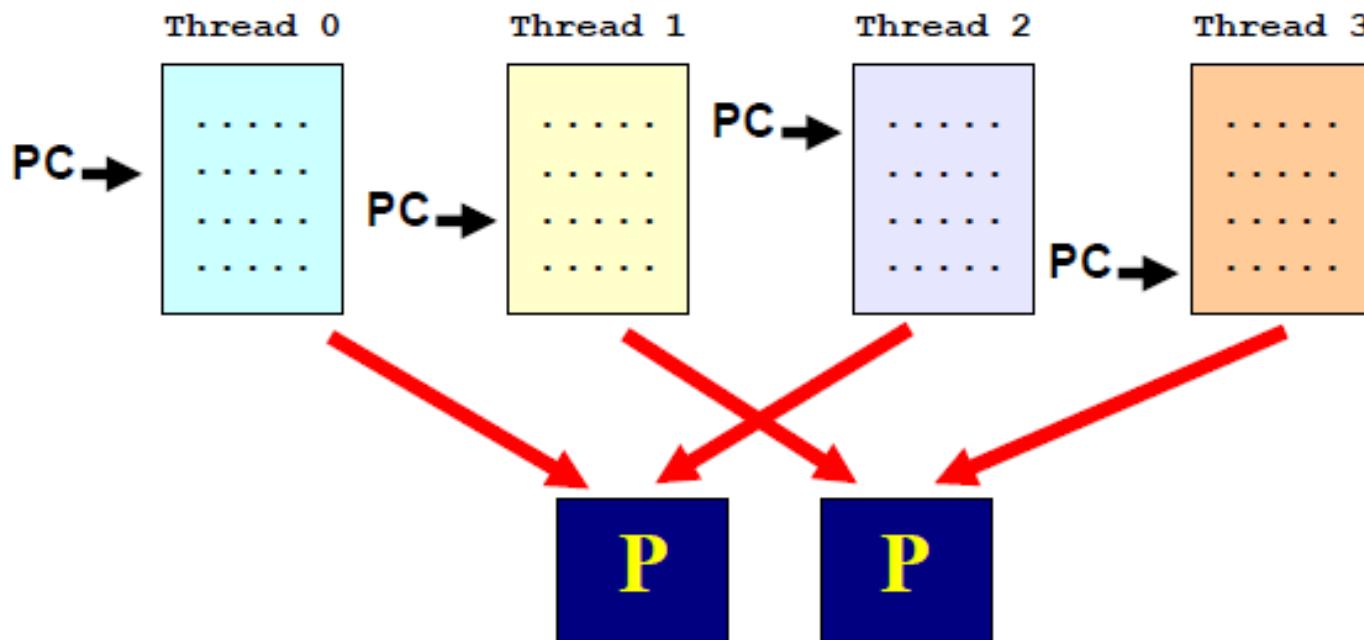
**In other words, it doesn't matter in what order those operations are performed**

- ◆ A sequence of machine instructions
- ◆ A collection of program statements
- ◆ An algorithm
- ◆ The problem you're trying to solve



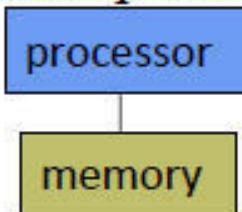
# What Is a Thread ?

- ◆ *Loosely said, a thread consists of a series of instructions with its own program counter ("PC") and state*
- ◆ *A parallel program executes threads in parallel*
- ◆ *These threads are then scheduled onto processors*

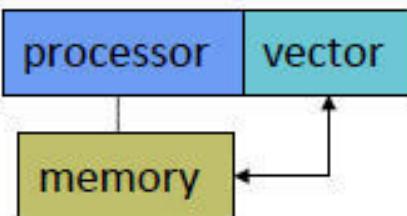


# *Parallel Architecture Types*

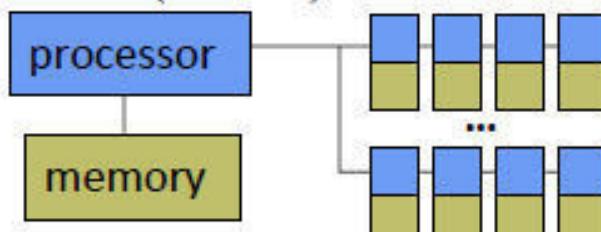
- Uniprocessor
  - Scalar processor



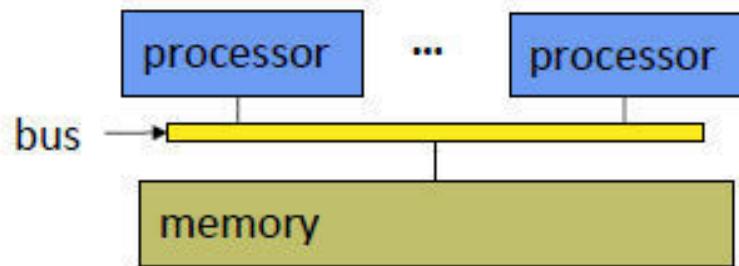
- Vector processor



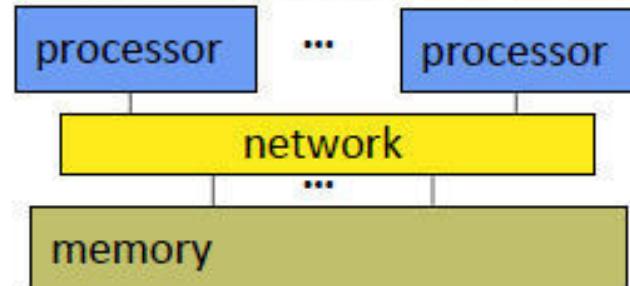
- Single Instruction Multiple Data (SIMD)



- Shared Memory Multiprocessor (SMP)
  - Shared memory address space
  - Bus-based memory system

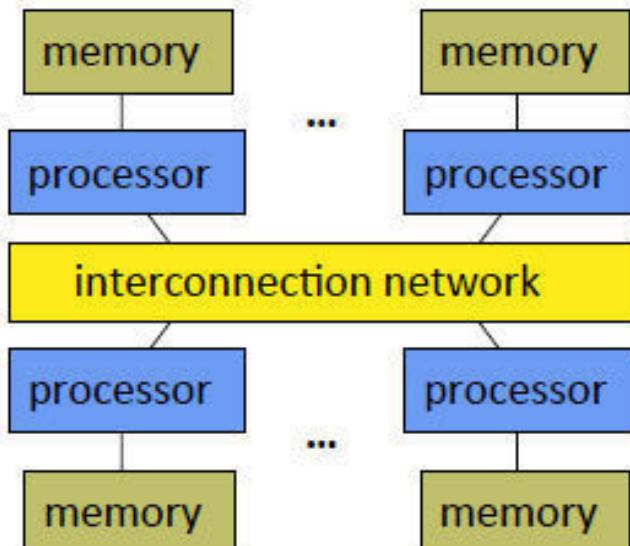


- Interconnection network



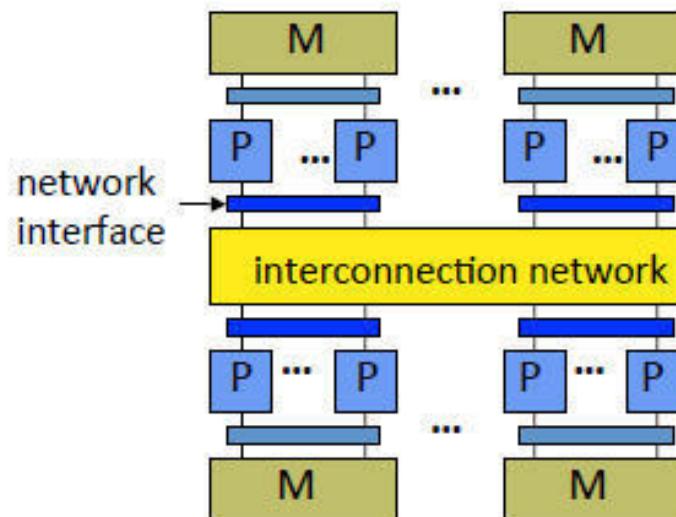
# Parallel Architecture Types (2)

- Distributed Memory Multiprocessor
  - Message passing between nodes



- Massively Parallel Processor (MPP)
  - Many, many processors

- Cluster of SMPs
  - Shared memory addressing within SMP node
  - Message passing between SMP nodes

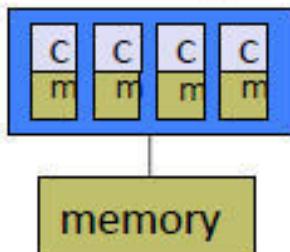


- Can also be regarded as MPP if processor number is large

# Parallel Architecture Types (3)

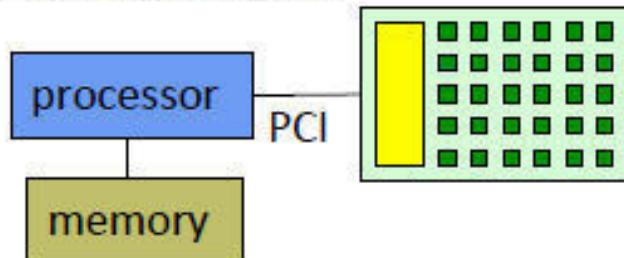
## □ Multicore

- Multicore processor

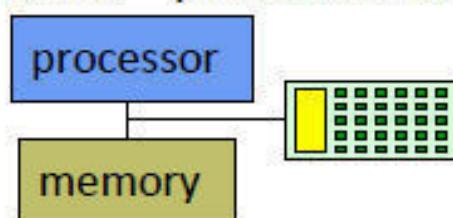


cores can be hardware multithreaded (hyperthread)

- GPU accelerator

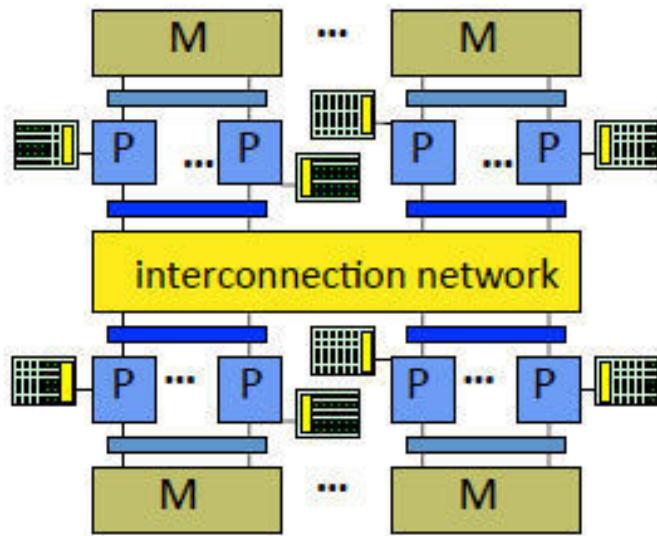


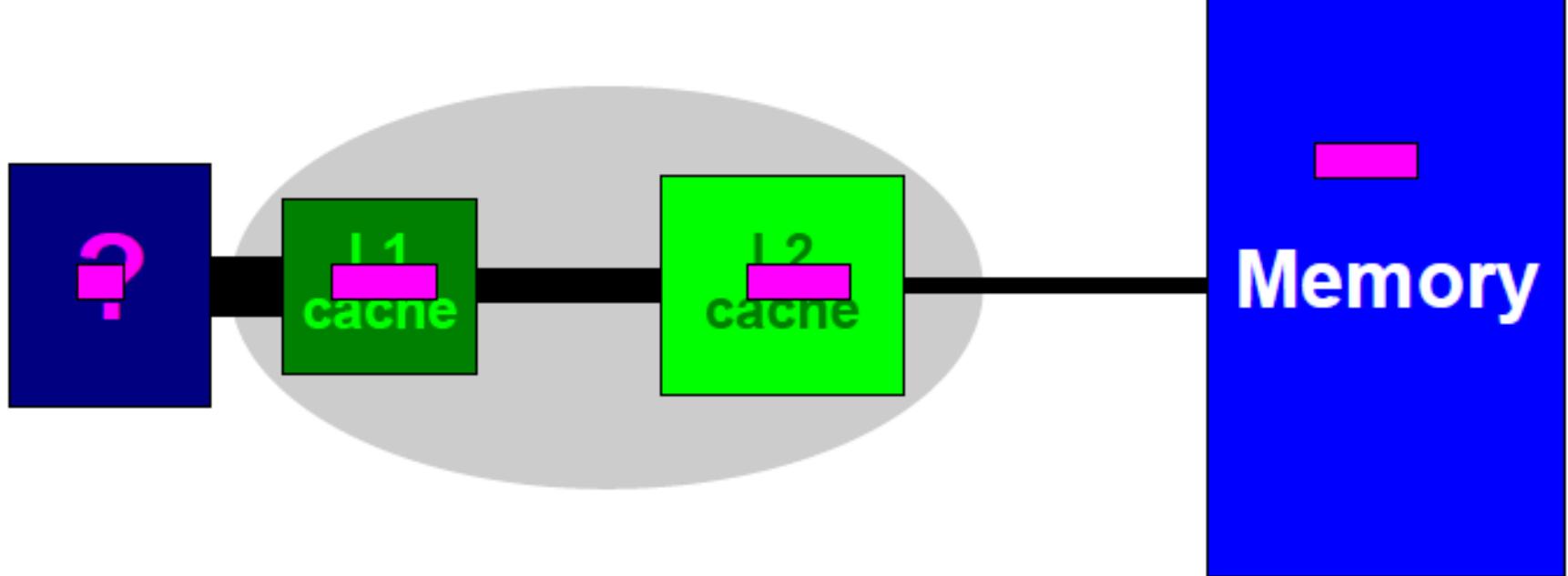
- “Fused” processor accelerator



- Multicore SMP+GPU Cluster

- Shared memory addressing within SMP node
- Message passing between SMP nodes
- GPU accelerators attached



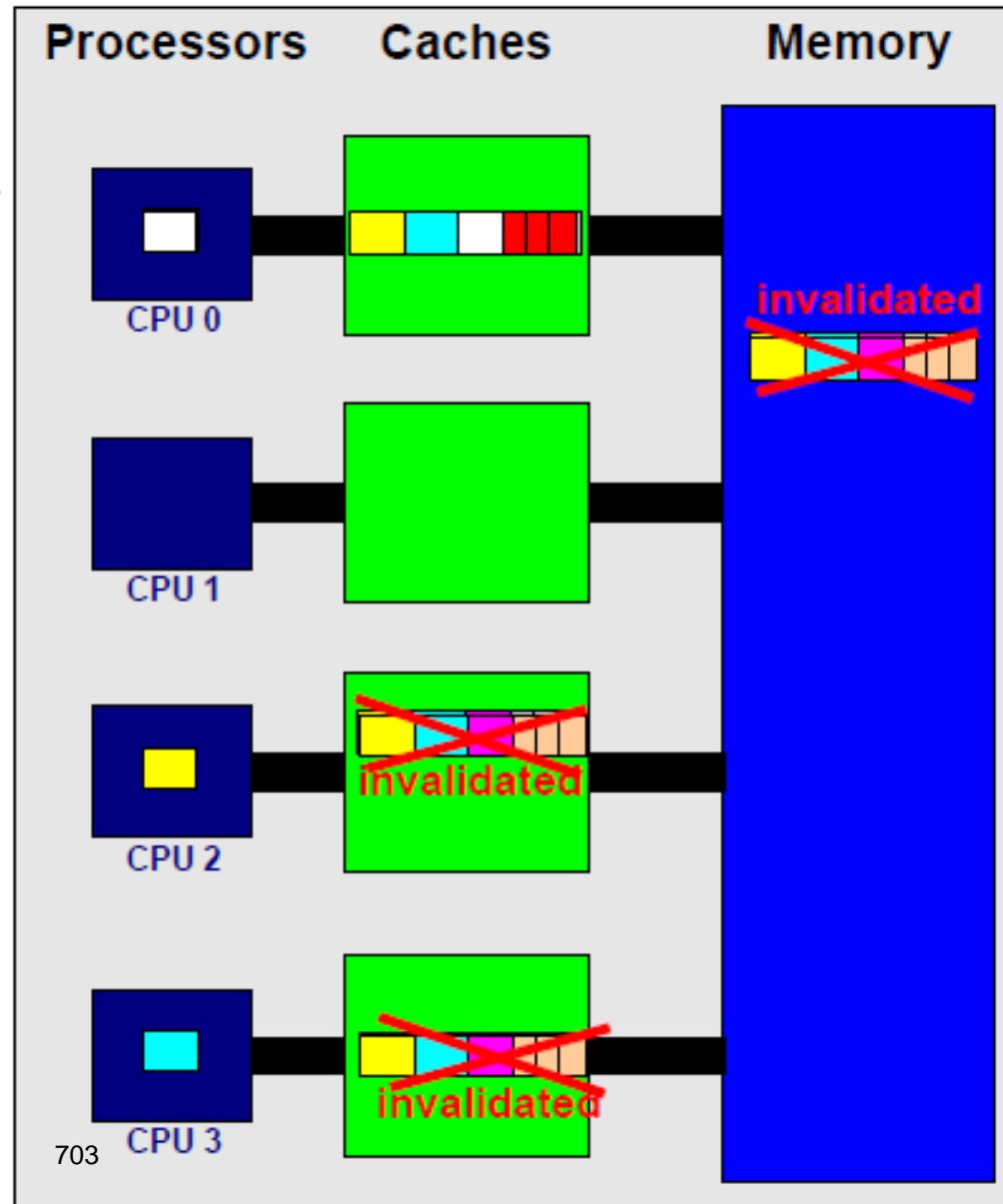


# Caches in a parallel computer

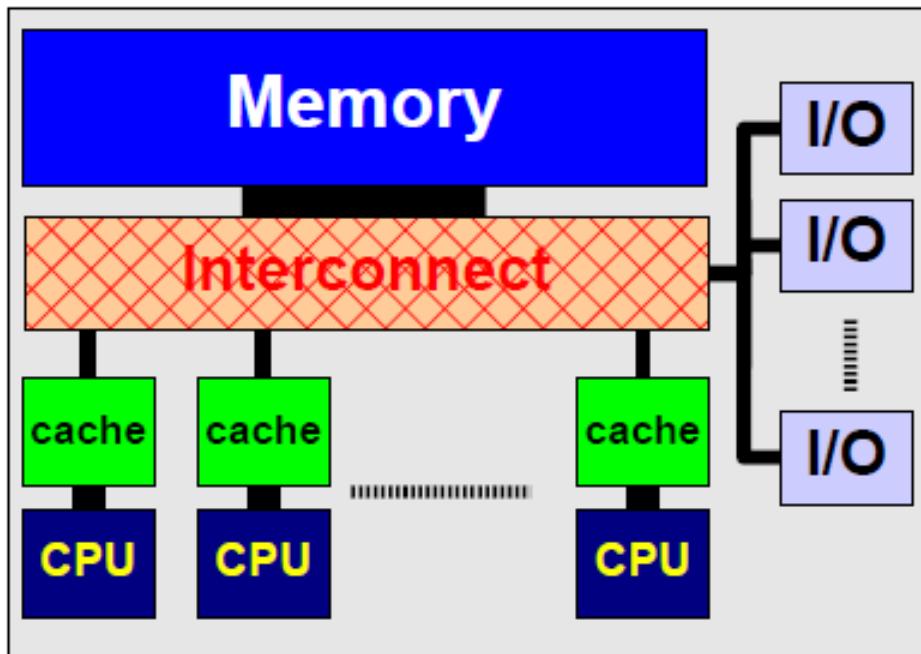
- A cache line starts in memory
- Over time multiple copies of this line may exist

## Cache Coherence ("cc"):

- ✓ Tracks changes in copies
- ✓ Makes sure correct cache line is used
- ✓ Different implementations possible
- ✓ Need hardware support to make it efficient



# Uniform Memory Access (UMA)



## Pro

- ✓ *Easy to use and to administer*
- ✓ *Efficient use of resources*

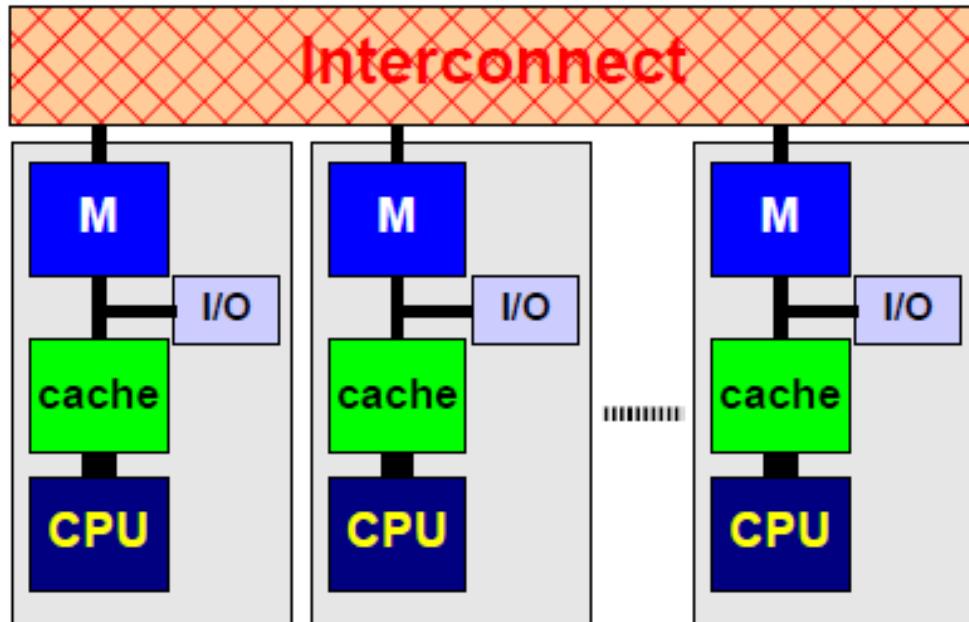
## Con

- ✓ *Said to be expensive*
- ✓ *Said to be non-scalable*

- *Also called "SMP" (Symmetric Multi Processor)*
- *Memory Access time is Uniform for all CPUs*
- *CPU can be multicore*
- *Interconnect is "cc":*
  - *Bus*
  - *Crossbar*
- *No fragmentation - Memory and I/O are shared resources*

- **Uniform memory access (UMA)** is a shared memory architecture used in parallel computers.
- All the processors in the UMA model share the physical memory uniformly.
- In an UMA architecture, access time to a memory location is independent of which processor makes the request or which memory chip contains the transferred data.
- Uniform memory access computer architectures are often contrasted with non uniform (NUMA) architectures.
- In the UMA architecture, each processor may use a private cache. Peripherals are also shared in some fashion.
- The UMA model is suitable for general purpose and time sharing applications by multiple users.
- It can be used to speed up the execution of a single large program in time critical applications.

# NUMA



- Also called "Distributed Memory" or NORMA (No Remote Memory Access)
- Memory Access time is Non-Uniform
- Hence the name "NUMA"
- Interconnect is not "cc":
  - Ethernet, Infiniband, etc, .....
- Runs 'N' copies of the OS
- Memory and I/O are distributed resources

## Pro

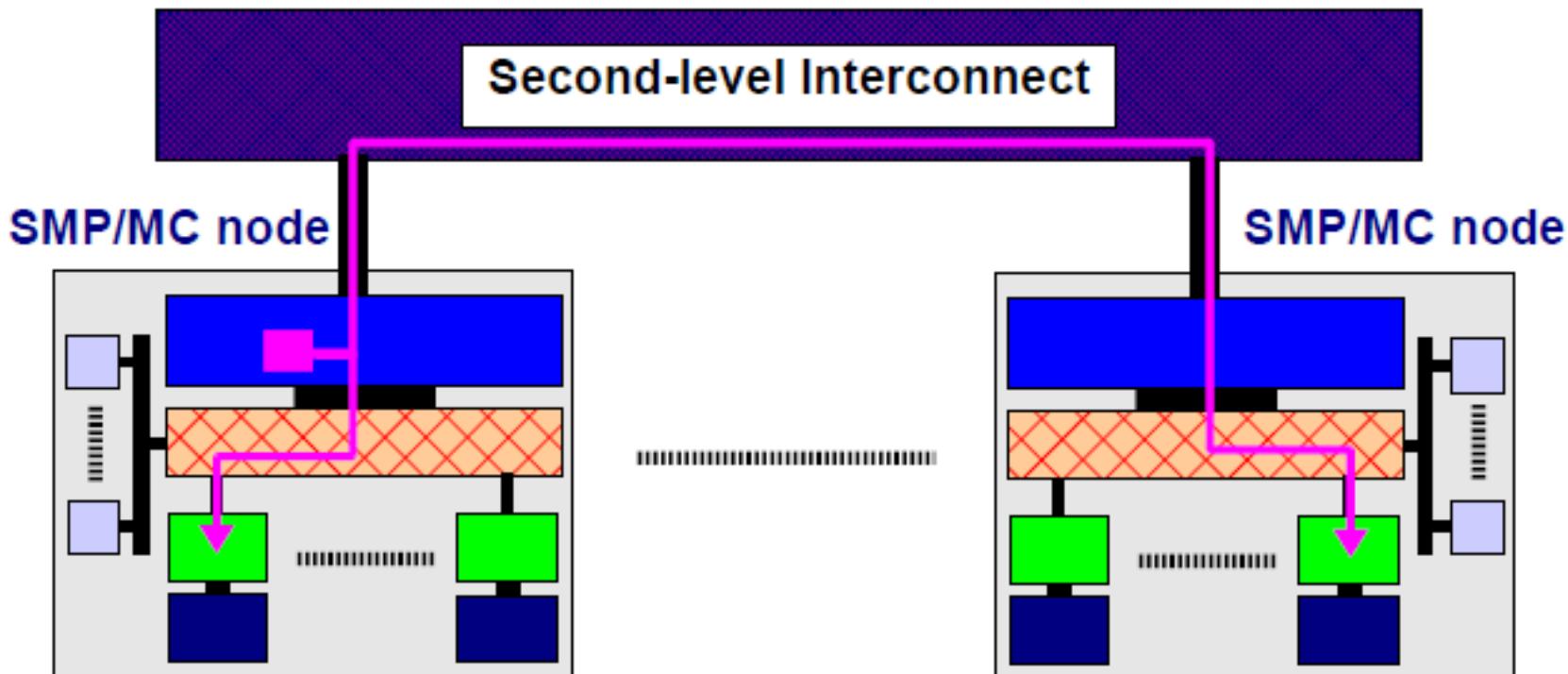
- ✓ Said to be cheap
- ✓ Said to be scalable

## Con

- ✓ Difficult to use and administer
- ✓ Inefficient use of resources

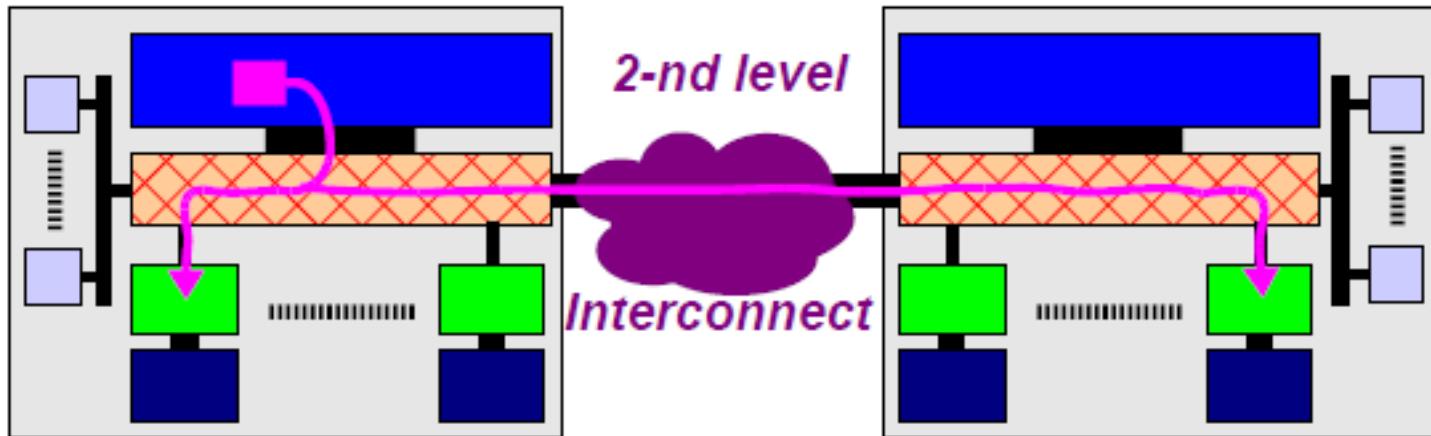
- **Non-uniform memory access (NUMA)** is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor.
- Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors).
- The benefits of NUMA are limited to particular workloads, notably on servers where the data is often associated strongly with certain tasks or users.

# The Hybrid Architecture



- ***Second-level interconnect is not cache coherent***
  - *Ethernet, Inf niband, etc, ....*
- ***Hybrid Architecture with all Pros and Cons:***
  - *UMA within one SMP/Multicore node*
  - *NUMA across nodes*

# cc-NUMA



- ❑ **Two-level interconnect:**
  - *UMA/SMP within one system*
  - *NUMA between the systems*
- ❑ **Both interconnects support cache coherence i.e. the system is fully cache coherent**
- ❑ **Has all the advantages ('look and feel') of an SMP**
- ❑ **Downside is the Non-Uniform Memory Access time**

# “Automatic” Parallelism in Modern Machines

- Bit level parallelism
  - within floating point operations, etc.
- Instruction level parallelism (ILP)
  - multiple instructions execute per clock cycle
- Memory system parallelism
  - overlap of memory operations with computation
- OS parallelism
  - multiple jobs run in parallel on commodity SMPs

Limits to all of these -- for very high performance, need user to identify, schedule and coordinate parallel tasks

## Amdahl's Law

---

**Amdahl's Law places a strict limit on the speedup that can be realized by using multiple processors. Two equivalent expressions for Amdahl's Law are given below:**

$$t_N = \left( f_p/N + f_s \right) t_1 \quad \text{Effect of multiple processors on run time}$$

$$S = 1/(f_s + f_p/N) \quad \text{Effect of multiple processors on speedup}$$

Where:

$f_s$  = serial fraction of code

$f_p$  = parallel fraction of code =  $1 - f_s$

$N$  = number of processors

# Overhead of Parallelism

---

- Given enough parallel work, this is the biggest barrier to getting desired speedup
- Parallelism overheads include:
  - cost of starting a thread or process
  - cost of communicating shared data
  - cost of synchronizing
  - extra (redundant) computation
- Each of these can be in the range of milliseconds (=millions of flops) on some systems
- Tradeoff: Algorithm needs sufficiently large units of work to run fast in parallel (I.e. large granularity), but not so large that there is not enough parallel work

# Load Imbalance

---

- Load imbalance is the time that some processors in the system are idle due to
  - insufficient parallelism (during that phase)
  - unequal size tasks
- Examples of the latter
  - adapting to “interesting parts of a domain”
  - tree-structured computations
  - fundamentally unstructured problems
- Algorithm needs to balance load

# Parallelism Is Everywhere

*Multiple levels of parallelism:*

<b>Granularity</b>	<b>Technology</b>	<b>Programming Model</b>
<b>Instruction Level</b>	<b>Superscalar</b>	<b>Compiler</b>
<b>Chip Level</b>	<b>Multicore</b>	<b>Compiler, OpenMP, MPI</b>
<b>System Level</b>	<b>SMP/cc-NUMA</b>	<b>Compiler, OpenMP, MPI</b>
<b>Grid Level</b>	<b>Cluster</b>	<b>MPI</b>

# Loosely Coupled Multiprocessor System

- It is a type of multiprocessing system in which, There is distributed memory instead of shared memory.
- In loosely coupled multiprocessor system, data rate is low rather than tightly coupled multiprocessor system.
- In loosely coupled multiprocessor system, modules are connected through MTS (Message transfer system) network.
- Data dependency will be less, so that maximum parallelism can be achieved.

# Tightly Coupled Multiprocessor System:

- It is a type of multiprocessing system in which, There is shared memory.
- In tightly coupled multiprocessor system, data rate is high rather than loosely coupled multiprocessor system.
- Data dependency will be high, so that parallelism will be minimized.

S.NO	LOOSELY COUPLED	TIGHTLY COUPLED
1.	There is distributed memory in loosely coupled multiprocessor system.	There is shared memory, in tightly coupled multiprocessor system.
2.	Loosely Coupled Multiprocessor System has low data rate.	Tightly coupled multiprocessor system has high data rate.
3.	The cost of loosely coupled multiprocessor system is less.	Tightly coupled multiprocessor system is more costly.
4.	In loosely coupled multiprocessor system, modules are connected through <b>Message transfer system</b> network.	While there is PMIN, IOPIN and ISIN networks.
5.	In loosely coupled multiprocessor, Memory conflicts don't take place.	While tightly coupled multiprocessor system have memory conflicts.
6.	Efficient when tasks running on different processors, has minimal interaction.	Efficient for high-speed or real-time processing.