



18CSC203J – COMPUTER ORGANIZATION AND ARCHITECTURE

Course Outcome

CLR-1:*Utilize the functional units of a computer*

CLO-1 :*Identify the computer hardware and how software interacts with computer hardware*



Topics Covered

- Functional Units of a computer
- Operational Concepts
- Bus Structures
- Memory Location and Addresses
- Memory Operations
- Instructions and Instruction Sequencing
- Addressing modes
- Problem Solving
- Introduction to Microprocessor
- Introduction to Assembly Language
- Writing of Assembly Language Programming
- ARM Processor: The Thumb instruction set
- Processor and CPU CORES
- Instruction Encoding Format
- Memory load and store instruction in ARM
- Basics of IO Operations



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Functional Units of a Computer



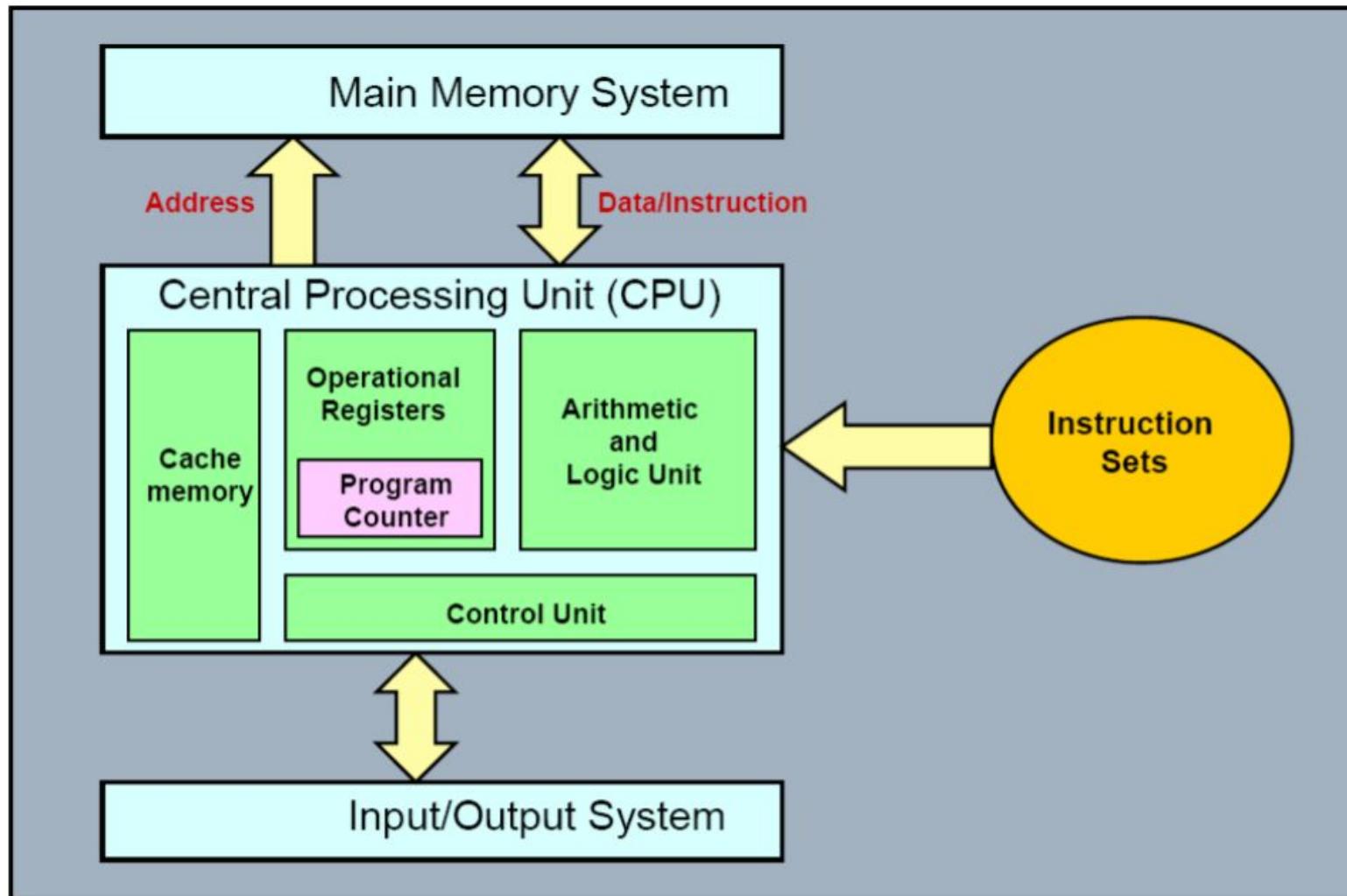
FUNCTIONAL UNITS OF COMPUTER

- **Input Unit**
- **Output Unit**
- **Central Processing Unit (ALU and Control Units)**
- **Memory**
- **Bus Structure**



SRM

◆ Basic Functional Unit of a Computer

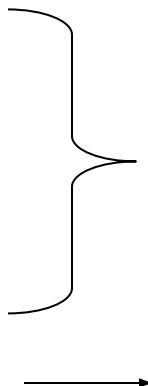




Functions

- **ALL computer functions are:**

- Data **PROCESSING**
- Data **STORAGE**
- Data **MOVEMENT**
- **CONTROL**



Data = Information

Coordinates How
Information is Used



Functions of a computer

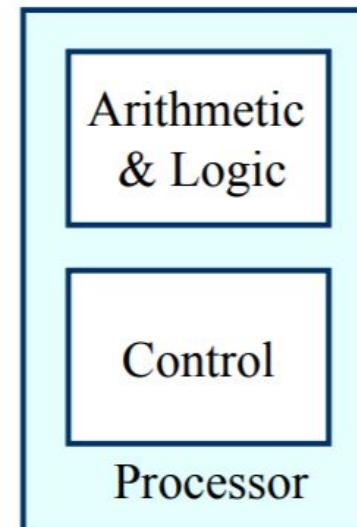
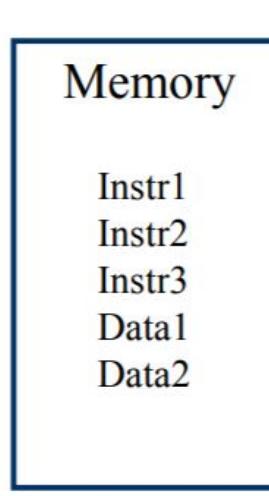
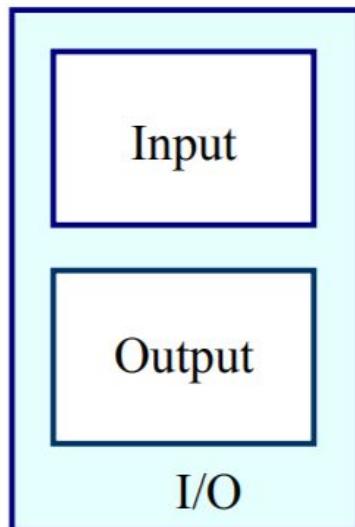
The operations performed by a computer using the functional units can be summarized as follows:

- It accepts information (program and data) through input unit and transfers it to the memory.
- Information stored in the memory is fetched, under program control, into an arithmetic and logic unit for processing.
- Processed information leaves the computer through an output unit.
- The control unit controls all activities taking place inside a computer.



Input unit accepts information:

- Human operators,
- Electromechanical devices (keyboard)
- Other computers



Output unit sends results of processing:

- To a monitor display,
- To a printer

Stores information:

- Instructions,
- Data

Control unit coordinates various actions

- Input,
- Output
- Processing

INPUT UNIT:

- Converts the external world data to a binary format, which can be understood by CPU

Eg: Keyboard, Mouse, Joystick etc

OUTPUT UNIT:

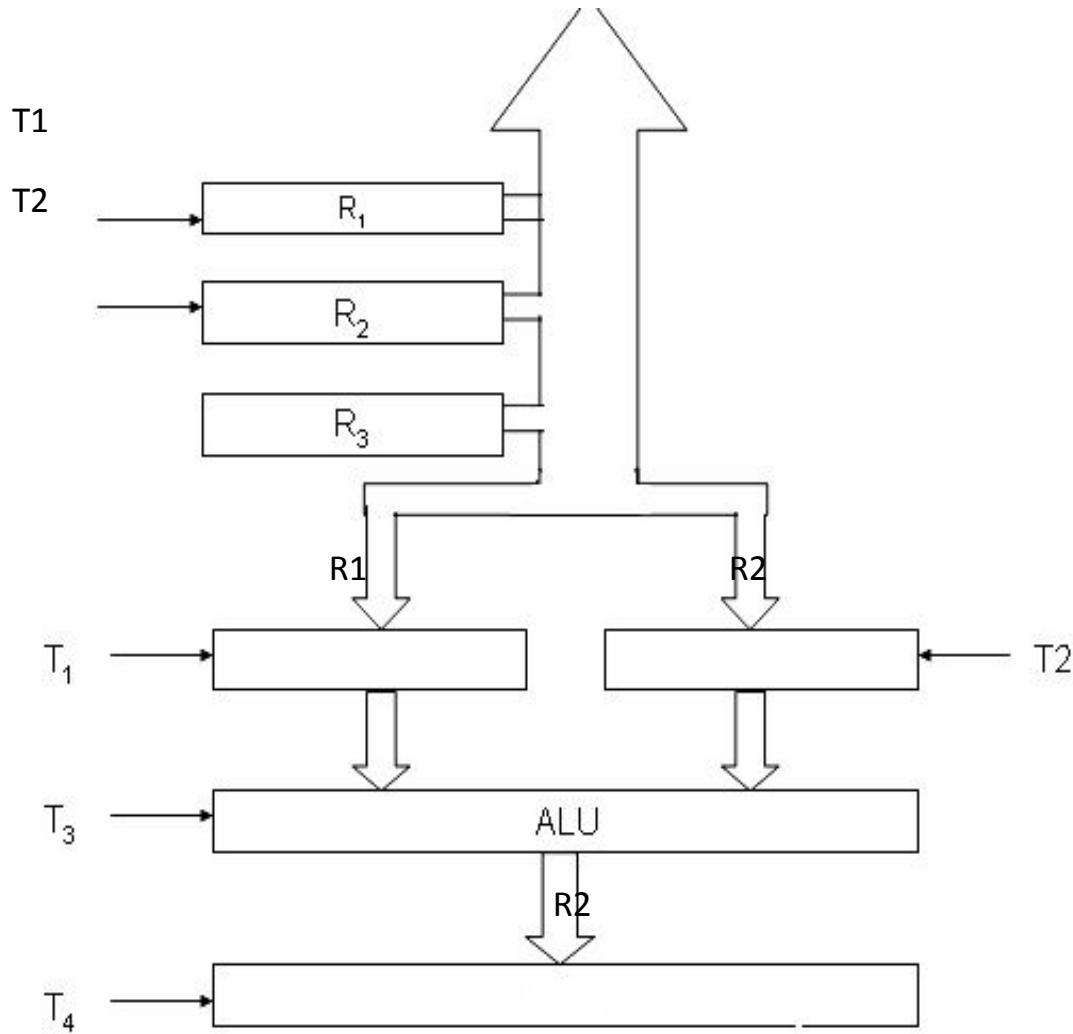
- Converts the binary format data to a format that a common man can understand

Eg: Monitor, Printer, LCD, LED etc

CPU (Central Processing Unit)

- The “brain” of the machine
- Responsible for carrying out computational task
- Contains ALU, CU, Registers
- ALU Performs Arithmetic and logical operations
- CU Provides control signals in accordance with some timings which in turn controls the execution process
- Register Stores data and result and speeds up the operation

CONTROL UNIT



- Control unit works with a reference signal called Processor clock
- Processor divides the operations into basic steps
- Each basic step is executed in one clock cycle

Example

Add R1, R2

T1 \longrightarrow **Enable R1**

T2 \longrightarrow **Enable R2**

▶

T3 \longrightarrow **Enable ALU for addition operation**

T4 \longrightarrow **Enable out put of ALU to store result of the operation**

MEMORY UNIT

- Stores data, results, programs
- Two class of storage (i) Primary (ii) Secondary
- Two types are RAM or R/W memory and ROM read only memory
- ROM is used to store data and program which is not going to change.
- Secondary storage is used for bulk storage or mass storage



Basic Operational Concepts

Basic Function of Computer

- To Execute a given task as per the appropriate program
- Program consists of list of instructions stored in memory

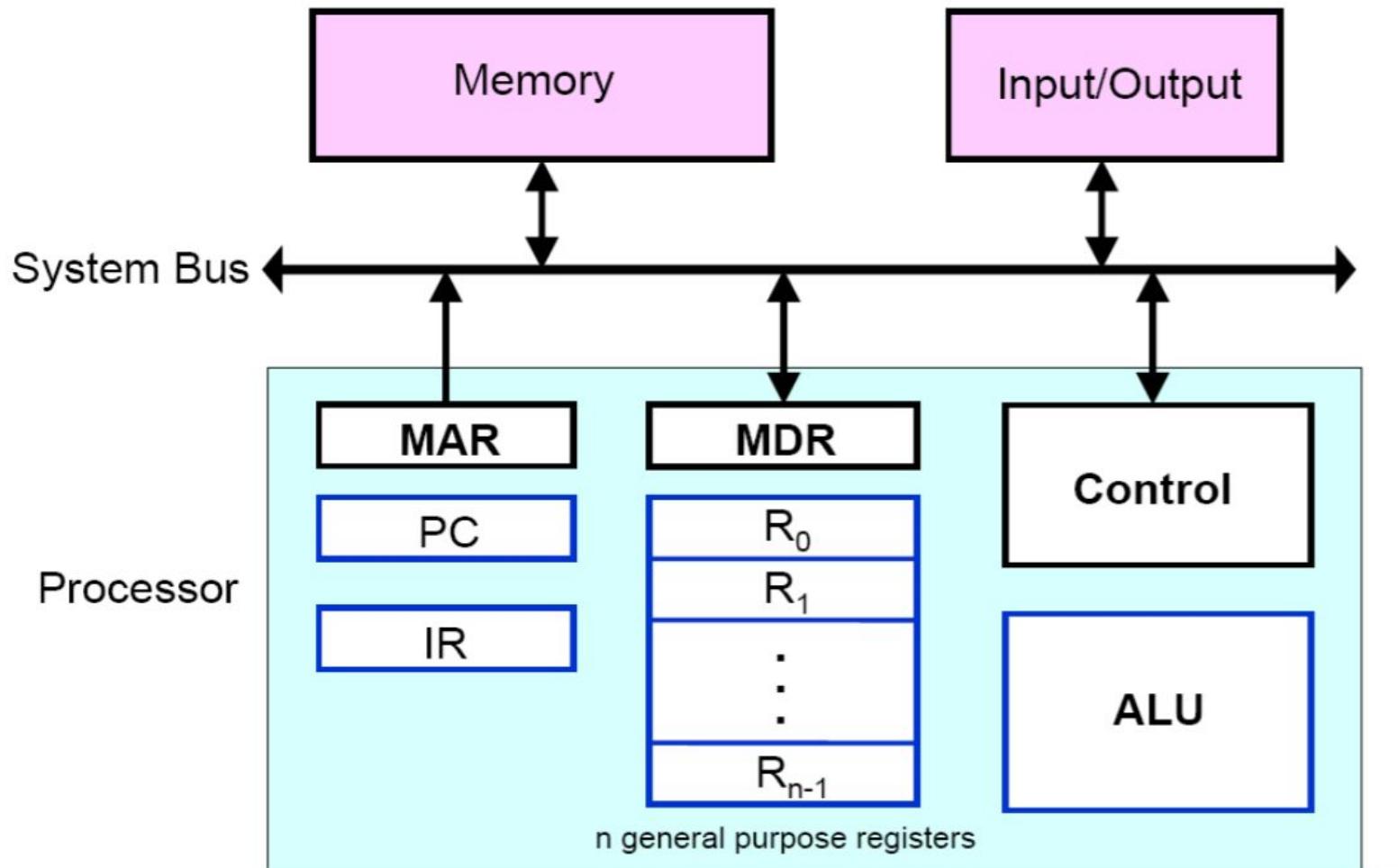
Review

- Activity in a computer is governed by instructions.
- To perform a task, an appropriate program consisting of a list of instructions is stored in the memory.
- Individual instructions are brought from the memory into the processor, which executes the specified operations.
- Data to be used as operands are also stored in the memory.

A Typical Instruction

Add R0, LOCA

- Add the operand at memory location LOCA to the operand in a register R0 in the processor.
- Place the sum into register R0.
- The original contents of LOCA are preserved.
- The original contents of R0 is overwritten.
- Instruction is fetched from the memory into the processor – the operand at LOCA is fetched and added to the contents of R0 – the resulting sum is stored in register R0.



Connections between Processor and memory

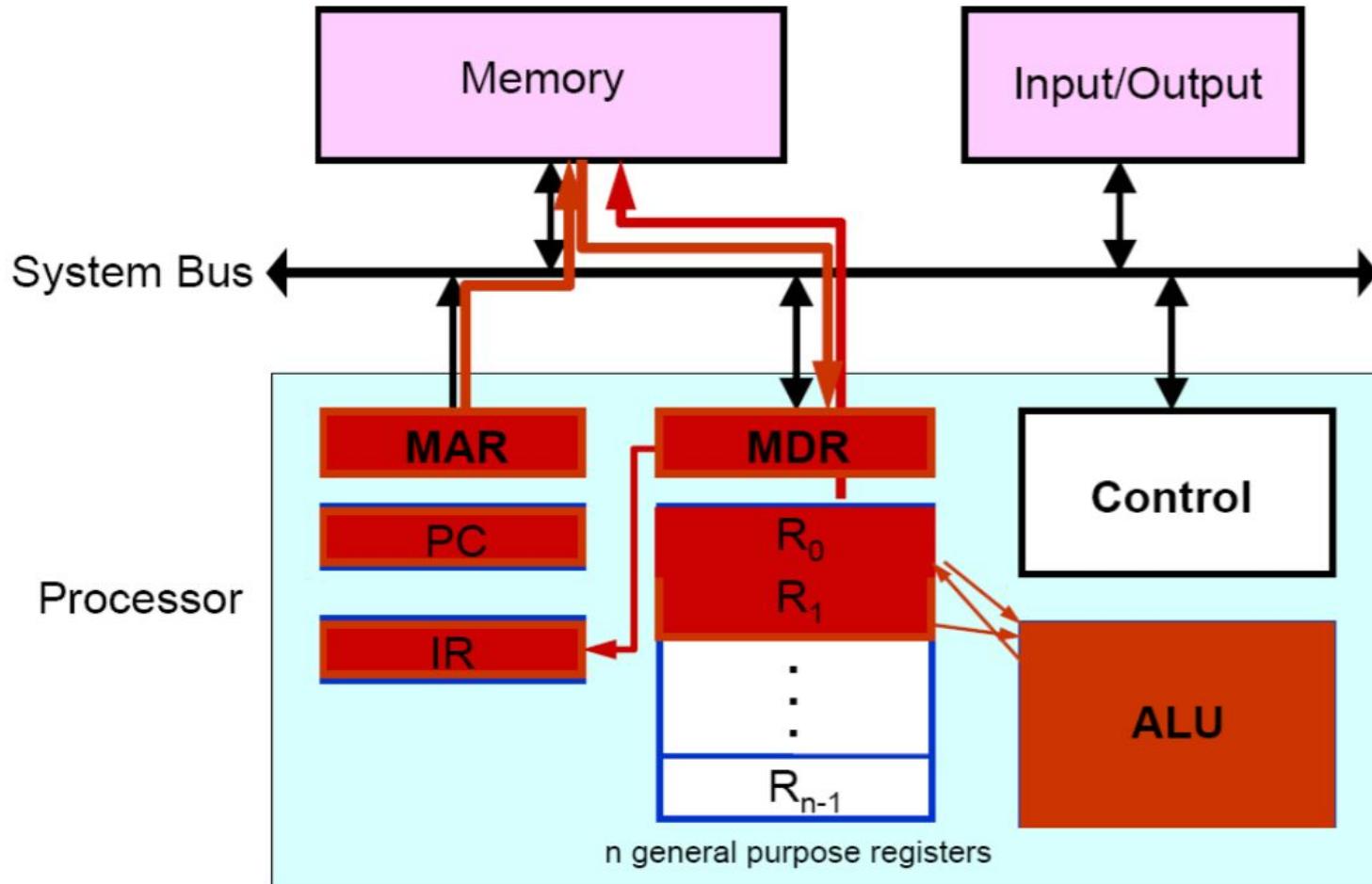
Registers

Registers are fast stand-alone storage locations that hold data temporarily. Multiple registers are needed to facilitate the operation of the CPU. Some of these registers are

- **Two registers-MAR (Memory Address Register) and MDR (Memory Data Register)** : To handle the data transfer between main memory and processor. MAR-Holds addresses, MDR-Holds data
- **Instruction register (IR)** : Hold the Instructions that is currently being executed
- **Program counter (PC)** : Points to the next instructions that is to be fetched from memory
- **General-purpose Registers:** are used for holding data, intermediate results of operations. They are also known as scratch-pad registers.



Basic Operational Concepts





INSTRUCTION FETCH – STEPS INVOLVED

- Program gets into the memory through an input device.
- Execution of a program starts by setting the PC to point to the first instruction of the program.
- The contents of PC are transferred to the MAR and a Read control signal is sent to the memory.
- The addressed word (here it is the first instruction of the program) is read out of memory and loaded into the MDR.
- The contents of MDR are transferred to the IR for instruction decoding



INSTRUCTION EXECUTION – STEPS INVOLVED

- The operation field of the instruction in IR is examined to determine the type of operation to be performed by the ALU.
- The specified operation is performed by obtaining the operand(s) from the memory locations or from GP registers.
 - 1) Fetching the operands from the memory requires sending the memory location address to the MAR and initiating a Read cycle.
 - 2) The operand is read from the memory into the MDR and then from MDR to the ALU.



INSTRUCTION EXECUTION – STEPS INVOLVED (Contd..)

3) The ALU performs the desired operation on one or more operands fetched in this manner and sends the result either to memory location or to a GP register.

4) The result is sent to MDR and the address of the location where the result is to be stored is sent to MAR and Write cycle is initiated.

Thus, the execute cycle ends for the current instruction and the PC is incremented to point to the next instruction for a new fetch cycle.



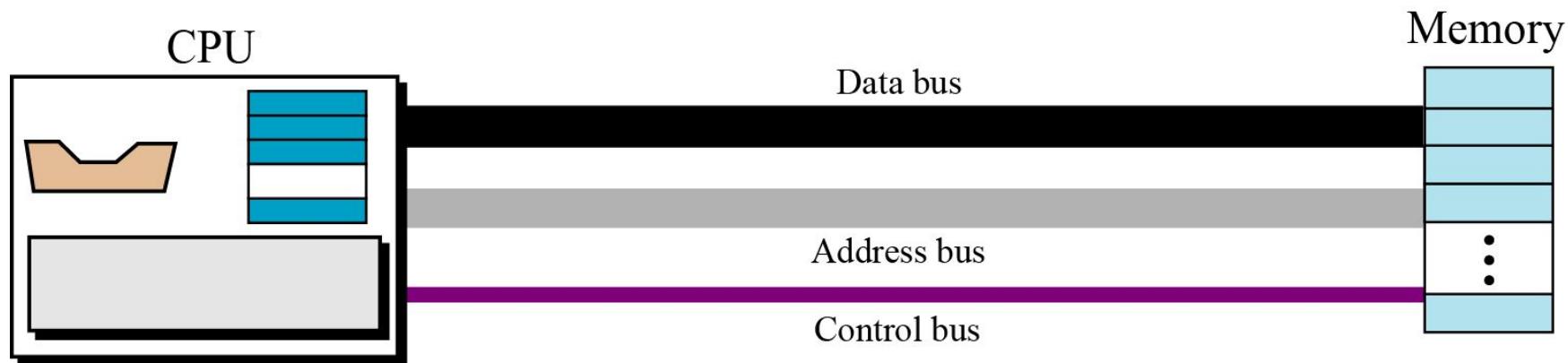
Interrupt

- An interrupt is a request from I/O device for service by processor
- Processor provides requested service by executing interrupt service routine (ISR)
- Contents of PC, general registers, and some control information are stored in memory .
- When ISR completed, processor restored, so that interrupted program may continue

BUS STRUCTURE

Connecting CPU and memory

The CPU and memory are normally connected by three groups of connections, each called a **bus**: *data bus*, *address bus* and *control bus*

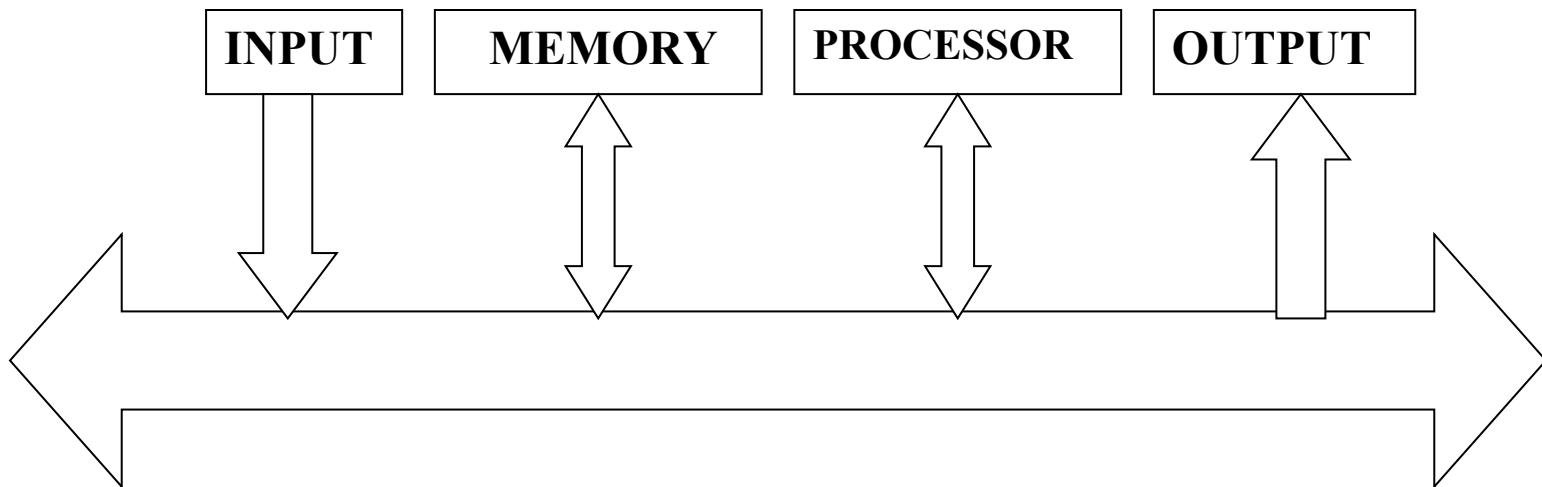


Connecting CPU and memory using three buses



BUS STRUCTURE

- Group of wires which carries information from CPU to peripherals or vice versa
- **Single bus structure:** Common bus used to communicate between peripherals and microprocessor



SINGLE BUS STRUCTURE



Drawbacks of the Single Bus Structure

- The devices connected to a bus vary widely in their speed of operation.
 - Some devices are relatively slow, such as printer and keyboard.
 - Some devices are considerably fast, such as optical disks.
 - Memory and processor units operate are the fastest parts of a computer.
- Efficient transfer mechanism thus is needed to cope with this problem.
 - A common approach is to include buffer registers with the devices to hold the information during transfers .
 - An another approach is to use two-bus structure and an additional transfer mechanism



TWO BUS STRUCTURE:

- **In two – bus structure :** One bus can be used to fetch instruction other can be used to fetch data, required for execution. The bus is said to perform two distinct functions. The main advantage of this structure is good operating speed but on account of more cost.

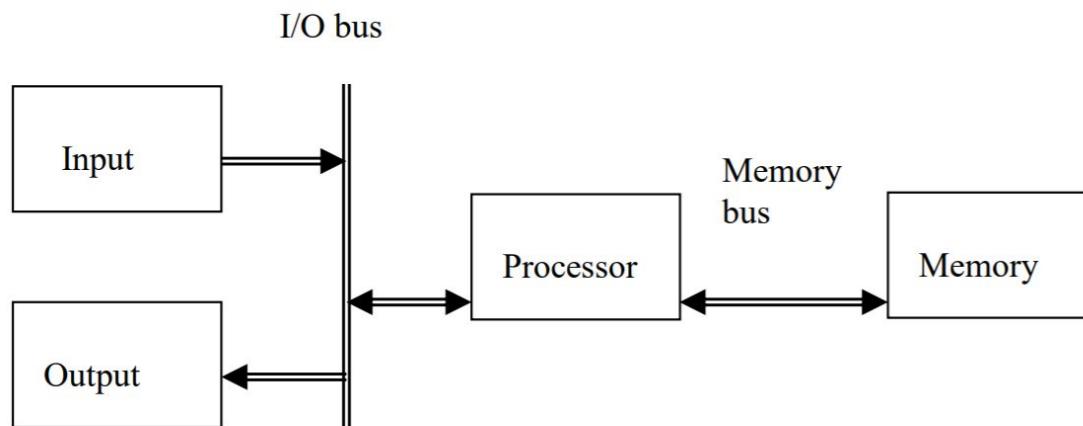
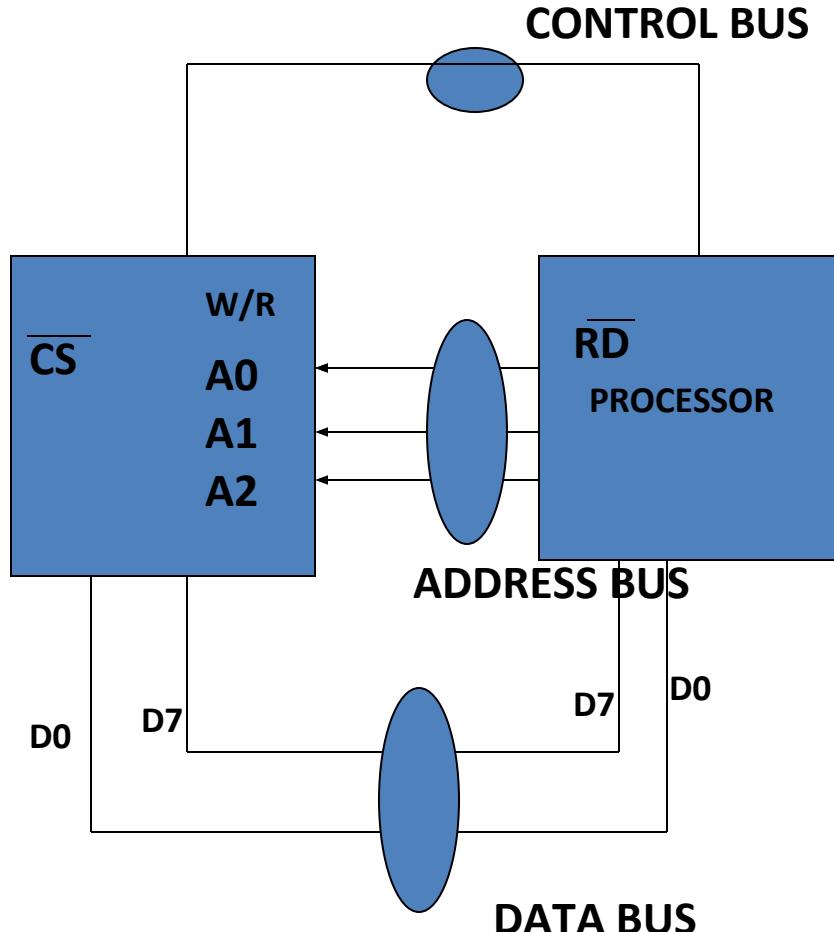


Figure 2.2 Two-bus Structure

MULTI BUS STRUCTURE

To improve performance **multi bus** structure can be used.

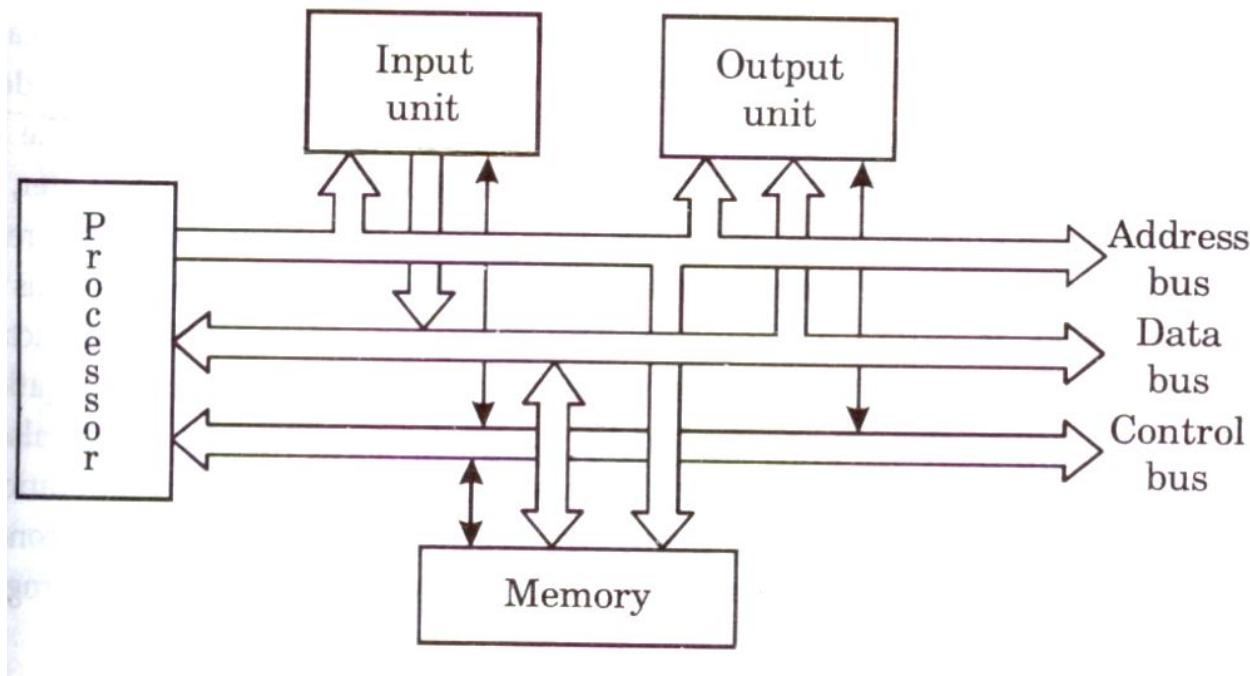


A ₂	A ₁	A ₀	Selected location
0	0	0	0 th Location
0	0	1	1 st Location
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	



- $2^3 = 8$ i.e. 3 address line is required to select 8 location
- In general $2^x = n$ where x number of address lines (address bit) and n is number of location
- **Address bus** : unidirectional : group of wires which carries address information bits form processor to peripherals (16,20,24 or more parallel signal lines)
- **Data bus**: bidirectional : group of wires which carries data information bit form processor to peripherals and vice – versa
- **Control bus**: bidirectional: group of wires which carries control signals form processor to peripherals and vice – versa

Figure below shows address, data and control bus and their connection with peripheral and microprocessor



Single bus structure showing the details of connection



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Memory Locations and Addresses



Memory Location and Addresses

- Memory consists of many millions of storage cells, each of which can store 1 bit.
- Data is usually accessed in n -bit groups. n is called word length.
- The memory of a computer can be schematically represented as a collection of words as shown in Figure 1.

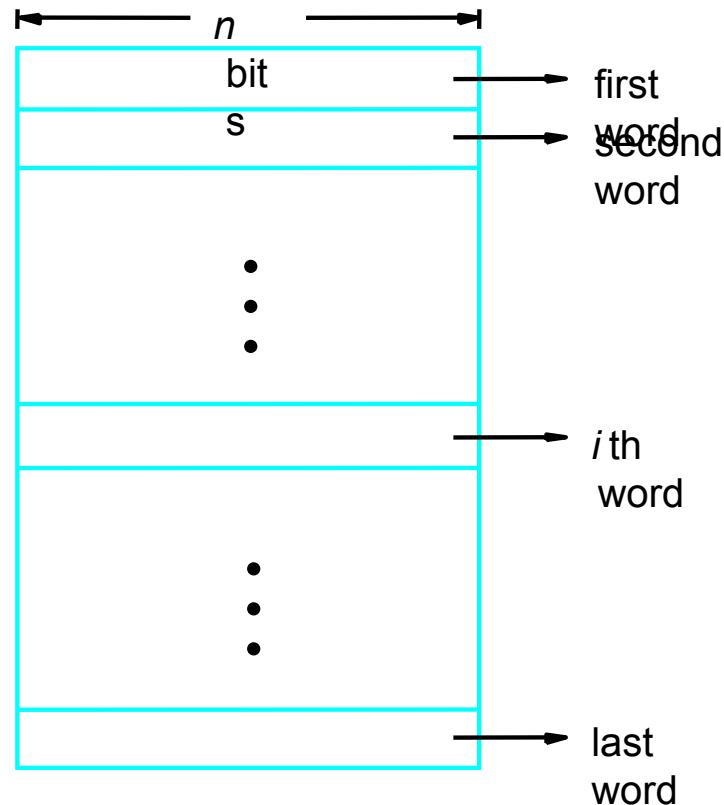
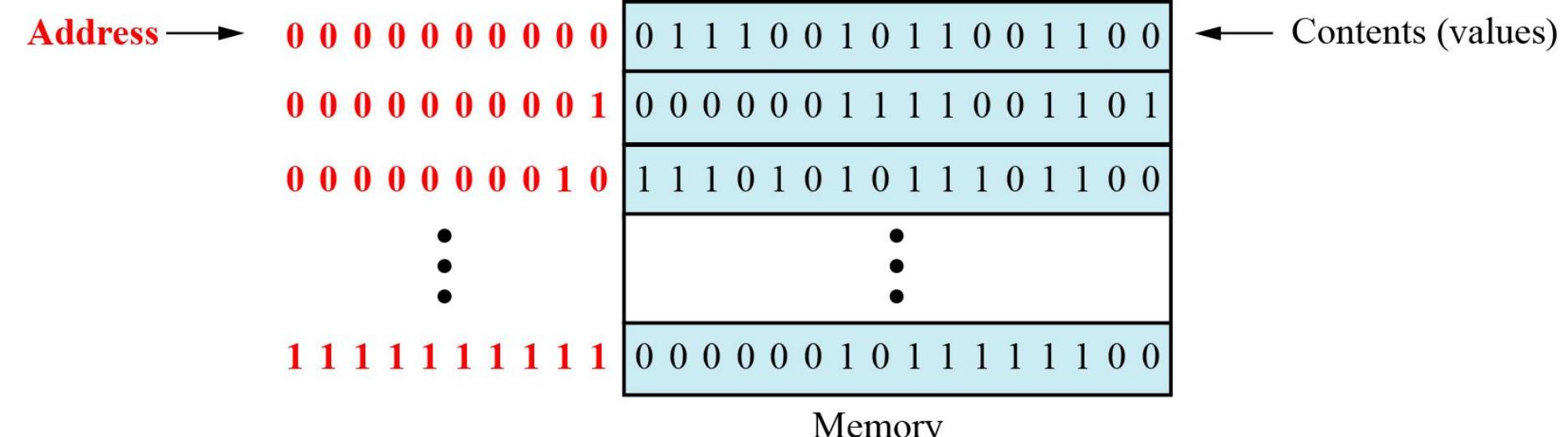


Figure 1 Main Memory words.



MEMORY LOCATIONS AND ADDRESSES

- **Main memory** is the second major subsystem in a computer. It consists of a collection of storage locations, each with a unique identifier, called an **address**.
- Data is transferred to and from memory in groups of bits called **words**. A word can be a group of 8 bits, 16 bits, 32 bits or 64 bits (and growing).
- If the word is 8 bits, it is referred to as a **byte**. The term “byte” is so common in computer science that sometimes a 16-bit word is referred to as a 2-byte word, or a 32-bit word is referred to as a 4-byte word.



Main memory

Address space

- To access a word in memory requires an identifier. Although programmers use a name to identify a word (or a collection of words), at the hardware level each word is identified by an address.
- The total number of uniquely identifiable locations in memory is called the **address space**.
- For example, a memory with 64 kilobytes (16 address line required) and a word size of 1 byte has an address space that ranges from 0 to 65,535.



Table 5.1 Memory units

<i>Unit</i>	<i>Exact Number of Bytes</i>	<i>Approximation</i>
kilobyte	2^{10} (1024) bytes	10^3 bytes
megabyte	2^{20} (1,048,576) bytes	10^6 bytes
gigabyte	2^{30} (1,073,741,824) bytes	10^9 bytes
terabyte	2^{40} bytes	10^{12} bytes

Memory addresses are defined using unsigned binary integers.

Example 1

A computer has 32 MB (megabytes) of memory. How many bits are needed to address any single byte in memory?

Solution

The memory address space is 32 MB, or 2^{25} ($2^5 \times 2^{20}$). This means that we need $\log_2 2^{25}$, or **25 bits**, to address each byte.

Example 2

A computer has 128 MB of memory. Each word in this computer is eight bytes. How many bits are needed to address any single word in memory?

Solution

The memory address space is 128 MB, which means 2^{27} . However, each word is eight (2^3) bytes, which means that we have 2^{24} words. This means that we need $\log_2 2^{24}$, or **24 bits**, to address each word.

MEMORY OPERATIONS

- Today, **general-purpose computers** use a set of instructions called a **program** to process data.
- A computer executes the program to create output data from input data
- Both program instructions and data operands are stored in memory
- Two basic operations requires in memory access
 - **Load operation (Read or Fetch)**-Contents of specified memory location are read by processor
 - **Store operation (Write)**- Data from the processor is stored in specified memory location



Assignment of Byte Address

- **Big-endian** and **little-endian** are terms that describe the order in which a sequence of bytes are stored in computer **memory**.
- **Big-endian** is an order in which the "**bigend**" (most significant value in the sequence) is stored first (at the lowest storage address).
- **Little-endian** is an order in which the "**Little end**" (least significant value in the sequence) is stored first (at the lowest storage address).



0x100 0x101 0x102 0x103



Big Endian

0x100 0x101 0x102 0x103



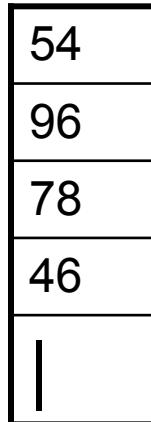
Little Endian



Assignment of byte addresses

- Little Endian (e.g., in DEC, Intel)
 - » low order byte stored at lowest address
 - » byte0 byte1 byte2 byte3
- Eg: 46,78,96,54 (32 bit data)
- H BYTE L BYTE

- 8000
- 8001
- 8002
- 8003
- 8004





Big Endian

- Big Endian (e.g., in IBM, Motorola, Sun, HP)
 - » high order byte stored at lowest address
 - » byte3 byte2 byte1 byte0
- Programmers/protocols should be careful when transferring binary data between Big Endian and Little Endian machines



Big-Endian and Little-Endian Assignments

Big-Endian: lower byte addresses are used for the most significant bytes of the word

Little-Endian: opposite ordering. lower byte addresses are used for the less significant bytes of the word

Word address	Byte address			
0	0	1	2	3
4	4	5	6	7
	•	•	•	•
$2^k - 4$	$2^k - 4$	$2^k - 3$	$2^k - 2$	$2^k - 1$

(a) Big-endian assignment

Byte address	3	2	1	0
0	3	2	1	0
4	7	6	5	4
	•	•	•	•
$2^k - 4$	$2^k - 1$	$2^k - 2$	$2^k - 3$	$2^k - 4$

(b) Little-endian assignment

Byte and word addressing.

- In case of 16 bit data, aligned words begin at byte addresses of 0,2,4,.....
- In case of 32 bit data, aligned words begin at byte address of 0,4,8,.....
- In case of 64 bit data, aligned words begin at byte addresses of 0,8,16,.....
- In some cases words can start at an arbitrary byte address also then, we say that word locations are **unaligned**



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

INSTRUCTION AND INSTRUCTION SEQUENCING



Introduction

- **Instruction:** is command to the microprocessor to perform a given task on specified data.
- **Instruction Set:** The entire group of these instructions are called instruction set.
- **instruction sequencing :** The order in which the instructions in a program are carried out.



Types of Instructions

- Most computer instructions are classified into 3 categories
 - Data transfer Instructions
 - To Transfer data from one location to another
 - Data Manipulation Instructions
 - To perform the operations by the ALU
 - Program control Instructions
 - To control the system



Data transfer Instructions

They are also called copy instructions.

Some instructions in 8086:

MOV -Copy from the source to the destination

LDA - Load the accumulator

STA - Store the accumulator

PUSH - Push the register pair onto the stack

POP - Pop off stack to the register pair



Data Manipulation Instructions

- To perform the operations by the ALU
- Three categories:
 - Arithmetic Instructions
 - Logical and bit manipulation instructions
 - Shift instructions



Arithmetic Instructions

Used to perform arithmetic operations

Some instruction in 8086

INC ↗ Increment the data by 1

DEC ↗ Decreases data by 1

ADD ↗ perform sum of data

ADC ↗ Add with carry bit.

MUL ↗ perform multiplication



Logical and bit manipulation instructions

Used to perform logical operations

Some instructions are:

AND ? bitwise AND operation

OR ? bitwise OR operation

NOT ? invert each bit of a byte or word

XOR ? Exclusive-OR operation over each bit

Shift instructions

used for shifting or rotating the contents of the register

Some instructions are:

SHR ☐ shift bits towards the right and put zero(S) in MSBs

ROL ☐ rotate bits towards the left, i.e. MSB to LSB and to Carry Flag [CF]

RCL ☐ rotate bits towards the left, i.e. MSB to CF and CF to LSB.



Instruction Formats

(Types of instruction based on the address field)

- A instruction in computer comprises of groups called fields.
- These field contains different information
- The most common fields are:

Operation field : specifies the operation to be performed like addition.

Address field : contain the location of operand

Mode field : specifies how to find the operand

- A instruction is of various length depending upon the number of addresses it contain.
- On the basis of number of address, instruction are classified as:
 - **Zero Address Instructions**
 - **One Address Instructions**
 - **Two Address Instructions**
 - **Three Address Instructions**



Zero Address Instructions

Used in stack based computers which do not use address field in instruction

- Location of all operands are defined implicitly
- Operands are stored in a structure called pushdown stack

Example: Evaluate $(A+B) * (C+D)$

- Using Zero-Address instruction
 1. **PUSHA** ; TOS $\leftarrow A$
 2. **PUSHB** ; TOS $\leftarrow B$
 3. **ADD** ; TOS $\leftarrow (A + B)$
 4. **PUSH C** ; TOS $\leftarrow C$
 5. **PUSHD** ; TOS $\leftarrow D$
 6. **ADD** ; TOS $\leftarrow (C + D)$
 7. **MUL** ; TOS $\leftarrow (C+D)*(A+B)$
 8. **POP X** ; M[X] \leftarrow TOS

One address Instruction

- This use a implied ACCUMULATOR register for data manipulation.
- One operand is in accumulator and other is in register or memory location.

Example: Evaluate $(A+B) * (C+D)$

- Using One-Address Instruction
 1. LOAD A ; $AC \leftarrow M[A]$
 2. ADD B ; $AC \leftarrow AC + M[B]$
 3. STORE T ; $M[T] \leftarrow AC$
 4. LOAD C ; $AC \leftarrow M[C]$
 5. ADD D ; $AC \leftarrow AC + M[D]$
 6. MUL T ; $AC \leftarrow AC * M[T]$
 7. STORE X ; $M[X] \leftarrow AC$



Two Address Instruction

This is common in commercial computers.

Here two address can be specified in the instruction.

Example: Evaluate $(A+B) * (C+D)$

Using Two address Instruction:

1. MOV R1,A ; $R1=M[A]$
2. ADD R1,B ; $R1=R1+M[B]$
3. MOV R2,C ; $R2=M[C]$
4. ADD R2,D ; $R2=R2+M[D]$
5. MUL R1,R2 ; $R1=R1*R2$
6. MOV X,R1 ; $M[X]=R1$



Three Address Instruction

This has three address field to specify a register or a memory location.

Program created are much short in size
creation of program much easier
does not mean that program will run much faster

Example: Evaluate $(A+B) * (C+D)$

Using Three address Instruction

1. ADD R1,A,B ; $R1=M[A]+M[B]$
2. ADD R2,C,D ; $R2=M[C]+M[D]$
3. MUL X,R1,R2 ; $M[X]=R1*R2$



Instruction Cycle

- the basic operational process of a computer.
- also known as **fetch-decode-execute cycle**
- This process is repeated continuously by CPU from boot up to shut down of computer.

steps that occur during an instruction cycle:

- 1. Fetch the Instruction**
- 2. Decode the Instruction**
- 3. Read the Effective Address**
- 4. Execute the Instruction**

1. Fetch the Instruction

The instruction is fetched from memory address that is stored in PC(Program Counter) and stored in the (instruction register) IR.

At the end of the fetch operation, PC is incremented by 1 and it then points to the next instruction to be executed.

2. Decode the Instruction

The instruction in the IR is decoded(Interpreted).



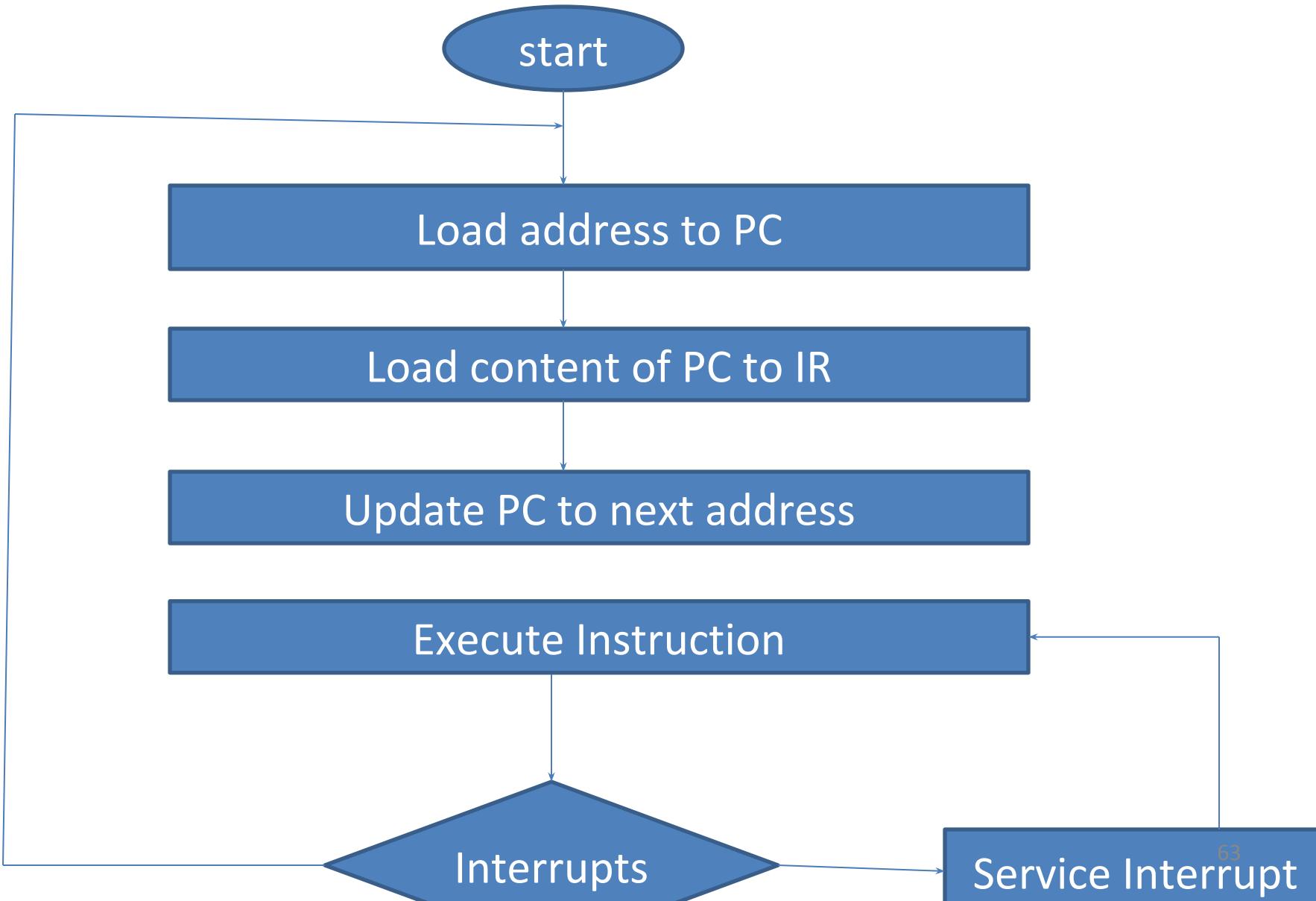
3. Read the Effective Address

If the instruction has an indirect address, the effective address is read from the memory. Otherwise operands are directly read in case of immediate operand instruction.

4. Execute the Instruction

The Control Unit passes the information in the form of control signals to the functional unit of CPU. The result generated is stored in main memory or sent to an output device.

The cycle is then repeated by fetching the next instruction. Thus in this way the instruction cycle is repeated continuously.





Addressing Modes

Different ways in which the location of the operand is specified in an instruction is referred as addressing modes

The purpose of using addressing mode is:

- To give the programming versatility to the user.
- To reduce the number of bits in addressing field of instruction.

Types of Addressing Modes

- Immediate Addressing
- Direct Addressing
- Indirect Addressing
- Register Addressing
- Register Indirect Addressing
- Relative Addressing
- Indexed Addressing
- Auto Increment
- Auto Decrement

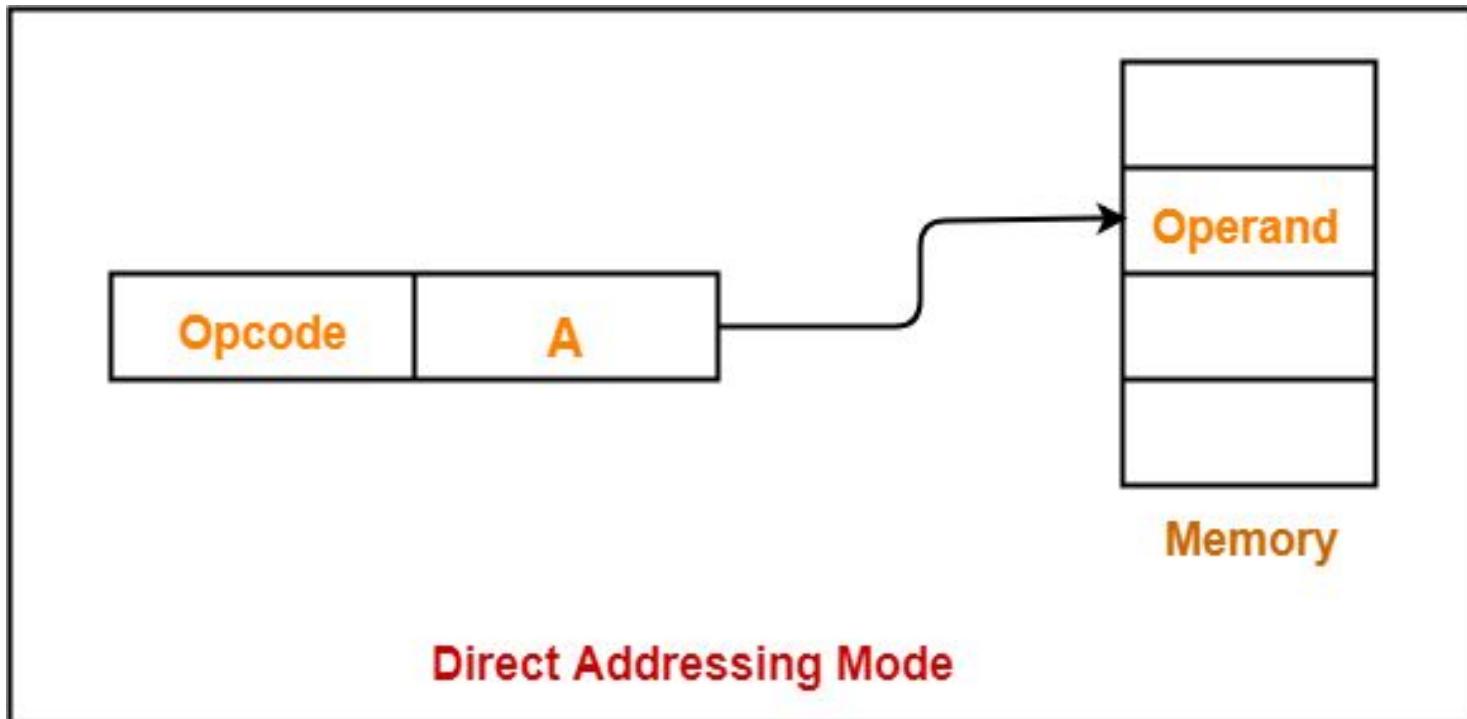
Immediate Addressing

- Operand is given explicitly in the instruction
- e.g. ADD 5
 - Add 5 to contents of accumulator
 - 5 is operand
- No memory reference to fetch data
- Fast
- Limited range



Direct Addressing

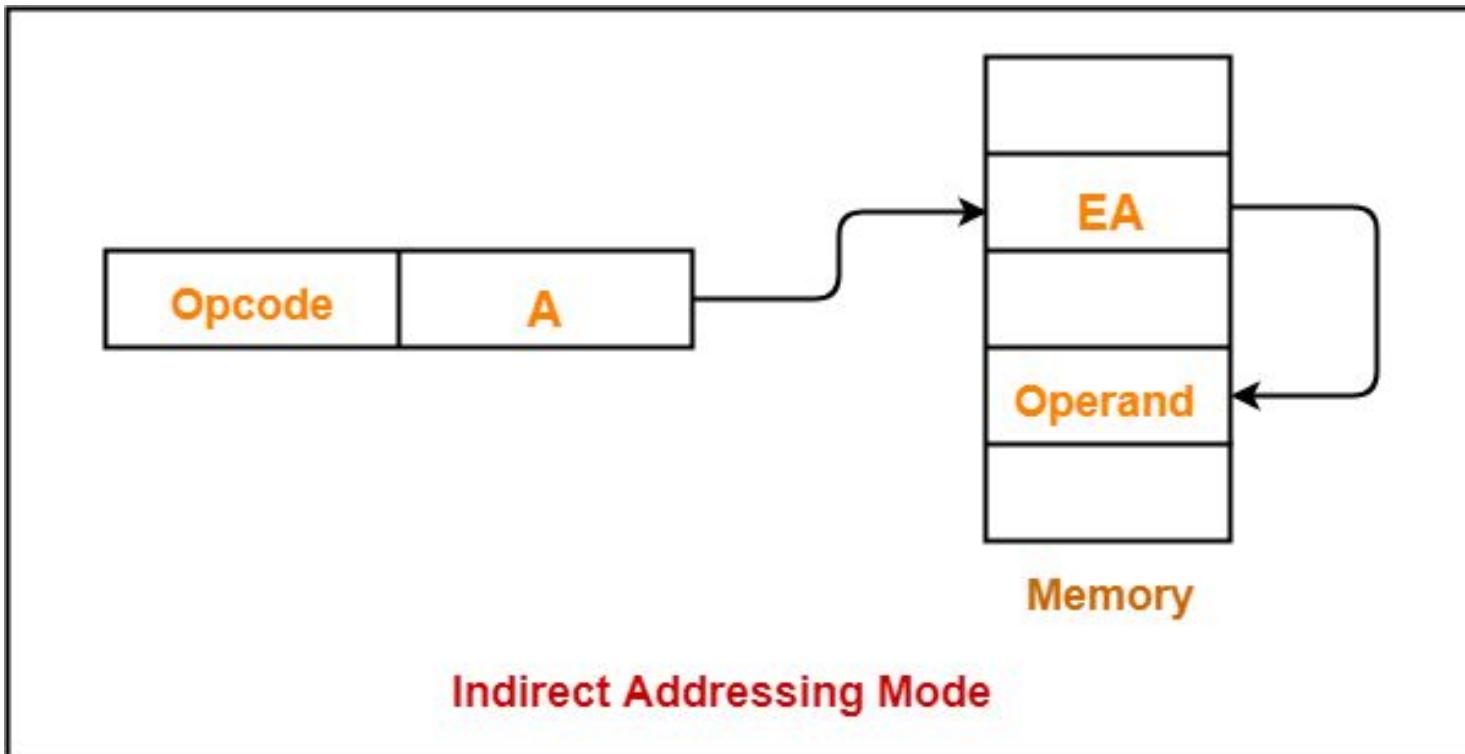
- Address field contains address of operand
- Effective address (EA) = address field (A)
- e.g. ADD A
 - Add contents of cell A to accumulator
 - Look in memory at address A for operand
- Single memory reference to access data
- No additional calculations to work out effective address
- Limited address space





Indirect Addressing (1)

- Memory cell pointed to by address field contains the address of (pointer to) the operand
Two references to memory are required to fetch the operand.
- Effective Address = [A]
 - Look in A, find address (A) and look there for operand
- e.g. ADD (A)
 - Add contents of cell pointed to by contents of A to the accumulator





Register Direct Addressing

In this addressing mode,

- The operand is contained in a register set.
- The address field of the instruction refers to a CPU register that contains the operand.

- No memory access
- Very fast execution
- Very limited address space
- Limited number of registers
- Very small address field needed
 - Shorter instructions
 - Faster instruction fetch

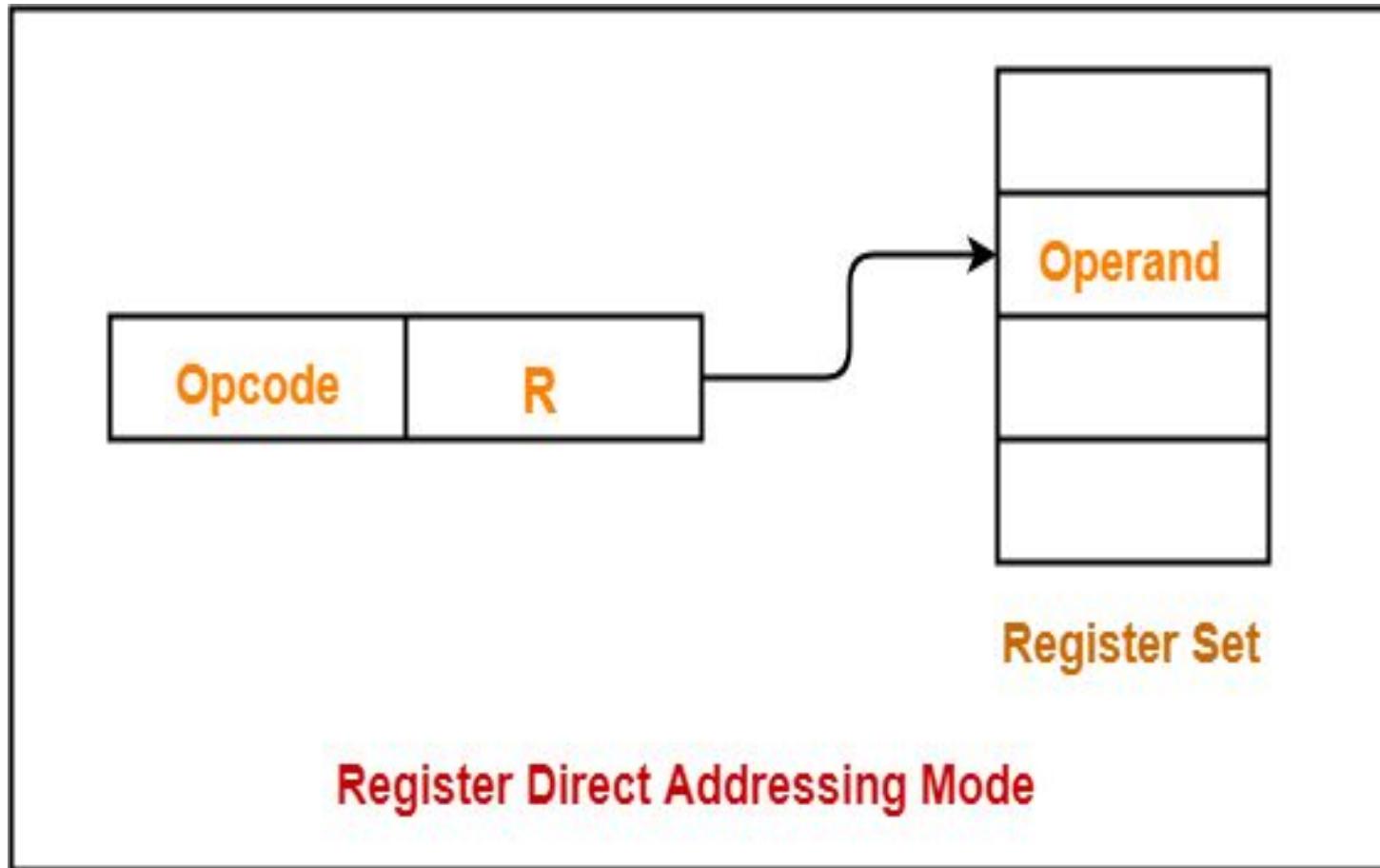


Eg:

ADD R will increment the value stored in the accumulator by the content of register R.

$$AC \leftarrow AC + [R]$$

- This addressing mode is similar to direct addressing mode.
- The only difference is address field of the instruction refers to a CPU register instead of main memory.





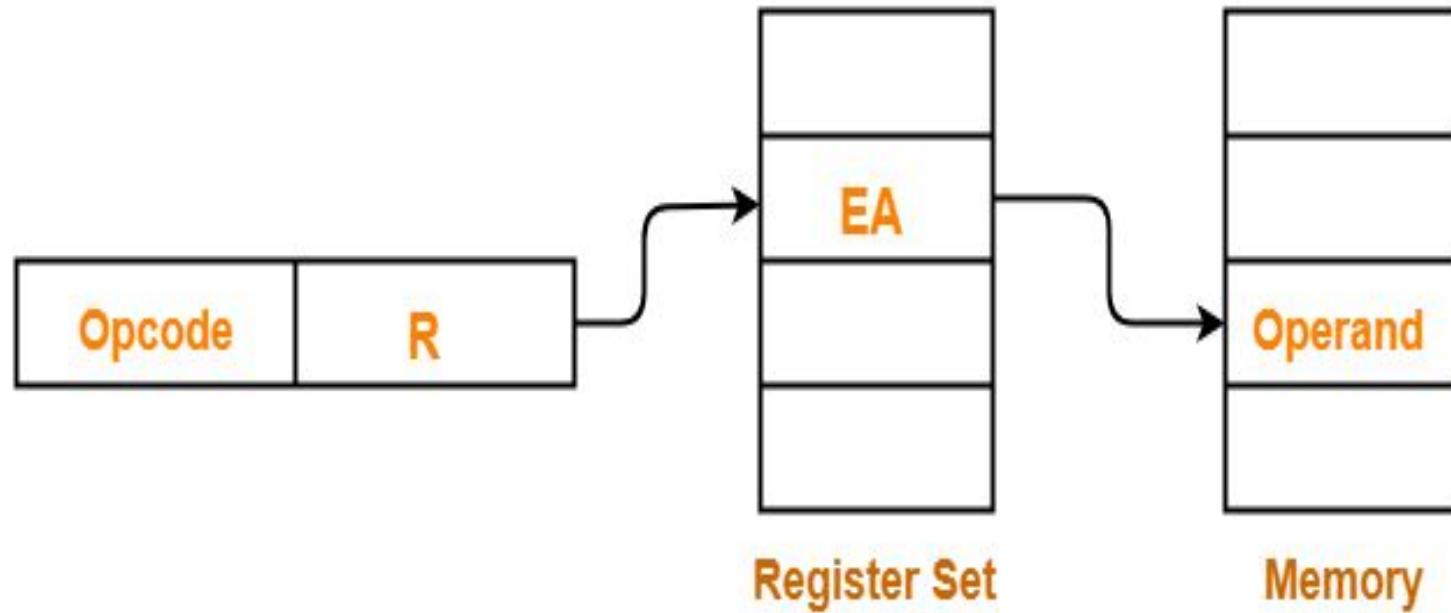
Register Indirect Addressing

- The address field of the instruction refers to a CPU register that contains the effective address of the operand.
- Only one reference to memory is required to fetch the operand

Eg:

ADD R will increment the value stored in the accumulator by the content of memory location specified in register R.

$$AC \leftarrow AC + [[R]]$$



Register Indirect Addressing Mode



Indexed Addressing

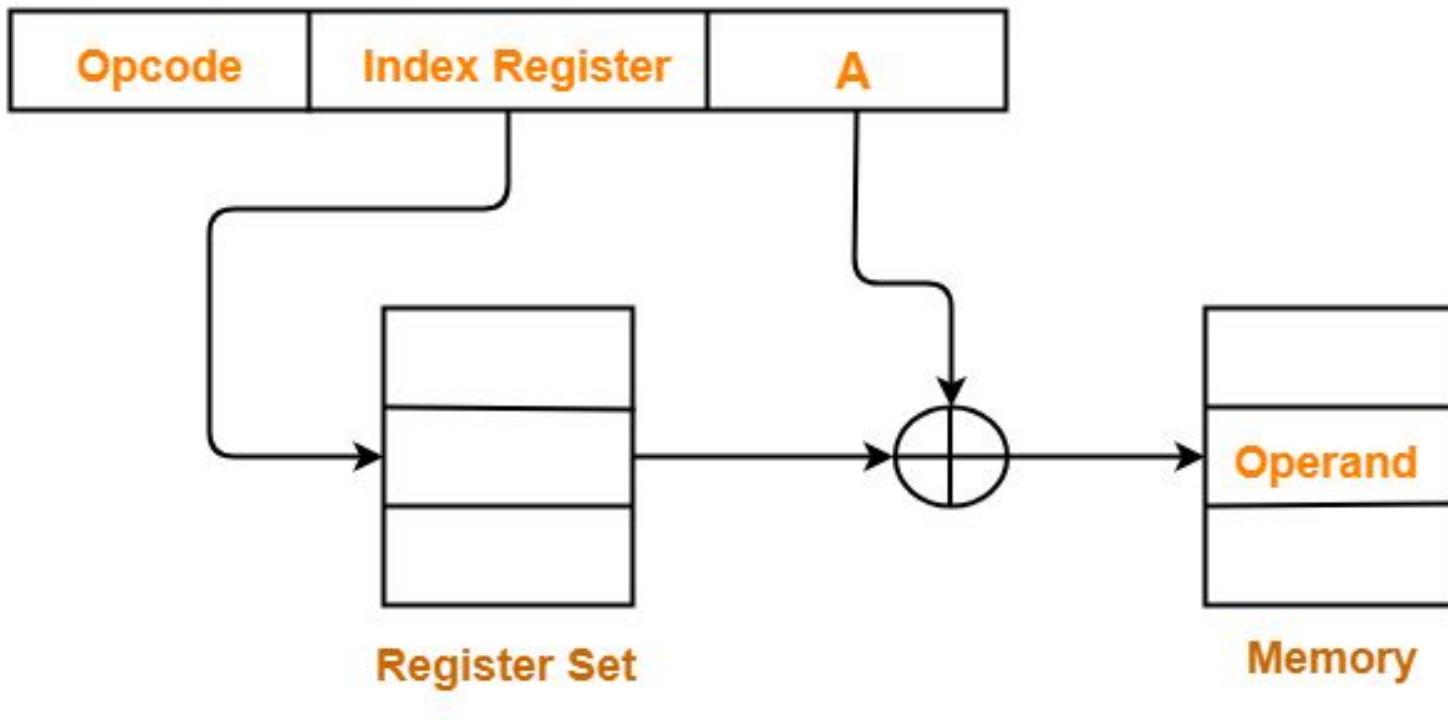
In this addressing mode,

- Effective address of the operand is obtained by adding the content of index register with the address part of the instruction.

Effective Address

= Content of Index Register +

Address part of the instruction



Indexed Addressing Mode



Relative Addressing

A version of displacement addressing

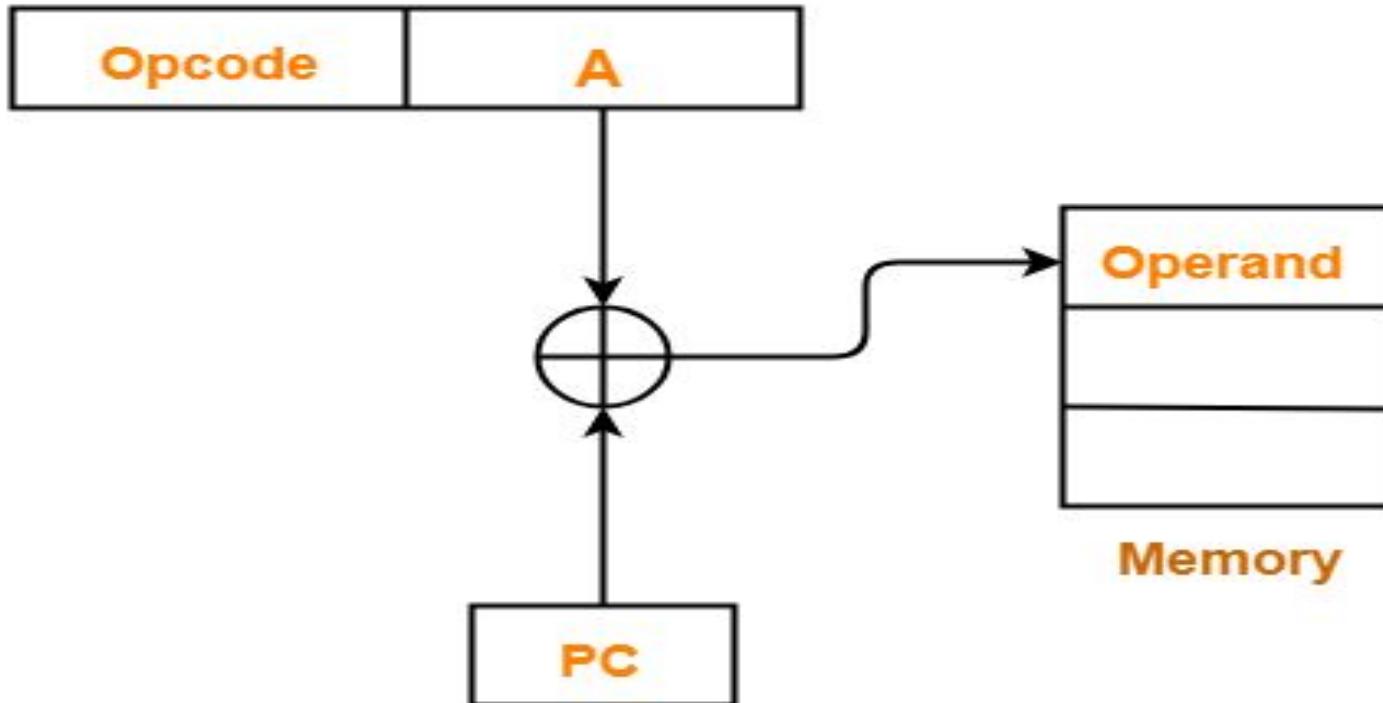
In this addressing mode,

- Effective address of the operand is obtained by adding the content of program counter with the address part of the instruction.

Effective Address

= Content of Program Counter +

Address part of the instruction



Relative Addressing Mode



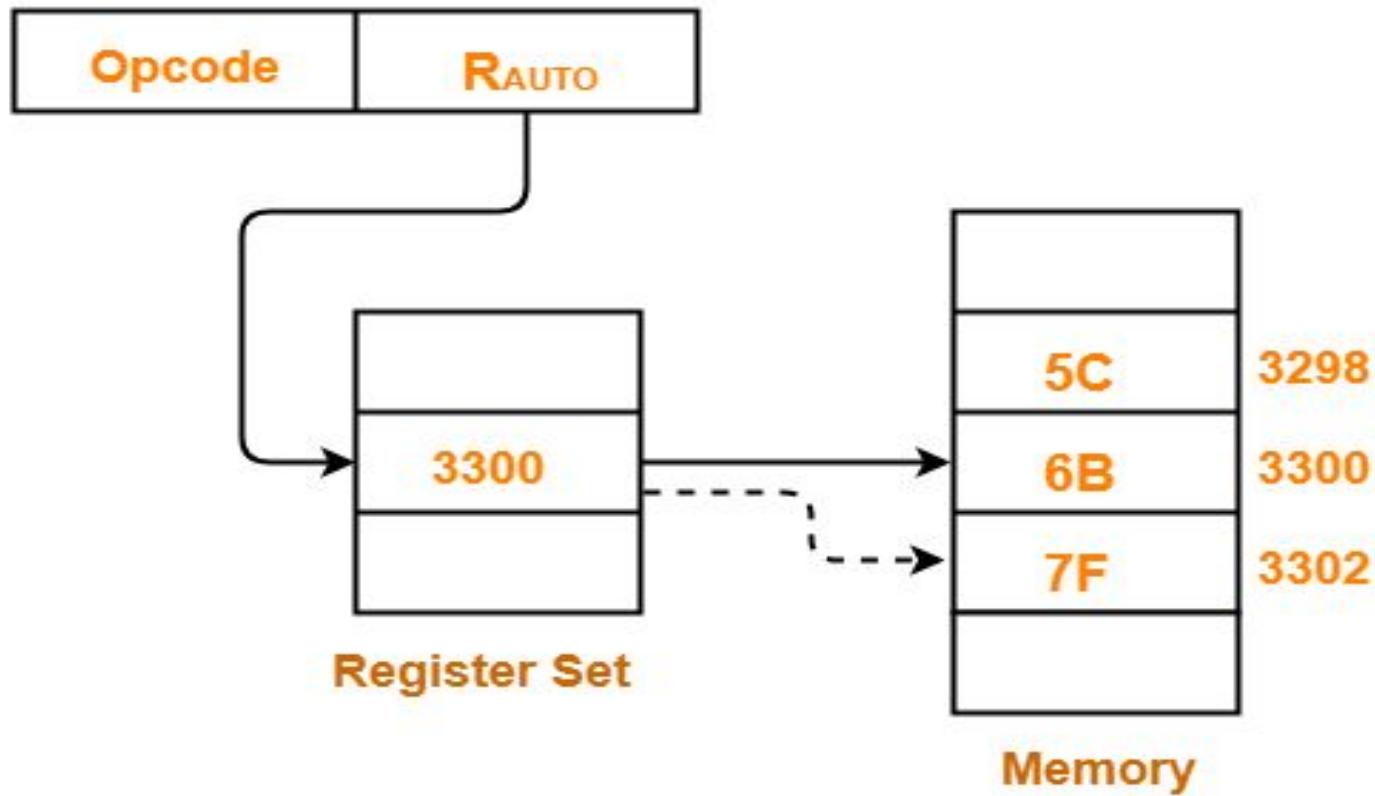
Auto increment mode

A special case of Register Indirect Addressing Mode where

Effective Address of the Operand
= Content of Register

In this addressing mode,

- After accessing the operand, the content of the register is automatically incremented by step size ‘d’.
- Step size ‘d’ depends on the size of operand accessed.
- Only one reference to memory is required to fetch the operand.



Auto-Increment Addressing Mode



Auto decrement mode

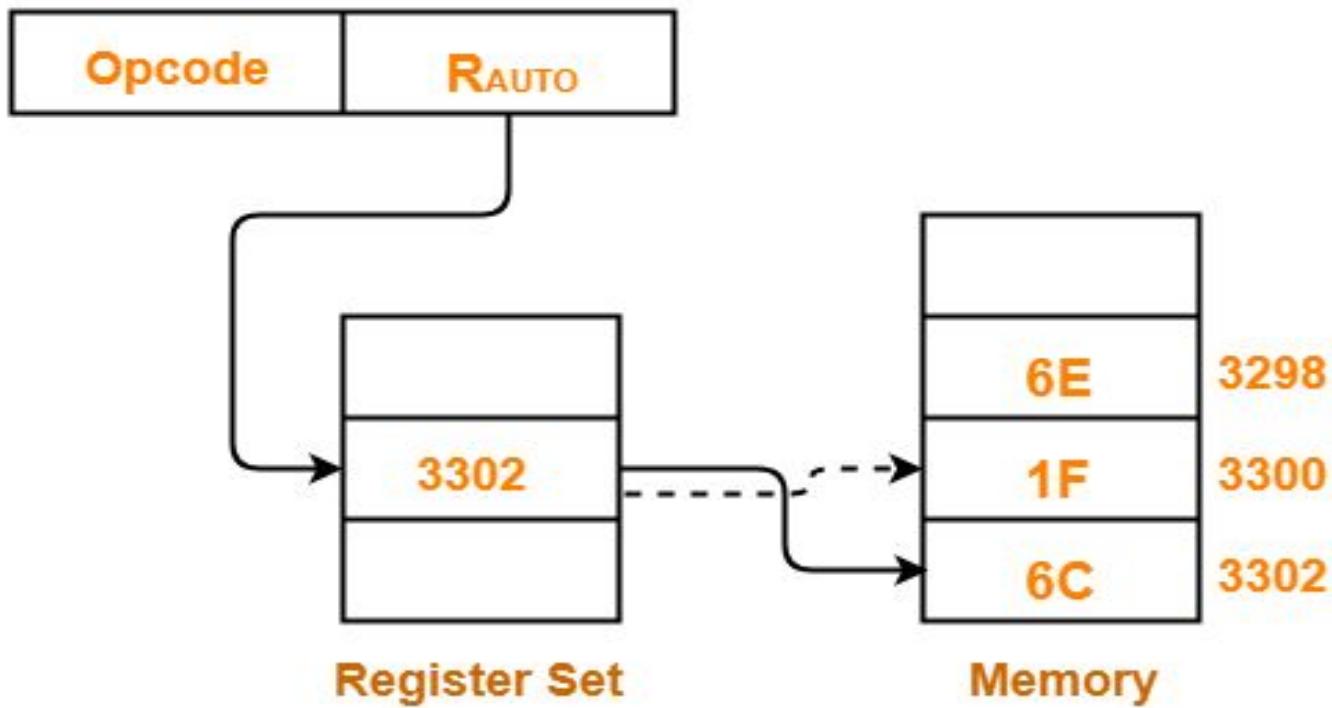
- A special case of Register Indirect Addressing Mode where

Effective Address of the Operand

= Content of Register – Step Size

In this addressing mode

- First, the content of the register is decremented by step size ‘d’.
- Step size ‘d’ depends on the size of operand accessed.
- After decrementing, the operand is read.
- Only one reference to memory is required to fetch the operand.





Microprocessors

Microprocessor : is a CPU on a single chip.

Microcontroller: If a microprocessor,

its associated support circuitry,

memory and

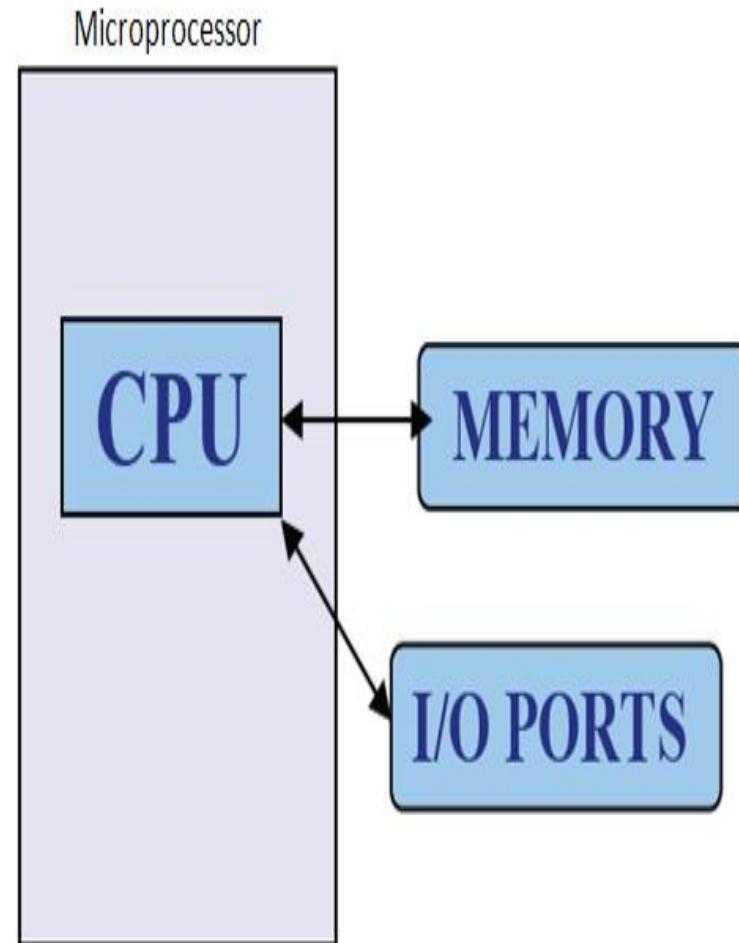
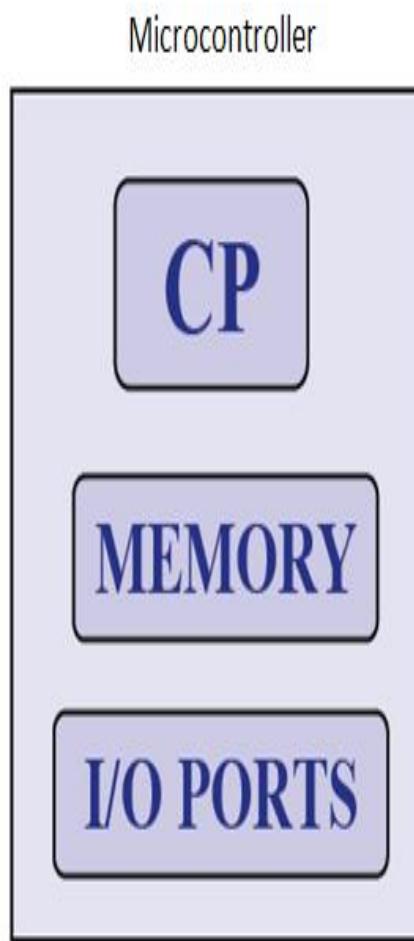
peripheral I/O components

are implemented on a single chip, it is a

microcontroller.

- We use AVR microcontroller as the example in our course study

What is Microprocessor and Microcontroller?

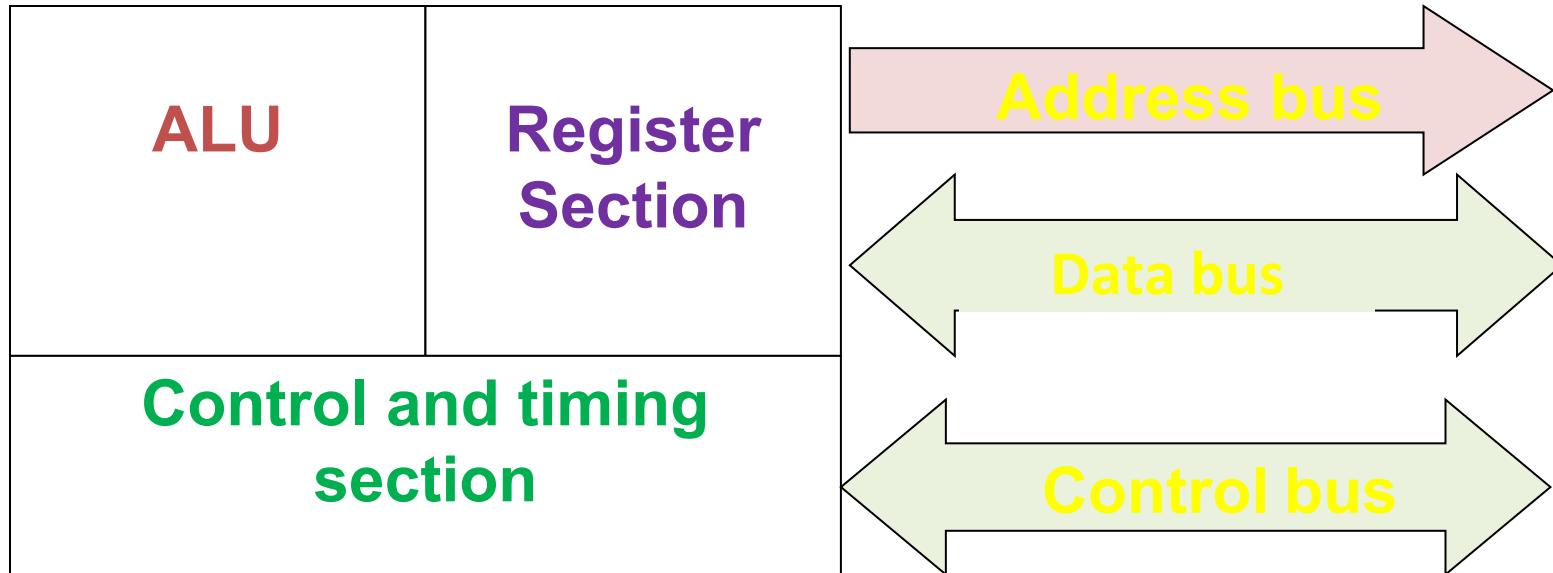




Sl No	Microprocessor	Microcontroller
1	CPU is stand-alone, RAM, ROM, I/O, timer are separate	CP, RAM, ROM, I/O and timer are all on single chip
2	Designer can decide on the amount of ROM, RAM and I/O ports	Fix amount of on-chip ROM, RAM, I/O ports
3	Expansive	Not Expansive
4	General purpose	Single purpose
5	Microprocessor based system design is complex and expensive	Microcontroller based system design is rather simple and cost effective
6	The instruction set of microprocessor is complex with large number of instructions.	The instruction set of a Microcontroller is very simple with the less number of instructions.



Internal structure and basic operation of microprocessor



Block diagram of a Microprocessor



- Microprocessor performs three main tasks:
 - data transfer between itself and the memory or I/O systems
 - simple arithmetic and logic operations
 - program flow via simple decisions



Microprocessor types

- Microprocessors can be characterized based on
 - the word size
 - 8 bit, 16 bit, 32 bit, etc. processors
 - Instruction set structure
 - RISC (Reduced Instruction Set Computer), CISC (Complex Instruction Set Computer)
 - Functions
 - General purpose, special purpose such image processing, floating point calculations
 - And more ...



Evolution of Microprocessors

- The first **microprocessor** was introduced in 1971 by Intel Corp.
- It was named Intel 4004 as it was a 4 bit processor.

Categories according to the generations or size

First Generation (4 - bit Microprocessors)

- could perform simple arithmetic such as addition, subtraction, and logical operations like Boolean OR and Boolean AND.
- had a control unit capable of performing control functions like
 - fetching an instruction from storage memory,
 - decoding it, and then
 - generating control pulses to execute it.



Second Generation (8 - bit Microprocessor)

- The second generation microprocessors were introduced in 1973 again by Intel.
- the first 8 - bit microprocessor which could perform arithmetic and logic operations on 8-bit words.

Third Generation (16 - bit Microprocessor)

- introduced in 1978
- represented by **Intel's 8086, Zilog Z800 and 80286,**
- 16 - bit processors with a performance like minicomputers.



Fourth Generation (32 - bit Microprocessors)

- Several different companies introduced the 32-bit microprocessors
- the most popular one is the **Intel 80386**

Fifth Generation (64 - bit Microprocessors)

- Introduced in 1995
- After 80856, Intel came out with a new processor namely Pentium processor followed by **Pentium Pro CPU**
- allows multiple CPUs in a single system to achieve multiprocessing.
- Other improved 64-bit processors are **Celeron, Dual, Quad, Octa Core processors**.

Typical microprocessors

- Most commonly used
 - 68K
 - Motorola
 - x86
 - Intel
 - IA-64
 - Intel
 - MIPS
 - Microprocessor without interlocked pipeline stages
 - ARM
 - Advanced RISC Machine
 - PowerPC
 - Apple-IBM-Motorola alliance
 - Atmel AVR
- A brief summary will be given later



8086 Microprocessor

- designed by Intel in 1976
- 16-bit Microprocessor having
- 20 address lines
- 16 data lines
- provides up to 1MB storage
- consists of powerful instruction set, which provides operations like multiplication and division easily.

supports two modes of operation

Maximum mode :

suitable for system having multiple processors

Minimum mode :

suitable for system having a single processor.

Features of 8086

- Has an instruction queue, which is capable of storing six instruction bytes
- First 16-bit processor having
 - 16-bit ALU
 - 16-bit registers
 - internal data bus
 - 16-bit external data bus



uses two stages of pipelining

1. Fetch Stage and
2. Execute Stage

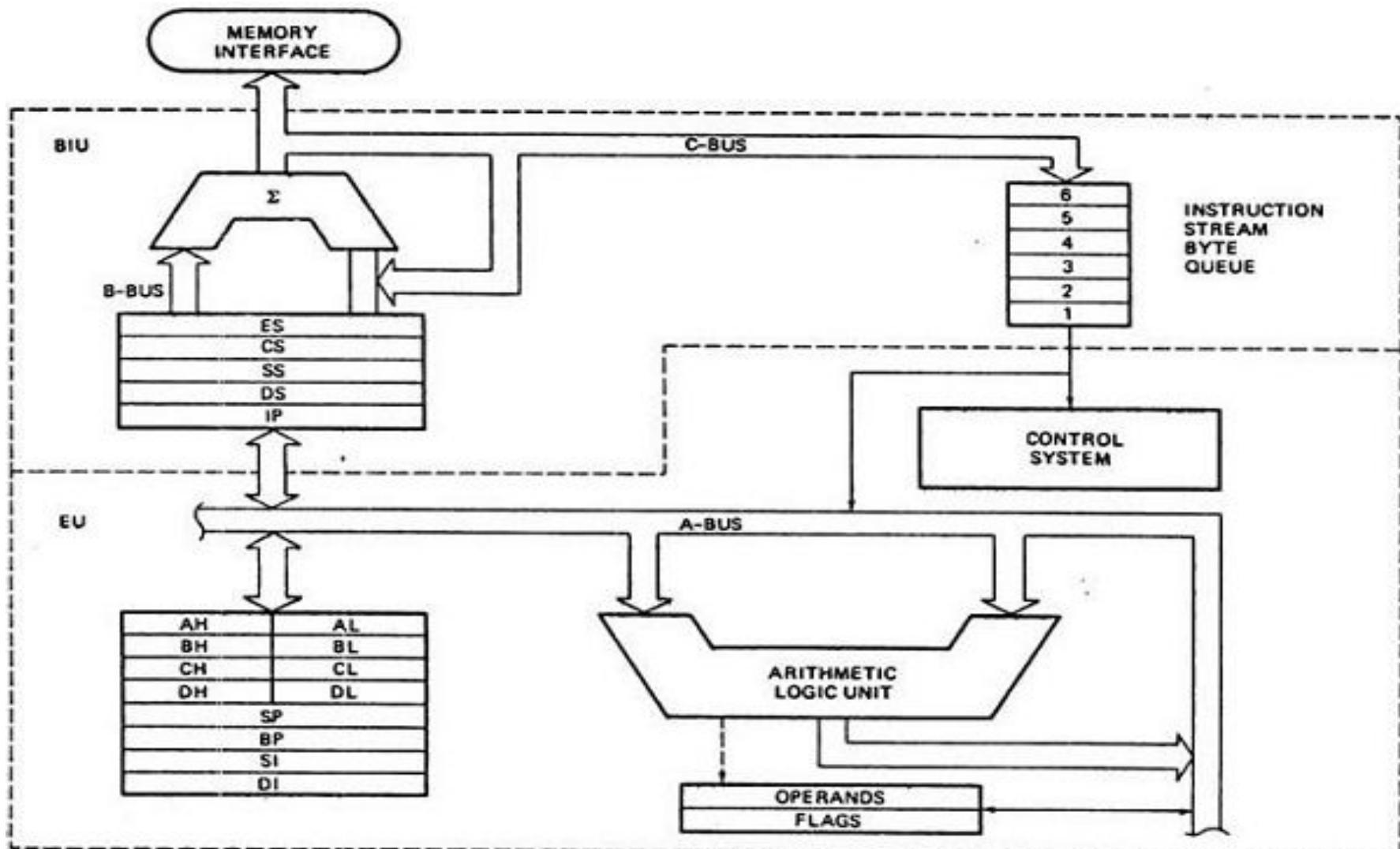
which improves performance.

Fetch stage : can pre-fetch up to 6 bytes of instructions and stores them in the queue.

Execute stage : executes these instructions.



Architecture of 8086





Segments in 8086

memory is divided into various sections called segments

Code segment : where you store the program.

Data segment : where the data is stored.

Extra segment : mostly used for string operations.

Stack segment : used to push/pop



General purpose registers

used to store temporary data within the microprocessor

AX – Accumulator

16 bit register

divided into two 8-bit registers AH and AL to perform 8-bit instructions also generally used for arithmetical and logical instructions

BX – Base register

16 bit register

divided into two 8-bit registers BH and BL to perform 8-bit instructions also

Used to store the value of the offset.



CX – Counter register

16 bit register

divided into two 8-bit registers CH and CL to
perform 8-bit instructions also

Used in looping and rotation

DX – Data register

16 bit register

divided into two 8-bit registers DH and DL to
perform 8-bit instructions also

Used in multiplication an input/output port addressing



Pointers and Index Registers

SP – Stack pointer

16 bit register

points to the topmost item of the stack

If the stack is empty the stack pointer will be (FFFE)H

It's offset address relative to stack segment

BP –Base pointer

16 bit register

used in accessing parameters passed by the stack

It's offset address relative to stack segment



SI – Source index register

16 bit register

used in the pointer addressing of data and
as a source in some string related operations

It's offset is relative to data segment

DI – Destination index register

16 bit register

used in the pointer addressing of data and
as a destination in string related operations

It's offset is relative to extra segment.



IP - Instruction Pointer

16 bit register

stores the address of the next instruction
to be executed

also acts as an offset for CS register.

Segment Registers

CS - Code Segment Register:

user cannot modify the content of these registers

Only the microprocessor's compiler can do this

DS - Data Segment Register:

The user can modify the content of the data segment.

SS - Stack Segment Registers:

used to store the information about the memory segment.

operations of the SS are mainly Push and Pop.

ES - Extra Segment Register:

By default, the control of the compiler remains in the DS where the user can add and modify the instructions

If there is less space in that segment, then ES is used

Also used for copying purpose.



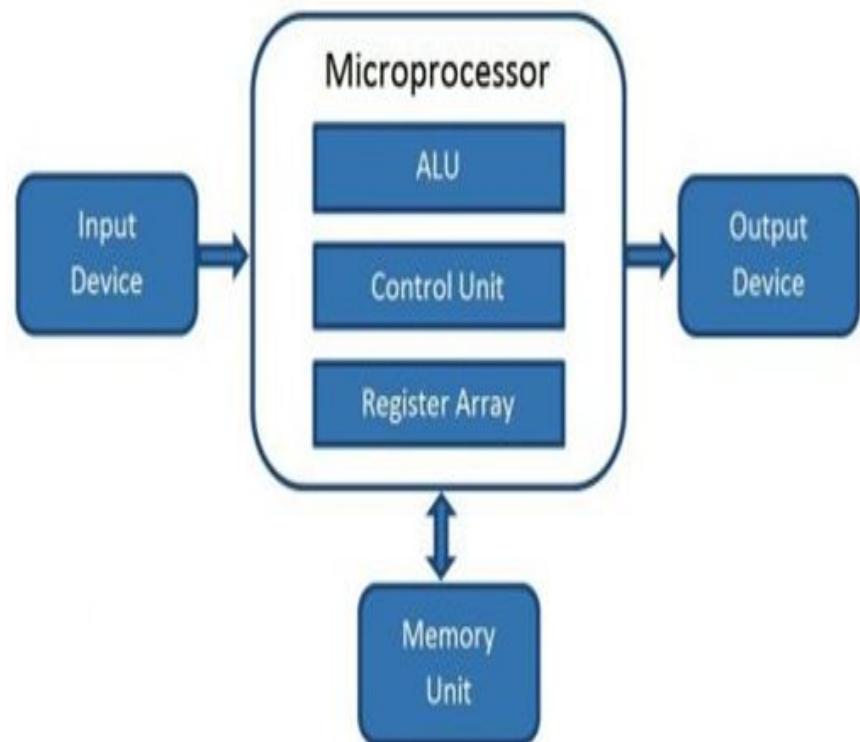
Flag or Status Register

- 16-bit register
- contains 9 flags
- remaining 7 bits are idle in this register
- These flags tell about the status of the processor after any arithmetic or logical operation
- IF the flag value is 1, the flag is set, and if it is 0, it is said to be reset.

Microcomputer

Block Diagram

- A digital computer with one microprocessor which acts as a CPU
- A complete computer on a small scale, designed for use by one person at a time
- called a personal computer (PC)
- a device based on a single-chip microprocessor
- includes laptops and desktops





Introduction to 8086 Assembly Language

Assembly Language Programming



Program Statements

- Program consist of statement, one per line.
- Each statement is either an **instruction**, which the assembler translate into machine code, or **assembler directive**, which instructs the assembler to perform some specific task, such as allocating memory space for a variable or creating a procedure.
- Both instructions and directives have up to four fields:
name operation operand(s) comment
- At least one blank or tab character must separate the fields



Program Statements

- An example of an instruction is

START:MOV CX,5 ; initialize counter

name operation operand(s) comment

- The name field consists of the label **START**:
 - The operation is **MOV**, the operands are **CX** and **5**
 - And the comment is **; initialize counter**

- An example of an assembler directive is

MAIN PROC

name operation operand(s) comment

- **MAIN** is the name, and the operation field contains **PROC**
 - This particular directive creates a procedure called **MAIN**

Program Statements



name operation operand(s) comment

- A Name field identifies a label, variable, or symbol.
It may contain any of the following character :
A,B.....Z ; a,b....z ; 0,1....9 ; ? ;
_ (underline) ; @ ; \$; . (period)

- Only the first 31 characters are recognized
- There is no distinction between uppercase and lower case letters.
- The first character may not be a digit
- If it is used, the period (.) may be used only as the first character.
- A programmer-chosen name may not be the same as an assembler reserved word.



Program Statements

name operation operand(s) comment



- Operation field is a predefined or reserved word
 - mnemonic - symbolic operation code.
 - The assembler translates a symbolic opcode into a machine language opcode.
 - Opcode symbols often describe the operation's function; for example, MOV, ADD, SUB
 - assembler directive - pseudo-operation code.
 - In an assembler directive, the operation field contains a pseudo-operation code (pseudo-op)
 - Pseudo-op are not translated into machine code; for example the PROC pseudo-op is used to create a procedure

Program Statements

name operation operand(s) comment



- An operand field specifies the data that are to be acted on by the operation.
- An instruction may have zero, one, or two operands. For example:
 - NOP No operands; does nothing
 - INC AX one operand; adds 1 to the contents of AX
 - ADD WORD1,2 two operands; adds 2 to the contents of memory word WORD1



Program Statements

name operation operand(s) comment



- The **comment** field is used by the programmer to say something about what the statement does.
- A semicolon marks the beginning of this field, and the assembler ignores anything typed after semicolon.
- Comments are optional, but because assembly language is low level, it is almost impossible to understand an assembly language program without comments.

Program Data and Storage

- Pseudo-ops to define data or reserve storage
 - DB - byte(s)
 - DW - word(s)
 - DD - doubleword(s)
 - DQ - quadword(s)
 - DT - tenbyte(s)
- These directives require one or more operands
 - define memory contents
 - specify amount of storage to reserve for run-time data

Defining Data

- Numeric data values
 - 100 - decimal
 - 100B - binary
 - 100H - hexadecimal
 - '100' - ASCII
 - "100" - ASCII
- Use the appropriate DEFINE directive (byte, word, etc.)
- A list of values may be used - the following creates 4 consecutive words
DW 40CH,10B,-13,0
- A ? represents an uninitialized storage location
DB 255,?, -128, 'X'

Naming Storage Locations

- Names can be associated with storage locations

ANum DB -4

DW 17

ONE

UNO DW 1

X DD ?

- These names are called variables

- ANum refers to a byte storage location, initialized to FCh
- The next word has no associated name
- ONE and UNO refer to the same word
- X is an uninitialized doubleword

- Multiple definitions can be abbreviated
Example:

message DB 'B'

DB 'y'

DB 'e'

DB 0DH

DB 0AH

can be written as

message DB 'B','y','e',0DH,0AH

- More compactly as

message DB 'Bye',0DH,0AH



Arrays

- Any consecutive storage locations of the same size can be called an array

X DW 40CH,10B,-13,0

Y DB 'This is an array'

Z DD -109236, FFFFFFFFH, -1, 100B

- Components of X are at X, X+2, X+4, X+6
- Components of Y are at Y, Y+1, ..., Y+15
- Components of Z are at Z, Z+4, Z+8, Z+12

DUP

- Allows a sequence of storage locations to be defined or reserved
- Only used as an operand of a define directive

DB 40 DUP (?) ; 40 words, uninitialized

DW 10h DUP (0) ; 16 words, initialized as 0

Table1 DW 10 DUP (?) ; 10 words, uninitialized

**message DB 3 DUP ('Baby') ; 12 bytes, initialized
; as BabyBabyBaby**

**Name1 DB 30 DUP ('?') ; 30 bytes, each
; initialized to ?**



DUP

- The DUP directive may also be nested

Example

```
stars DB 4 DUP(3 DUP ('*'),2 DUP ('?'),5 DUP ('!'))
```

Reserves 40-bytes space and initializes it as

```
***??!!!!**??!!!!**??!!!!**??!!!!
```

```
matrix DW 10 DUP (5 DUP (0))
```

defines a 10X5 matrix and initializes its elements to zero.

This declaration can also be done by

```
matrix DW 50 DUP (0)
```



Word Storage

- Word, doubleword, and quadword data are stored in reverse byte order (in memory)

Directive	Bytes in Storage
DW 256	00 01
DD 1234567H	67 45 23 01
DQ 10	0A 00 00 00 00 00 00 00
X DW 35DAh	DA 35

Low byte of X is at X, high byte of X is at X+1



Word Storage

Symbol Table

- * Assembler builds a symbol table so we can refer to the allocated storage space by the associated label

Example

			name	offset
value	DW	0	value	0
sum	DD	0	sum	2
marks	DW	10 DUP (?)	marks	6
message	DB	'The grade is:', 0	message	26
char1	DB	?	char1	40

Named Constants

- Symbolic names associated with storage locations represent addresses
- Named constants are symbols created to represent specific values determined by an expression
- Named constants can be numeric or string
- Some named constants can be redefined
- No storage is allocated for these values



Equal Sign Directive

- name = expression
 - expression must be numeric
 - these symbols may be redefined at any time

maxint = 7FFFh

count = 1

DW count

count = count * 2

DW count

EQU Directive

- name EQU expression
 - expression can be string or numeric
 - Use < and > to specify a string EQU
 - these symbols cannot be redefined later in the program

sample EQU 7Fh

aString EQU <1.234>

message EQU <This is a message>

Data Transfer Instructions

- **MOV target, source**
 - reg, reg
 - mem, reg
 - reg, mem
 - mem, immed
 - reg, immed
- Sizes of both operands must be the same
- reg can be any non-segment register except IP cannot be the target register
- MOV's between a segment register and memory or a 16-bit register are possible

Sample MOV Instructions

b db 4Fh

w dw 2048

mov bl,dh

mov ax,w

mov ch,b

mov al,255

mov w,-100

mov b,0

- When a variable is created with a define directive, it is assigned a default size attribute (byte, word, etc)
- You can assign a size attribute using LABEL

LoByte LABEL BYTE

aWord DW 97F2h



Program Segment Structure

- Data Segments
 - Storage for variables
 - Variable addresses are computed as offsets from start of this segment
- Code Segment
 - contains executable instructions
- Stack Segment
 - used to set aside storage for the stack
 - Stack addresses are computed as offsets into this segment
- Segment directives
 - .data**
 - .code**
 - .stack size**



Instruction types

Data transfer instructions

8086 instruction set

IN Input byte or word from port
LAHF Load AH from flags
LDS Load pointer using data segment
LEA Load effective address
LES Load pointer using extra segment
MOV Move to/from register/memory
OUT Output byte or word to port
POP Pop word off stack
POPF Pop flags off stack
PUSH Push word onto stack
PUSHF Push flags onto stack
SAHF Store AH into flags
XCHG Exchange byte or word
XLAT Translate byte

Additional 80286 instructions

INS Input string from port
OUTS Output string to port
POPA Pop all registers
PUSHA Push all registers

Additional 80386 instructions

LFS Load pointer using FS
LGS Load pointer using GS
LSS Load pointer using SS
MOVSX Move with sign extended
MOVZX Move with zero extended
POPAD Pop all double (32 bit) registers
POPD Pop double register
POPFD Pop double flag register
PUSHAD Push all double registers
PUSHD Push double register
PUSHFD Push double flag register

Additional 80486 instruction

BSWAP Byte swap

Additional Pentium instruction

MOV Move to/from control register

Arithmetic instructions

8086 instruction set

AAA ASCII adjust for addition

AAD ASCII adjust for division

AAM ASCII adjust for multiply

AAS ASCII adjust for subtraction

ADC Add byte or word plus carry

ADD Add byte or word

CBW Convert byte or word

CMPC Compare byte or word

CWD Convert word to double-word

DAA Decimal adjust for addition

DAS Decimal adjust for subtraction

DEC Decrement byte or word by one

DIV Divide byte or word

IDIV Integer divide byte or word

IMUL Integer multiply byte or word

INC Increment byte or word by one

MUL Multiply byte or word (unsigned)

NEG Negate byte or word

SBB Subtract byte or word and carry (borrow)

SUB Subtract byte or word

Additional 80386 instructions

CDQ Convert double-word to quad-word

CWDE Convert word to double-word

Additional 80486 instructions

CMPXCHG Compare and exchange

XADD Exchange and add

Additional Pentium instruction

CMPXCHG8B Compare and exchange 8 bytes

Bit manipulation instructions

8086 instruction set

AND Logical **AND** of byte or word

NOT Logical **NOT** of byte or word

OR Logical **OR** of byte or word

RCL Rotate left trough carry byte or word

RCR Rotate right trough carry byte or word

ROL Rotate left byte or word

ROR Rotate right byte or word

SAL Arithmetic shift left byte or word

SAR Arithmetic shift right byte or word

SHL Logical shift left byte or word

SHR Logical shift right byte or word

TEST Test byte or word

XOR Logical exclusive-**OR** of byte or word

Additional 80386 instructions

BSF Bit scan forward

BSR Bit scan reverse

BT Bit test

BTC Bit test and complement

BTR Bit test and reset

BTS Bit test and set

SETcc Set byte on condition

SHLD Shift left double precision

SHRD Shift right double precision



String instructions

8086 instruction set

CMPS	Compare byte or word string
LODS	Load byte or word string
MOVS	Move byte or word string
MOVSB(MOVSW)	Move byte string (word string)
REP	Repeat
REPE (REPZ)	Repeat while equal (zero)
REPNE (REPNZ)	Repeat while not equal (not zero)
SCAS	Scan byte or word string
STOS	Store byte or word string



Program Skeleton

```
.model small
.stack 100H
.data
    ;declarations
.code
main proc
    ;code
main endp
    ;other procs
end main
```

- Select a memory model
- Define the stack size
- Declare variables
- Write code
 - organize into procedures
- Mark the end of the source file
 - optionally, define the entry point

EXAMPLE : Adding two 8 bit numbers

```
DATA SEGMENT          ; Data Segment
N1
3n2 DB 12H
N2 DB 21H
RES DB ?
DATA ENDS
CODE SEGMENT          ; Code segment
ASSUME CS: CODE, DS: DATA
START: MOV AX, DATA
MOV DS, AX
MOV AL, N1
MOV BL, N2
ADD AL, BL
MOV RES, AL
INT 21H
CODE ENDS
END START
```



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

The ARM Architecture



ARM Ltd

- Founded in November 1990
- Designs the ARM range of RISC processor cores
- Licenses ARM core designs to semiconductor partners who fabricate and sell to their customers.
- Also develop technologies to assist with the design-in of the ARM architecture





ARM Partnership Model





ARM Powered Products





Data Sizes and Instruction Sets

- The ARM is a 32-bit architecture.
- When used in relation to the ARM:
 - Byte means 8 bits
 - Halfword means 16 bits (two bytes)
 - Word means 32 bits (four bytes)
- Most ARM's implement two instruction sets
 - 32-bit ARM Instruction Set
 - 16-bit Thumb Instruction Set
- Jazelle cores can also execute Java bytecode

Processor Modes

- The ARM has seven basic operating modes:
 - User : unprivileged mode under which most tasks run
 - FIQ : entered when a high priority (fast) interrupt is raised
 - IRQ : entered when a low priority (normal) interrupt is raised
 - Supervisor : entered on reset and when a Software Interrupt instruction is executed
 - Abort : used to handle memory access violations
 - Undef : used to handle undefined instructions
 - System : privileged mode using the same registers as user mode



The ARM Register Set

Current Visible Registers

Abort Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
spsr

User

r13 (sp)
r14 (lr)

Banked out Registers

r8
r9
r10
r11
r12

r13 (sp)
r14 (lr)

r13 (sp)
r14 (lr)

r13 (sp)
r14 (lr)

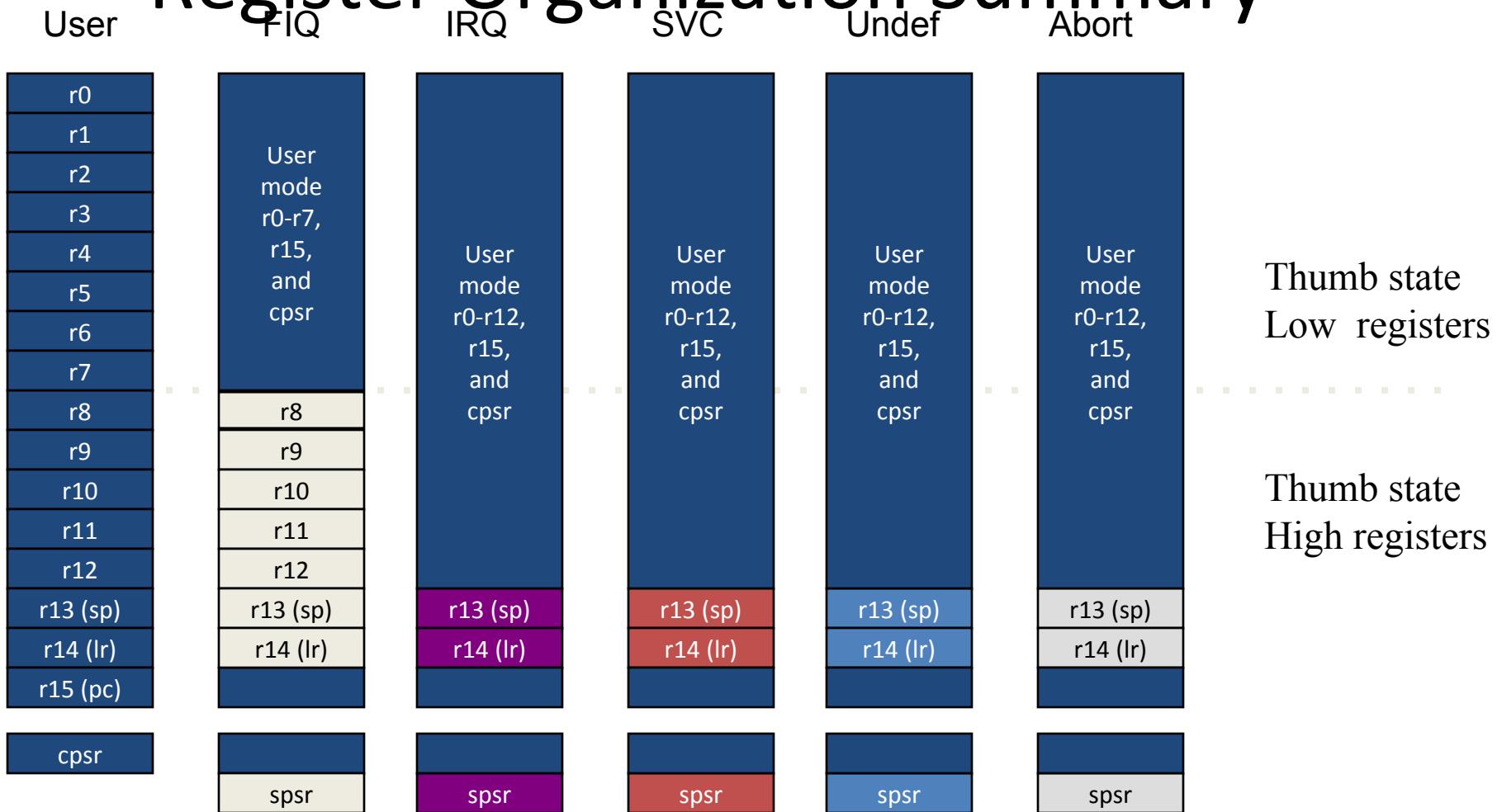
spsr

spsr

spsr

spsr

Register Organization Summary



Note: System mode uses the User mode register set



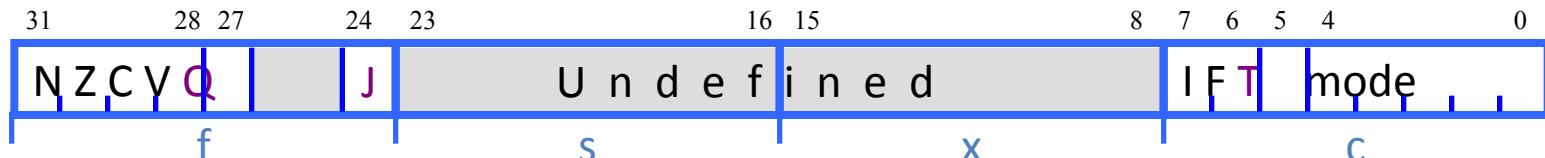
The Registers

- ARM has 37 registers all of which are 32-bits long.
 - 1 dedicated program counter
 - 1 dedicated current program status register
 - 5 dedicated saved program status registers
 - 30 general purpose registers
- The current processor mode governs which of several banks is accessible. Each mode can access
 - a particular set of r0-r12 registers
 - a particular r13 (the stack pointer, sp) and r14 (the link register, lr)
 - the program counter, r15 (pc)
 - the current program status register, cpsr

Privileged modes (except System) can also access

- a particular spsr (saved program status register)

Program Status Registers



- Condition code flags
 - N = Negative result from ALU
 - Z = Zero result from ALU
 - C = ALU operation Carried out
 - V = ALU operation oVerflowed
- Sticky Overflow flag - Q flag
 - Architecture 5TE/J only
 - Indicates if saturation has occurred
- J bit
 - Architecture 5TEJ only
 - J = 1: Processor in Jazelle state
- Interrupt Disable bits.
 - I = 1: Disables the IRQ.
 - F = 1: Disables the FIQ.
- T Bit
 - Architecture xT only
 - T = 0: Processor in ARM state
 - T = 1: Processor in Thumb state
- Mode bits
 - Specify the processor mode

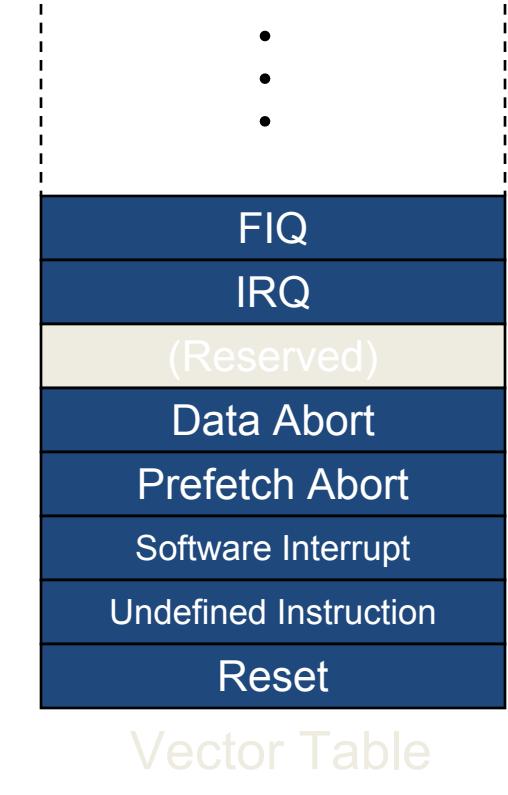


Program Counter (r15)

- When the processor is executing in ARM state:
 - All instructions are 32 bits wide
 - All instructions must be word aligned
 - Therefore the **pc** value is stored in bits [31:2] with bits [1:0] undefined (as instruction cannot be halfword or byte aligned).
- When the processor is executing in Thumb state:
 - All instructions are 16 bits wide
 - All instructions must be halfword aligned
 - Therefore the **pc** value is stored in bits [31:1] with bit [0] undefined (as instruction cannot be byte aligned).
- When the processor is executing in Jazelle state:
 - All instructions are 8 bits wide
 - Processor performs a word access to read 4 instructions at once

Exception Handling

- When an exception occurs, the ARM:
 - Copies CPSR into SPSR_<mode>
 - Sets appropriate CPSR bits
 - Change to ARM state
 - Change to exception mode
 - Disable interrupts (if appropriate)
 - Stores the return address in LR_<mode>
 - Sets PC to vector address
 - To return, exception handler needs to:
 - Restore CPSR from SPSR_<mode>
 - Restore PC from LR_<mode>
- This can only be done in ARM state.



Vector table can be at 0xFFFF0000 on ARM720T and on ARM9/10 family devices



Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.
 - This improves code density *and* performance by reducing the number of forward branch instructions.

```
CMP    r3,#0
BEQ    skip
ADD    r0,r1,r2
skip
```

```
CMP    r3,#0
ADDNE r0,r1,r2
```

- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using “S”. CMP does not need “S”.

```
loop
...
SUBS r1,r1,#1
BNE loop
```

decrement r1 and set flags

if Z flag clear then branch



Condition Codes

- The possible condition codes are listed below:
 - Note AL is the default and does not need to be specified

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N!=V
AL	Always	



Examples of conditional execution

- Use a sequence of several conditional instructions

```
if (a==0) func(1);
    CMP r0,#0
    MOVEQ r0,#1
    BLEQ func
```

- Set the flags, then use various condition codes

```
if (a==0) x=0;
if (a>0) x=1;
    CMP r0,#0
    MOVEQ r1,#0
    MOVG T r1,#1
```

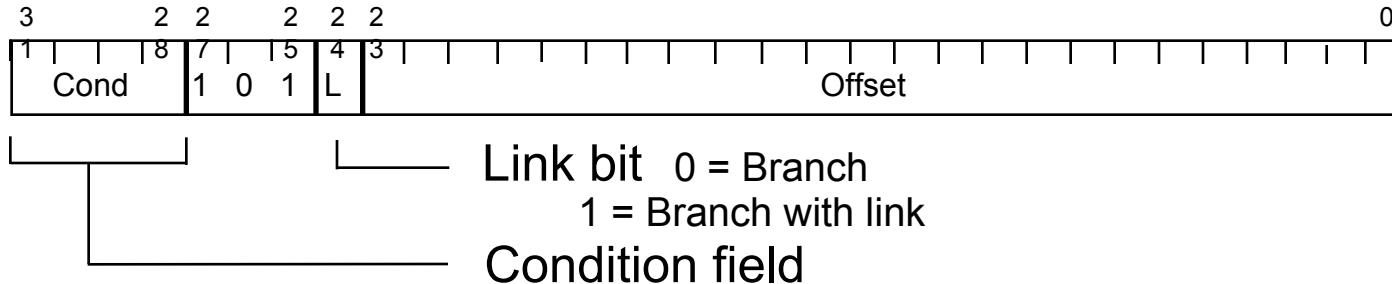
- Use conditional compare instructions

```
if (a==4 || a==10) x=0;
    CMP r0,#4
    CMPNE r0,#10
    MOVEQ r1,#0
```



Branch instructions

- Branch : B{<cond>} label
- Branch with Link : BL{<cond>} subroutine_label



- The processor core shifts the offset field left by 2 positions, sign-extends it and adds it to the PC
 - ± 32 Mbyte range
 - How to perform longer branches?



Data processing Instructions

- Consist of :
 - Arithmetic: ADD ADC SUB SBC RSB RSC
 - Logical: AND ORR EOR BIC
 - Comparisons: CMP CMN TST TEQ
 - Data movement: MOV MVN
- These instructions only work on registers, NOT memory.
- Second operand is sent to the ALU via barrel shifter.



EXAMPLE

- **Arithmetic Operations**

ADD r0,r1,r2 ;r0:=r1+r2

ADC r0,r1,r2 ;r0:=r1+r2+C

SUB r0,r1,r2 ;r0:=r1-r2

SBC r0,r1,r2 ;r0:=r1-r2+C-1

RSB r0,r1,r2 ;r0:=r2-r1, reverse subtraction

RSC r0,r1,r2 ;r0:=r2-r1+C-1

- **Syntax:**

- <Operation>{<cond>} {S} Rd, Rn, Operand2

By default data processing operations *do no affect the condition flags*



Barrel Shifter & Memory Instructions



The Barrel Shifter

LSL : Logical Left Shift



Multiplication by a power of 2

ASR: Arithmetic Right Shift



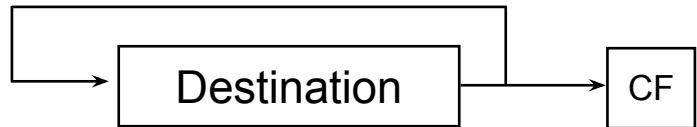
Division by a power of 2,
preserving the sign bit

LSR : Logical Shift Right



Division by a power of 2

ROR: Rotate Right



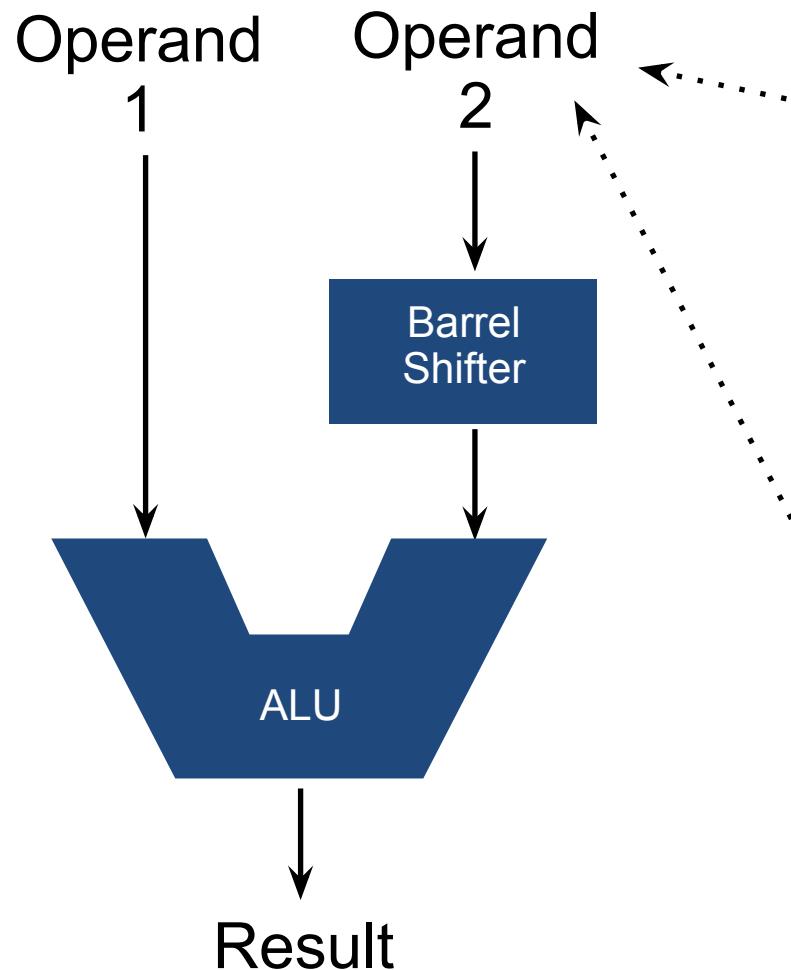
Bit rotate with wrap around
from LSB to MSB

RRX: Rotate Right Extended



Single bit rotate with wrap around
from CF to MSB

Using the Barrel Shifter: The Second Operand



Register, optionally with shift operation

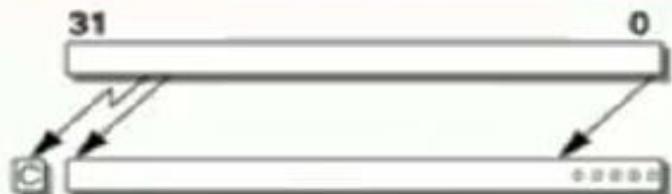
- Shift value can be either be:
 - 5 bit unsigned integer
 - Specified in bottom byte of another register.
- Used for multiplication by constant

Immediate value

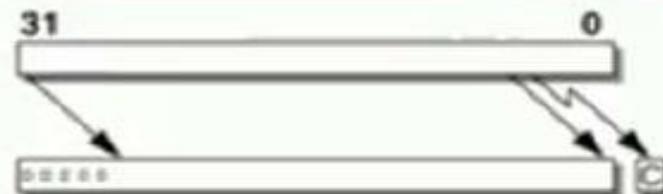
- 8 bit number, with a range of 0-255.
 - Rotated right through even number of positions
- Allows increased range of 32-bit constants to be loaded directly into registers



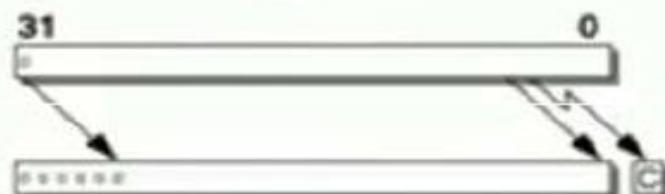
EXAMPLE



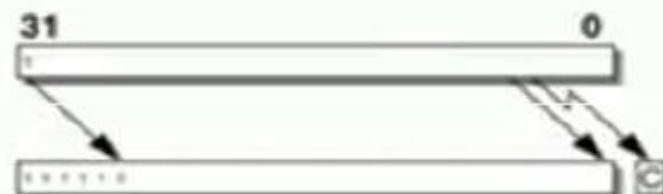
LSL # 5



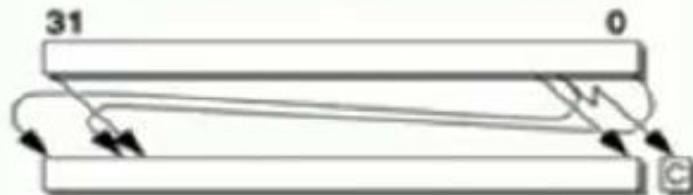
LSR # 5



ASR # 5 – positive operand



ASR # 5 – negative operand



ROR # 5



RRX

Load / Store Instructions

- The ARM is a Load / Store Architecture:
 - Does not support memory to memory data processing operations
 - Must move data values into registers before using them
- This might sound inefficient, but in practice isn't:
 - Load data values from memory into registers
 - Process data in registers using a number of data processing instructions which are not slowed down by memory access
 - Store results from registers out to memory
- The ARM has three sets of instructions which interact with main memory. These are:
 - Single register data transfer (LDR / STR)
 - Block data transfer (LDM/STM)
 - Single Data Swap (SWP)

Single register data transfer

LDR: **LoaD** words from memory into a **Register**

STR: **STore** words from a **Register** into memory

Basic syntax:

<LDR/STR>{cond}{type} Rd, [Rn, addressing]

where Rd = destination (for LDR) & source (for STR)

 Rn = Base address register

 cond = condition flag

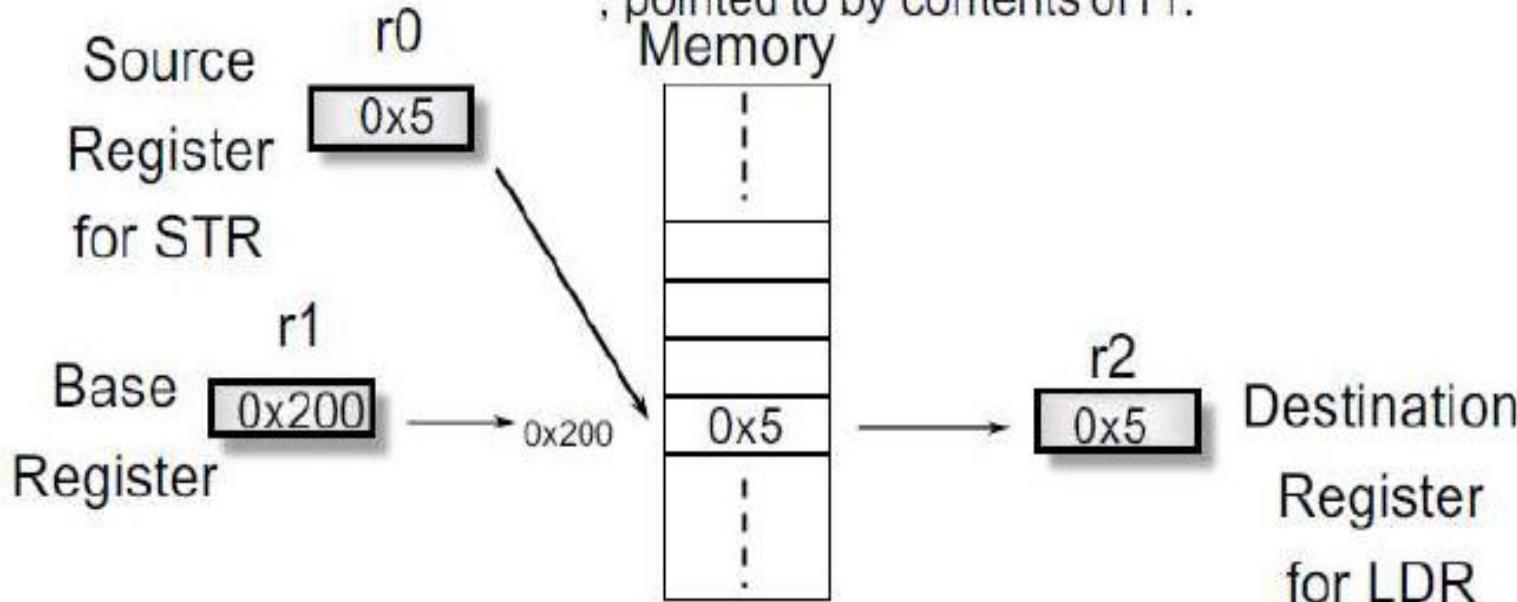
 type = byte, halfword, word(default), signed
 & unsigned

Base Register

- The memory location to be accessed is held in a base register

– STR r0, [r1] ; Store contents of r0 to location pointed to ; by contents of r1.

– LDR r2, [r1] ; Load r2 with contents of memory location ; pointed to by contents of r1.



Multiple-Register Load-Store(LDM/STM)

The multiple-register load-store instructions support the transfer of a block of data in one instruction, through the use of **Multiple registers**.

Basic instructions: LDM and STM

Usually used with a suffix: IA, IB, DA, DB

IA: increment after

IB: increment before

DA: decrement after

DB: decrement before



(LDM/STM) OPERATIONS

- Syntax:

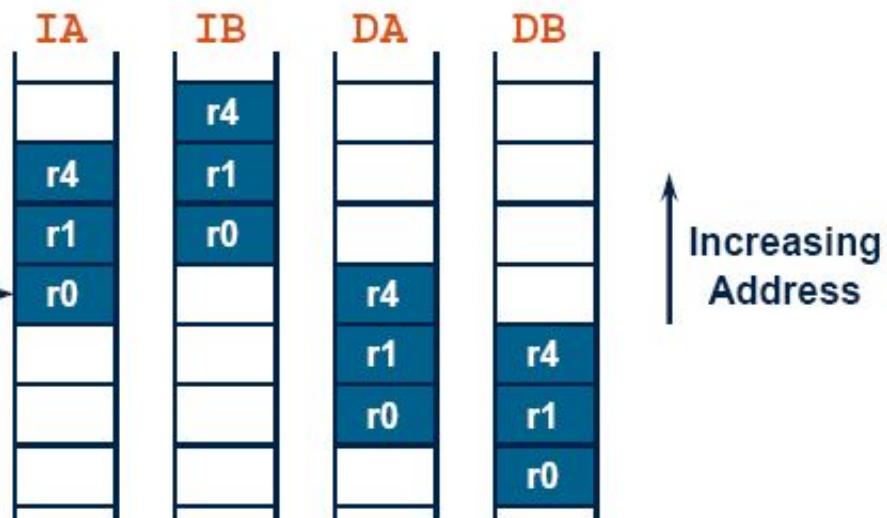
<**LDM | STM**>{<cond>}<addressing_mode> Rb{!}, <register list>

- 4 addressing modes:

LDMIA / STMIA	increment after
LDMIB / STMIB	increment before
LDMDA / STMDA	decrement after
LDMDB / STMDB	decrement before

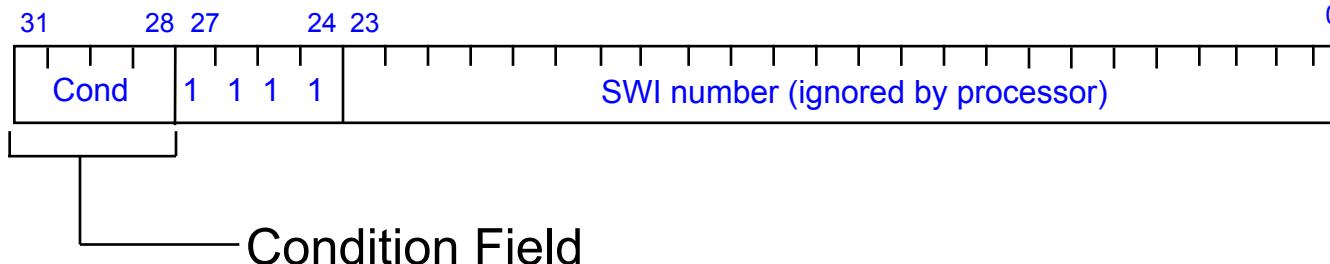
LDMxx r10, {r0,r1,r4}
STMxx r10, {r0,r1,r4}

Base Register (Rb) **r10**





Software Interrupt (SWI)

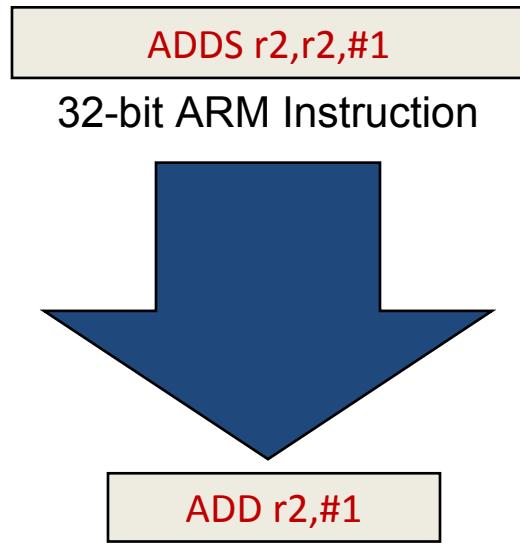


- Causes an exception trap to the SWI hardware vector
- The SWI handler can examine the SWI number to decide what operation has been requested.
- By using the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.
- Syntax:

SWI {<cond>} <SWI number>

Thumb

- Thumb is a 16-bit instruction set
 - Optimized for code density from C code (~65% of ARM code size)
 - Improved performance from narrow memory
 - Subset of the functionality of the ARM instruction set
- Core has additional execution state - Thumb
 - Switch between ARM and Thumb using **BX** instruction

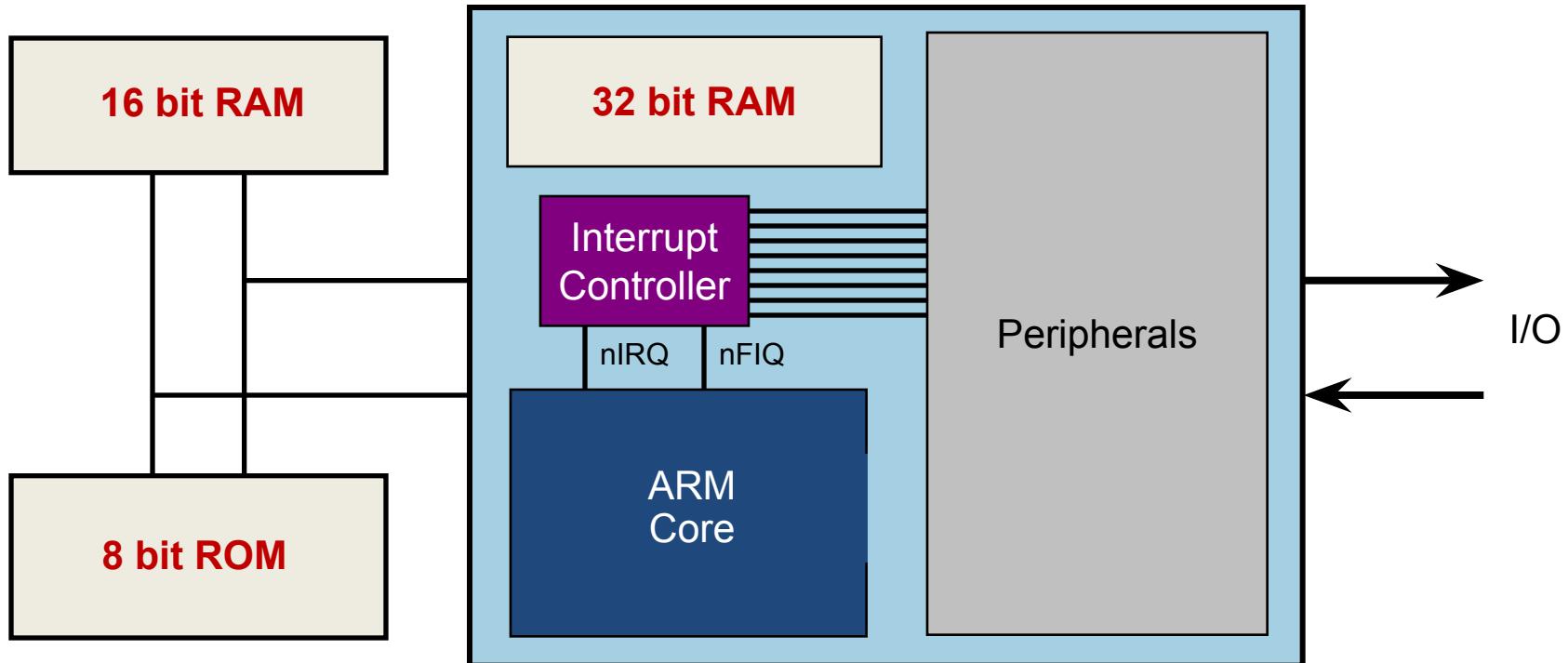


For most instructions generated by compiler:

- Conditional execution is not used
- Source and destination registers identical
- Only Low registers used
- Constants are of limited size
- Inline barrel shifter not used



Example ARM-based System





SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

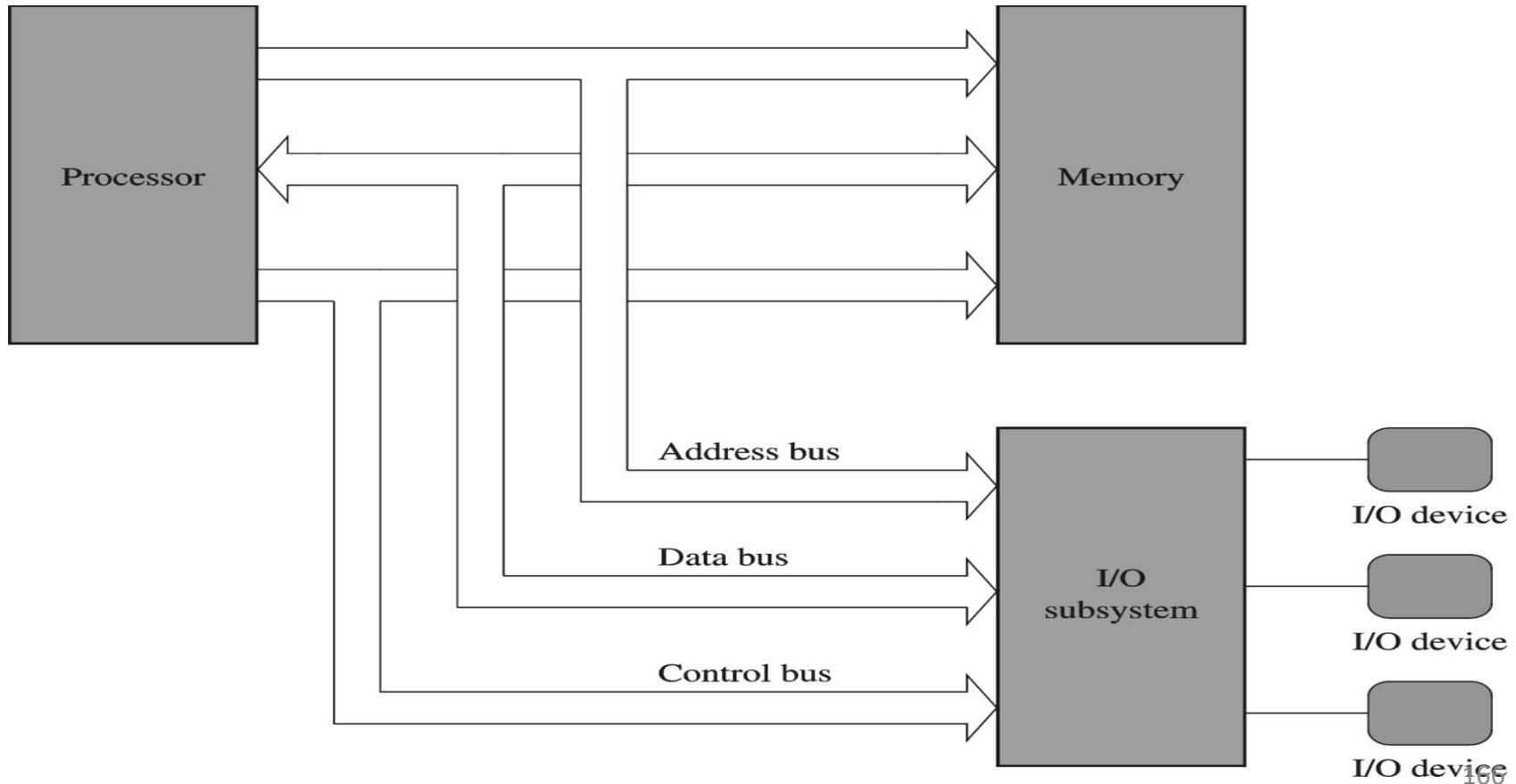
Basics of Input - Output Operations

Input - Output Interface

- Input Output Interface provides a method for transferring information between internal storage and external I/O devices.
- Peripherals connected to a computer need special communication links for interfacing them with the central processing unit.



COMMUNICATION BETWEEN CPU, MEMORY AND I/O DEVICES





BUS

- A bus is a bunch of wires through which data or address or control signals flow.
- The microprocessor communicates with the memory and the Input/Output devices via the three buses, viz., **data bus, address bus and control bus**.
- Data flow through the DB, while address comes out of the AB and CB controls the activities of the microprocessor system at any instant of time.

Input - Output Interface

The purpose of communication link is to resolve the differences that exist between the central computer and each peripheral.



The Major Differences are:-

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation of the CPU and memory, will differ. Therefore, a **conversion of signal values** may be needed.
2. The data transfer rate of peripherals is usually slower than the transfer rate of CPU and consequently, a **synchronization** mechanism may be needed.

3. Data codes and formats in the peripherals differ from the word format in the CPU and memory.
4. The **operating modes of peripherals** are different from each other and must be controlled so as not to disturb the operation of other peripherals connected to the CPU.



Input - Output Interface

- **To Resolve these differences**, computer systems include special hardware components between the CPU and Peripherals to supervise and synchronize all input and output transfers.
- These components are called **Interface Units** because they interface between the processor bus and the peripheral devices.

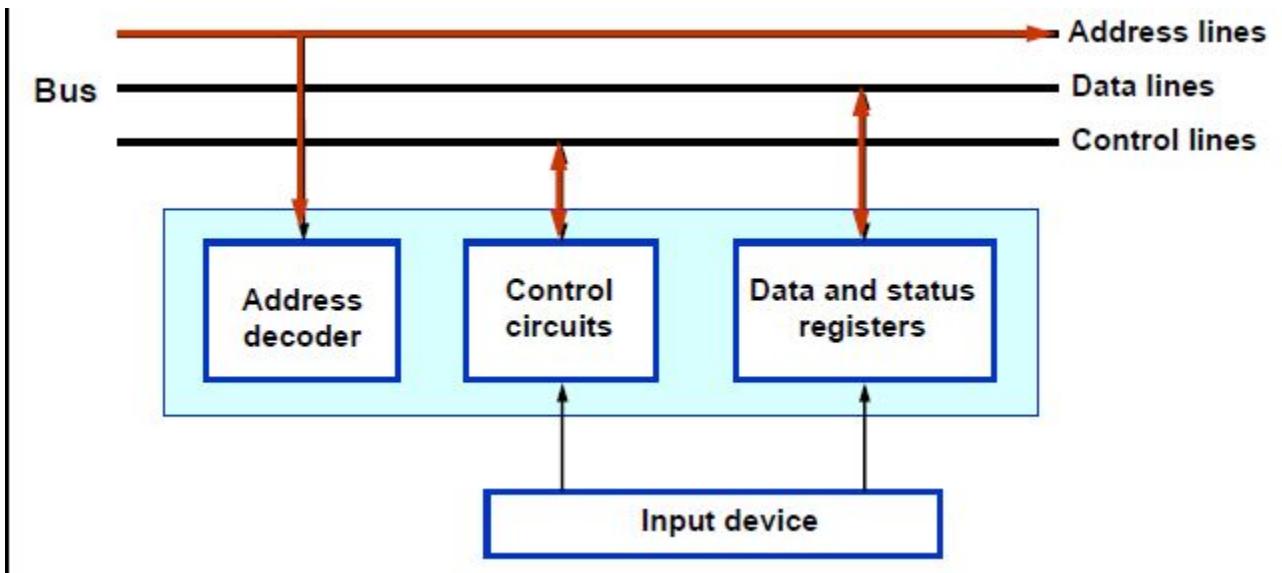


I/O BUS and Interface Module

- It defines the typical link between the processor and several peripherals.
- The I/O Bus consists of **data lines, address lines and control lines**.
- The I/O bus from the processor is attached to all peripherals interface.
- To communicate with a particular device, the processor places a device address on address lines.



Interface Module



I/O BUS and Interface Module

- Each Interface **decodes** the address and control received from the I/O bus, interprets them for peripherals and provides signals for the **peripheral controller**.
- It is also synchronizes the data flow and supervises the transfer between peripheral and processor.
- Each peripheral has its own controller. **For example**, the printer controller controls the paper motion, the print timing.

The control lines are referred as an I/O command. The commands are as following:

Control command- A control command is issued to activate the peripheral and to inform it what to do.

Status command- A status command is used to test various status conditions in the interface and the peripheral.



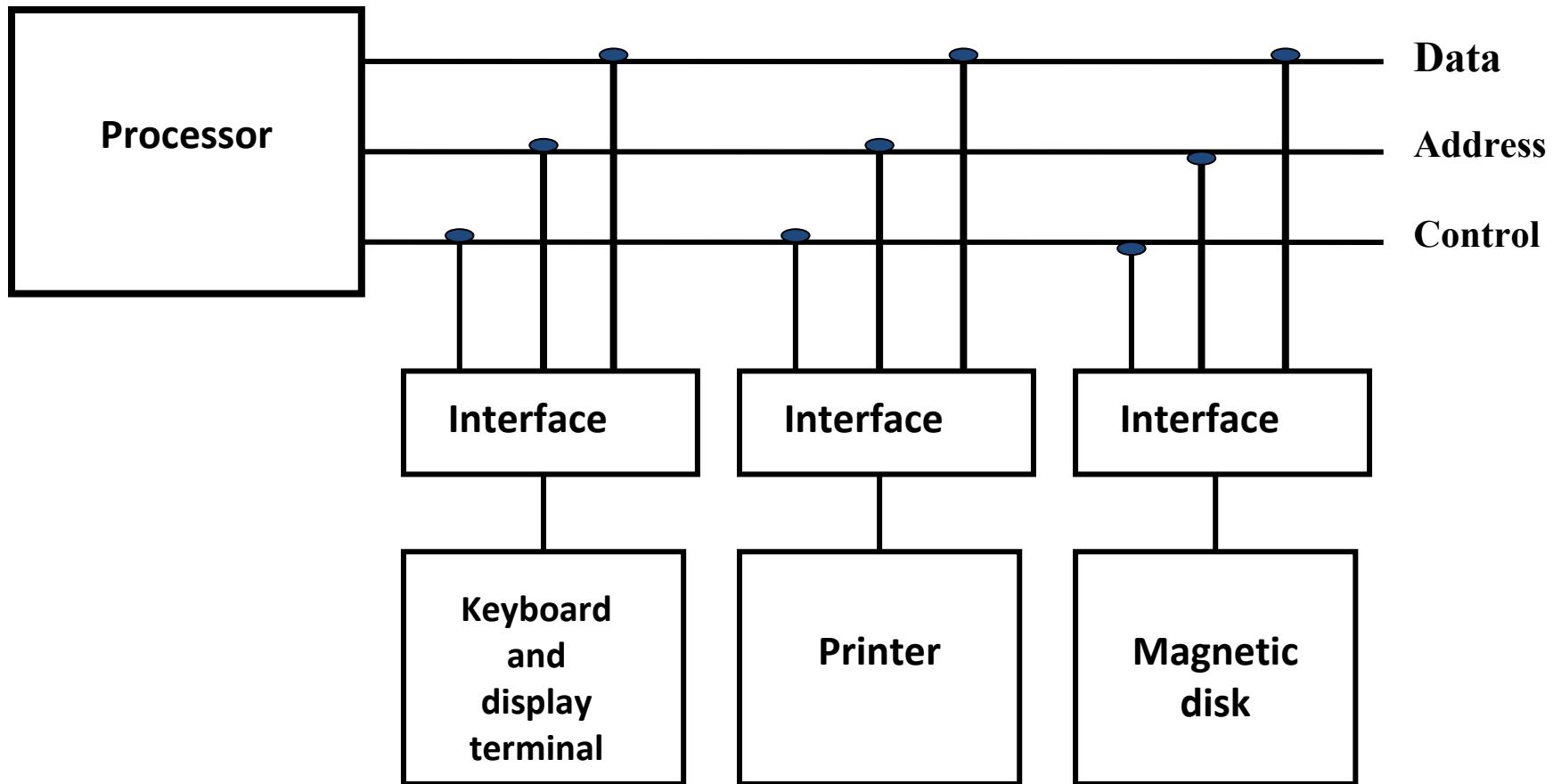
I/O BUS and Interface Module

Output data command- A data output command causes the interface to respond by transferring **data from the bus into one of its registers.**

Input data command- The data input command is the opposite of the data output. In this case the interface receives one item of **data from the peripheral** and places it in its buffer register.



I/O BUS and Interface Module



Connection of I/O bus to input-output devices

I/O Versus Memory Bus

There are 3 ways that computer buses can be used to communicate with memory and I/O:

- i. Use **two Separate buses**, one for memory and other for I/O.
- ii. Use **one common bus** for both memory and I/O but separate control lines for each.
- iii. Use **one common bus for memory and I/O with common control lines.**

Unit 2

18CSC203J-Computer Organization and Architecture





Course Outcome

- CLR-2:*Analyze the functions of arithmetic units like adders, multipliers etc.*
- *CLR-6:Simulate simple fundamental units like half adder, full adder etc.*
- CLO-2:*Apply Boolean algebra as related in designing computer logic through simple combinational and sequential logic circuits*



Table of Contents

- Addition and subtraction of signed numbers
- Design of fast adders – Ripple Carry Adder and Carry Look ahead Adder
- Multiplication of positive numbers
- Signed operand multiplication
- Fast multiplication - Bit pair recoding of Multipliers
- Carry Save Addition of Summands
- Integer division – Restoring Division and Non Restoring Division
- Floating point numbers and operations



Integer Representation

- Only have 0 & 1 to represent everything
- Positive numbers stored in binary
- e.g. $41=00101001$
- No minus sign
- No period
- Sign-Magnitude
- One's Complement
- Two's compliment



- In integer
- MSB-Most significant Bit
- LSB-Least Significant Bit

Most Significant Bit (MSB)

This bit has the highest value (greatest weight) and is located at the far left of the bit string.

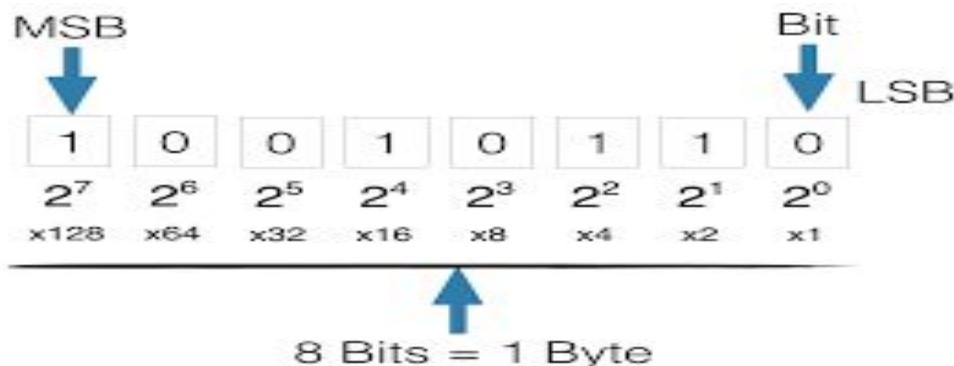
Least Significant Bit (LSB)

This bit has the lowest value (bit position zero) and is located at the far right of the bit string.

Bit Position	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Data	1	0	1	1	0	1	1	1	1	1	1	0	1	0	0	1	
MSB	↑															LSB	↑
sign bit	1	1	0	1												magnitude	1

4 bit signed binary representation

If MSB is 0 then the integer is +ve number
MSB is 1 then the integer is –ve number



The value of bits in signed and unsigned binary numbers

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-------	-------	-------	-------	-------	-------	-------	-------

Unsigned	$27 = 128$	$26 = 64$	$25 = 32$	$24 = 16$	$23 = 8$	$22 = 4$	$21 = 2$	$20 = 1$
----------	------------	-----------	-----------	-----------	----------	----------	----------	----------

Signed	$-(27) = -128$	$26 = 64$	$25 = 32$	$24 = 16$	$23 = 8$	$22 = 4$	$21 = 2$	$20 = 1$
--------	----------------	-----------	-----------	-----------	----------	----------	----------	----------



Integer Representation (cont'd)

- Sign Magnitude: One's Complement Two's Complement

$$000 = +0 \quad 000 = +0 \quad 000 = +0$$

$$001 = +1 \quad 001 = +1 \quad 001 = +1$$

$$010 = +2 \quad 010 = +2 \quad 010 = +2$$

$$011 = +3 \quad 011 = +3 \quad 011 = +3$$

$$100 = -0 \quad 100 = -3 \quad 100 = -4$$

$$101 = -1 \quad 101 = -2 \quad 101 = -3$$

$$110 = -2 \quad 110 = -1 \quad 110 = -2$$

$$111 = -3 \quad 111 = -0 \quad 111 = -1$$

- Issues: balance, number of zeros, ease of operations

- Which one is best? Why?

SUMMARY OF THE TABLE



- SIGN & MAGNITUDE SYSTEM: Negative value is obtained by changing the sign bit (MSB)
- SIGNED 1'S COMPLEMENT: Negative number is obtained by complementing each bit of the corresponding positive number i.e $(2^n - 1) - N$
- SIGNED 2'S COMPLEMENT: Negative number is obtained by taking 2's complement of positive number
 - ❖ $2^n - N$
 - ❖ Range: - (2^{n-1}) to + $(2^{n-1} - 1)$



Range of Values

$$n = 8$$

n 8 bit number

is Complement $(2^n - 1) - N$

2's Complement $-(2^{n-1}) + (2^{n-1} - 1)$

Example

$$n = 8 \\ -(2^{8-1}) = -2^7 = -128 \text{ minimum}$$

$$+(2^{8-1}) - 1 = 2^7 - 1 = +127 \text{ maximum}$$

* BINARY ARITHMETIC

* for all digital Computer and digital System

* Binary addition, Subtraction, Multiplication

and division.

Binary Addition

Truth table

A	B	CARRY	SUM
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Example :

$$\begin{array}{r}
 & 8 & 4 & 2 & 1 \\
 & 0 & 0 & 1 & 0 \\
 & 0 & 0 & 1 & 1 \\
 \hline
 & 0 & 1 & 0 & 1
 \end{array} = 5$$

$$\begin{array}{r}
 & 1 & 6 & 8 & 4 & 2 & 1 \\
 & 0 & 1 & 0 & 0 & 1 \\
 \hline
 & 1 & 0 & 1 & 1 & 0 \\
 \hline
 & 1 & 1 & 1 & 1 & 1
 \end{array} = 31$$

Binary Subtraction



- Subtraction and Borrow, these two words will be used very frequently for the binary subtraction. There four rules of the binary subtraction.



A	B	BORROW	DIFFERENCE
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

$$\begin{array}{r} 0011010 \\ - 001100 \\ \hline 00001110 \end{array}$$

1 1 borrow

$$\begin{array}{r} 0011010 \\ - 0001100 \\ \hline 0001110 \end{array} = 26_{10}$$
$$= 12_{10}$$
$$= 14_{10}$$



Binary Subtraction

A	B	Borrow	Difference
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

By Using is Δ 2's Complement

Ex:

$$+5: 0101 \xrightarrow{1's} 1010 \\ +2 = 0010 \xrightarrow{1's} 1101$$

$$\begin{array}{r} -5 \\ -2 \\ \hline -7 \end{array}$$

$$\begin{array}{r} 1010 \\ +1 \\ \hline 1011 \end{array} \rightarrow -5$$

$$\begin{array}{r} 1101 \\ +1 \\ \hline 1110 \end{array} \rightarrow -2$$

$$\begin{array}{r} 1011 \\ 1110 \\ \hline 1001 \end{array}$$

Carry
NSB is -ve take 2's

$$\begin{array}{r} 1001 \rightarrow 0110 \\ - (7) \quad \hline 0111 \end{array}$$

$$2-4 \Rightarrow 2 + (-4) = -2 + 4 \Rightarrow 0100 \\ = \begin{array}{r} 1011 \\ - 0100 \\ \hline 0100 \end{array}$$

$$\begin{array}{r} 0010 \\ + 1100 \\ \hline 1110 \end{array}$$

$$\begin{array}{r} 0001 \\ - 0010 \\ \hline 0010 = 2 \end{array}$$

$\therefore -2$

2-4

$$\begin{array}{r} 0010 \\ 0100 \\ \hline 1110 \end{array}$$

$$\begin{array}{r} - 0010 \\ \hline 1111 \end{array}$$

(-2)

5-6

$$\begin{array}{r} 0010 \\ 0110 \\ \hline 1111 \end{array}$$

$$\begin{array}{r} - 0000 \\ + 1 \\ \hline 0001 = 1 \end{array}$$

$\therefore -1$



Binary Multiplication

- Binary multiplication is similar to decimal multiplication. It is simpler than decimal multiplication because only 0s and 1s are involved. There are four rules of the binary multiplication.



Case	A	x	B	Multiplication
1	0	x	0	0
2	0	x	1	0
3	1	x	0	0
4	1	x	1	1

Example:

$$0011010 \times 001100 = 100111000$$

$$\begin{array}{r} 0011010 = 26_{10} \\ \times 0001100 = 12_{10} \\ \hline & 0000000 \\ & 0000000 \\ & 0011010 \\ & 0011010 \\ \hline & 0100111000 = 312_{10} \end{array}$$



Binary Division

- Binary division is similar to decimal division. It is called as the long division procedure

$$101010 / 000110 = 000111$$

$$\begin{array}{r} 1\ 1\ 1 \\ \hline 000110) 101010 \\ - 110 \\ \hline 101 \\ - 110 \\ \hline 101 \\ - 110 \\ \hline 0 \end{array} = 7_{10}$$
$$= 42_{10}$$
$$= 6_{10}$$



Half Adder

- Half adder is a combinational logic circuit with two input and two output. The half adder circuit is designed to add two single bit binary number A and B. It is the basic building block for addition of two single bit numbers. This circuit has two outputs carry and sum.





Truth Table and Circuit Diagram

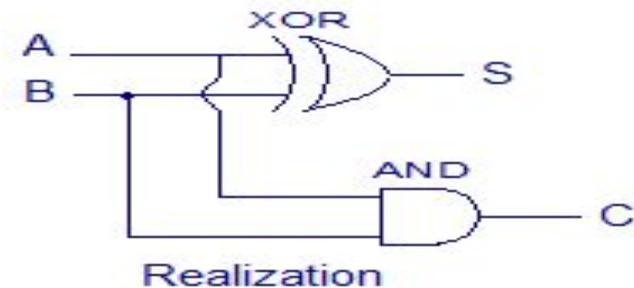
Inputs		Outputs	
A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Truth table

$$\text{SUM } S = A \cdot \bar{B} + \bar{A} \cdot B$$

$$\text{CARRY } C = A \cdot B$$

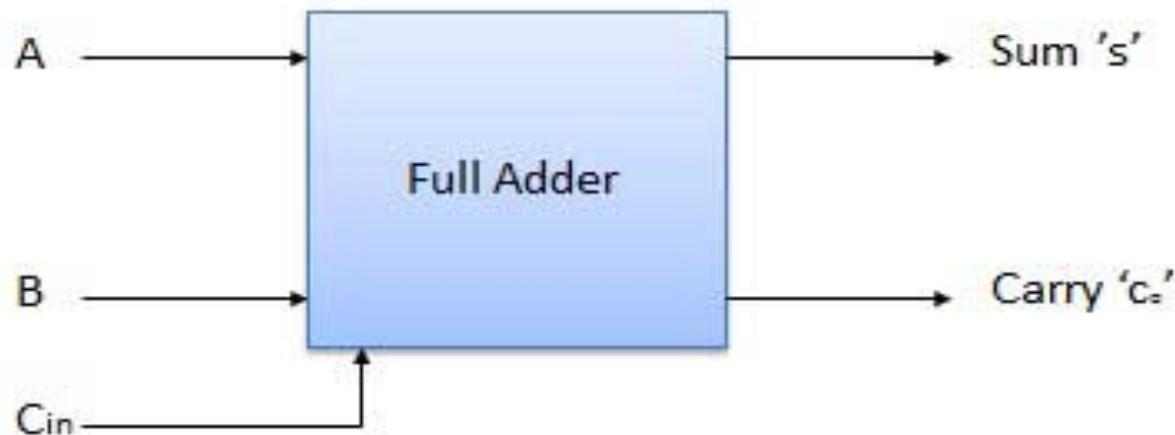
Boolean Expression





Full Adder

- Full adder is developed to overcome the drawback of Half Adder circuit. It can add two one-bit numbers A and B, and carry c. The full adder is a three input and two output combinational circuit.



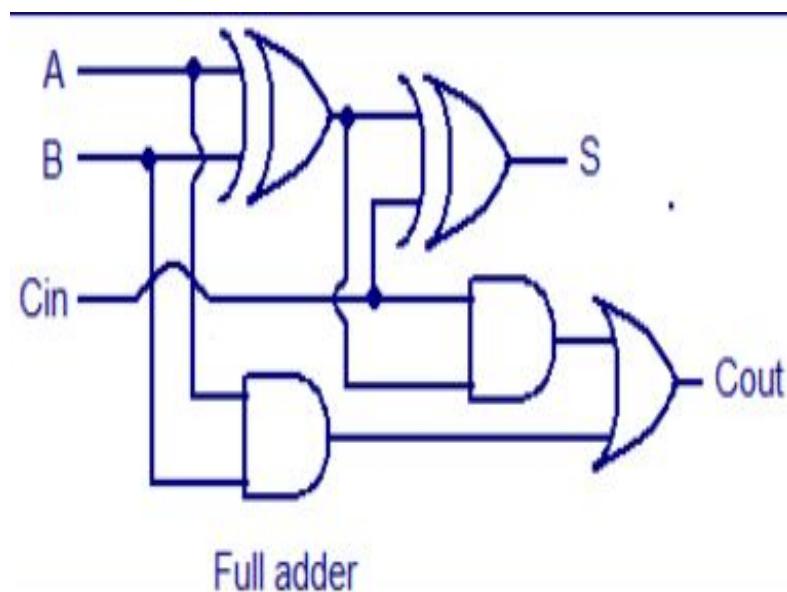


Truth Table and Circuit Diagram

$$\begin{aligned}
 S &= A\overline{BC} + \overline{A}\overline{B}C + ABC + \overline{ABC} \\
 &= C(AB + \overline{A}\overline{B}) + \overline{C}(\overline{AB} + A\overline{B}) \\
 &= C(\overline{AB} + A\overline{B}) + \overline{C}(\overline{AB} + A\overline{B}) \\
 &= C(\overline{A \oplus B}) + \overline{C}(A \oplus B) = A \oplus B \oplus C
 \end{aligned}$$

$$\begin{aligned}
 C_{out} &= \overline{ABC} + A\overline{B}C + ABC + \overline{ABC} \\
 &= (\overline{AB} + A\overline{B})C + AB(\overline{C} + C) \\
 &= (A \oplus B).C + AB.
 \end{aligned}$$

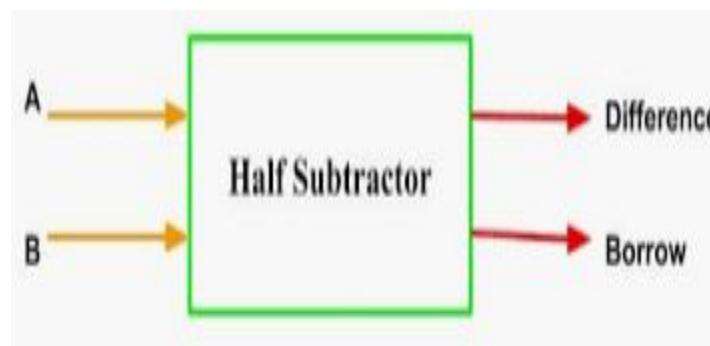
Inputs			Output	
A	B	Cin	S	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1





Half Subtractor

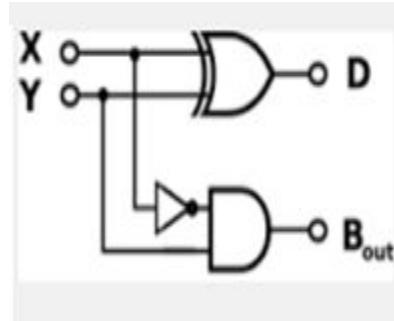
- Half subtractor is a combination circuit with two inputs and two outputs (difference and borrow). It produces the difference between the two binary bits at the input and also produces a output (Borrow) to indicate if a 1 has been borrowed. In the subtraction ($A-B$), A is called as Minuend bit and B is called as Subtrahend bit.





Truth Table and Circuit Diagram

A	B	BORROW	DIFFERENCE
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0



From the truth table, Boolean Expression can be derived as:

$$D = A'B + AB' = A \oplus B$$

$$B_o = A'B$$



Half Subtractors

Half Subtractor

A	B	B'	D
0	0	0	0
0	1	0	1
1	0	0	1
1	1	0	0

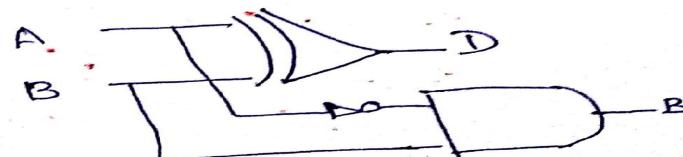
$$\begin{aligned}D &= A'B + AB' \\&= A \oplus B\end{aligned}$$

$$B' = A'B$$

Eg.

$$\begin{array}{r} 8 - 2 \\ - 2 \\ \hline 6 \end{array}$$
$$\begin{array}{r} 1000 \\ - 0010 \\ \hline 0110 \end{array}$$
$$= 6$$

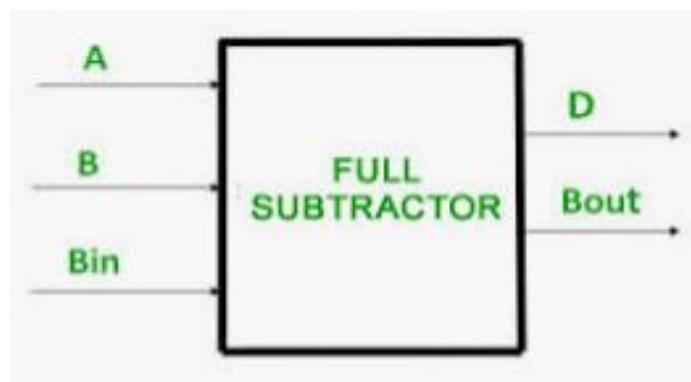
$$\begin{array}{r} 2 - 8 \\ - 0010 \\ \hline 1010 \end{array}$$
$$- 0110$$
$$= - 6$$





Full Subtractor

- The disadvantage of a half subtractor is overcome by full subtractor. The full subtractor is a combinational circuit with three inputs A, B, Bin and two output D and Bo. A is the minuend, B is subtrahend, Bin is the borrow produced by the previous stage, D is the difference output and Bo is the borrow output.





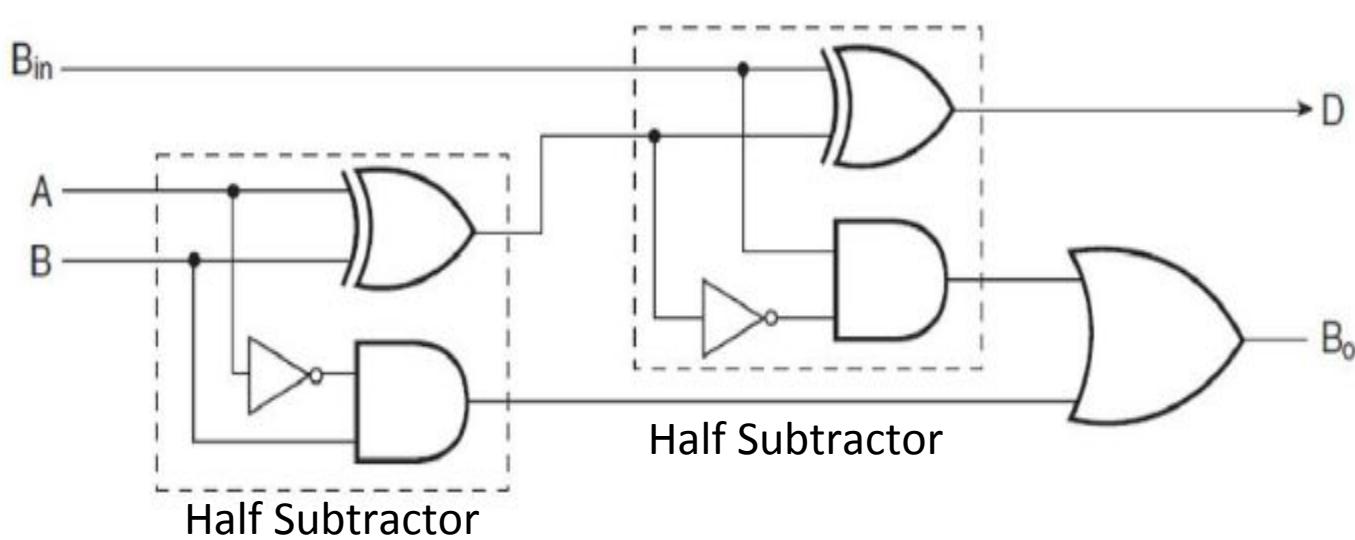
INPUT			OUTPUT	
A	B	Bin	D	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

D_i

$$\begin{aligned}
 & A'B'B_{in} + A'BB_{in}' + AB'B_{in}' + ABB_{in} \\
 & = A'(B'B_{in} + BB_{in}') + A(B'B_{in}' + BB_{in}) \\
 & = A'(B \oplus B_{in}) + A(B \oplus B_{in})' \\
 & = A \oplus B \oplus B_{in}
 \end{aligned}$$

B_0

$$\begin{aligned}
 & A'B'B_{in} + A'BB_{in}' + A'BB_{in} + ABB_{in} \\
 & = A'B'B_{in} + ABB_{in} + A'B(B_{in} + B_{in}') \\
 & = B_{in}(A \oplus B)' + A'B
 \end{aligned}$$



Ripple Carry Adder



Typical Ripple Carry Addition is a Serial Process:

- Addition starts by adding LSBs of the augend and addend.
- Then next position bits of augend and addend are added along with the carry (if any) from the preceding bit.
- This process is repeated until the addition of MSBs is completed.
- Speed of a ripple adder is limited due to carry propagation or carry ripple.
- Sum of MSB depends on the carry generated by LSB.



Ripple Carry Adder

Example: 4-bit Carry Ripple Adder

- Assume to add two operands A and B where

$$A = A_3 \ A_2 \ A_1 \ A_0$$

$$B = B_3 \ B_2 \ B_1 \ B_0$$

$$A = \begin{array}{cccc} 1 & 0 & 1 & 1 \end{array} +$$

$$B = \begin{array}{cccc} 1 & 1 & 0 & 1 \end{array}$$

$$A+B = \begin{array}{cccc} 1 & 1 & 0 & 0 \end{array}$$

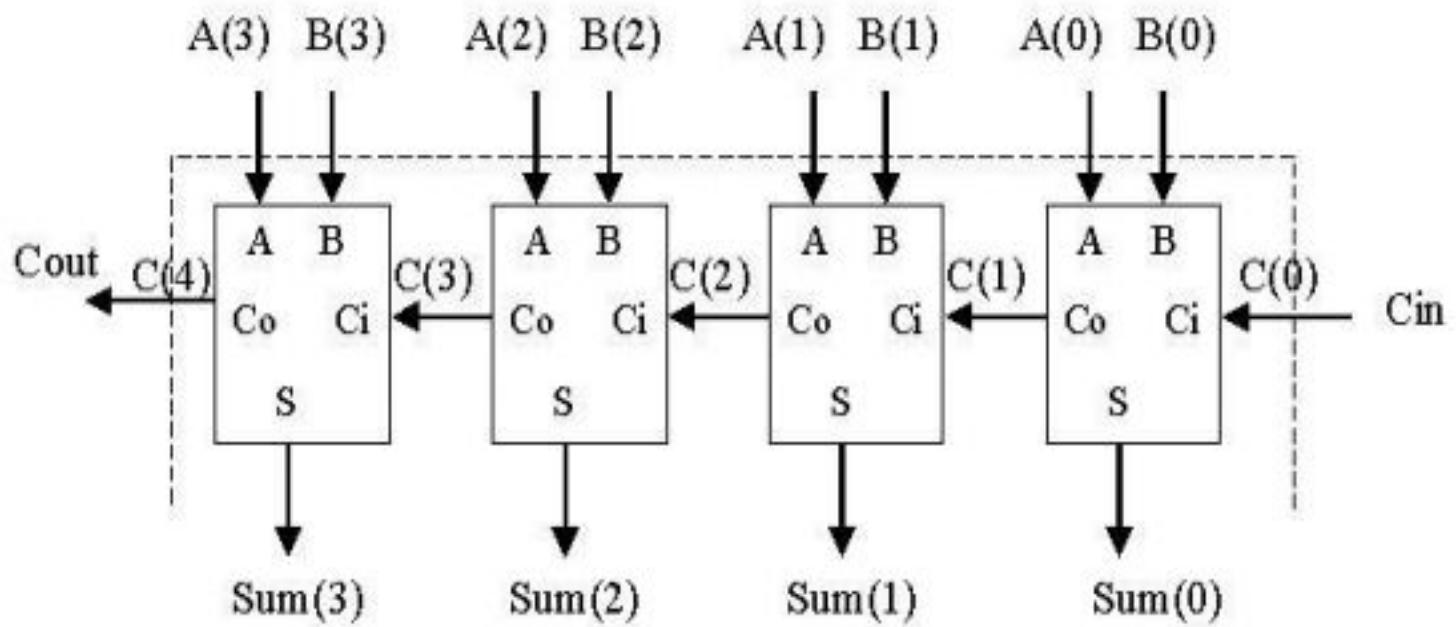
$$\text{Cout } S_3 \ S_2 \ S_1 \ S_0$$

Ripple Carry Adder



Carry Propagation

- From the above example it can be seen that we are adding 3 bits at a time sequentially until all bits are added.
- A full adder is a combinational circuit that performs the arithmetic sum of three input bits: augends A_i , addend B_i and carry in C_{in} from the previous adder.
- Its result contain the sum S_i and the carry out, C_{out} to the next stage.





example

Ripple Carry Adder

It is useful for performing Multibit addition

E1

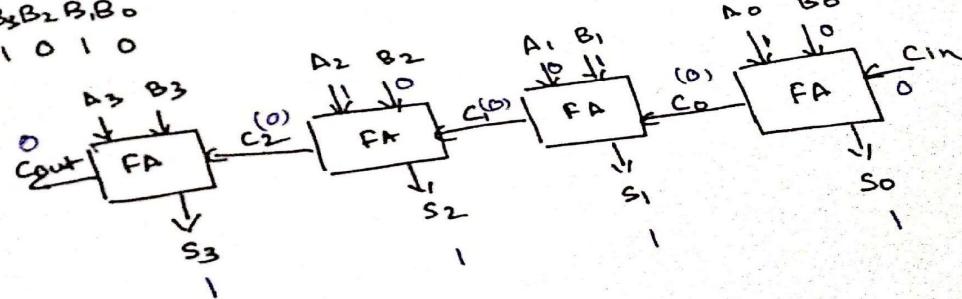
$$A = 0101, B = 1010 \quad C_{in} = 0 \text{ (Initial Condition)}$$

$$\begin{matrix} A_3 & A_2 & A_1 & A_0 \\ 0 & 1 & 0 & 1 \end{matrix}$$

$$\begin{matrix} B_3 & B_2 & B_1 & B_0 \\ 1 & 0 & 1 & 0 \end{matrix}$$

$$\begin{array}{r}
 5 \\
 \hline
 10 \\
 \hline
 15
 \end{array}
 \begin{array}{r}
 0101 \\
 +1010 \\
 \hline
 1111
 \end{array}$$

$S_3 = 1$, $S_2 = 1$, $S_1 = 1$, $S_0 = 1$
 $C_3 = 0$, $C_2 = 0$, $C_1 = 0$, $C_0 = 0$



Ripple Carry Adder



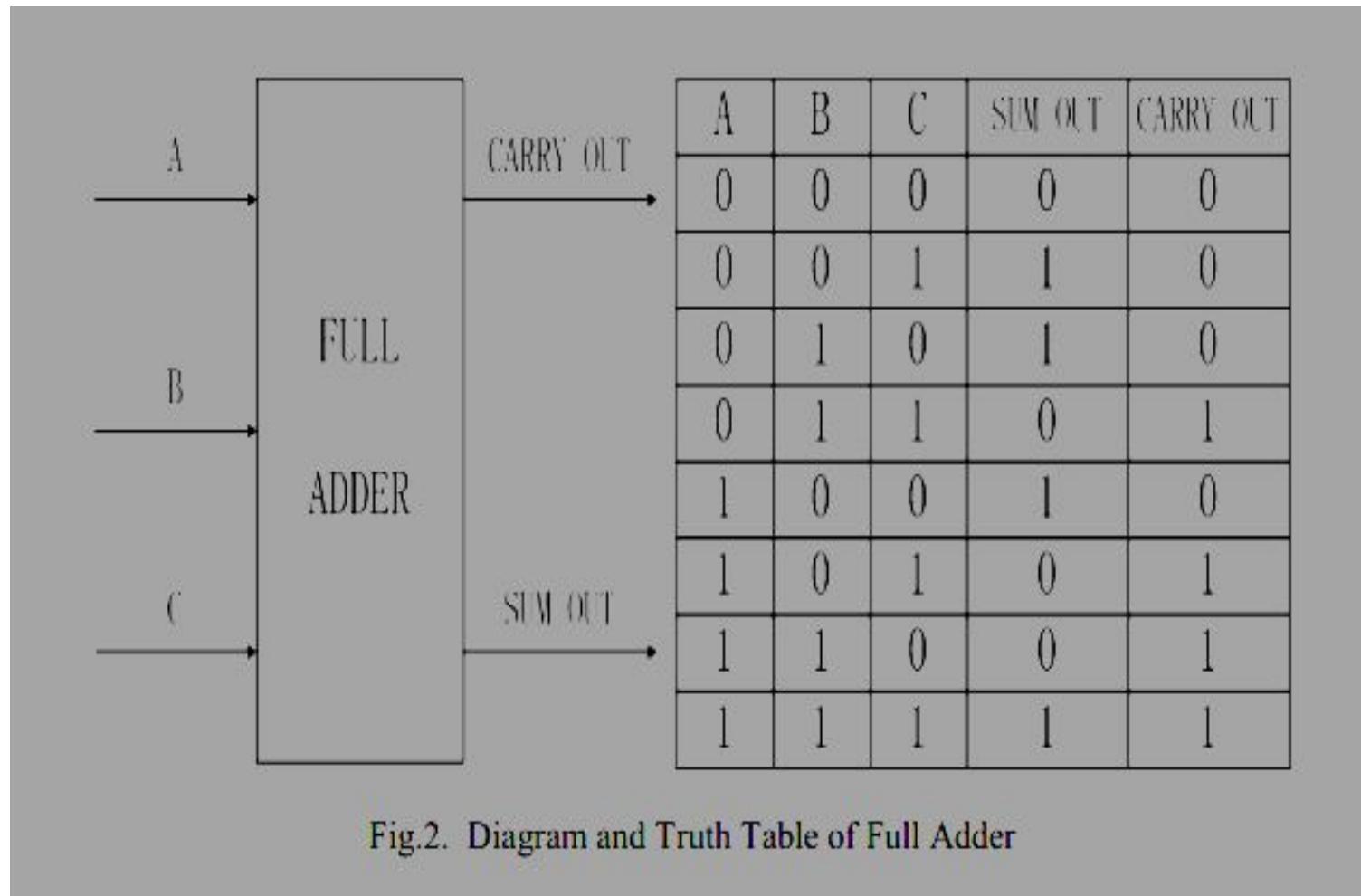
4-bit Adder

- A 4-bit adder circuit can be designed by first designing the 1-bit full adder and then connecting the four 1-bit full adders to get the 4-bit adder as shown in the diagram above.
- For the 1-bit full adder, the design begins by drawing the Truth Table for the three input and the corresponding output SUM and CARRY.
- The Boolean Expression describing the binary adder circuit is then deduced.
- The binary full adder is a three input combinational circuit which satisfies the truth table given below.



Ripple Carry Adder

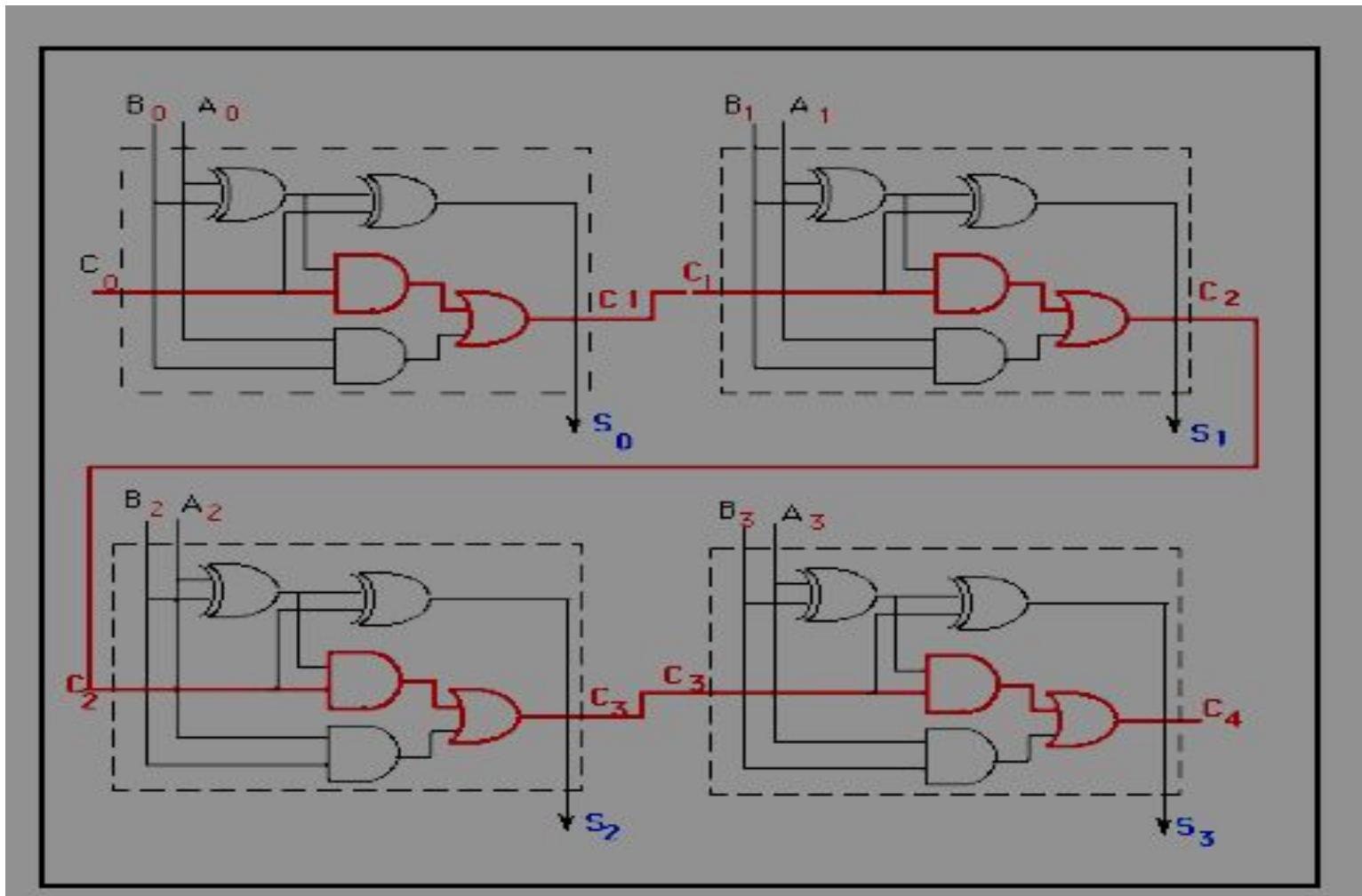
Full Adder





Ripple Carry Adder

4-bit Adder





Draw back of Ripple carry adder:

- * bit execution in serial manner.
- * so more delay due to carry propagation

To reduce the delay:

Carry look ahead Adder (fast adder)

Consider full adder Sum and Carry

$$S = A \oplus B \oplus C$$

$$C = C(A \otimes B) + AB$$

CLA each full adder is going to generate the carry simultaneously

In if in full adder any carry is identified then other carry will get generated.

for doing this two variables are important.

$P_i \rightarrow$ carry propagate

$C_i \rightarrow$ carry generate



Design of Fast adder: Carry Look-ahead Adder

- A carry-look ahead adders (CLA) is a type of adder used in digital logic. A carry-look ahead adder improves speed by reducing the amount of time required to determine carry bits.
- It can be contrasted with the simpler, but usually slower, ripple carry adder for which the carry bit is calculated alongside the sum bit, and each bit must wait until the previous carry has been calculated to begin calculating its own result and carry bits (see adder for detail on ripple carry adders)



- The carry-look ahead adder calculates one or more carry bits before the sum, which reduces the wait time to calculate the result of the larger value bits.
- In a ripple adder the delay of each adder is 10 ns , then for 4 adders the delay will be 40 ns.
- To overcome this delay Carry Look-ahead Adder is used.



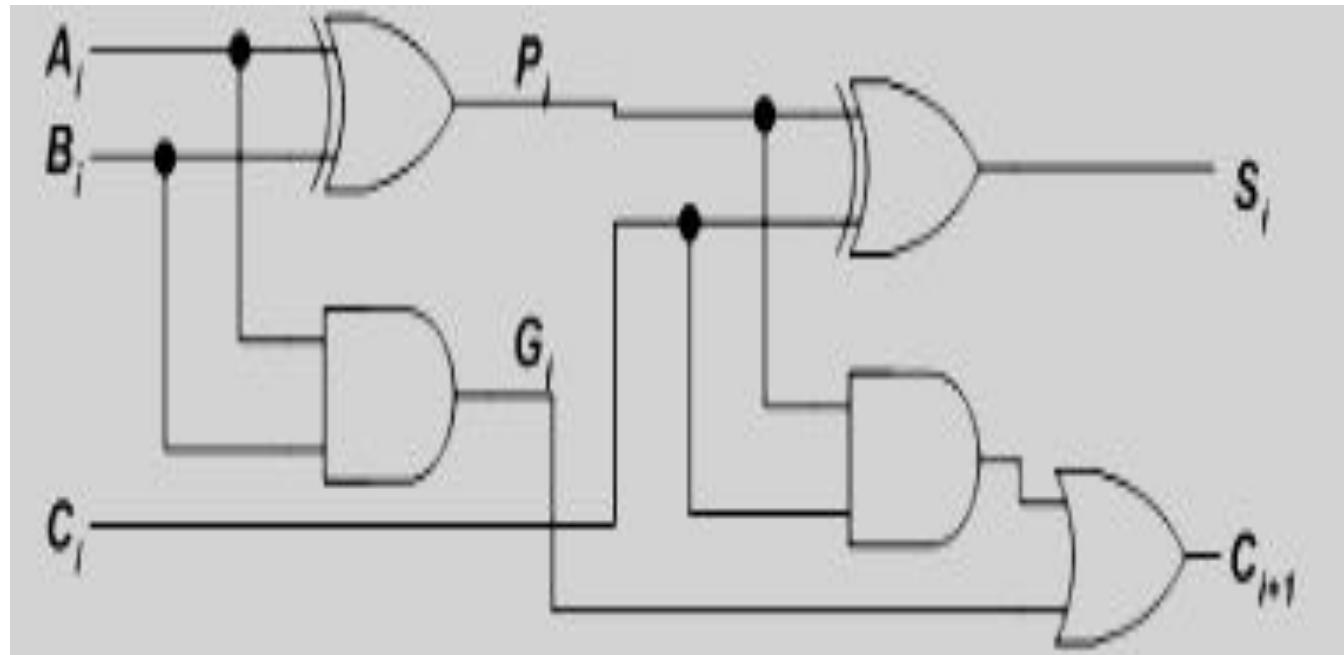
Carry Look-ahead Adder

- Different logic design approaches have been employed to overcome the carry propagation delay problem of adders.
- One widely used approach employs the principle of carry look-ahead solves this problem by calculating the carry signals in advance, based on the input signals.
- This type of adder circuit is called as carry look-ahead adder (CLA adder). A carry signal will be generated in two cases:
 - (1) when both bits A_i and B_i are 1, or
 - (2) when one of the two bits is 1 and the carry-in (carry of the previous stage) is 1.



Carry Look-ahead Adder

The Figure shows the full adder circuit used to add the operand bits in the I^{th} column; namely A_i & B_i and the carry bit coming from the previous column (C_i).





Carry Look-ahead Adder

- G_i is known as the carry Generate signal since a carry (C_{i+1}) is generated whenever $G_i = 1$, regardless of the input carry (C_i).
- P_i is known as the carry propagate signal since whenever $P_i = 1$, the input carry is propagated to the output carry, i.e., $C_{i+1} = C_i$ (note that whenever $P_i = 1$, $G_i = 0$).
- Computing the values of P_i and G_i only depend on the input operand bits (A_i & B_i) as clear from the Figure and equations.
- Thus, these signals settle to their steady-state value after the propagation through their respective gates.



Carry Look-ahead Adder

- Computed values of all the P_i 's are valid one XOR-gate delay after the operands A and B are made valid.
- Computed values of all the G_i 's are valid one AND-gate delay after the operands A and B are made valid.
- The Boolean expression of the carry outputs of various stage $C_1 = G_0 + P_0 C_0$

$$\begin{aligned}C_2 &= G_1 + P_1 C_1 = G_1 + P_1 (G_0 + P_0 C_0) \\&= G_1 + P_1 G_0 + P_1 P_0 C_0\end{aligned}$$

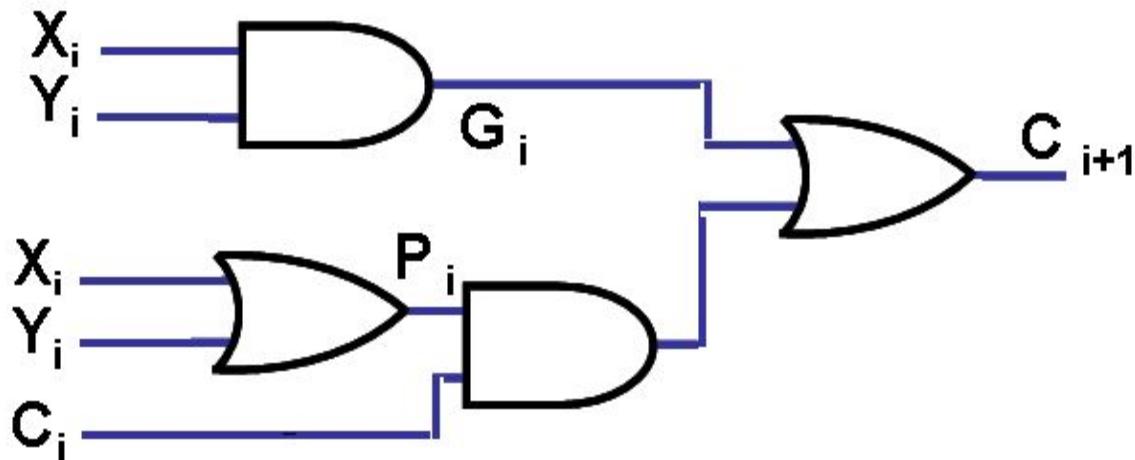
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

$$\begin{aligned}C_4 &= G_3 + P_3 C_3 \\&= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0\end{aligned}$$



Carry Look-ahead Adder

$$\begin{aligned}\text{Carry } C_{i+1} &= X_i Y_i + Y_i C_i + C_i X_i \\ &= X_i Y_i + C_i (X_i + Y_i) \\ &= G_i + C_i P_i. \quad (\text{Generate Carry} + C_i * \text{Propagate Carry})\end{aligned}$$

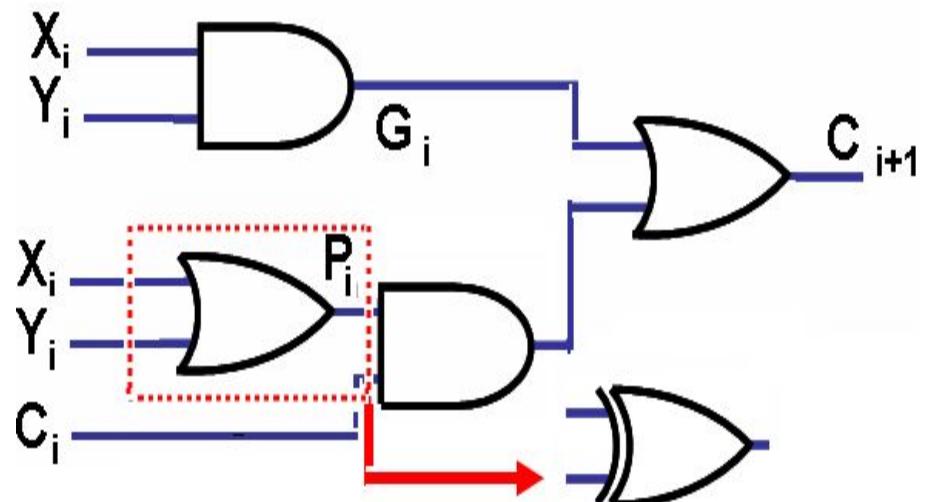


$$\begin{aligned}\text{Carry } C_{i+1} &= G_i + C_i P_i \\ \text{i.e. } C_i &= (G_{i-1} + P_{i-1} C_{i-1}) \quad \& \\ C_{i-1} &= (G_{i-2} + P_{i-2} C_{i-2}).\end{aligned}$$

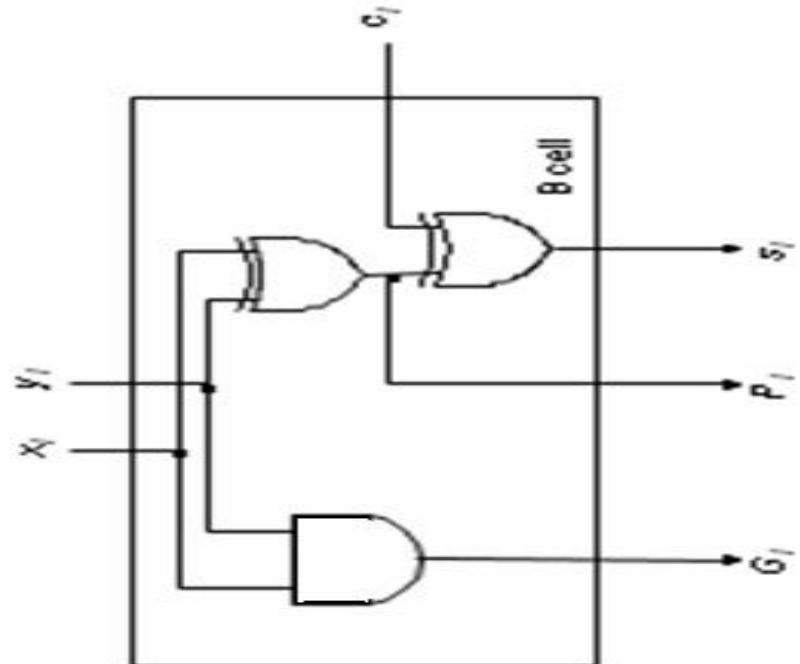


Carry Look-ahead Adder

- $C_{i+1} = G_i + C_i P_i.$



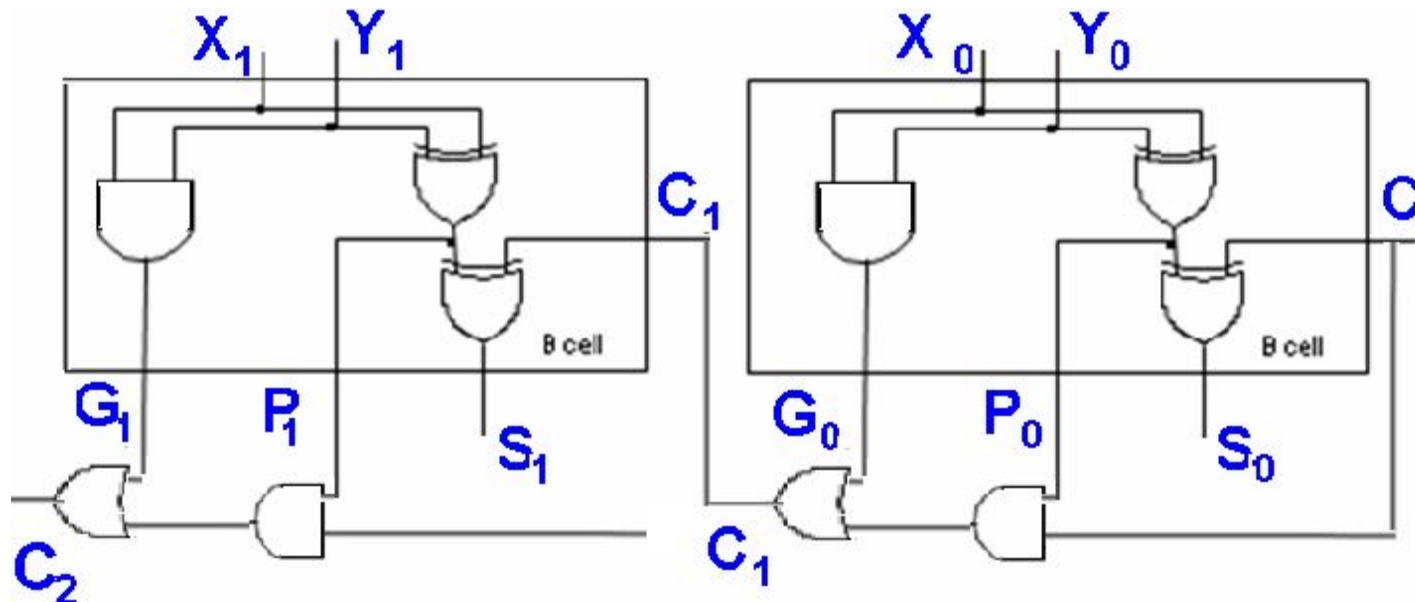
always, except when $X_i = 1 \& Y_i = 1$. But, then $G_i = 1$ to make $C_{i+1} = 1$; hence Bit cell





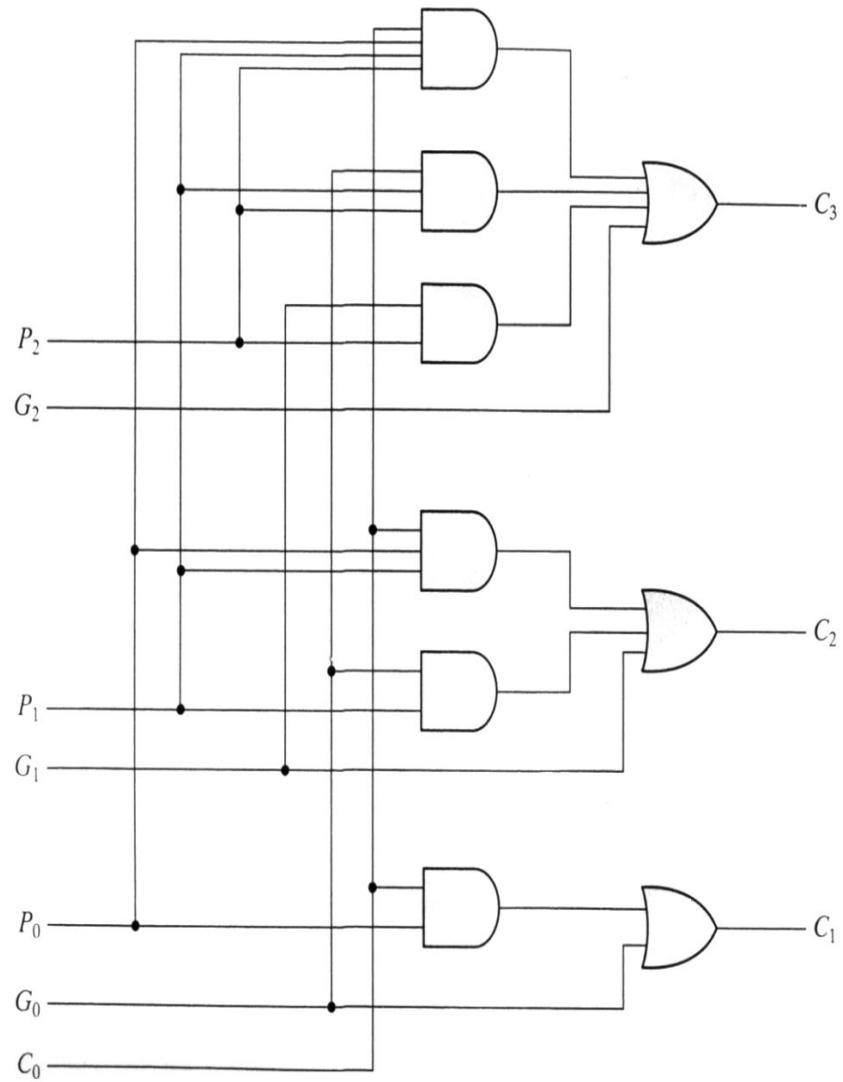
Carry Look-ahead Adder

- In general $C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \dots + P_i P_{i-1} \dots P_1 G_0 + P_i P_{i-1} \dots P_0 G_0$.
- C_{i+1} -- in 3 Gates delay; S_i – in 4 Gates delay irrespective of n. C_1 in 3 gates delay, C_2 in 3 gates delay, C_3 in 3 gates delay and so on.
- S_0 in 4 gates delay, S_1 in 4 gates delay, S_2 in 4 gates delay and so on.





Carry Look-ahead Adder



Implementing these expressions (for a 4-bit adder), results in the logic diagram. (IC- 74182)

$$c_1 = G_0 + P_0 c_0$$

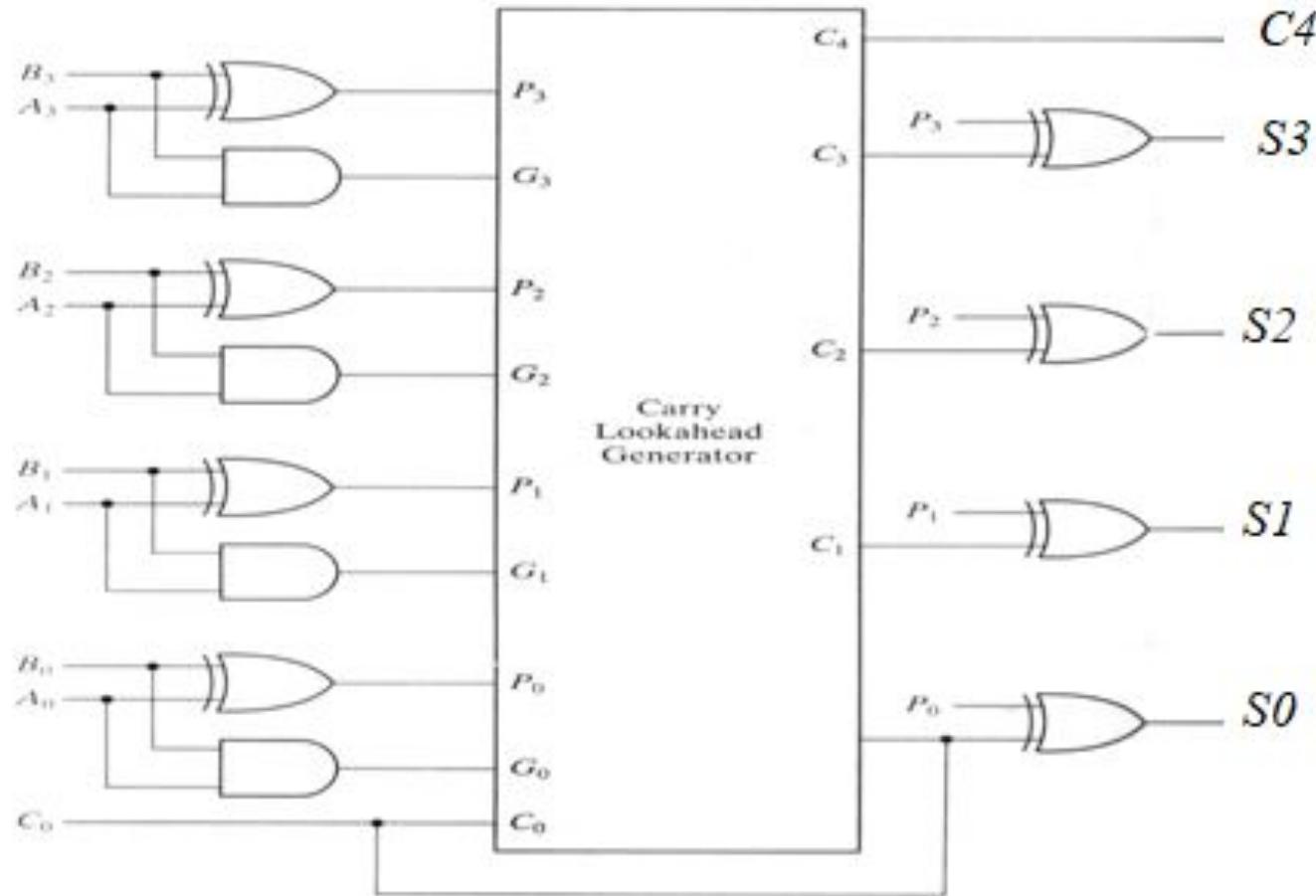
$$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$$

$$c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$

$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$



4-bit Carry Look-ahead Adder





4-bit Carry Look-ahead Adder

- independent of n,
- the n-bit addition process requires only four gate delays (against $2n$)
- increasing ‘n’ increases gate fan-in requirements ($C_4 - \text{fan_in} = 5$)
- Longer version adders - cascading



Binary Multiplier

- Multiplication of binary numbers is performed in the same way as with decimal numbers. The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit.
- The result of each such multiplication forms a partial product. Successive partial products are shifted one bit to the left.
- The product is obtained by adding these shifted partial products.
- Example 1: Consider multiplication of two numbers, say A and B (2 bits each),
$$C = A \times B.$$



Binary Multiplier

- Unsigned number multiplication
- two n-bit numbers; 2n-bit result

$$\begin{array}{r} 1010 \\ \times 1011 \\ \hline 1010 \\ 1010 \\ 0000 \\ 1010 \\ \hline 1101110 \end{array}$$

→ **Multiplicand**
→ **Multiplier**
→ **Partial product 1**
→ **Partial product 2**
→ **Partial product 3**
→ **Partial product 4**



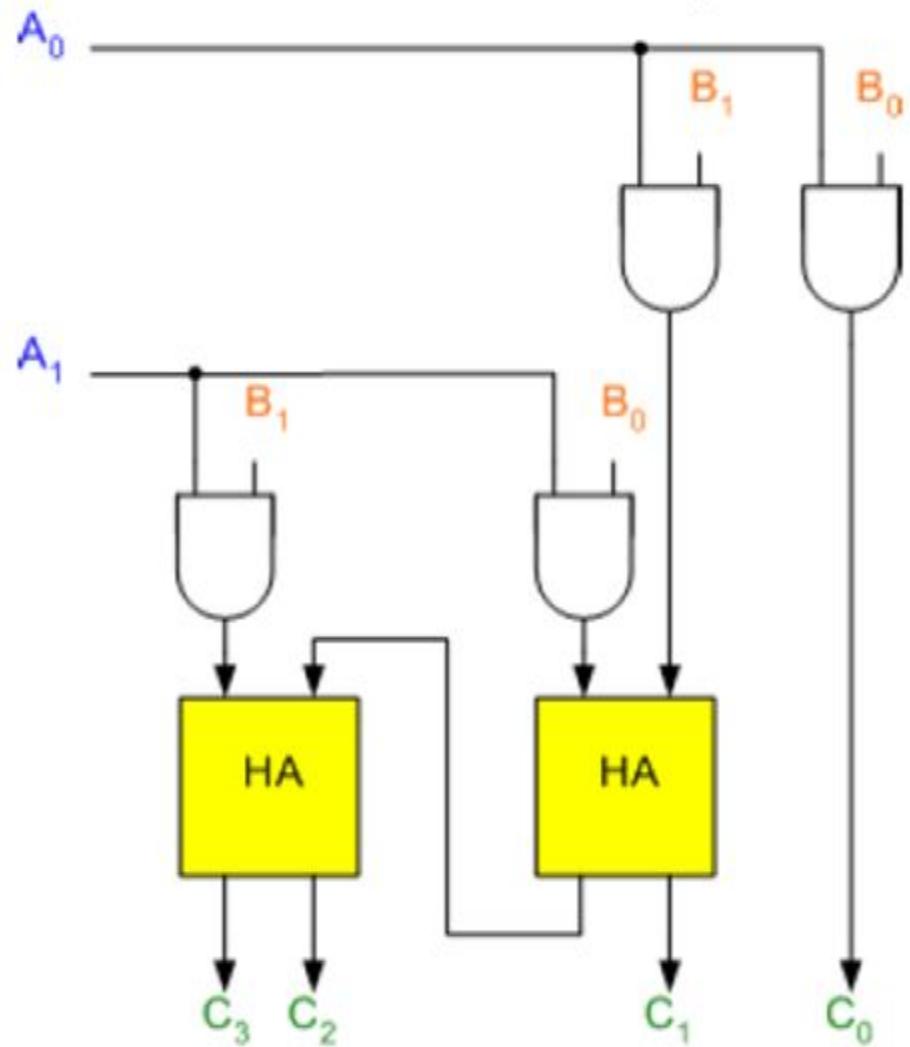
Binary Multiplier

- The first partial product is formed by multiplying the B₁B₀ by A₀. The multiplication of two bits such as A₀ and B₀ produces a 1 if both bits are 1; otherwise it produces a 0 like an AND operation. So the partial products can be implemented with AND gates.
- The second partial product is formed by multiplying the B₁B₀ by A₁ and is shifted one position to the left.
- The two partial products are added with two half adders (HA). Usually there are more bits in the partial products, and then it will be necessary to use FAs.



Binary Multiplier

$$\begin{array}{r} & \textcolor{red}{B_1} & \textcolor{red}{B_0} \\ & \times \textcolor{blue}{A_1} & \textcolor{blue}{A_0} \\ \hline & \textcolor{blue}{A_0} \textcolor{red}{B_1} & \textcolor{blue}{A_0} \textcolor{red}{B_0} \\ \hline \textcolor{blue}{A_1} \textcolor{red}{B_1} & \textcolor{blue}{A_1} \textcolor{red}{B_0} \\ \hline \textcolor{green}{C_3} & \textcolor{green}{C_2} & \textcolor{green}{C_1} & \textcolor{green}{C_0} \end{array}$$





Binary Multiplier

- The least significant bit of the product does not have to go through an adder, since it is formed by the output of the first AND gate as shown in the Figure.



Shift-and-Add Multiplier

- Shift-and-add multiplication is similar to the multiplication performed by paper and pencil.
- This method adds the multiplicand X to itself Y times, where Y denotes the multiplier.
- To multiply two numbers by paper and pencil, the algorithm is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by a single digit of the multiplier and placing the intermediate product in the appropriate positions to the left of the earlier results.



Shift-and-Add Multiplier

- As an example, consider the multiplication of two unsigned 4-bit numbers,
- 13 (1101) and 11 (1011).

$$\begin{array}{r} 1 \ 1 \ 0 \ 1 \\ \times 1 \ 0 \ 1 \ 1 \\ \hline 1 \ 1 \ 0 \ 1 \\ 1 \ 1 \ 0 \ 1 \\ 0 \ 0 \ 0 \ 0 \\ \hline 1 \ 1 \ 0 \ 1 \end{array} \quad \begin{array}{l} (13)_{10} \text{ Multiplicand M} \\ (11)_{10} \text{ Multiplier Q} \\ \qquad \qquad \qquad \boxed{\qquad \qquad \qquad \qquad} \text{ Partial products} \\ (143)_{10} \text{ Product P} \end{array}$$

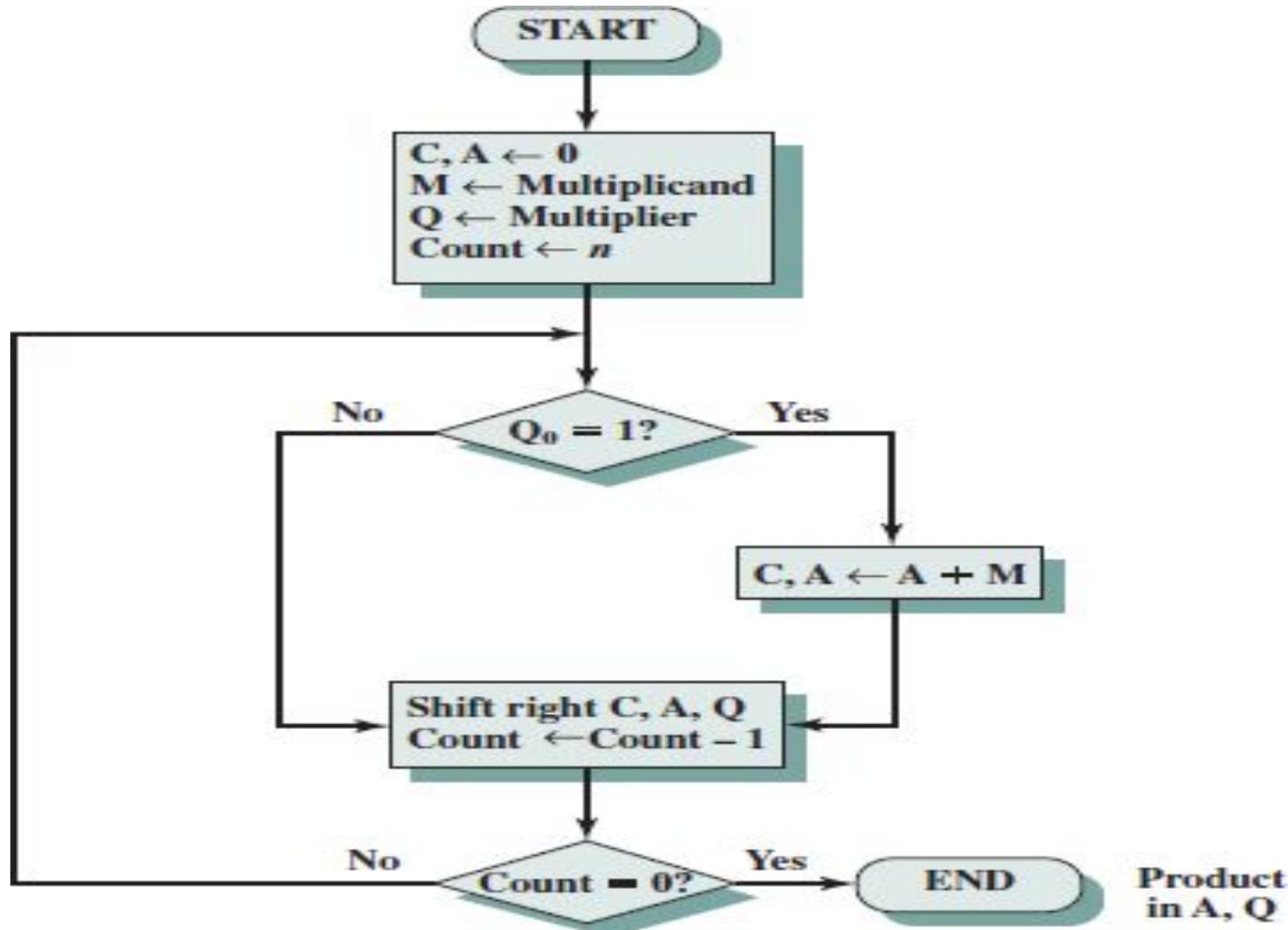


Shift-and-Add Multiplier

- In the case of binary multiplication, since the digits are 0 and 1, each step of the multiplication is simple.
- If the multiplier digit is 1, a copy of the multiplicand ($1 \times$ multiplicand) is placed in the proper positions;
- If the multiplier digit is 0, a number of 0 digits ($0 \times$ multiplicand) are placed in the proper positions.
- Consider the multiplication of positive numbers. The first version of the multiplier circuit, which implements the shift-and-add multiplication method for two n-bit numbers, is shown in Figure.



Shift - and - Add multiplier





Shift-and-Add Multiplier

- For Example, Perform the multiplication 13×11 (1101×1011). Finally, both A and Q contains the result of product.

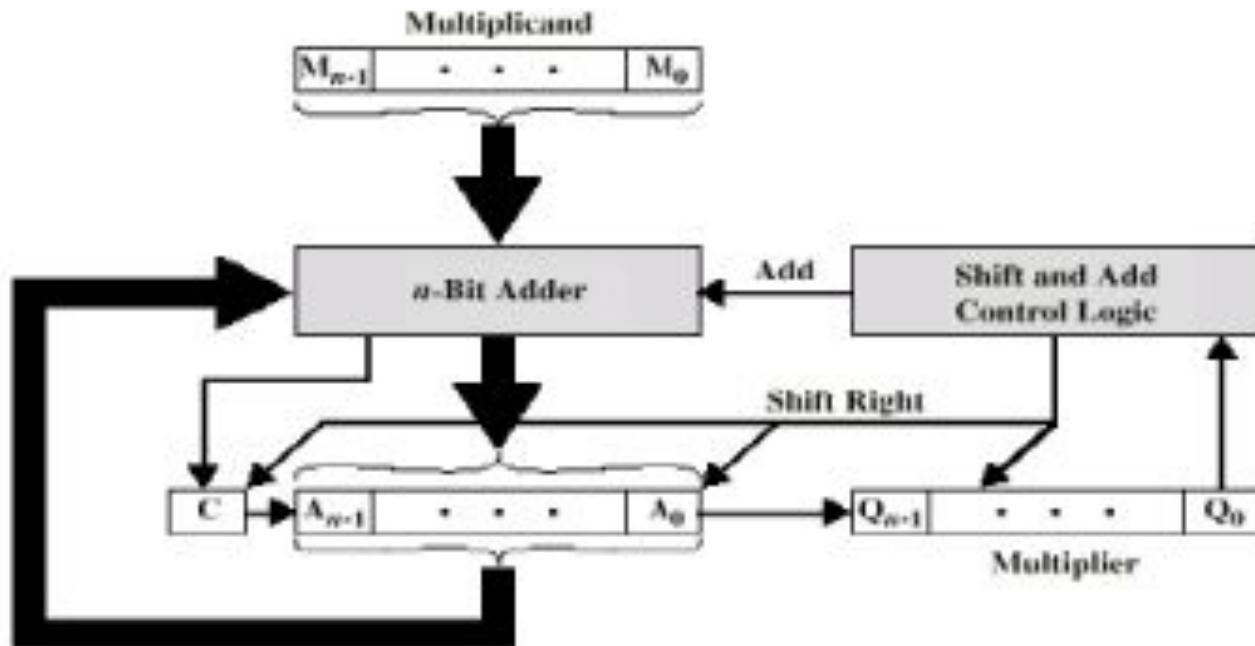


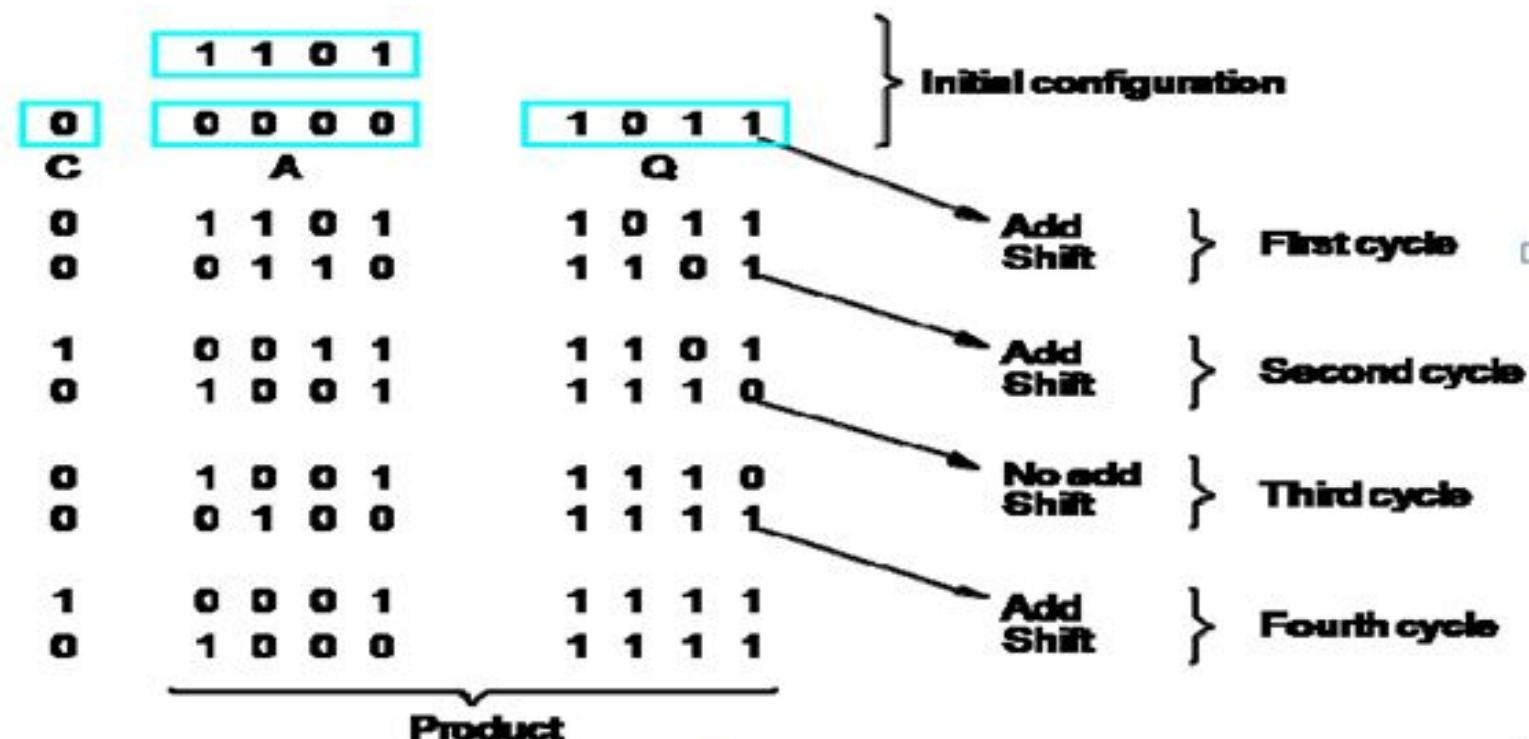
Fig: Block diagram of multiplication



Shift-and-Add Multiplier

Example 2:

- $A = 0000$ M (Multiplicand) $\times 13$ (1 1 0 1)
Q(Multiplier) $\times 11$ (1 0 1 1).



$$11 \times 13 \quad 11 = 1011 \text{ (H)} \quad N=4$$

$$13 = 1101 \text{ (A)}$$

N	C	A	α	H	
4	0	0000	1101	1011	$q_0 = 1 \rightarrow A = A + H, RS, N-1$
	Q	1011 0101	1101 1110	1011 1011	$\frac{0000}{1011} \quad 4-1=3$
3	0	0101	1110	1011	$q_0 = 0, RS, N-1$
	Q	0010	1111	1011	$3-1=2$
2	0	0010	1111	1011	$q_0 = 1, A = A + H, RS, N-1$
	Q	1101 0110	1111	1011	$\frac{0010}{1011} \quad 2-1=1$
1	0	0110	1111	1011	$q_0 = 1, A = A + H, RS, N-1$
	Q	0001	1111	1011	
	0	1000	1111	1011	$\frac{0110}{1011} \quad N-1$ $\frac{1000}{1000} \quad 1-1=0$
					Try for <u>12x10</u>
$N=0$ Stop the process					
Product = A0					
$1000 \ 1111$					
$= 1430$					

Signed Multiplication - Booth Algorithm



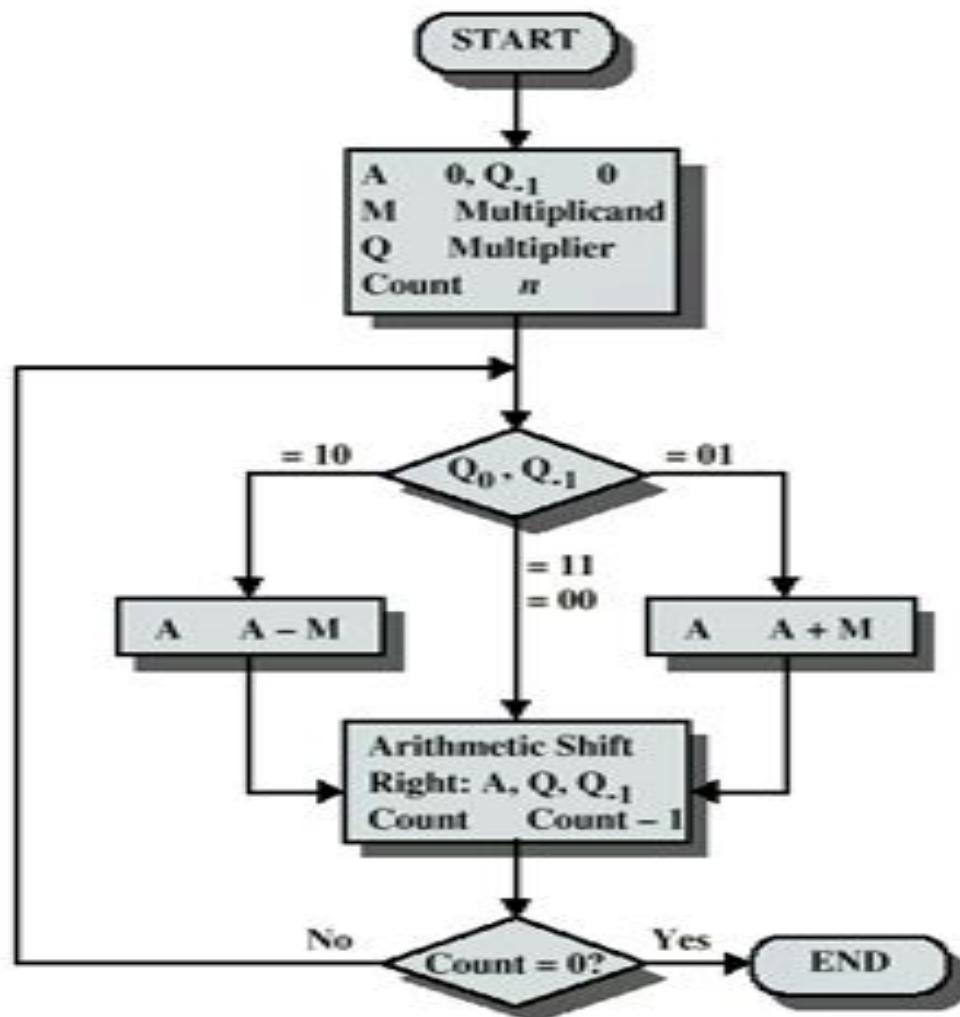


Signed Multiplication - Booth Algorithm

- A powerful algorithm for signed number multiplication is Booth's algorithm which generates a $2n$ bit product and treats both positive and negative numbers uniformly.
- This algorithm suggest that we can reduce the number of operations required for multiplication by representing multiplier as a difference between two numbers.



Signed Multiplication - Booth Algorithm





Signed Multiplication - Booth Algorithm

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A	A - M } First
1100	1001	1	0111	Shift	} Cycle
1110	0100	1	0111	Shift	} Second Cycle
0101	0100	1	0111	A	A + M } Third
0010	1010	0	0111	Shift	} Cycle
0001	0101	0	0111	Shift	} Fourth Cycle

Signed Multiplication

BOOTH RECODING OF MULTIPLIERS

Signed Multiplication

- Considering 2's-complement signed operands, what will happen to $(-13) \times (+11)$ if following the same method of unsigned multiplication?

Sign extension is shown in blue

$$\begin{array}{r} & 1 & 0 & 0 & 1 & 1 & (-13) \\ & 0 & 1 & 0 & 1 & 1 & (+11) \\ \hline & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & (-143) \end{array}$$

Sign extension of negative multiplicand.



Example for Signed Number Multiplication

①

$$-7 \times 3 \quad q_1 = 3 = 0011 \quad q_0 = 0$$

$$M = -7$$

$$N = 4$$

$$A = 0111 \xrightarrow{1's} 1000 \xrightarrow{2's} 1001 \quad -M = 0111$$

	A	q_1	q_0	M	Steps	①	②	S
4	0000	0011	0	1001	$q_{10} = 0$ $q_{11} = 1$	$A = A - H$ $= A + (-M)$ $= 0000$ $\underline{0111}$ 0111	ARS $N = N-1$ $4-1=3$	
3	0111	0011	0	1001				
	↓111	↓111						
0011	1001	10	1001					
2	0001	1101	1	1001	$q_{10} = 0$ $q_{11} = 1$	ARS $N = N-1$		
	1010	1100	1	1001				
	↓1111	↓1111						
	1101	0110	0	1001				
1	1101	0110	0	1001	$q_{10} = 0$ $q_{11} = 0$	ARS $N = N-1$		
	↓1111	↓1111						
	1110	1011	0	1001				
0	1110	1011	0	1001				

Product = AB

$$= \underline{1110} \quad 1011$$

$$= - [\underline{1110} \quad 1011) \downarrow 1's]$$

$$= \underline{\quad \quad \quad 00010100} \quad \downarrow 2's$$

$$\begin{array}{r} 00010101 \\ +1 \\ \hline 00010101 \end{array} \quad \begin{matrix} \downarrow 2^3 & \downarrow 2^2 & \downarrow 2^1 \\ 4 & 2 & 1 \\ 21 \end{matrix}$$

-21"

Example 2

Perform Multiplication for $(-15) \times (-13)$

$$M = -15$$

$$\begin{array}{r} 15 = \\ \begin{array}{r} S \quad 8 \quad 4 \quad 2 \quad 1 \\ 0 \quad 1 \quad 1 \quad 1 \end{array} \\ i's = 1 \quad 0 \quad 0 \quad 0 \quad 0 \\ d's = \underline{\underline{1 \quad 0 \quad 0 \quad 0}} \end{array}$$

$$M = 10001$$

$$-M = 01111$$

$$B = -13$$

$$\begin{array}{r} 13 = \\ \begin{array}{r} S \quad 8 \quad 4 \quad 2 \quad 1 \\ 0 \quad 1 \quad 1 \quad 0 \end{array} \\ i's \rightarrow 1 \quad 0 \quad 0 \quad 1 \quad 0 \\ d's \rightarrow \underline{\underline{1 \quad 0 \quad 0 \quad 1}} \end{array}$$

$$q_1 = 10011, q_0 = 0$$

$$N = 5$$

N	A	q_V	q_0	M	$q_0 = 0, A = A - M$ $q_1 = 1, A = A + (M)$ $\text{ARS} N = N - 1$
5	00000	10011	0	10001	
	01111	10011	0	10001	
	00111	11001	1	10001	

N	A	q_V	q_0	M	$q_0 = 1, q_1 = 1 \Rightarrow \text{ARS} \rightarrow N - 1$
4	00111	11001	1	10001	
	00011	11100	1	10001	

2

N	A	q_V	q_0	M	$q_0 = 0$ $q_1 = 0$ $\text{ARS} \rightarrow N - 1$
2	11010	01110	0	10001	
	11101	00111	0	10001	
1	11101	00111	0	10001	$q_0 = 0, A = A - M$ $q_1 = 1, A = A + (-M)$ 11101 01111 101100
0	00110	00011	1	10001	

$$\text{Product} = AB$$

$$00110 \quad 00011 = 195$$

$$-15 \times -13 = 195$$

Try for 1115×-13

$$2) -6 \times 7$$



The BOOTH RECODED MULTIPLIER

- Booth multiplication reduces the number of additions for intermediate results, but can sometimes make it worse as we will see.
- Booth multiplier recoding table

Multiplier		Version of multiplicand selected by bit
Bit i	Bit $i-1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$



The BOOTH RECODED MULTIPLIER

- Booth Recoding:

(i) 30_{10} : 0 1 1 1 1 0 0

• $+1 \ 0 \ 0 \ 0 \ -1 \ 0$

• (ii) 100_{10} : 0 1 1 0 0 1 0 0 0

•

• $+1 \ 0 \ -1 \ 0 \ +1 \ -1 \ 0 \ 0$

• (iii) 985_{10} : 0 0 1 1 1 1 0 1 1 0 0 1 0

• $0 \ +1 \ 0 \ 0 \ 0 \ -1 \ +1 \ 0 \ -1 \ 0 \ +1 \ -1$



BOOTH Algorithm

Booth algorithm treats both +ve &
-ve operands equally.

(a) + Md X + Mr

Ex: $0\ 1\ 1\ 0\ 1\ (+13) \times 0\ 1\ 0\ 1\ 1\ (+11)$

+1-1+1 0 -1

1 1 1 1 1 1 0 0 1 1

0 0 0 0 0 0 0 0 0

0 0 0 0 1 1 0 1

1 1 1 0 0 1 1

0 0 1 1 0 1

0 0 1 0 0 0 1 1 1 1 (+143)



BOOTH Algorithm

Booth algorithm treats both +ve &
-ve operands equally.

(b) - Md X + Mr

Ex: $1\ 0\ 0\ 1\ 1\ (-13) \times 0\ 1\ 0\ 1\ 1\ (+11)$

+1-1+1 0 -1

0	0	0	0	0	0	1	1	0	1
0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	1	1		
0	0	0	1	1	0	1			
1	1	0	0	1	1				

1 0 1 0 0 0 0 0 0 1 (-143)





BOOTH Algorithm

Booth algorithm treats both +ve &
-ve operands equally.

(c) + Md X - Mr

Ex: $01101 (+13) \times 10101 (-11)$

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

$1\ 0\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ (-143)$

→



BOOTH Algorithm

Booth algorithm treats both +ve &
-ve operands equally.

(d) - Md X - Mr

Ex:
$$\begin{array}{r} 10011 \text{ (-13)} \\ \times 00101 \text{ (-11)} \\ \hline -1+1-1 \quad +1-1 \\ 0000000000000 \\ 00000011101 \\ 1111100111 \\ 000011101 \\ 1110011 \\ 001101 \\ \hline 00100011111 \text{ (+143)} \\ \hline \end{array}$$



BOOTH Algorithm

Good multiplier

0	0	0	0	1	1	1	1	1	0	0	0	0	1	1	1
0	0	0	+1	0	0	0	0	-1	0	0	0	+1	0	0	-1

Count = 4 Vs 8 speed improvement

Ordinary multiplier

1	1	0	0	0	1	0	1	1	0	1	1	1	1	0	0
0	-1	0	0	+1	-1	+1	0	-1	+1	0	0	0	-1	0	0

Count = 7 Vs 9 no speed improvement

Worst-case multiplier

0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1

Count = 16 Vs 8 speed worsened.

On an average no improvement in speed



Example 2

$$-12 \times -3$$

Multiplicand
 $-12 = 01100$

Multiplicand
 $-3 = 00011$

$\begin{array}{r} 1's \\ +1 \\ \hline 10011 \end{array}$ $\xrightarrow{2's}$ 10100

$\begin{array}{r} 1's \\ +1 \\ \hline 11100 \end{array}$ $\xrightarrow{2's}$ 11101

$-12 = 10100, -3 = 11101$

Recode the Multiplicand in (-3)

$\begin{array}{r} 11101 \\ \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\ 00-1+1-1 \end{array}$

Add implied zero

\therefore Recoded Value is 00-1+1-1

(Sign bit extension) $10100 \times 00-1+1-1$

0	0	0	0	0	1	1	0	0
1	1	1	1	0	1	0	0	
0	0	0	1	1	0	0		
0	0	0	0	0	0	0		
0	0	0	0	0	0	0		

$$-12 \times -3 = 36$$

when doing multiplication by -1, the result is 2's complement of multiplicand

2's of
12 is 01100

$$\begin{array}{r} 2^5 2^4 2^3 2^2 2^1 2^0 \\ \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\ 32 \quad 4 \\ = 36 \end{array}$$

Try for $\underline{\underline{-11}} \times \underline{\underline{6}}$



FAST MULTIPLICATION

- 1.BIT PAIR RECODING OF MULTIPLIERS
- 2.CARRY SAVE ADDITION OF SUMMANDS



Fast Multiplication

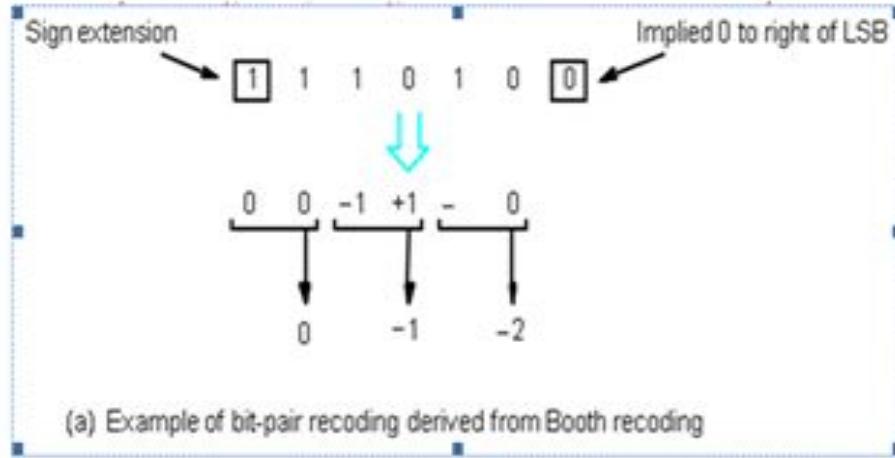
- There are two techniques for speeding up the multiplication operation.
- The first technique guarantees that the maximum number of summands (versions of the multiplicand) that must be added is $n/2$ for n -bit operands.
- The second technique reduces the time needed to add the summands (carry-save addition of summands method).



Bit-Pair Recoding of Multipliers

- This bit-pair recoding technique halves the maximum number of summands. It is derived from the Booth algorithm.
- Group the Booth-recoded multiplier bits in pairs, and observe the following: The pair (+1 -1) is equivalent to the pair (0 +1).
- That is, instead of adding $-1 \times M$ at shift position i to $+1 \times M$ at position $i + 1$, the same result is obtained by adding $+1 \times M$ at position i . Other examples are: (+1 0) is equivalent to (0 +2), (-1 +1) is equivalent to (0 -1). and so on

Bit-Pair Recoding of Multipliers



original bit pair <i>i+1 i</i>	Bit to right <i>i-1</i>	Bit pair Recoded <i>y_{i+1} y_i</i>	Multiplier value
0 0	0	0 0	0
0 0	1	0 1	+1
0 1	0	1 -1	+1
0 1	1	1 0	+2
1 0	0	-1 0	-2
1 0	1	-1 1	-1
1 1	0	0 -1	-1
1 1	1	0 0	0



BIT PAIR RECODING OF MULTIPLIERS

$i+1$	i	$i-1$	Multiplicand Selected position i
0	0	0	$+0 \times M$
0	0	1	$+1 \times M$
0	1	0	$+1 \times M$
0	1	1	$+2 \times M$
1	0	0	$-2 \times M$
1	0	1	$-1 \times M$
1	1	0	$-1 \times M$
1	1	1	$0 \times M$



Bit-Pair Recoding of Multipliers

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



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



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

FIG – 15: Multiplication requiring $n/2$ summands.



Carry-Save Addition of Summands

- A **carry-save adder** is a type of digital adder, used to efficiently compute the sum of three or more binary numbers.
- A carry-save adder (CSA), or 3-2 adder, is a very fast and cheap adder that does not propagate carry bits.
- A Carry Save Adder is generally used in binary multiplier, since a binary multiplier involves addition of more than two binary numbers after multiplication.
- It can be used to speed up addition of the several summands required in multiplication
- It differs from other digital adders in that it outputs two (or more) numbers, and the answer of the original summation can be achieved by adding these outputs together.
- A big adder implemented using this technique will usually be much faster than conventional addition of those numbers.

Fast Multiplication

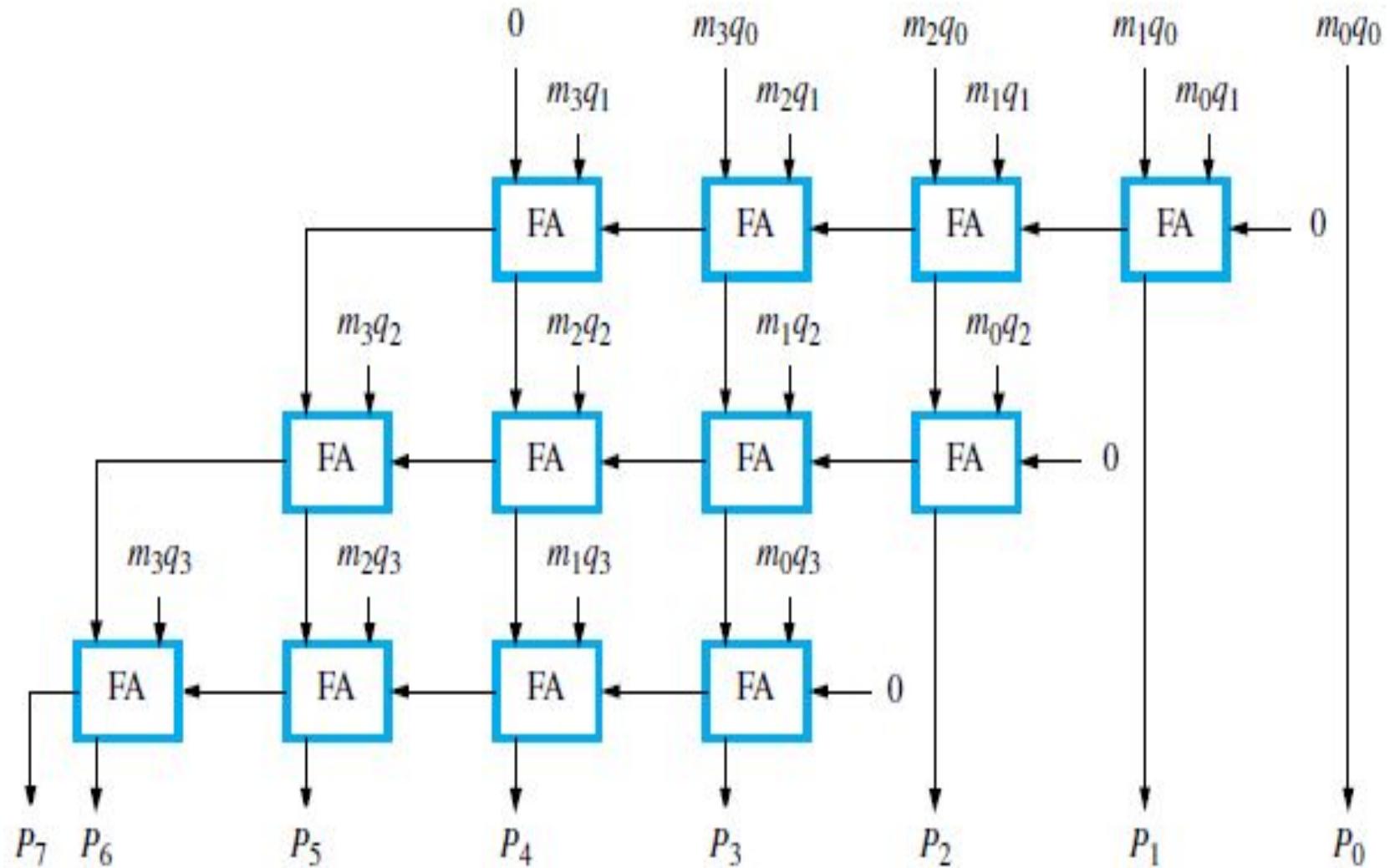
- Bit pair recoding reduces summands by a factor of 2
- Summands are reduced by carry save addition
- Final product can be generated by using carry look ahead adder



Carry-Save Addition of Summands

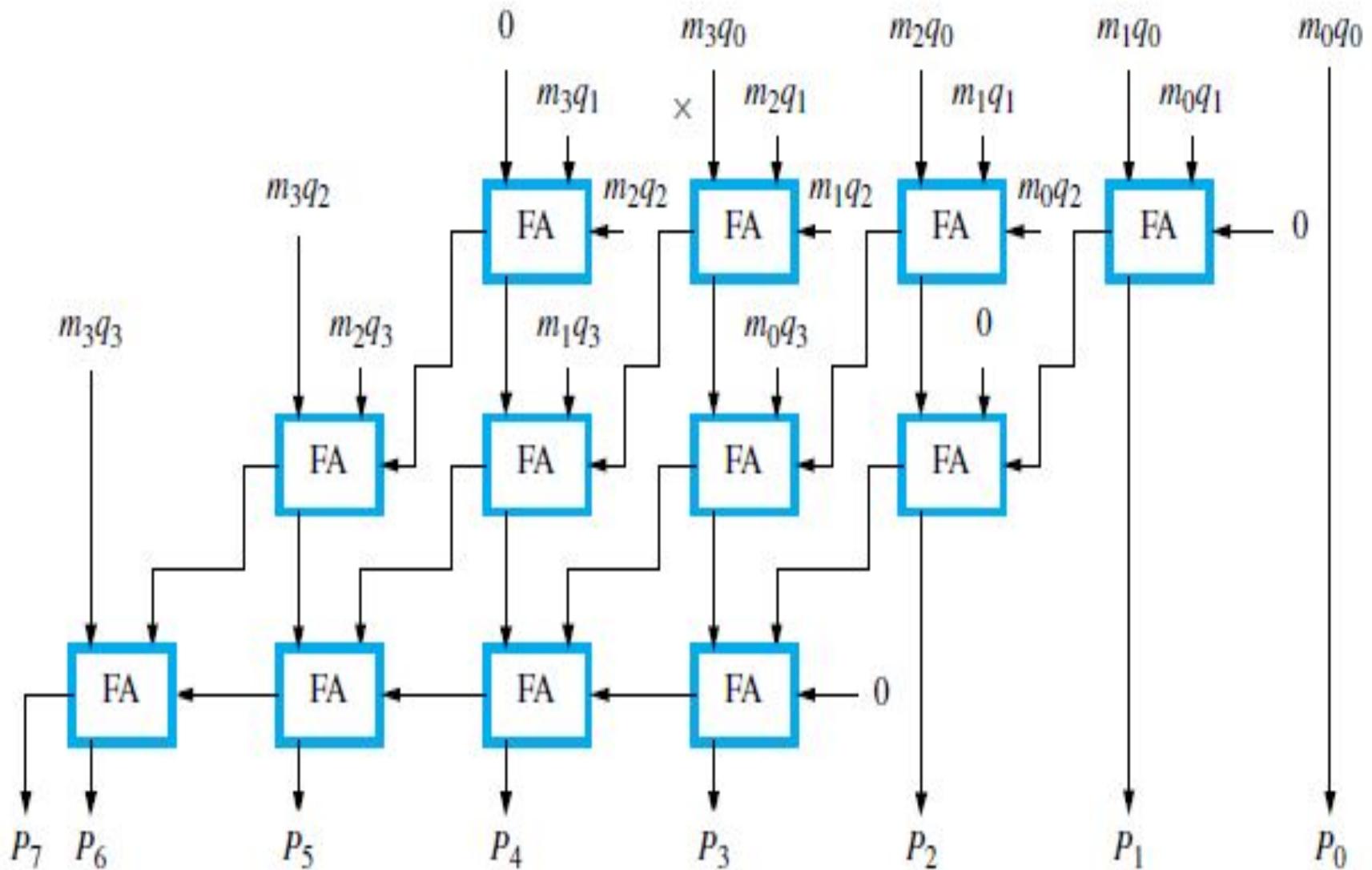
- **Disadvantage of the Ripple Carry Adder** - Each full adder has to wait for its carry-in from its previous stage full adder. This increase propagation time. This causes a delay and makes ripple carry adder extremely slow. RCar is **very slow** when adding many bits.
- **Advantage of the Carry Look ahead Adder** - This is an improved version of the Ripple Carry Adder. Fast parallel adder. It generates the carry-in of each full adder simultaneously without causing any delay. So, CLAr is faster (because of reduced propagation delay) than RCar.
- **Disadvantage the Carry Look-ahead Adder** - It is **costlier** as it reduces the propagation delay by more **complex hardware**. It gets more complicated as the number of bits increases.

Ripple Carry Array





Carry Save Array





Carry-Save Addition of Summands

- Consider the addition of many summands, We can:
 - Group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay
 - Group all of the S and C vectors into threes, and perform carry-save addition of them, generating a further set of S and C vectors in one more full-adder delay
 - Continue with this process until there are only two vectors remaining
 - They can be added in a Ripple Carry Adder (RPA) or Carry Look-ahead Adder (CLA) to produce the desired product

Carry-Save Addition of Summands

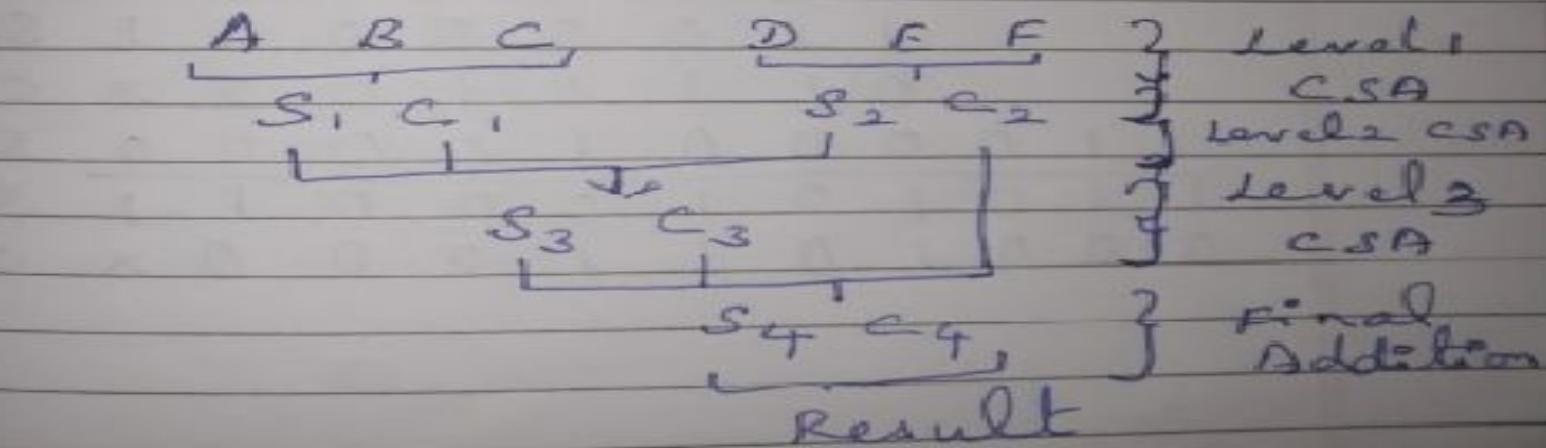


Carry Save Addition:

$$45 \times 63$$

$$\begin{array}{r} 45 \\ 63 \\ \hline \end{array} \quad \begin{array}{r} 101101 \\ 111111 \\ \hline \end{array}$$

$$\begin{array}{r} 101101 \\ 01101x \\ 01101xx \\ 101101xxx \\ 101101xxx \\ 101101xxx \\ \hline \end{array} \quad \begin{array}{l} A \\ B \\ C \\ D \\ E \\ F \\ \hline \end{array}$$



Carry-Save Addition of Summands



$$\begin{array}{r} 101101A \\ 101101B \\ 101101 \times C \\ \hline 11000011 S_1 \\ 01111100 \times C_1 \end{array}$$

$$\begin{array}{r} 101101 \times \times \times D \\ 101101 \times \times \times E \\ 101101 \times \times \times F \\ \hline 110000110000 S_2 \\ 0111110000 \times C_2 \end{array}$$

$$\begin{array}{r} 11000011 S_1 \\ 01111100 \times C_1 \\ 110000110000 S_2 \\ 11010100011 S_3 \\ 00010110000 \times C_3 \end{array}$$



Carry-Save Addition of Summands

Handwritten notes illustrating the Carry-Save Addition of Summands:

Summand 1: 11010100011 s_1

Summand 2: 00010110000 $\times s_2$

Summand 3: 01111000000 $\times s_3$

Summand 4: 10101000000 $\times s_4$

Carry Save Adder (CSA) diagram:

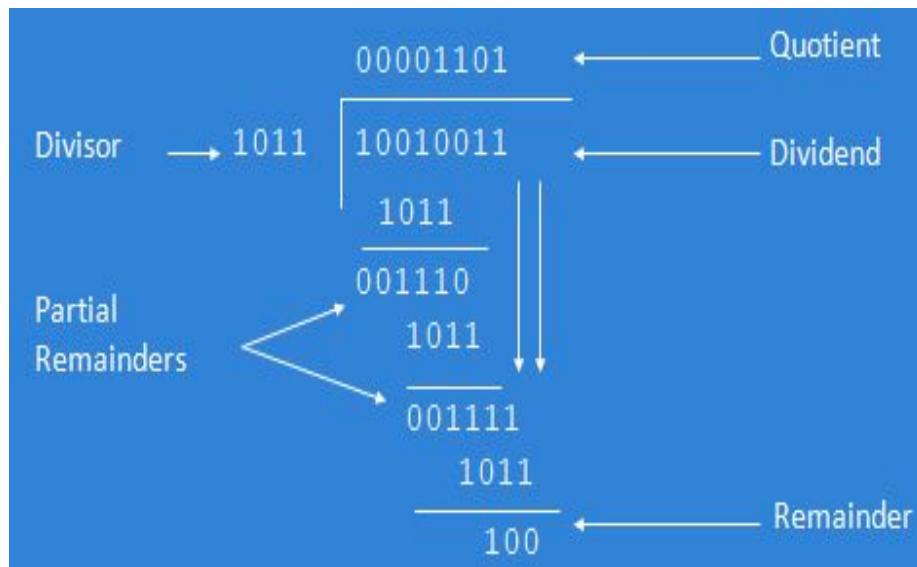
Input 1: 11010100011
Input 2: 00010110000
Input 3: 01111000000
Input 4: 10101000000
Carry In: 1111
Sum: 1011000100011 $\rightarrow 2835$

Product is 2835



Integer Division

- More complex than multiplication
- Negative numbers are really bad!
- Based on long division





Integer Division

- Decimal Division
- Binary Division

$$\begin{array}{r} 21 \\ 13 \) 274 \\ \underline{-26} \\ 14 \\ \underline{-13} \\ 1 \end{array}$$

$$\begin{array}{r} 10101 \\ 1101 \) 100010010 \\ \underline{-1101} \\ 10000 \\ \underline{-1101} \\ 1110 \\ \underline{-1101} \\ 1 \end{array}$$

Figure: Longhand division examples



Integer Division

Longhand Division operates as follows:

- Position the divisor appropriately with respect to the dividend and performs a subtraction.
- If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed.
- If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction



Restoring Division

- Similar to multiplication circuit
- An n-bit positive divisor is loaded into register M and an n-bit positive dividend is loaded into register Q at the start of the operation.
- Register A is set to 0
- After the division operation is complete, the n-bit quotient is in register Q and the remainder is in register A.
- The required subtractions are facilitated by using 2's complement arithmetic.
- The extra bit position at the left end of both A and M accommodates the sign bit during subtractions.



Restoring Division

Strategy for unsigned division:

Shift the dividend one bit at a time starting from MSB into a register.

Subtract the divisor from this register.

If the result is negative ("didn't go"):

- Add the divisor back into the register.
- Record 0 into the result register.

If the result is positive:

- Do not restore the intermediate result.
- Set a 1 into the result register.

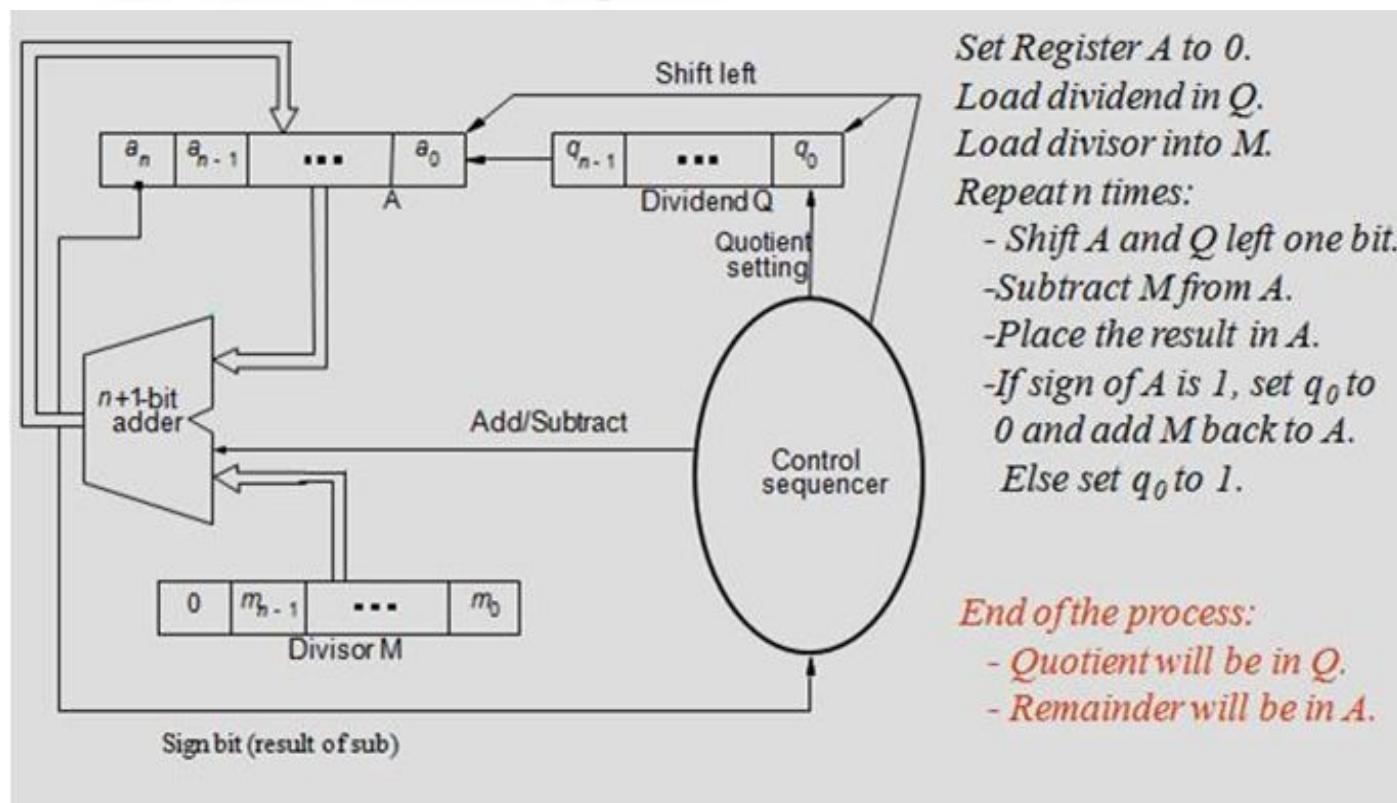


Figure: Logic Circuit arrangement for binary Division (Restoring)



Restoring Division

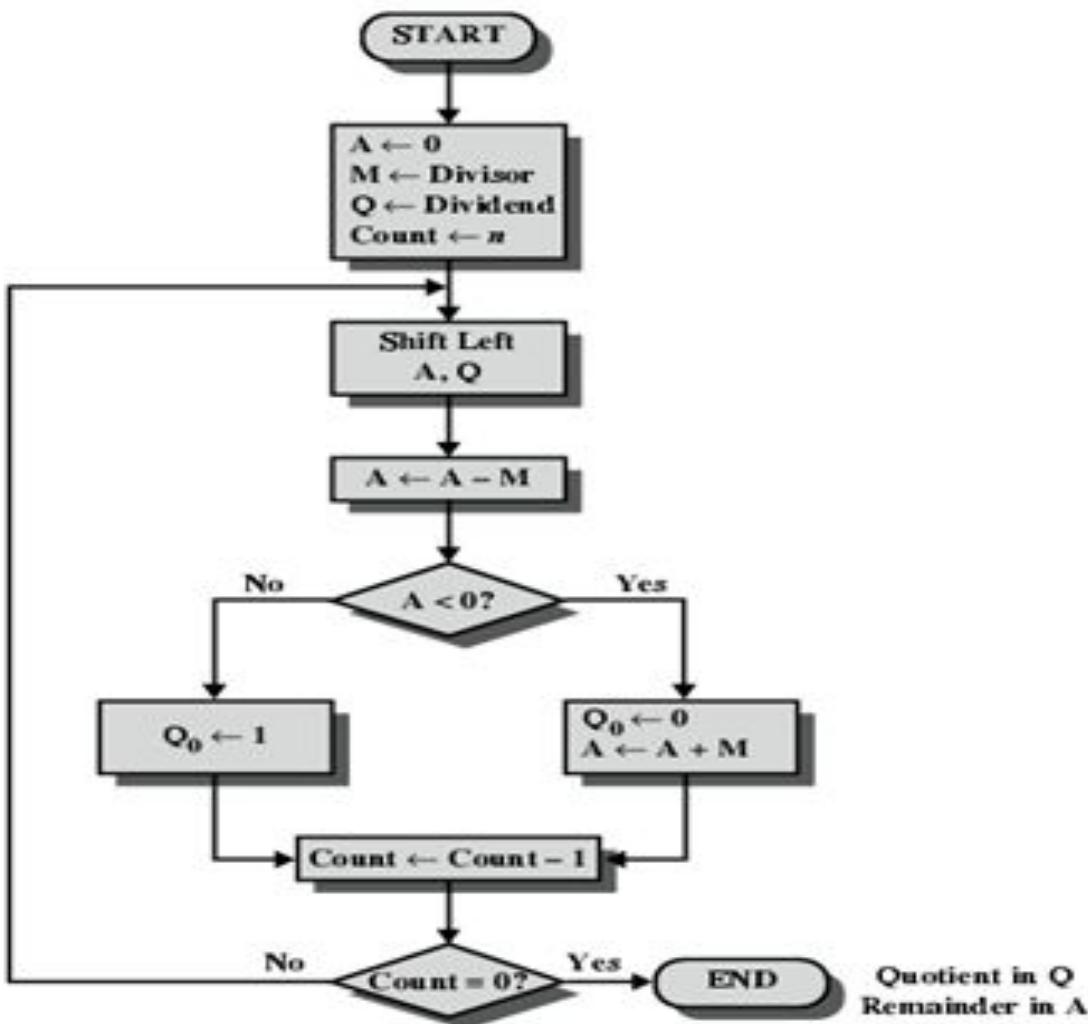


Figure: Flowchart for Restoring Division



Restoring Division

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	
Set q_0	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	
Set q_0	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	
Set q_0	0 0 0 0 1	
Shift	0 0 0 1 0	0 0 0 1
Subtract	1 1 1 0 1	0 0 1 □
Set q_0	1 1 1 1 1	
Restore	1 1	
	0 0 0 1 0	0 0 1 0

Remainder Quotient

10
11) 1000
11
—
10

The diagram illustrates the process of restoring division through four cycles. The dividend is 1000, and the divisor is 11. The quotient is 10, and the remainder is 10. The steps are as follows:

- First cycle:** Initial values. Subtraction results in a borrow (0 0 0 0 1 - 1 1 1 0 1 = 0 0 0 1 0).
- Second cycle:** After setting q_0 , the quotient bit is 1. Subtraction results in a borrow (0 0 0 1 0 - 1 1 1 0 1 = 0 0 0 0 1).
- Third cycle:** After setting q_0 , the quotient bit is 0. Subtraction results in a borrow (0 0 0 0 1 - 1 1 1 0 1 = 0 0 0 1 0).
- Fourth cycle:** After setting q_0 , the quotient bit is 1. Subtraction results in a borrow (0 0 0 1 0 - 1 1 1 0 1 = 0 0 1 0).

Arrows indicate the flow of bits from the previous step's result to the current subtraction and set operation. The quotient is built up as 1 1 1 1 1 0 0 1 0.



Restoring Division

Restoring Division			
Count	A	[Dividend]	$B \div 3$
4 Initial	0 0 0 0 0	1 0 0 0	$\begin{array}{r} 10 \\ 11 \longdiv{1000} \\ \underline{-11} \\ 10 \end{array}$
	0 0 0 1 1 [M]		
Shift left Aq	0 0 0 0 1	0 0 0 □	Divisor
Subtract ($A < A-M$)	1 1 1 0 1		$M = 0 0 0 1 1$
$A < 0; Q_0 < 0$	① 1 1 0		$R' \Rightarrow 1 1 1 0 0$
ADD ($A < A-M$) Restore	0 0 0 1 1	0 0 0 [0]	(R) 2's 1 1 1 0 1
?			1
Shift left Aq	0 0 0 1 0	0 0 1 0 □	
Subtract ($A < A-M$)	1 1 1 0 1		
$A < 0; Q_0 < 0$	① 1 1 1 1		
ADD ($A < A-M$) Restore	1 1	0 0 0 1 0	
		0 0 0 0 0	
2			
Shift left Aq	0 0 1 0 0	0 0 0 0 □	
Subtract ($A < A-M$)	1 1 1 0 1		
$A > 0; Q_0 < 1$	② 0 0 0 1		
		0 0 0 0 1	
1			
Shift left Aq	0 0 0 1 0	0 0 0 1 □	
Subtract ($A < A-M$)	1 1 1 0 1		
$A < 0; Q_0 < 0$	③ 1 1 1 1		
ADD ($A < A-M$) Restore	1 1	0 0 0 1 0	
		0 0 0 0 0	
0			
		Remainder Quotient	
	[0 0 1 0]	[0 0 1 0]	



Non-Restoring Division

- Initially Dividend is loaded into register Q,
and n-bit Divisor is loaded into register M
- Let M' is 2's complement of M
- Set Register A to 0
- Set count to n
- SHL AQ denotes shift left AQ by one position leaving Q_0 blank.
- Similarly, a square symbol in Q_0 position denote, it is to be calculated later



Non-Restoring Division

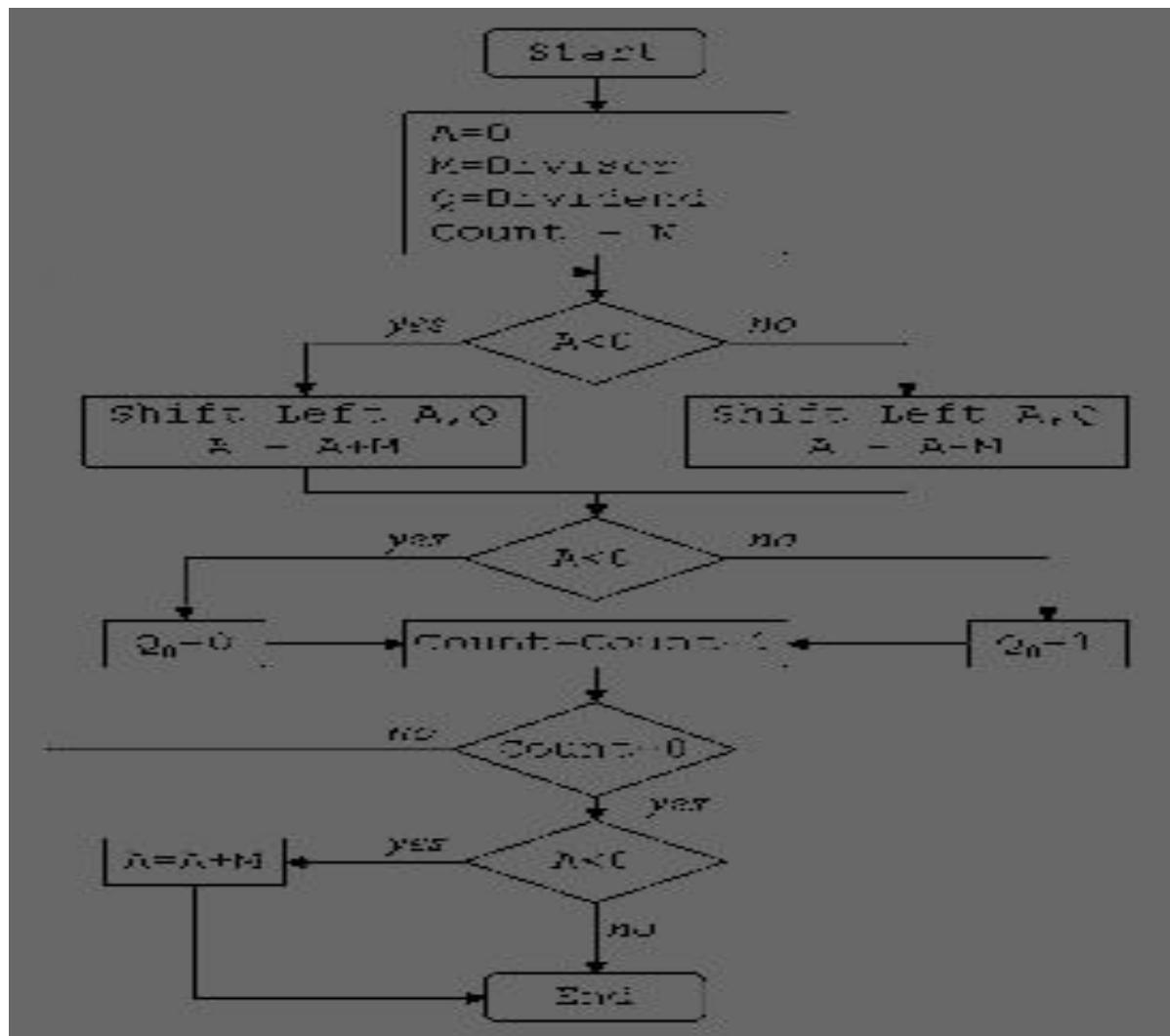


Figure: Flowchart for Non-Restoring Division



Non-Restoring Division

Restoring division can be improved using non-restoring algorithm

The effect of restoring algorithm actually is:

If A is positive, we shift it left and subtract M , that is compute $2A - M$

If A is negative, we restore it $(A + M)$, shift it left, and subtract M , that is, $2(A + M) - M = 2A + M$.

Set q_0 to 1 or 0 appropriately.

Non-restoring algorithm is:

Set A to 0.

Repeat n times:

If the sign of A is positive:

Shift A and Q left and subtract M . Set q_0 to 1.

Else if the sign of A is negative:

Shift A and Q left and add M . Set q_0 to 0.

If the sign of A is 1, add A to M .



Non-Restoring Division

$\begin{array}{r} 10 \\ 11) 1000 \\ \hline 11 \\ \hline 10 \end{array}$	<table border="0" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%;">Initially</td><td style="width: 30%; text-align: center;">0 0 0 0 0</td><td style="width: 30%; text-align: center;">1 0 0 0</td><td rowspan="4" style="width: 15%; vertical-align: middle; font-size: 2em;">}</td></tr> <tr> <td>Shift</td><td style="text-align: center;">0 0 0 0 1</td><td style="text-align: center;">0 0 0 □</td></tr> <tr> <td>Subtract</td><td style="text-align: center;"><u>1 1 1 0 1</u></td><td></td></tr> <tr> <td>Set q_0</td><td style="text-align: center;">1 1 1 1 0</td><td style="text-align: center;">0 0 0 0</td></tr> </table>	Initially	0 0 0 0 0	1 0 0 0	}	Shift	0 0 0 0 1	0 0 0 □	Subtract	<u>1 1 1 0 1</u>		Set q_0	1 1 1 1 0	0 0 0 0
Initially	0 0 0 0 0	1 0 0 0	}											
Shift	0 0 0 0 1	0 0 0 □												
Subtract	<u>1 1 1 0 1</u>													
Set q_0	1 1 1 1 0	0 0 0 0												
	<table border="0" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%;">Shift</td><td style="width: 30%; text-align: center;">1 1 1 0 0</td><td style="width: 30%; text-align: center;">0 0 0 □</td><td rowspan="4" style="width: 15%; vertical-align: middle; font-size: 2em;">}</td></tr> <tr> <td>Add</td><td style="text-align: center;"><u>0 0 0 1 1</u></td><td></td></tr> <tr> <td>Set q_0</td><td style="text-align: center;">1 1 1 1 1</td><td style="text-align: center;">0 0 0 0</td></tr> <tr> <td></td><td></td><td></td></tr> </table>	Shift	1 1 1 0 0	0 0 0 □	}	Add	<u>0 0 0 1 1</u>		Set q_0	1 1 1 1 1	0 0 0 0			
Shift	1 1 1 0 0	0 0 0 □	}											
Add	<u>0 0 0 1 1</u>													
Set q_0	1 1 1 1 1	0 0 0 0												
	<table border="0" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%;">Shift</td><td style="width: 30%; text-align: center;">1 1 1 1 0</td><td style="width: 30%; text-align: center;">0 0 0 □</td><td rowspan="4" style="width: 15%; vertical-align: middle; font-size: 2em;">}</td></tr> <tr> <td>Add</td><td style="text-align: center;"><u>0 0 0 1 1</u></td><td></td></tr> <tr> <td>Set q_0</td><td style="text-align: center;">0 0 0 0 1</td><td style="text-align: center;">0 0 0 1</td></tr> <tr> <td></td><td></td><td></td></tr> </table>	Shift	1 1 1 1 0	0 0 0 □	}	Add	<u>0 0 0 1 1</u>		Set q_0	0 0 0 0 1	0 0 0 1			
Shift	1 1 1 1 0	0 0 0 □	}											
Add	<u>0 0 0 1 1</u>													
Set q_0	0 0 0 0 1	0 0 0 1												
	<table border="0" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%;">Shift</td><td style="width: 30%; text-align: center;">0 0 0 1 0</td><td style="width: 30%; text-align: center;">0 0 1 □</td><td rowspan="4" style="width: 15%; vertical-align: middle; font-size: 2em;">}</td></tr> <tr> <td>Add</td><td style="text-align: center;"><u>1 1 1 0 1</u></td><td></td></tr> <tr> <td>Set q_0</td><td style="text-align: center;">1 1 1 1 1</td><td style="text-align: center;">0 0 1 0</td></tr> <tr> <td></td><td></td><td></td></tr> </table>	Shift	0 0 0 1 0	0 0 1 □	}	Add	<u>1 1 1 0 1</u>		Set q_0	1 1 1 1 1	0 0 1 0			
Shift	0 0 0 1 0	0 0 1 □	}											
Add	<u>1 1 1 0 1</u>													
Set q_0	1 1 1 1 1	0 0 1 0												
	$\overbrace{\begin{array}{r} 1 1 1 1 1 \\ 0 0 0 1 1 \\ \hline 0 0 0 1 0 \end{array}}^{\text{Quotient}}$	$\overbrace{\begin{array}{r} \\ \\ \\ \\ \end{array}}^{\text{Remainder}}$	}											
	<table border="0" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%;">Add</td><td style="width: 30%; text-align: center;">1 1 1 1 1</td><td style="width: 30%; text-align: center;">0 0 0 1 0</td><td rowspan="2" style="width: 15%; vertical-align: middle; font-size: 2em;">}</td></tr> <tr> <td></td><td style="text-align: center;"><u>0 0 0 1 1</u></td><td></td></tr> </table>	Add	1 1 1 1 1	0 0 0 1 0	}		<u>0 0 0 1 1</u>		}	}				
Add	1 1 1 1 1	0 0 0 1 0	}											
	<u>0 0 0 1 1</u>													
		$\overbrace{\begin{array}{r} \\ \\ \\ \\ \end{array}}^{\text{Remainder}}$												



Non-Restoring Division

Non-Restoring Division			
Count	A	Q (Dividend)	$\frac{A}{M}$
4 Initial	0 0 0 0 0	1 0 0 0	$\begin{array}{r} 1 0 \\ 1 1 \end{array} \overline{) 1 0 0 0} \\ \underline{- 1 0} \\ 1 0 \end{array}$
	0 0 0 1 1 (M)	0 0 0 □	
	$A \rightarrow +ve; \text{Shift left } A \&$ $A < A-M; \cancel{\text{Subtract}}$		
3	1 1 1 0 1	0 0 0 □	Divisor
	$A < 0, Q_0 < 0$ $A \rightarrow -ve, \text{Shift left } A \&$ $A < A+M; \text{Add}$		$M = 0 0 0 1 1$ 1's 1 1 1 0 0
	1 1 1 0 0	0 0 0 □	
	0 0 0 1 1	0 0 0 □	2's 1 1 1 0 1
2	1 1 1 1 1	0 0 □ □	
	$A \rightarrow -ve, \text{Shift left } A \&$ $A < A+M, \text{Add}$		
	1 1 1 1 0	0 0 □ □	
	0 0 0 1 1	0 0 □ □	
	$A \neq 0; Q_0 < 1$ (+ve)		
1	0 0 0 1 0	0 0 □ 1 □	
	$A \rightarrow +ve; \text{Shift left } A \&$ $A < A-M; \text{Subtract}$		
	1 1 1 0 1	0 0 □ 1 □	
	$A < 0; Q_0 < 0$		
	1 1 1 1 1	0 0 □ 1 □	
0	$A \rightarrow -ve; \text{ADD}$ $A < A+M$		
	0 0 0 1 1	0 0 □ 1 □	
	0 0 0 1 0	0 0 □ 1 □	
	Remainder Quotient		
	[0 0 1 0 0 0 1 0]		



Floating Point Numbers and Operations

Floating Point Number Representation (FPR)

\pm Significant \times Base^{+Exponent}

Example: $+10.55 \times 10^{55}$

Why FPR?

- Consider a very small number 0.00000000005
- As it consists of many 0's after the decimal point, it requires more number bits for representation
- 0.0000000005 is represented as 0.5×10^{-10} , it will require only fewer bits
- Consider a very large number 50000000000 which can be represented as $5 \times 10^{10} (+5)$
- **Consider the example $0.123 \times 10^4 = 0.0123 \times 10^5 = 123 \times 10^3$**
- But we need a fixed and single representation for floating point numbers. For this normalization is required.



Floating Point Numbers and Operations

Normalization rules for floating point number

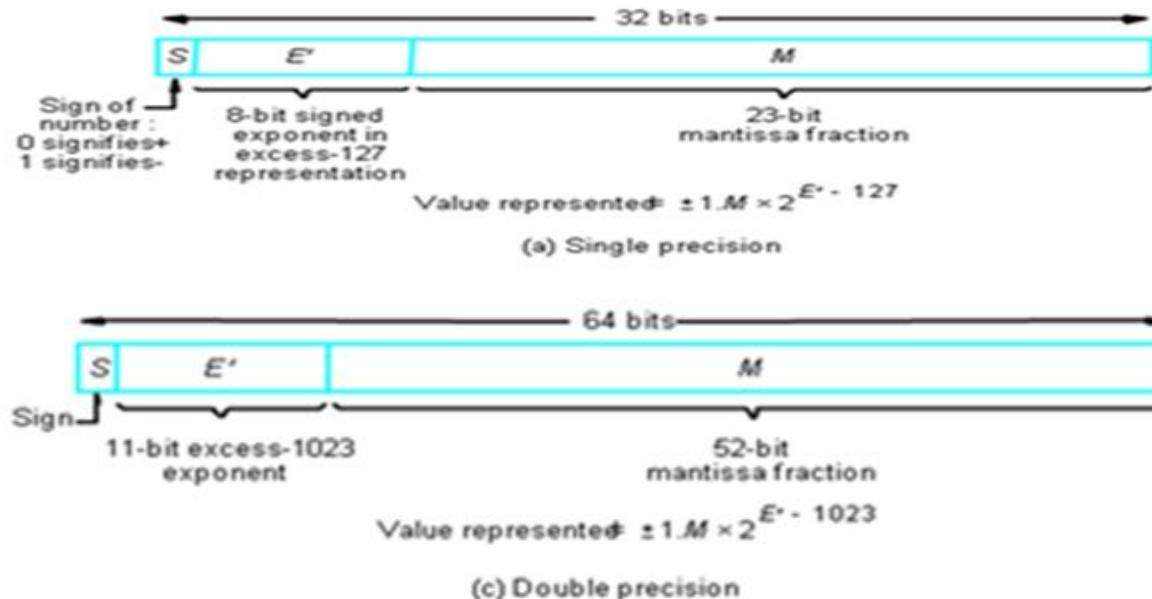
There are 2 rules:

- (1) The integer part should be 0
- (2) $0.d_1d_2\dots d_n \times B^{\pm E}$ then $d_1 > 0$ and all $d_i \geq 0$

In the example $0.123 \times 10^4 = \cancel{0.0123} \times \cancel{10^5} = \cancel{123} \times \cancel{10^3}$, the strikethrough representations are wrong with respect to normalization rules.

Two floating point representation techniques are,

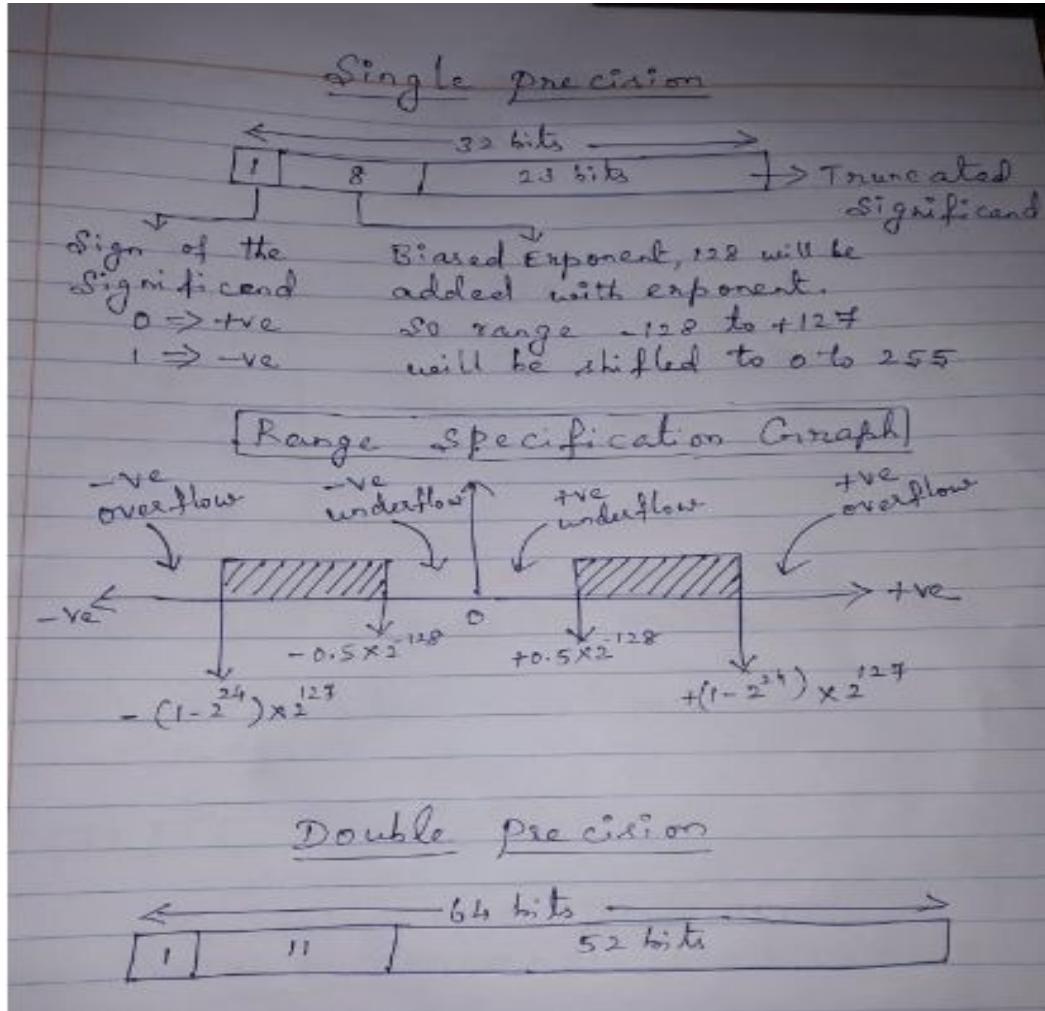
- (1) Single precision (32 bits or 4 bytes)
- (2) Double precision (64 bits or 8 bytes)





Floating Point Numbers and Operations

Single Precision and Double Precision Format





Floating Point Numbers and Operations

Decimal number into IEEE 754 32-bit floating point number

To represent a number in IEEE 754 32-bit floating point notation

263.3

2 | 263
2 | 131 - 1
2 | 65 - 1
2 | 32 - 1
2 | 16 - 0
2 | 8 - 0
2 | 4 - 0
2 | 2 - 0
1 - 0

263 : 1 00000111
0.3 : 0100110011...

$0.3 \times 2 | 0.6 | 0$
 $0.6 \times 2 | 1.2 | 1$
 $0.2 \times 2 | 0.4 | 0$
 $0.4 \times 2 | 0.8 | 0$
 $0.8 \times 2 | 1.6 | 1$
 $0.6 \times 2 | 1.2 | 1$
0
0
1
:

(1) 263.3 \rightarrow 100000111.0100110011... 1

(2) Scientific notation \downarrow
1.000001110100110011... $\times 2^8$
Mantissa

(3) [1] [8] [23] $\xrightarrow{32\text{ bits}}$ Exponent bias
 $127 + 8 = 135$ \rightarrow 127

0 10001110000011101000110011...

263.3 \rightarrow 0100 0011 1000 0011 1010 0110 0110 0110
↓
Decimal Number 32-bit IEEE 754 number



Floating Point Numbers and Operations

Floating Point Arithmetic Addition/Subtraction

Steps to add/subtract two floating point numbers:

1. Compare the magnitudes of the exponents and make suitable alignment to the number with the smaller magnitude of exponent
2. Perform the addition/subtraction
3. Perform normalization by shifting the resulting mantissa and adjusting the resulting exponent

Example: Add 1.1100×2^4 and 1.1000×2^2

1. Alignment: 1.1000×2^2 has to be aligned to 0.0110×2^4 1.1100

2. Addition: Add the numbers to get 10.0010×2^4 0.0110

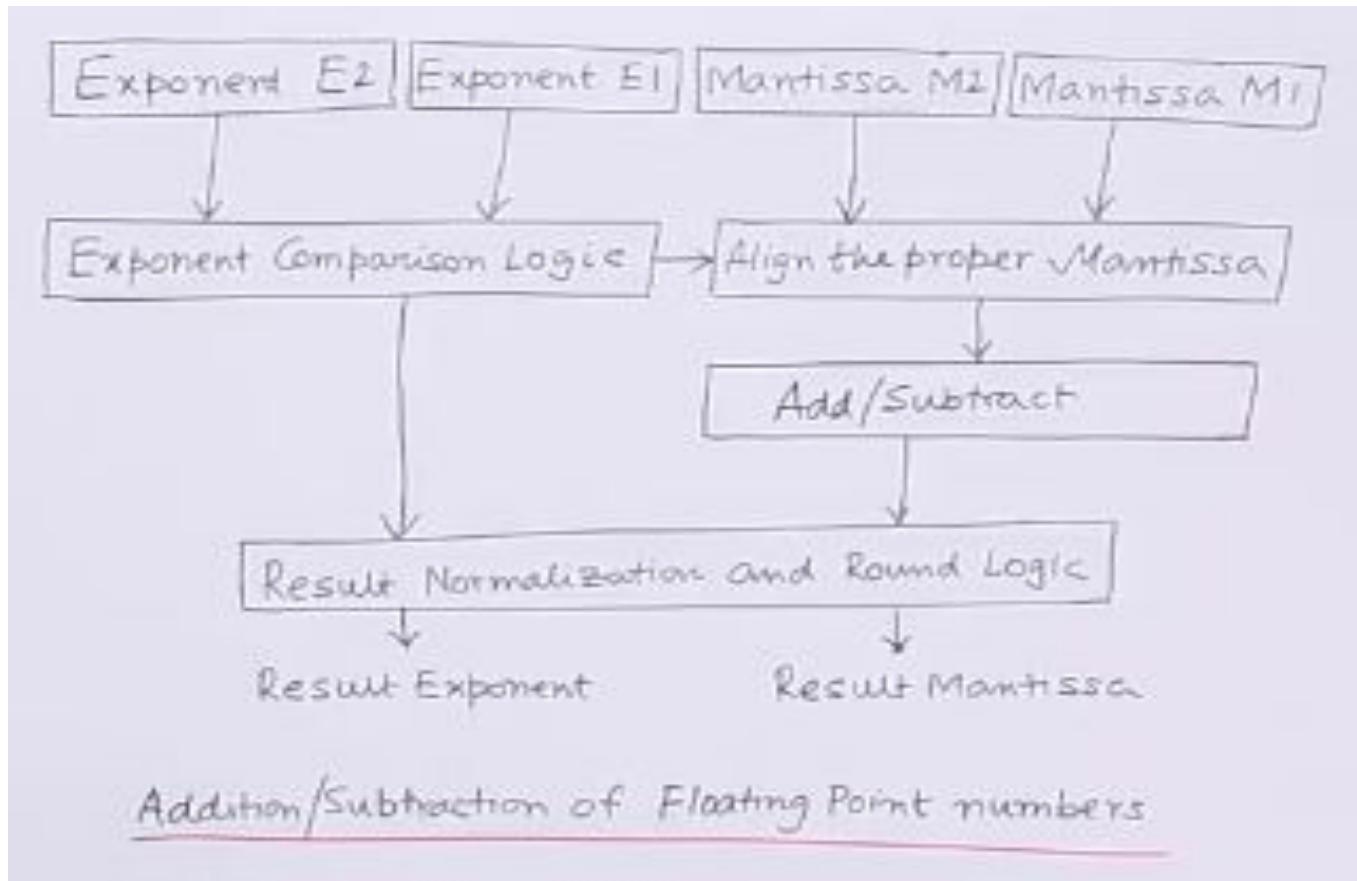
3. Normalization: Find normalized result 10.0110

0.1000×2^6 (assuming 4-bits are allowed after decimal point)



Floating Point Numbers and Operations

Floating Point Arithmetic Addition/Subtraction





Reference Links

Ripple Carry Adder

- <https://www.gatevidyalay.com/tag/advantages-of-ripple-carry-adder/>

Floating Point Numbers and Operations

- <https://www.youtube.com/watch?v=XOMTNy2qiZ0&t=72s>
- <https://www.youtube.com/watch?v=8afbTaA-gOQ>
- <https://www.youtube.com/watch?v=w7NQTb1FTDU>

18CSC203J – COMPUTER ORGANIZATION AND ARCHITECTURE

UNIT-III

Course Outcome

CLR-3: Understand the concepts of Pipelining and basic processing units

CLO-3 : Analyze the detailed operation of Basic Processing units and the performance of Pipelining

Topics Covered

- Fundamental concepts of basic processing unit
- Performing ALU operation
- Execution of complete instruction, Branch instruction
- Multiple bus organization
- Hardwired control,
- Generation of control signals
- Micro-programmed control, Microinstruction
- Micro-program Sequencing
- Micro instruction with Next address field
- Basic concepts of pipelining
- Pipeline Performance
- Pipeline Hazards-Data hazards, Methods to overcome Data hazards
- Instruction Hazards
- Hazards on conditional and Unconditional Branching
- Control hazards
- Influence of hazards on instruction sets

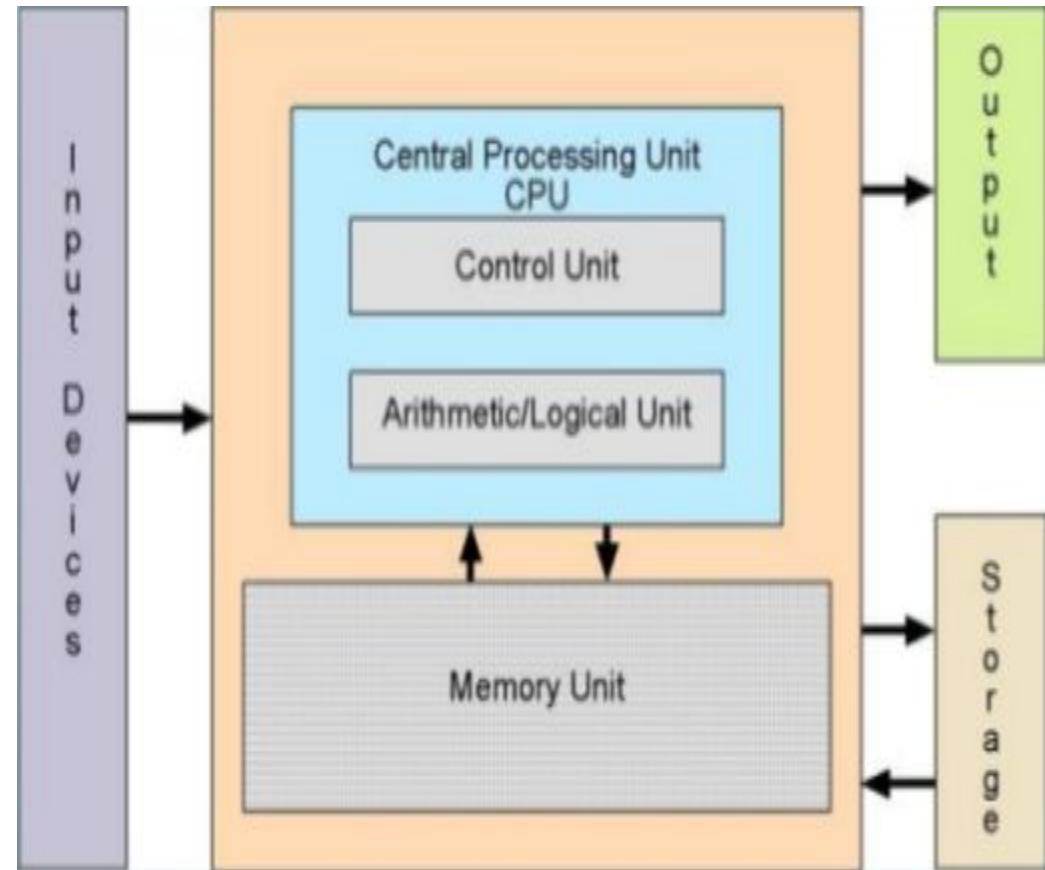
PROCESSING UNIT

FUNCTIONS OF CPU:

- CPU carries out all forms of data processing tasks.
- It saves information, intermediate results and instructions.
- CPU monitors the functionality of all computer components.

COMPONENTS OF CPU:

- Memory or storage unit:** The memory unit stores all the instructions and data. This unit provides data to other units of the computer if necessary. Also known as main memory, or **RAM** (Random Access Memory).
- Control unit:** This unit monitors all computing processes but does not execute actual data processing.
- Arithmetic Logic Unit (ALU):** This does all the calculations and makes the decisions.



FUNDAMENTAL CONCEPTS OF BASIC PROCESSING UNIT

- Processor fetches one instruction at a time and perform the specified operation.
- Instructions are fetched from successive memory locations except for branch/ jump instruction.
- The address of the next instruction to be executed is tracked by the Program Counter (PC) register.
- Instruction Register (IR) contains instruction that is currently executed.
- Instruction execution happens in three phases:
 - ✓ Fetch: Fetch the instruction from the specified memory
 - ✓ Decode: Determined the opcode and the operands
 - ✓ Execute: Run the instruction

EXECUTING AN INSTRUCTION

- Fetch the contents of memory location pointed by the PC. The contents of this memory location is loaded to the **IR-Fetch phase**
 $\text{IR} \leftarrow [\text{PC}]$
- Increment the PC by 4 (assume the word size as 4)
 $\text{PC} \leftarrow [\text{PC}] + 4$
- Carry out the actions specified by the instruction in the **IR-Execution phase**
- Datapath:** collection of functional units such as ALU or multipliers that perform data processing operations.
- MDR:** Two inputs and two outputs since data can be loaded from memory or processor bus.
- MAR:** Input line is connected to internal bus and output line to external bus
- Control lines:** connected to instruction decoder and control logic block to issue control signals
- R0-R(n-1):** General Purpose registers whose numbers vary between processors.
- TEMP, Y and Z:** temporary registers used by the processor during instruction execution

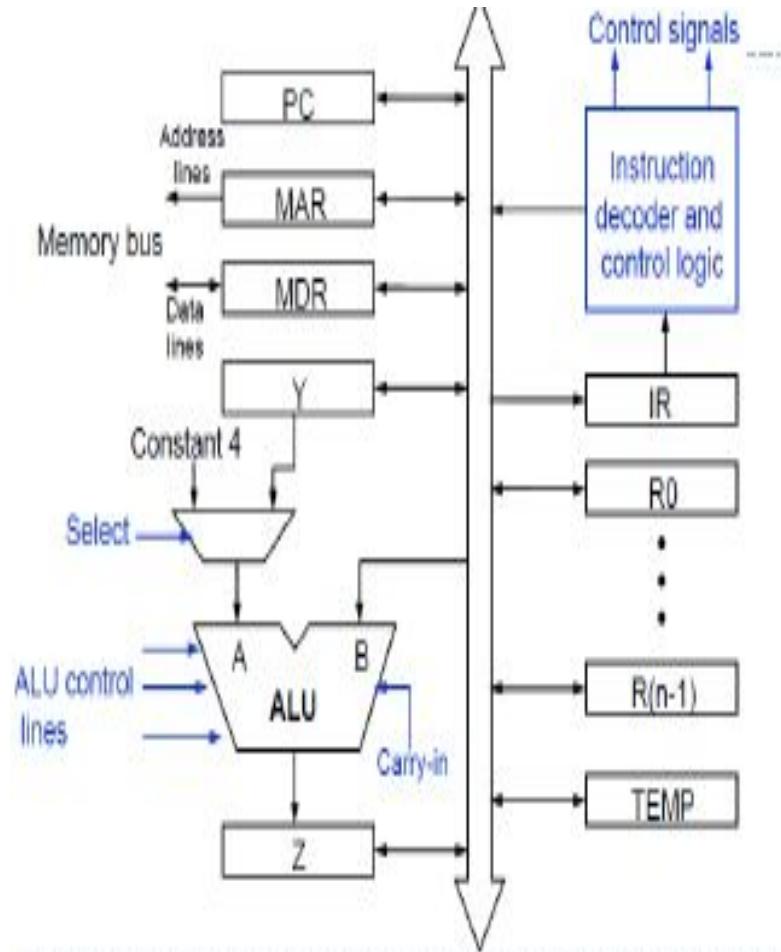


Fig : Single bus organization of datapath

- Transfer a word of data from one processor register to another or to the ALU
- Perform Arithmetic and Logic operation and store the result in processor register
- Fetch the contents of a given memory location and load the, into processor register.
- Store a word of data from a processor register into a given memory location

REGISTER TRANSFERS

- For each register two control signals are used :
 - ✓ To place the contents of that register on the bus
 - ✓ To load the data on the bus into register
- The input and output of register are connected to the bus through switches controlled by the signals Rin and Rout.

PERFORMING AN ARITHMETIC OR LOGIC OPERATION

- The ALU is a combinational circuit that has no internal storage.
- ALU gets the two operands from MUX and bus. The result is temporarily stored in register Z.
- The sequence of operations to add contents of R1 and R2 and store it in R3:
 1. $R1_{out}, Y_{in}$
 2. $R2_{out}, \text{Select}Y, \text{Add}, Z_{in}$
 3. $Z_{out}, R3_{in}$

FETCHING WORD FROM MEMORY

- Address will be loaded into MAR from PC
- Read operation will be issued in the bus and the data will be loaded into MDR.
- Wait for Memory Function Completed (WMFC):** This signal is issued to make the processor to wait for the reply from the memory (MFC).

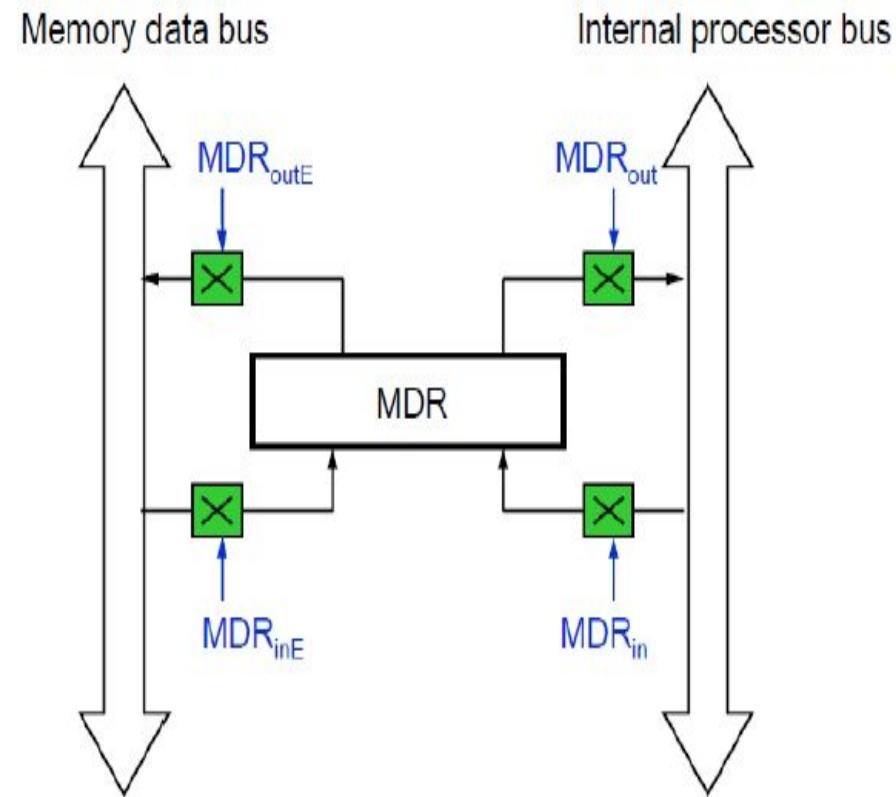


Fig: Fetching word form Memory

Move (R1), R2

1. $R1_{out}, \text{MAR}_{in}, \text{Read}$
2. $\text{MDR}_{inE}, \text{WMFC}$
3. $\text{MDR}_{out}, R2_{in}$

EXECUTION OF COMPLETE INSTRUCTION

- Initiate instruction fetch by loading contents of PC to MAR.
- Send Read request to memory.

Incrementing PC

- Contents of PC is loaded to register B.
- Set select signal to 4, which make multiplexer to select const 4, which is then added to B. The result is available in Z.
- The content of Z is loaded to PC.

Load IR

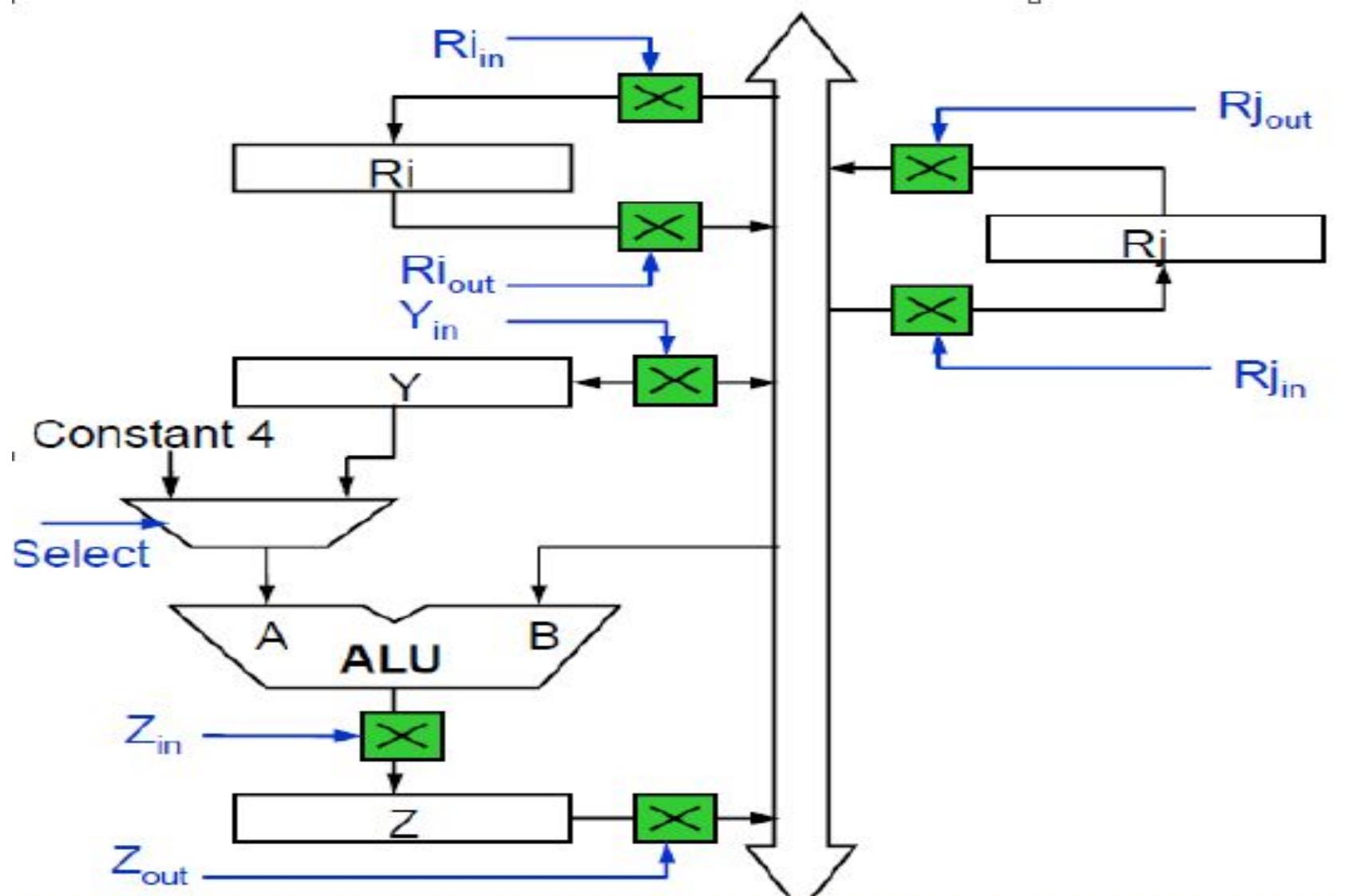
- The word is fetched from the memory and loaded into IR
- Decoding: Instruction decoding circuit interprets contents of IR.
- Execution: Control circuitry is activated to issue relevant control signals.
- Load MAR with contents of R3. Initiate memory read. Transfer R1 to Y.

ADD (R3), R1

Step	Action
1	$PC_{out}, MAR_{in}, Read, Select4 Add, Z_{in}$
2	$Z_{out}, PC_{in}, Y_{in}, WMFC$
3	MDR_{out}, IR_{in}
4	$R3_{out}, MAR_{in}, Read$
5	$R1_{out}, Y_{in}, WMFC$
6	$MDR_{out}, SelectY, Add, Z_{in}$
7	$Z_{out}, R1_{in}, End$

- Perform addition on Y and MDR after selecting Y.
- Load the result from Z to R1.

SIGNAL DIAGRAM



EXECUTION OF BRANCH INSTRUCTIONS

- Branch instruction replaces the contents of PC with the branch target address, which is usually obtained by adding an offset X given in the branch instruction.
- The offset X is usually the difference between the branch target address and the address immediately following the branch instruction.
- Two types of branch: conditional and unconditional branch

UnConditional Branch:

- **Offset value** : branch target address –address immediately following the branch instruction.
- Offset value is obtained from IR and is available in X.
- Y contains the updated PC value. Add the offset address along with PC to find the branch target.
- Now load the calculated branch target from Z to PC.

Step	Action
1	$PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}$
2	$Z_{out}, PC_{in}, Y_{in}, WMFC$
3	MDR_{out}, IR_{in}
4	$Offset\text{-}field\text{-}of\text{-}IR_{out}, Add, Z_{in}$
5	Z_{out}, PC_{in}, End

EXECUTION OF CONDITIONAL BRANCH

- Check status of condition codes before loading branch target into PC

Condition Codes

- The CPU maintains a set of single-bit *condition code* registers.
- These registers are set by arithmetic and logical operations and can be tested to perform conditional branches.

Examples:

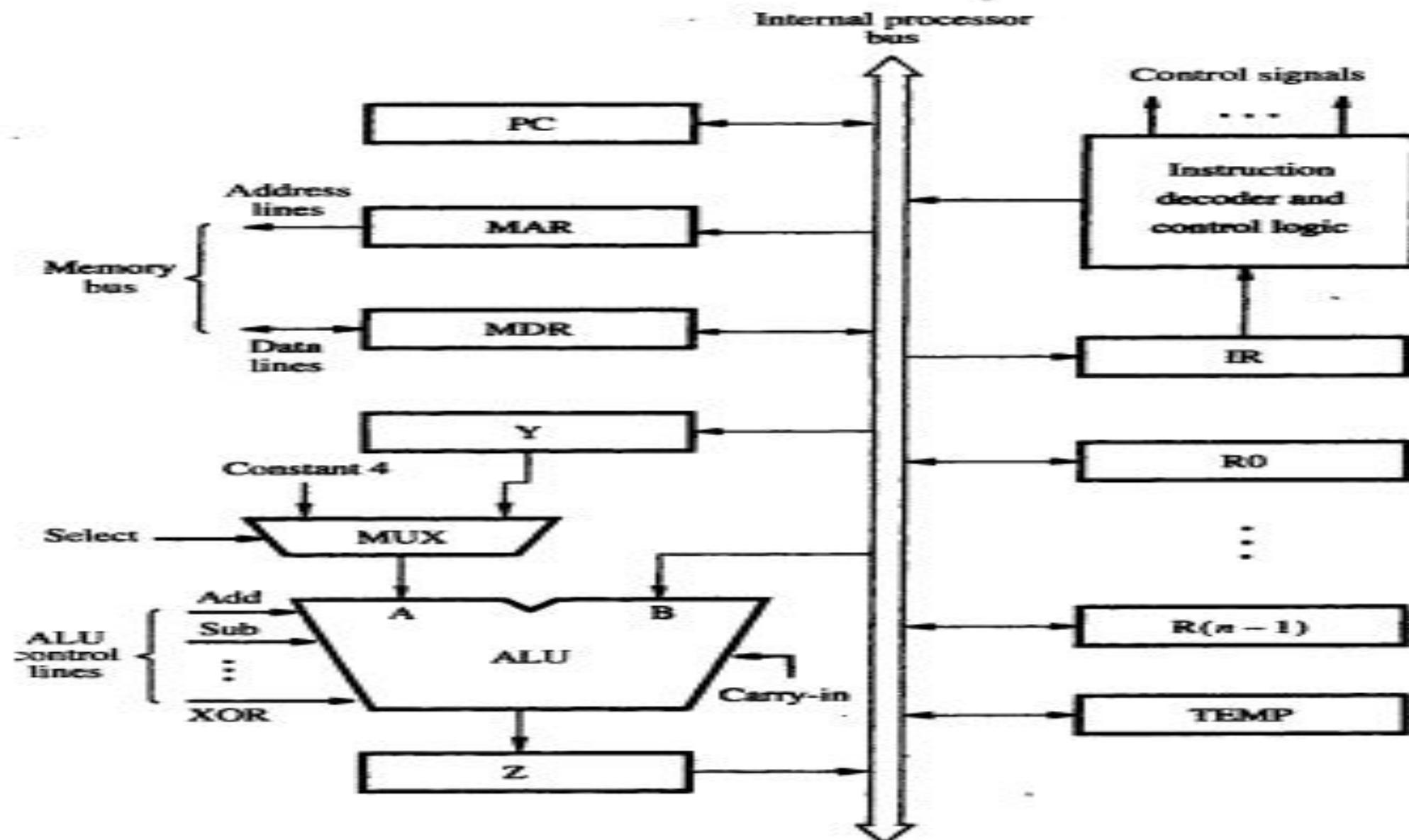
- ✓ **Carry flag.** If set, the most recent operation generated a carry out of the most significant bit. Use to detect overflow for unsigned operations.
- ✓ **Zero flag.** The most recent operation yielded zero.
- ✓ **Sign flag.** The most recent operation yielded a negative value.
- ✓ **Overflow flag.** The most recent operation caused a two's complement overflow – either negative or positive. **Offset-field-of-IR_{out}, Add, Z_{in}, If N = 0 then End**

Branch on negative

- If N=0 then the processor return to step 1 after step 4.
- If N=1 then load a new value into PC to perform branching.

Multiple bus organization

SINGLE BUS ORGANISATION OF A DATAPATH



MULTIPLE BUS ORGANIZATION

- Single bus structure takes long time for transfer of data in a clock cycle
- To avoid the above said process ,multiple internal paths are enabled in commercial processors .
- This allows to have several transfers to take place in parallel

Multi bus

- The next slide has the three bus structure which is connecting registers and the ALU of processors
- Register file: is a single block of all general purpose registers
- Register file has three ports.
- First Two output ports are connected to different registers placed on the Bus A and Bus B.
- One more third port connected to Bus C during the same clock cycle.

Cont...

- Buses A and B used to Transfer the source operands to A and B inputs of ALU.(Arithmetic and logic operations are performed)
- The result is transferred over the Bus C.
- Three bus arrangement avoids the usage of registers Y and Z.
- Incrementer Unit: is used to increment the PC by 4 which eliminates the usage of ALU.

Cont...

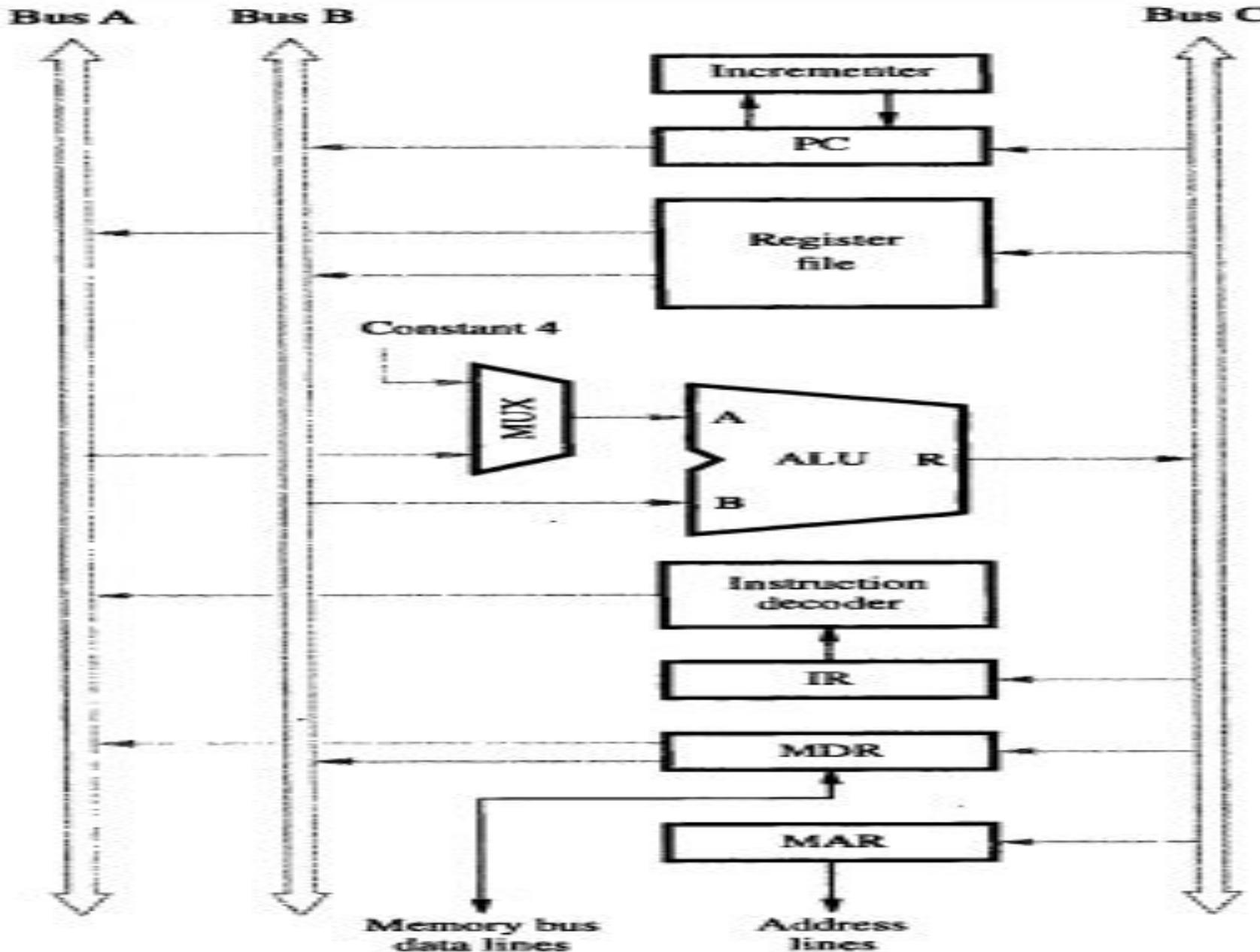
- It increments the other addresses such as the memory addresses in LoadMultiple and StoreMultiple instructions.
- Three operand instruction
- Add R4,R5,R6
- STEP1: The contents of the PC are passed through the ALU using R=B control signal and loaded into MAR for(Memory Read operation) and the PC is incremented by 4

Cont..

- Step2: The processor waits for MFC (WMFC-Wait for Memory Function completed)
- STEP3: MFC loads the data received into MDR. And again data is transferred into IR
- STEP4: Final execution phase requires only one control step

Providing more paths for data transfer results in reduction of clock cycles needed to execute an instruction.

MULTIPLE BUS ORGANISATION



Multiple-Bus Organization

- Add R4, R5, R6
-

Step	Action
1	PC _{out} , R=B, MAR _{in} , Read, IncPC
2	WMF C
3	MDR _{outB} , R=B, IR _{in}
4	R4 _{outA} , R5 _{outB} , SelectA, Add, R6 _{in} , End

Figure 7.9. Control sequence for the instruction. Add R4,R5,R6,
for the three-bus organization in Figure 7.8.

Hardwired Control

Overview

- To execute instructions, the processor must have some means of generating the control signals needed in the proper sequence.
- Two categories: hardwired control and microprogrammed control
- Hardwired system can operate at high speed; but with little flexibility.

7 Steps:

Cont...

- The before slide depicts the sequence of control signals
- Every step is completed in one clock period
- Counter is used to keep track of the control steps.
- Control signals are determined by following information
 1. Contents of the control step counter
 2. Contents of instruction register
 3. Contents of condition code flags
 4. External input signals, such as MFC and interrupt requests

Decoder/encoder block

- Is a combinational circuit that generates the required control outputs depending on the state of all its input.
- The control unit organization of the decoder and encoder block is shown in the next slide

Control Unit Organization

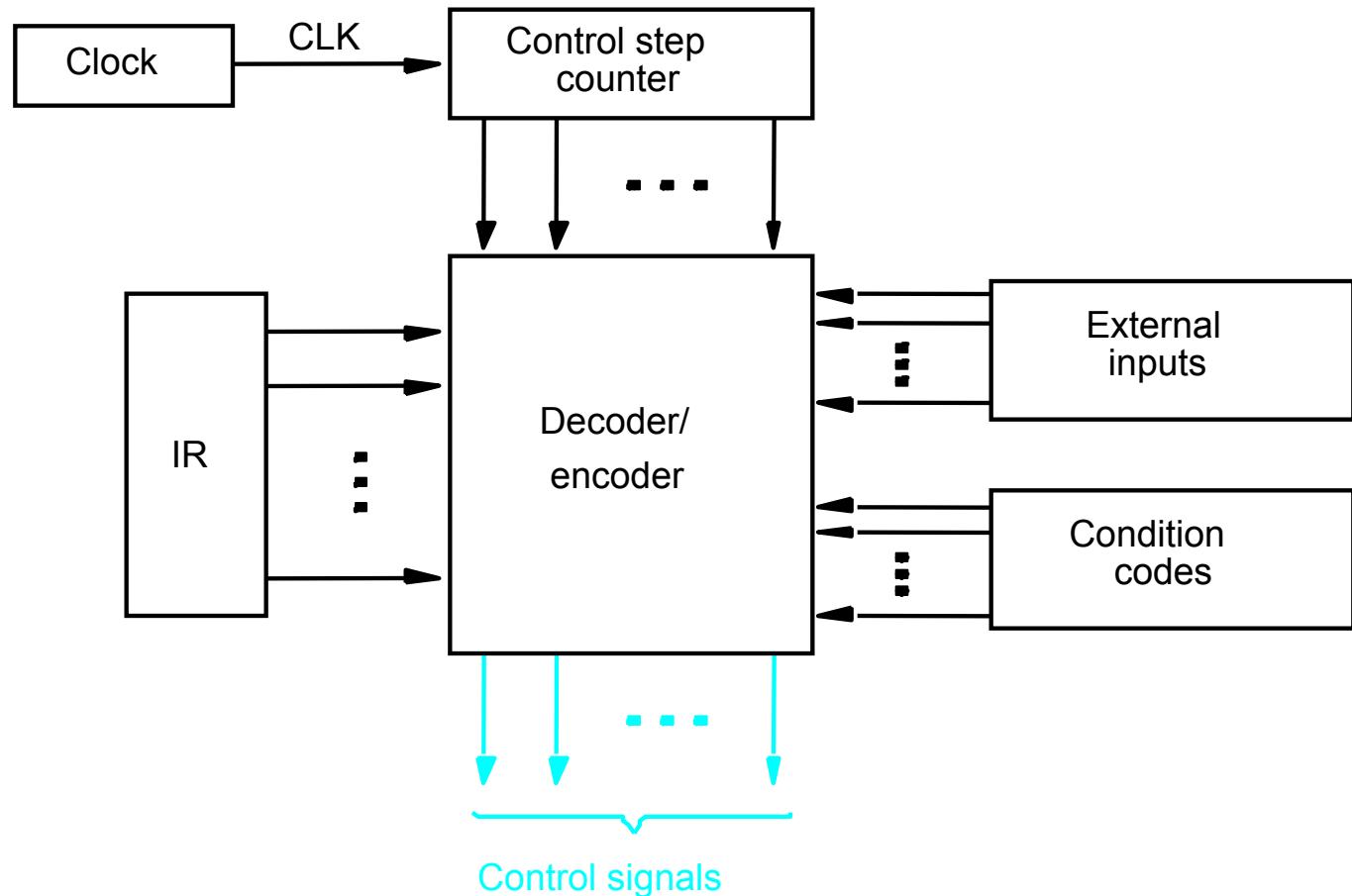


Figure 7.10. Control unit organization.

Detailed Block Description

Explanation of decoding and encoding

- The step decoder provides a separate signal line for each step or time slot in the control sequence
- Output of the instruction decoder consists of separate line for each machine instruction.
- Any instruction loaded into IR, one of the output lines INS1 through INS_m is set to 1 and all others are set to 0.
- The input signals to the encoder are combined to generate individual control signals Y_{in}, PC_{out}

Generating Z_{in}

- $Z_{in} = T_1 + T_6 \cdot \text{ADD} + T_4 \cdot \text{BR} + \dots$
- The signal is asserted during time slot T_1 for all instruction , T_6 for an ADD instruction , T_4 for unconditional branch instruction

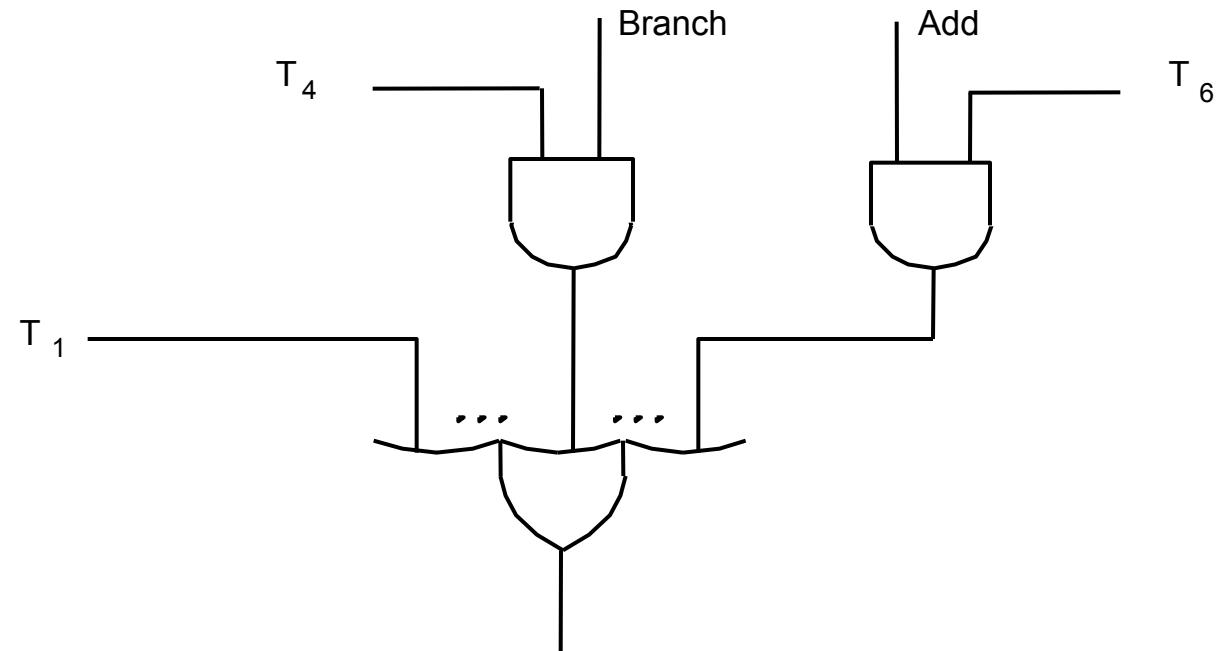


Figure 7.12. Generation of the Z_{in} control signal for the processor in Figure 7.1.

Generating End

- $\text{End} = T_7 \bullet \text{ADD} + T_5 \bullet \text{BR} + (T_5 \bullet N + T_4 \bullet N) \bullet \text{BRN} + \dots$

End signal

- End signRal starts a new instruction fetch cycle by resetting the control step counter to its starting value.
- Control signal called RUN.
- RUN set as 1 ,causes the counter to be incremented by one at the end of every clock cycle.
- RUN equal to 0 , the counter stops counting
- WMFC signal is issued to cause the processor to wait for the reply from memory

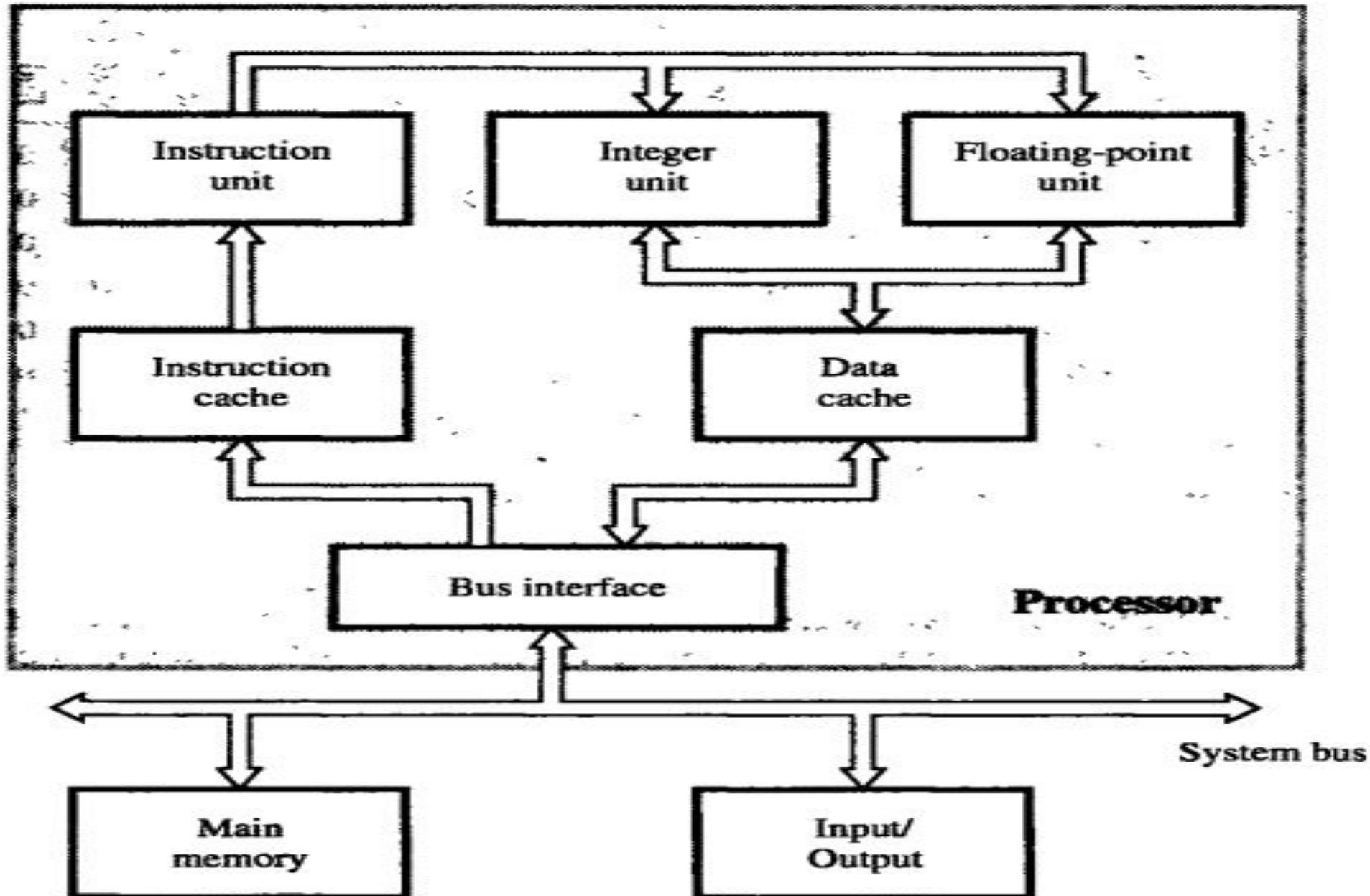
Cont...

- These control unit organization viewed as state machine(one state to another in every clock cycle)
- Output of the state machines are control signals.
- Sequence of operations carried out by machine is determined by the wiring of logic elements,hence the name “hardwired”.
- Controller operates at high speed

Complete Processor

- This structure has instruction unit that fetches from instruction cache or from the main memory.
- Processing unit to handle integer and floating point data.
- Data cache is introduced between these units and main memory
- Separate cache for instruction and data
- Processor is connected to system bus.
- Bus interface is used to connect to rest of memory and input output unit.

Complete Processor



Microprogrammed Control

A control unit whose binary control variables are stored in memory is called a micro programmed control unit.

Microprogrammed Control Unit

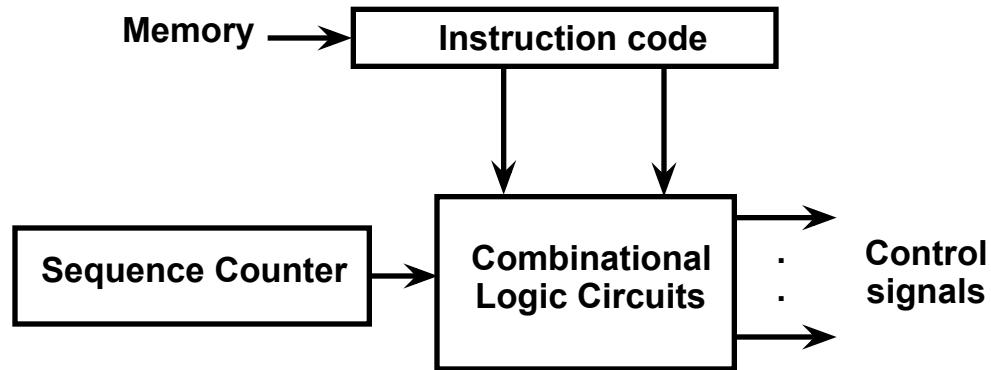
- Control signals
 - Group of bits used to select paths in multiplexers, decoders, arithmetic logic units
- Control variables
 - Binary variables specify microoperations
 - Certain microoperations initiated while others idle
- Control word
 - String of 1's and 0's represent control variables

Microprogrammed Control Unit

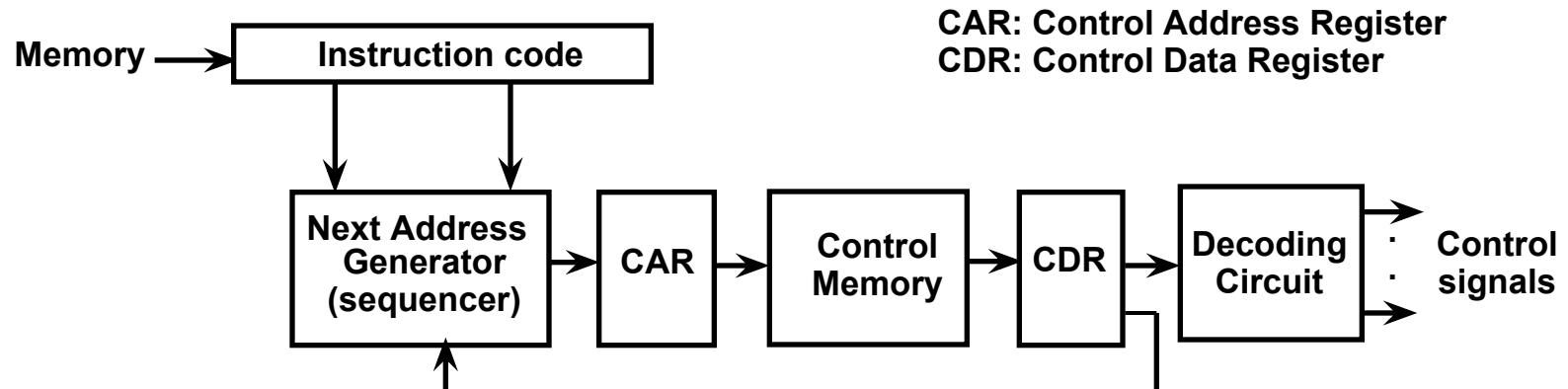
- Control memory
 - Memory contains control words
- Microinstructions
 - Control words stored in control memory
 - Specify control signals for execution of microoperations
- Microprogram
 - Sequence of microinstructions

Control Unit Implementation

- Hardwired

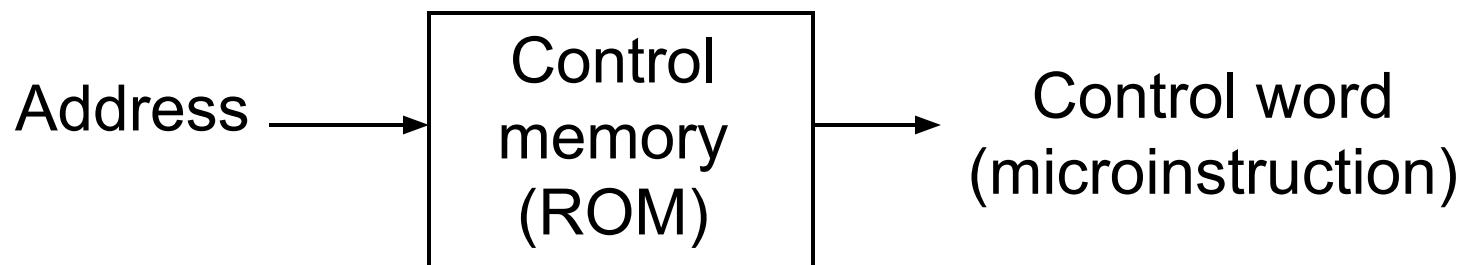


- Microprogrammed

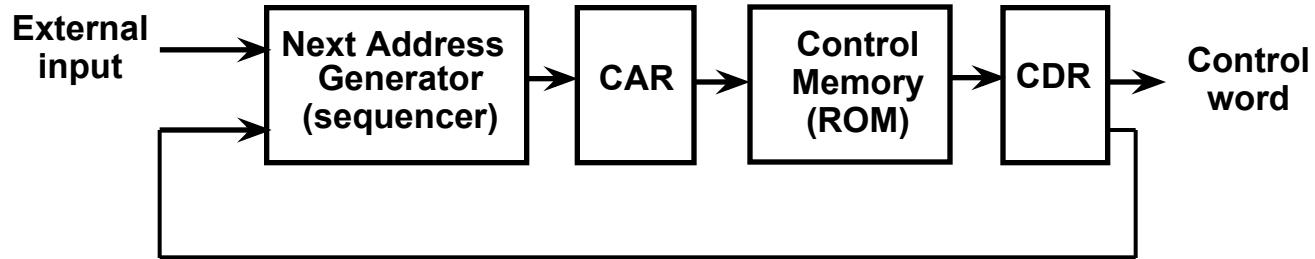


Control Memory

- Read-only memory (ROM)
- Content of word in ROM at given address specifies microinstruction
- Each computer instruction initiates series of microinstructions (microprogram) in control memory
- These microinstructions generate microoperations to
 - Fetch instruction from main memory
 - Evaluate effective address
 - Execute operation specified by instruction
 - Return control to fetch phase for next instruction



Microprogrammed Control Organization



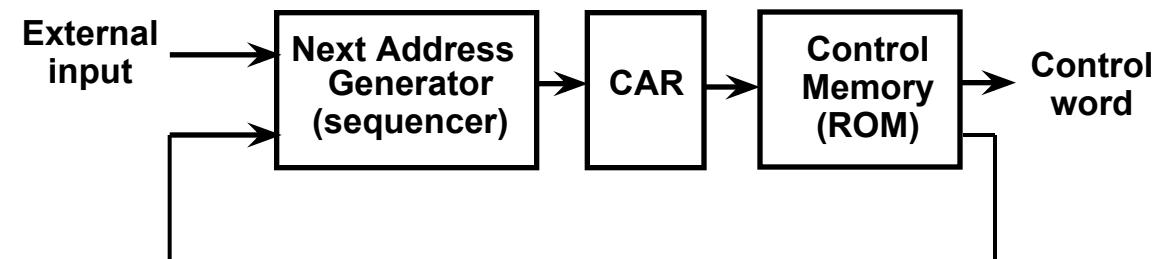
- Control memory
 - Contains microprograms (set of microinstructions)
 - Microinstruction contains
 - Bits initiate microoperations
 - Bits determine address of next microinstruction
- Control address register (CAR)
 - Specifies address of next microinstruction

Microprogrammed Control Organization

- Next address generator (microprogram sequencer)
 - Determines address sequence for control memory
- Microprogram sequencer functions
 - Increment CAR by one
 - Transfer external address into CAR
 - Load initial address into CAR to start control operations

Microprogrammed Control Organization

- Control data register (CDR)- or pipeline register
 - Holds microinstruction read from control memory
 - Allows execution of microoperations specified by control word simultaneously with generation of next microinstruction
- Control unit can operate without CDR



Microinstruction Sequencing:

A micro-program control unit can be viewed as consisting of two parts:

The control memory that stores the microinstructions.

Sequencing circuit that controls the generation of the next address.

Microinstruction Sequencing:

A micro-program sequencer attached to a control memory inputs certain bits of the microinstruction, from which it determines the next address for control memory. A typical sequencer provides the following address-sequencing capabilities:

Increment the present address for control memory.

Branches to an address as specified by the address field of the micro instruction.

Branches to a given address if a specified status bit is equal to 1.

Transfer control to a new address as specified by an external source (Instruction Register).

Has a facility for subroutine calls and returns.

Microinstruction Sequencing:

Depending on the current microinstruction condition flags, and the contents of the instruction register, a control memory address must be generated for the next micro instruction.

There are three general techniques based on the format of the address information in the microinstruction:

Two Address Field.

Single Address Field.

Variable Format

Two address field

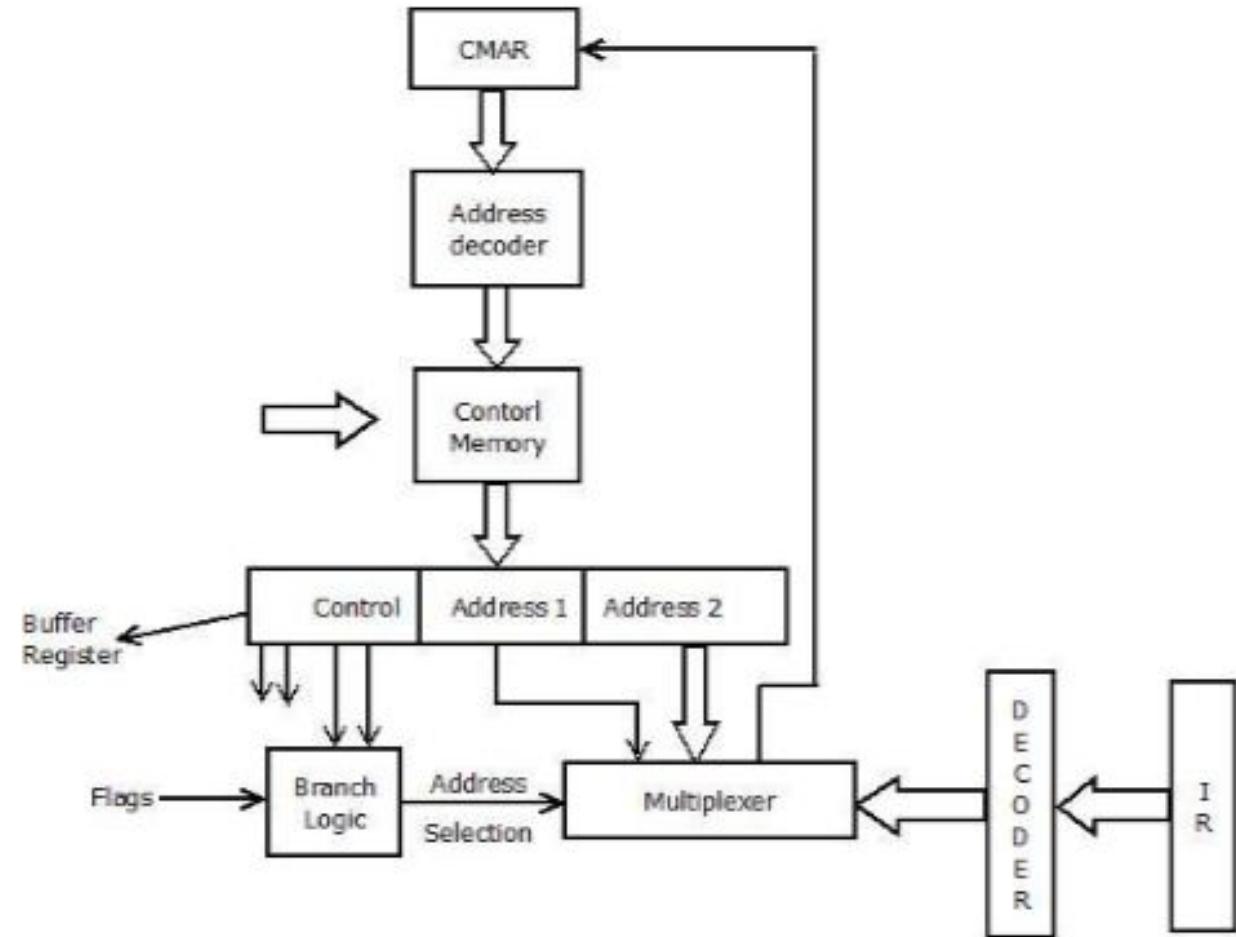
The simplest approach is to provide two address field in each microinstruction and multiplexer is provided to select:

Address from the second address field.

Starting address based on the OPcode field in the current instruction.

The address selection signals are provided by a branch logic module whose input consists of control unit flags plus bits from the control partition of the micro instruction.

Two address field



Single address field

Two-address approach is simple but it requires more bits in the microinstruction. With a simpler approach, we can have a single address field in the micro instruction with the following options for the next address.

Address Field.

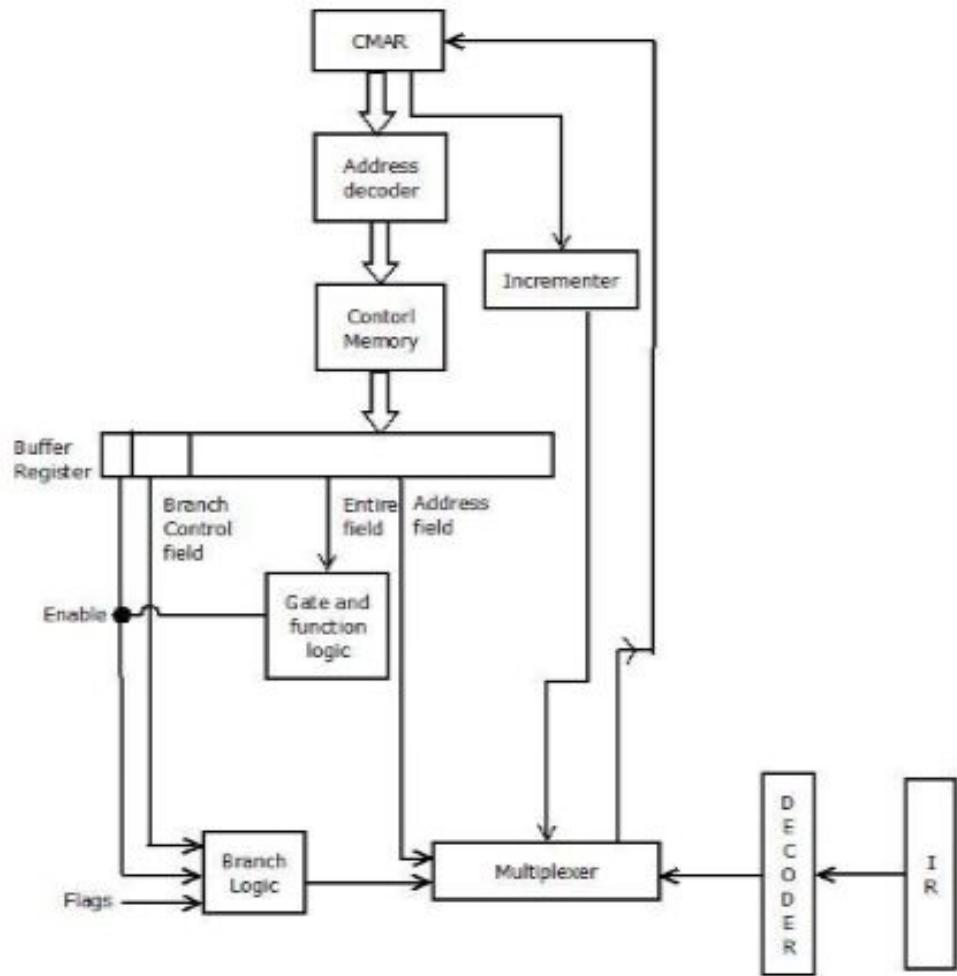
Based on OPcode in instruction register.

Next Sequential Address.

enter image description here

The address selection signals determine which option is selected. This approach reduces the number of address field to one. In most cases (in case of sequential execution) the address field will not be used. Thus the microinstruction encoding does not efficiently utilize the entire microinstruction.

Single address field

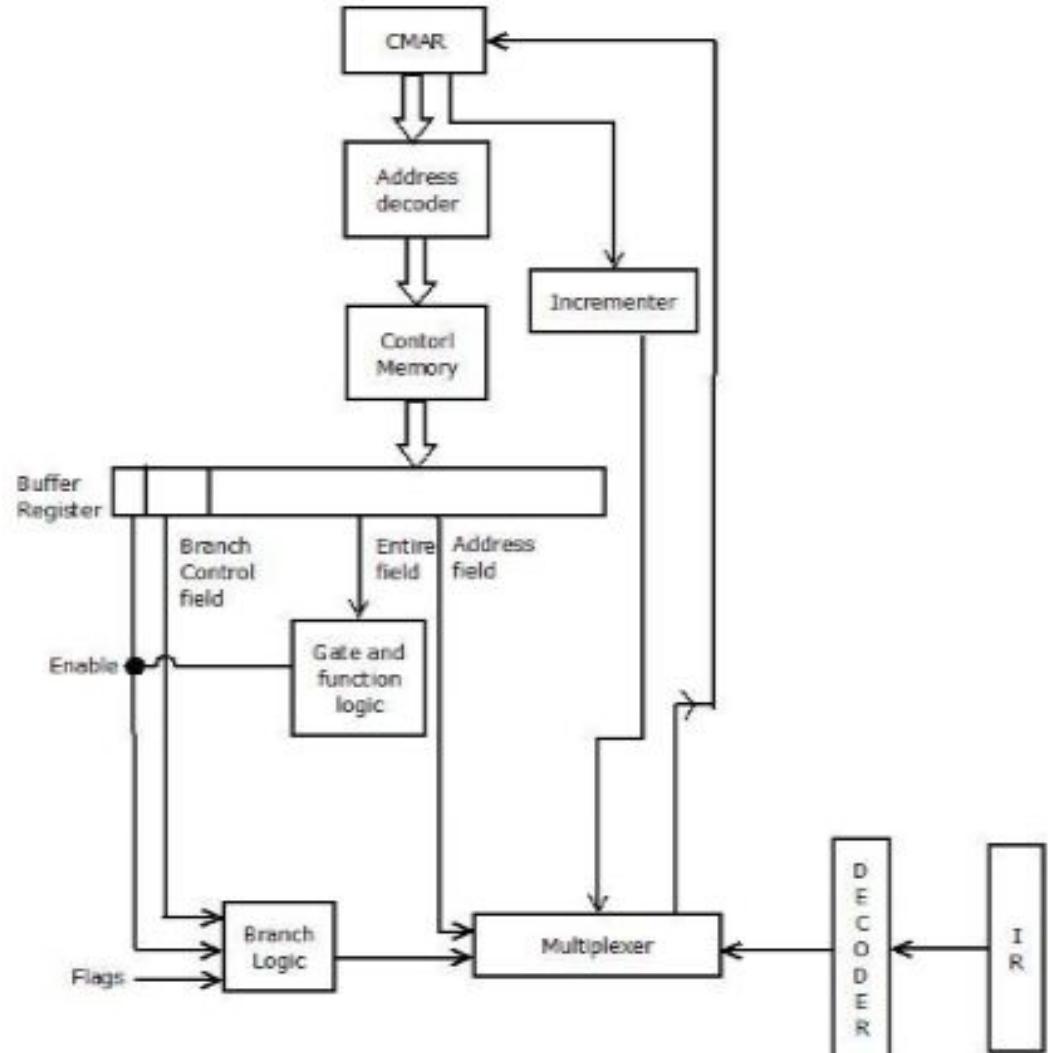


Variable Format

In this approach, there are two entirely different microinstruction formats. One bit designates which format is being used. In this first format, the remaining bits are used to activate control signals.

In the second format, some bits drive the branch logic module, and the remaining bits provide the address. With the first format, the next address is either the next sequential address or an address derived from the instruction register. With the second format, either a conditional or unconditional branch is specified.

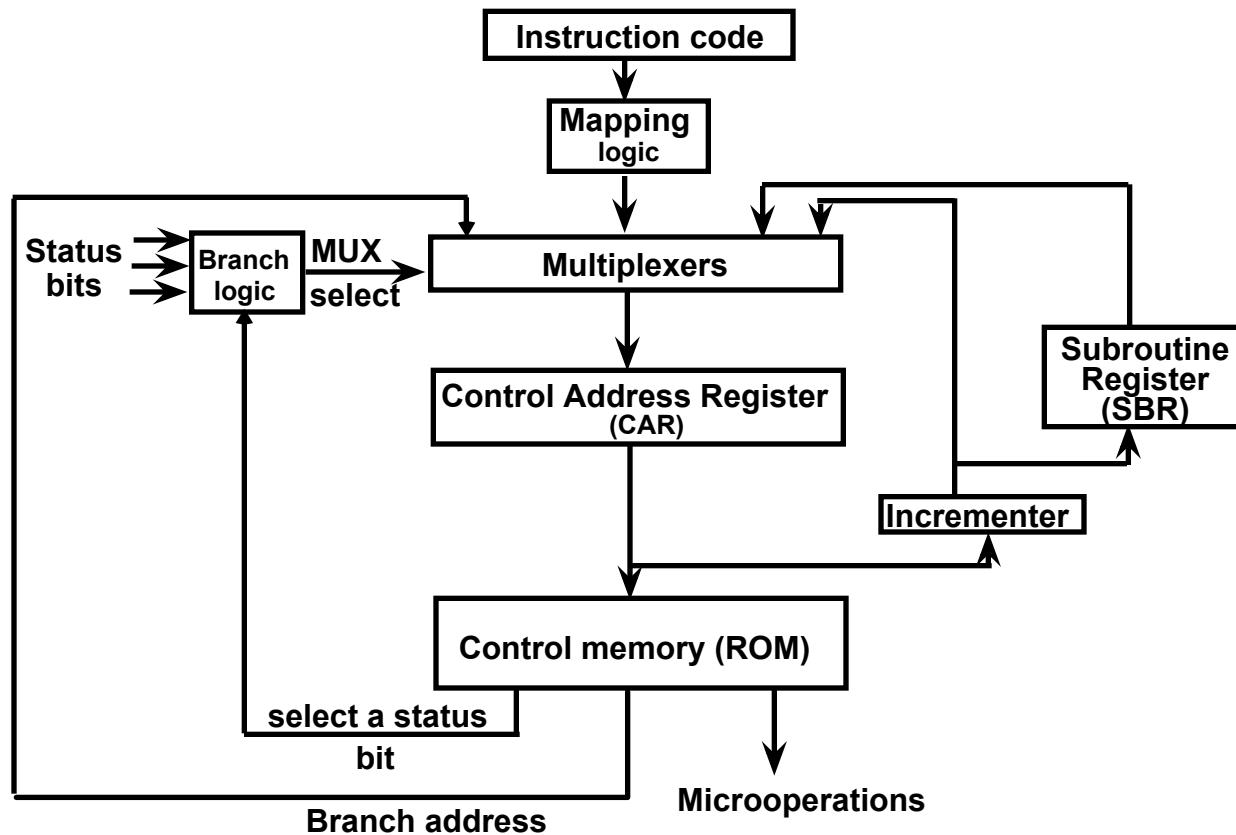
Variable Format



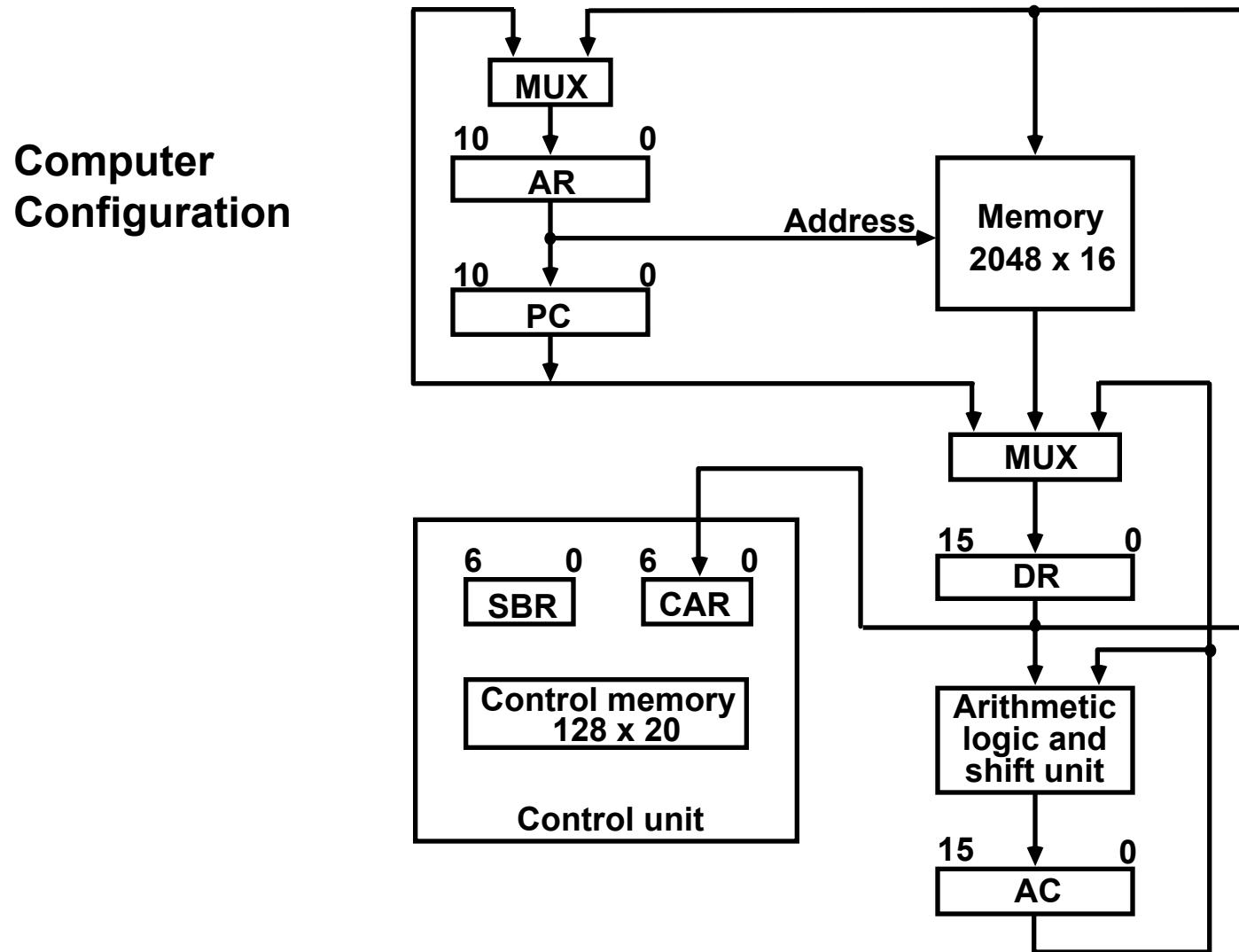
Address Sequencing

- Address sequencing capabilities required in control unit
 - Incrementing CAR
 - Unconditional or conditional branch, depending on status bit conditions
 - Mapping from bits of instruction to address for control memory
 - Facility for subroutine call and return

Address Sequencing

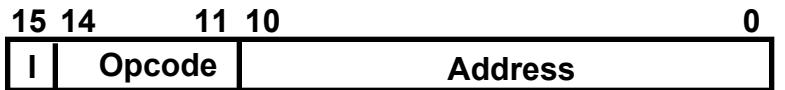


Microprogram Example



Microprogram Example

Computer instruction format

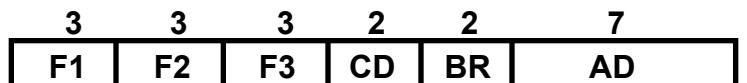


Four computer instructions

Symbol	OP-code	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	if ($AC < 0$) then ($PC \leftarrow EA$)
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

Microinstruction Format



F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

Microinstruction Fields

F1	Microoperation	Symbol
000	None NOP	
001	AC \leftarrow AC + DR	ADD
010	AC \leftarrow 0	CLRAC
011	AC \leftarrow AC + 1	INCAC
100	AC \leftarrow DR	DRTAC
101	AR \leftarrow DR(0-10)	DRTAR
110	AR \leftarrow PC	PCTAR
111	M[AR] \leftarrow DRWRITE	

F2	Microoperation	Symbol
000	None NOP	
001	AC \leftarrow AC - DR	SUB
010	AC \leftarrow AC \vee DR	OR
011	AC \leftarrow AC \wedge DR	AND
100	DR \leftarrow M[AR]READ	
101	DR \leftarrow AC	ACTDR
110	DR \leftarrow DR + 1	INCDR
111	DR(0-10) \leftarrow PC	PCTDR

F3	Microoperation	Symbol
000	None NOP	
001	AC \leftarrow AC \oplus DR	XOR
010	AC \leftarrow AC'	COM
011	AC \leftarrow shl AC	SHL
100	AC \leftarrow shr AC	SHR
101	PC \leftarrow PC + 1	INCPC
110	PC \leftarrow AR	ARTPC
111	Reserved	

Microinstruction Fields

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	DR(15)	I	Indirect address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC

BR	Symbol	Function
00	JMP	CAR \leftarrow AD if condition = 1 CAR \leftarrow CAR + 1 if condition = 0
01	CALL	CAR \leftarrow AD, SBR \leftarrow CAR + 1 if condition = 1
10	RET	CAR \leftarrow CAR + 1 if condition = 0
11	MAP	CAR(2-5) \leftarrow DR(11-14), CAR(0,1,6) \leftarrow 0

Symbolic Microinstruction

- **Sample Format**

Label:	Micro-ops	CD	BR	AD
---------------	------------------	-----------	-----------	-----------

- Label may be empty or may specify symbolic address terminated with colon
- Micro-ops consists of 1, 2, or 3 symbols separated by commas
- CD one of {U, I, S, Z}
 - U: Unconditional Branch
 - I: Indirect address bit
 - S: Sign of AC
 - Z: Zero value in AC
- BR one of {JMP, CALL, RET, MAP}
- AD one of {Symbolic address, NEXT, empty}

Fetch Routine

- Fetch routine
 - Read instruction from memory
 - Decode instruction and update PC

Microinstructions for fetch routine:

```

AR ← PC
DR ← M[AR], PC ← PC + 1
AR ← DR(0-10), CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0
    
```

Symbolic microprogram for fetch routine:

ORG 64	
FETCH:	PCTAR U JMP NEXT
	READ, INCPC U JMP NEXT
	DRTAR U MAP

Binary microporogram for fetch routine:

Binary address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

Symbolic Microprogram

- Control memory: 128 20-bit words
- First 64 words: Routines for 16 machine instructions
- Last 64 words: Used for other purpose (e.g., fetch routine and other subroutines)
- Mapping: OP-code XXXX into 0XXXX00, first address for 16 routines are 0(0 0000 00), 4(0 0001 00), 8, 12, 16, 20, ..., 60

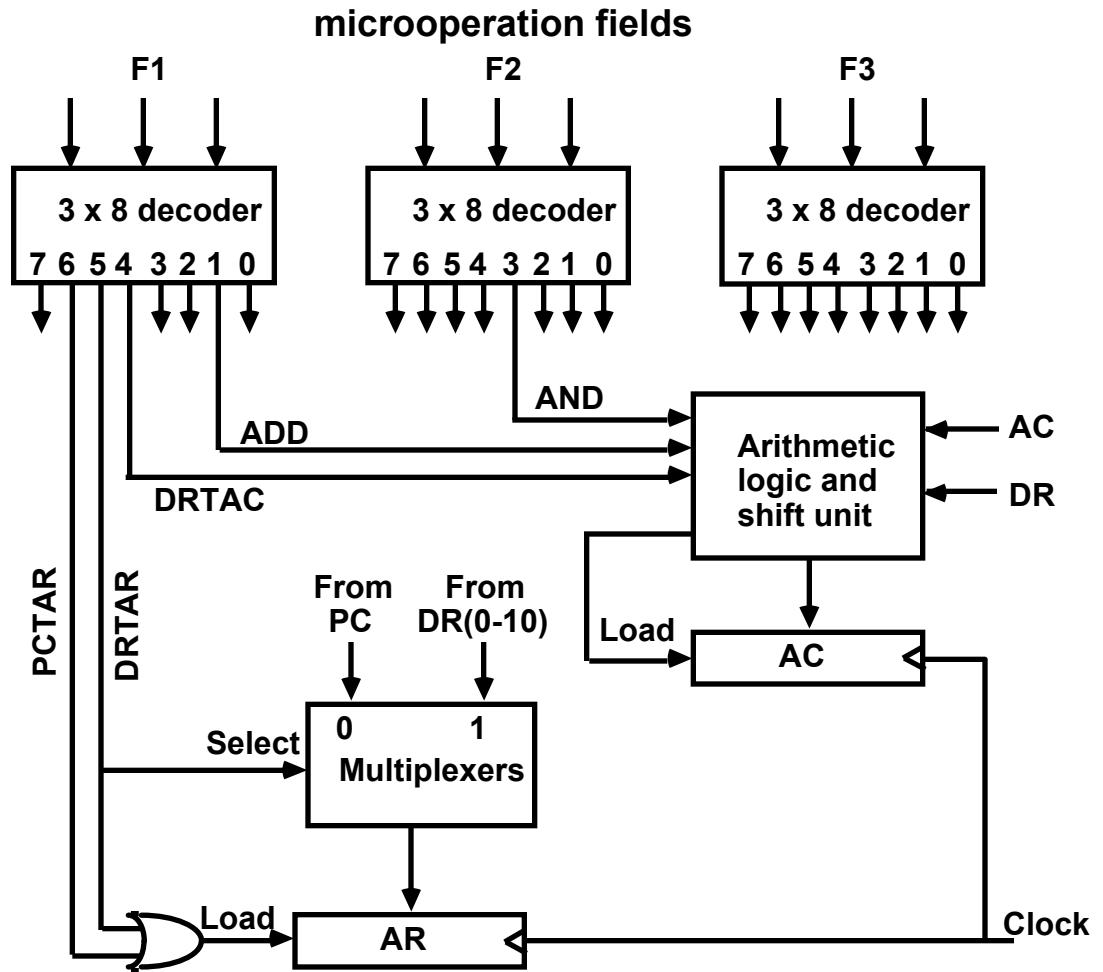
Partial Symbolic Microprogram

Label	Microops	CD	BR	AD
ADD:	ORG 0			
	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
BRANCH:	ORG 4			
	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
	NOP	I	CALL	INDRCT
OVER:	ARTPC	U	JMP	FETCH
	ORG 8			
	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
STORE:	WRITE	U	JMP	FETCH
	ORG 12			
	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
EXCHANGE:	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
	PCTAR	U	JMP	NEXT
FETCH:	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
	READ	U	JMP	NEXT
	DRTAR	U	RET	
INDRCT:				

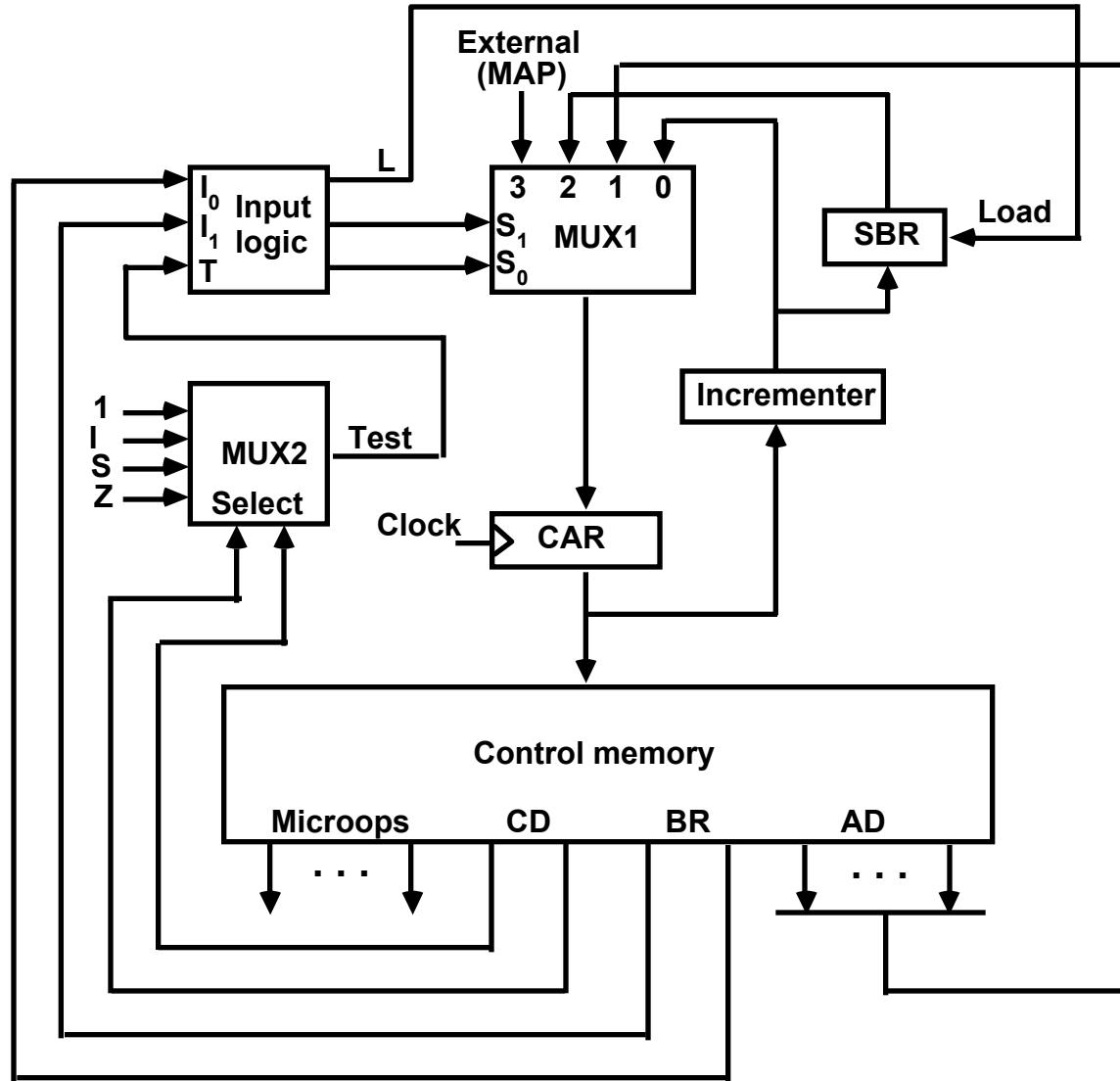
Binary Microprogram

Micro Routine	Address			Binary Microinstruction					
	Decimal	Binary		F1	F2	F3	CD	BR	
AD									
ADD	0	0000000	000 000	000	000	01	01	1000011	
		1	0000001	000	100	000	00		00
	0000010			001	000	000	00		
	1000000			000	000	000	00		
	1000000			000	000	000	00		
BRANCH	4	0000100		000	000	000	10		
	0000110			000	000	000	00		
	5	0000101		000	000	000	00	00	1000000
	6	0000110		000	000	000	01	01	1000011
	7	0000111		000	000	110	00	00	1000000
STORE	8	0001000		000	000	000	01	01	1000011
	9	0001001		000	101	000	00	00	0001010
	10	0001010		111	000	000	00	00	
	1000000								
	11	0001011		000	000	000	00	00	
EXCHANGE	12	0001100		000	000	000	01	01	1000011
	0001110			001	000	000	00	00	
	13	0001101		000	000	000	00	00	
	14	0001110		100	101	000	00	00	
	0001111			111	000	000	00	00	
	1000000								

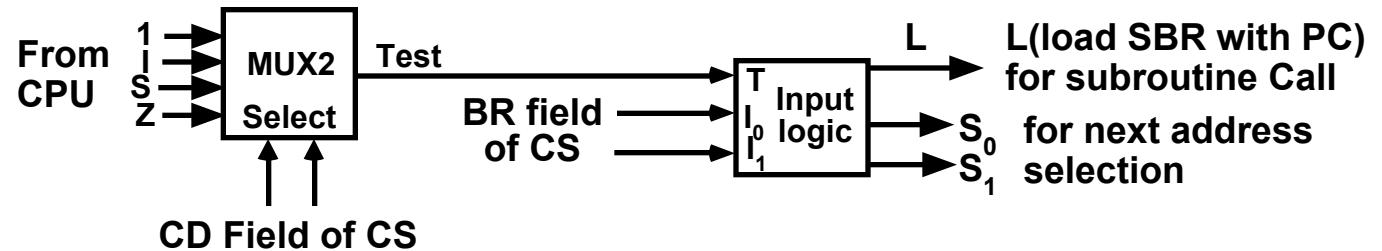
Design of Control Unit



Microprogram Sequencer



Input Logic for Microprogram Sequencer



Input Logic

I ₁ I ₀ T	Meaning	Source of Address	S ₁	S ₀
L				
000	In-Line	CAR+1	00	0
001	JMP	CS(AD)	01	0
010	In-Line	CAR+1	00	0
011	CALL	CS(AD) and SBR <- CAR+1	01	1
10x	RET	SBR	10	0
11x	MAP	DR(11-14)	11	0

$$S_1 = I_1$$

$$S_0 = I_0 I_1 + I_1' T$$

$$L = I_1' I_0 T$$

Address Sequencing

Microinstructions are stored in control memory in groups, with each group specifying a routine.

To appreciate the address sequencing in a micro-program control unit, let us specify the steps that the control must undergo during the execution of a single computer instruction.

Step-1

- An initial address is loaded into the control address register when power is turned on in the computer.
- This address is usually the address of the first microinstruction that activates the instruction fetch routine.
- The fetch routine may be sequenced by incrementing the control address register through the rest of its microinstructions.
- At the end of the fetch routine, the instruction is in the instruction register of the computer.

Step-2

- The control memory next must go through the routine that determines the effective address of the operand.
- A machine instruction may have bits that specify various addressing modes, such as indirect address and index registers.
- The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction.
- When the effective address computation routine is completed, the address of the operand is available in the memory address register.

Step-3

- The next step is to generate the microoperations that execute the instruction fetched from memory.
- The microoperation steps to be generated in processor registers depend on the operation code part of the instruction.
- Each instruction has its own micro-program routine stored in a given location of control memory.
- The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a mapping process.
- A mapping procedure is a rule that transforms the instruction code into a control memory address.

Step-4

- Once the required routine is reached, the microinstructions that execute the instruction may be sequenced by incrementing the control address register.
- Micro-programs that employ subroutines will require an external register for storing the return address.
- Return addresses cannot be stored in ROM because the unit has no writing capability.
- When the execution of the instruction is completed, control must return to the fetch routine.
- This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine.

Basic Concepts of pipelining

How to improve the performance of the processor?

1. By introducing faster circuit technology
2. Arrange the hardware in such a way that, more than one operation can be performed at the same time.

What is Pipeining?

It is the process of arrangement of hardware elements in such way that, simultaneous execution of more than one instruction takes place in a pipelined processor so as to increase the overall performance.

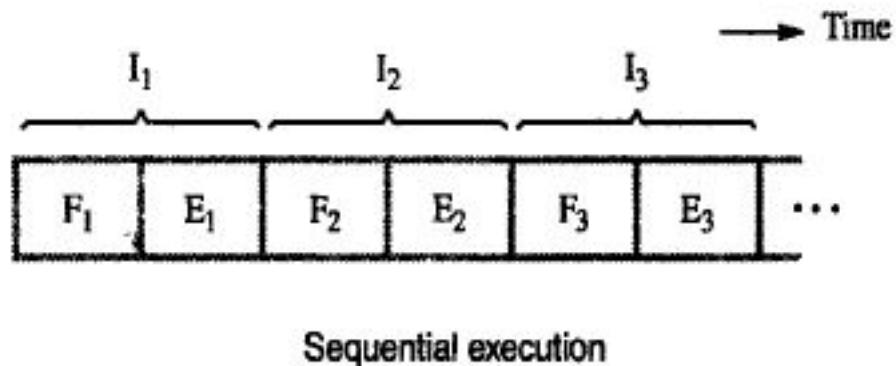
What is Instruction Pipeining?

- The number of instruction are pipelined and the execution of current instruction is overlapped by the execution of the subsequent instruction.
- It is a instruction level parallelism where execution of current instruction does not wait until the previous instruction has executed completely.

Basic idea of Instruction Pipelining

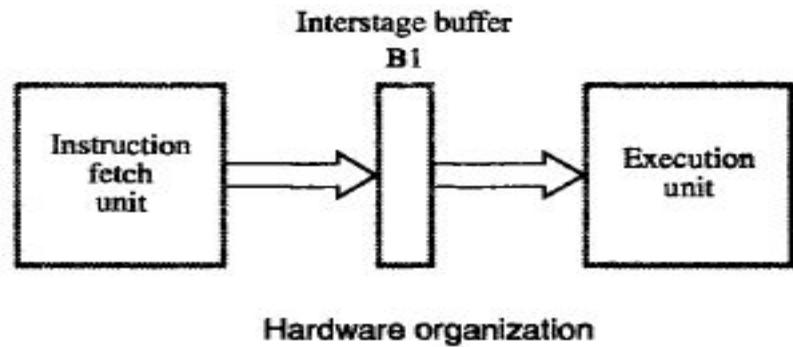
Sequential Execution of a program

- The processor executes a program by fetching(F_i) and executing(E_i) instructions one by one.



Hardware organization and instruction pipeline

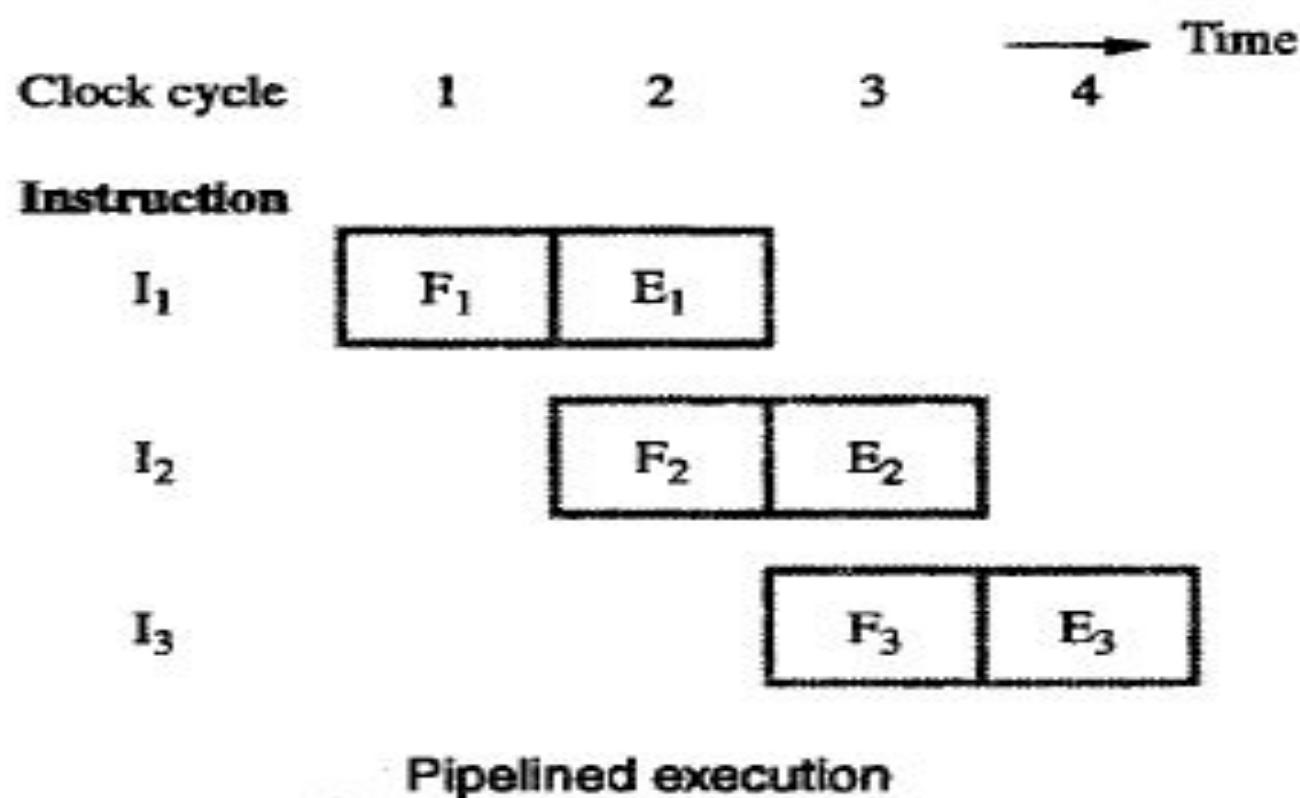
- Consists of 2 hardware units one for fetching and another one for execution as follows.
- Also has intermediate buffer to store the fetched instruction



2 stage pipeline

- Execution of instruction in pipeline manner is controlled by a clock.
- In the first clock cycle, fetch unit fetches the instruction I1 and store it in buffer B1.
- In the second clock cycle, fetch unit fetches the instruction I2 , and execution unit executes the instruction I1 which is available in buffer B1.
- By the end of the second clock cycle, execution of I1 gets completed and the instruction I2 is available in buffer B1.
- In the third clock cycle, fetch unit fetches the instruction I3 , and execution unit executes the instruction I2 which is available in buffer B1.
- In this way both fetch and execute units are kept busy always.

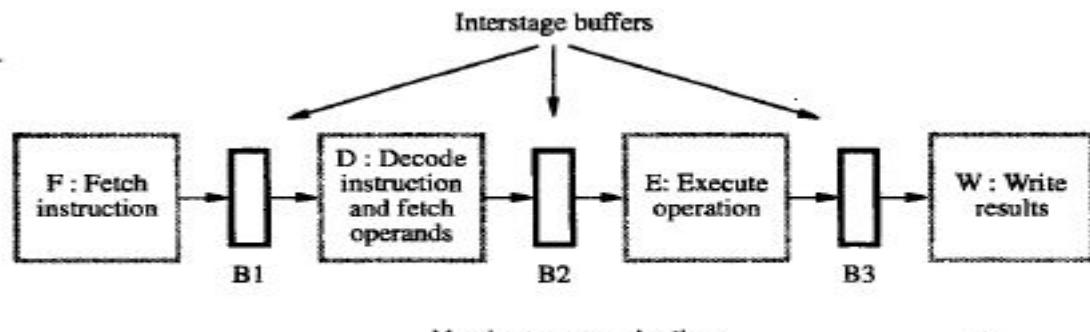
Contd...



Hardware organization for 4 stage pipeline

- Pipelined processor may process each instruction in 4 steps.

1. Fetch(F): Fetch the Instruction
 2. Decode(D): Decode the Instruction
 3. Execute (E) : Execute the Instruction
 4. Write (W) : Write the result in the destination location
- 4 distinct hardware units are needed as shown below.

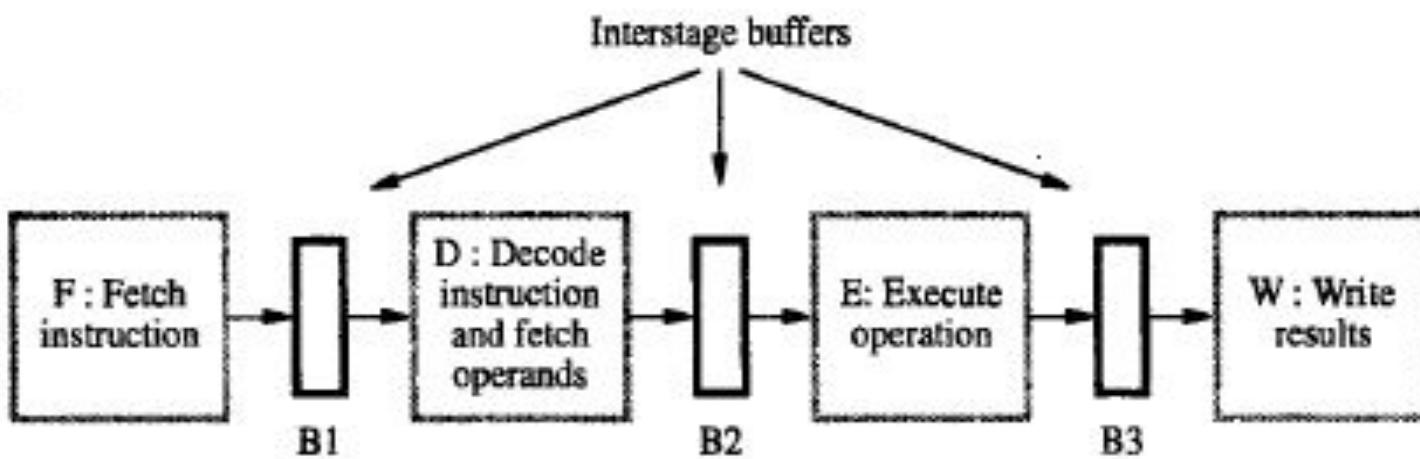


A 4-stage pipeline.

Execution of instruction in 4 stage pipeline

- In the first clock cycle, fetch unit fetches the instruction I1 and store it in buffer B1.
- In the second clock cycle, fetch unit fetches the instruction I2 , and decode unit decodes instruction I1 which is available in buffer B1.
- In the third clock cycle fetch unit fetches the instruction I3 , and decode unit decodes instruction I2 which is available in buffer B1 and execution unit executes the instruction I1 which is available in buffer B2.
- In the fourth clock cycle fetch unit fetches the instruction I4 , and decode unit decodes instruction I3 which is available in buffer B1, execution unit executes the instruction I2 which is available in buffer B2 and write unit write the result of I1.

Contd...



Hardware organization

A 4-stage pipeline.

Role of cache memory in Pipelining

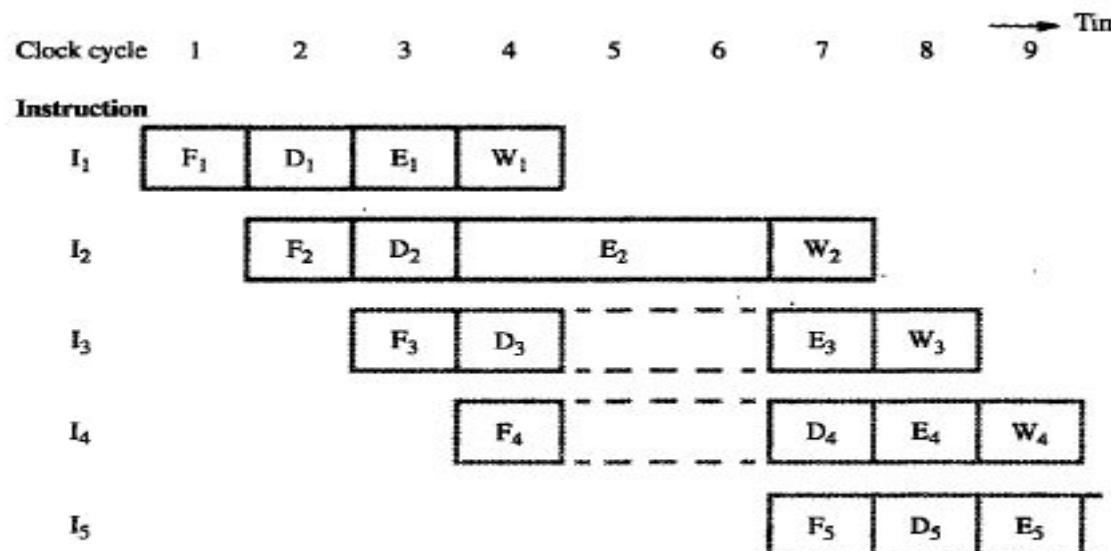
- Each stage of the pipeline is controlled by a clock cycle whose period is that the fetch, decode, execute and write steps of any instruction can each be completed in one clock cycle.
- However the access time of the main memory may be much greater than the time required to perform basic pipeline stage operations inside the processor.
- The use of cache memories solve this issue.
- If cache is included on the same chip as the processor, access time to cache is equal to the time required to perform basic pipeline stage operations .

Pipeline Performance

- Pipelining increases the CPU instruction throughput - the number of instructions completed per unit of time.
- The increase in instruction throughput means that a program runs faster and has lower total execution time.
- Example in 4 stage pipeline, the rate of instruction processing is 4 times that of sequential processing.
- Increase in performance is proportional to no. of stages used.
- However, this increase in performance is achieved only if the pipelined operation is continued without any interruption.
- But this is not the case always.

Contd...

- Consider the scenario, where one of the pipeline stage may require more clock cycle than the other.
- For example, consider the following figure where instruction I2 takes 3 cycles to completes its execution(cycle 4,5,6)
- In cycle 5,6 the write stage must be told to do nothing, because it has no data to work with.



Effect of an execution operation taking more than one clock cycle.

What is Pipeline Hazard?

- Meanwhile, the information in buffer B2 must remain there until the execute stage has completed.
- It shows that stage 2 and stage 1 are blocked from accepting new instructions.
- Thus the steps D4 and F5 must be postponed which is shown in the previous figure.
- The previous figure shows that the pipelined operations are paused for 2 clock cycles and resumes the pipelined operations in cycle 7.
- Any condition that causes the pipeline to pause/stall is called **hazard**.
- **Types of hazard:**

1. Data hazard

It is a condition in which either the source or the destination operands are not available at the expected time in the pipeline. The previous figure shows an example of data hazard.

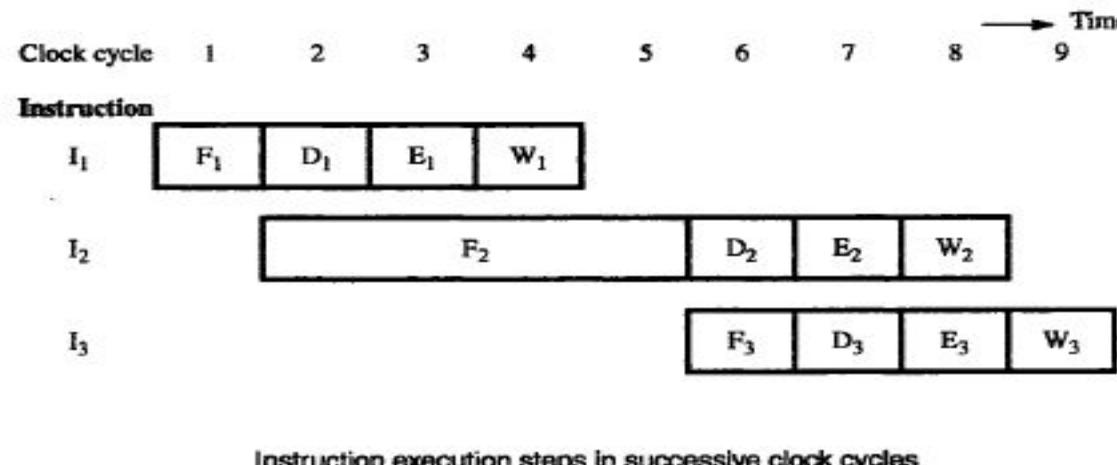
2. Control or Instruction hazard

It is a condition in which the pipelined operations are paused because of the delay in the availability of an instruction.

3. Structural hazard

Example for Instruction hazard

- The following figure illustrate that, Instruction I1 is fetched from cache in clock cycle1 and its execution proceeds normally.
- In clock cycle2, the fetch operation of I2 from cache miss and it suspends the further fetch requests and wait for the arrival of I2.
- The figure shows that, I2 arrived and loaded into B1 at the end of clock cycle5. Now the normal pipeline operation resumes.



Contd...

- Note that the decode unit is suspended from clock cycle3 to clock cycle5,
- The execute unit is suspended from clock cycle4 to clock cycle6
- The write unit is suspended from clock cycle5 to clock cycle7
- These suspended period is called stalls/bubbles in the pipeline

What is Structural hazard?

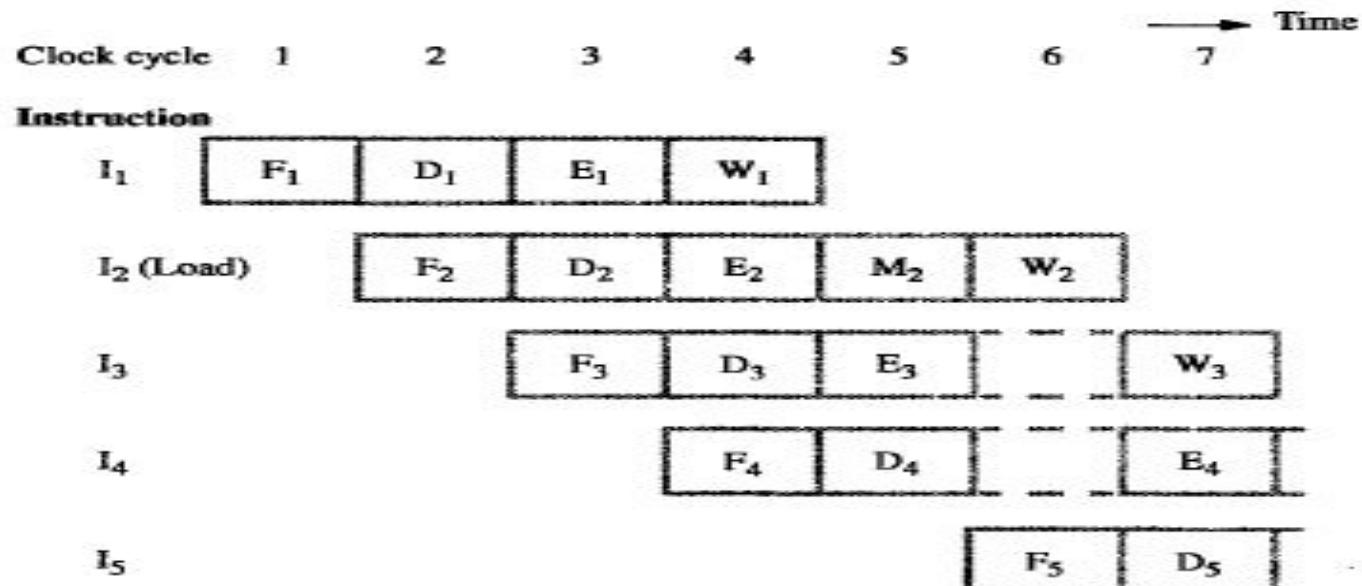
- This hazard arises in a situation when 2 instruction tries to use the same hardware resources at a time.
- Consider a situation, where one instruction may need to access memory either in execute or write stage whereas the other instruction may need to access memory in fetch stage .
- Also assumes that the instruction and data are available in cache.
- In this scenario, only one instruction can proceed while the other is suspended.
- Consider the instruction

I2 : Load X(R1),R2

- Here the memory address, $X + [R1]$ is computed at clock cycle 4, then the memory access takes place in clock cycle 5. (i.e) I2 execution takes 2 clock cycles.

Contd...

- It means that memory access at clock cycle 5 is written into register R2 in clock cycle 6.
- It causes the pipeline operation is suspended for clock cycle 6, since both I2 and I3 require access to the register file in clock cycle 6.



Effect of a Load instruction on pipeline timing.

Important point to consider

- It is very important that, pipeline operation does not execute the individual instruction faster, instead it increases the overall performance of program execution
- At any point of time, if any one of the stages in pipeline can not complete its operation in one clock cycle, then the pipeline is suspended and some performance degradation occurs.
- So the goal is to identify all hazards and find ways to minimize their impact.

Data Hazards

- It is a condition in which pipeline is stalled/suspended coz of either the source or the destination operands are not available at the expected time in the pipeline .
- Consider the following 2 instruction

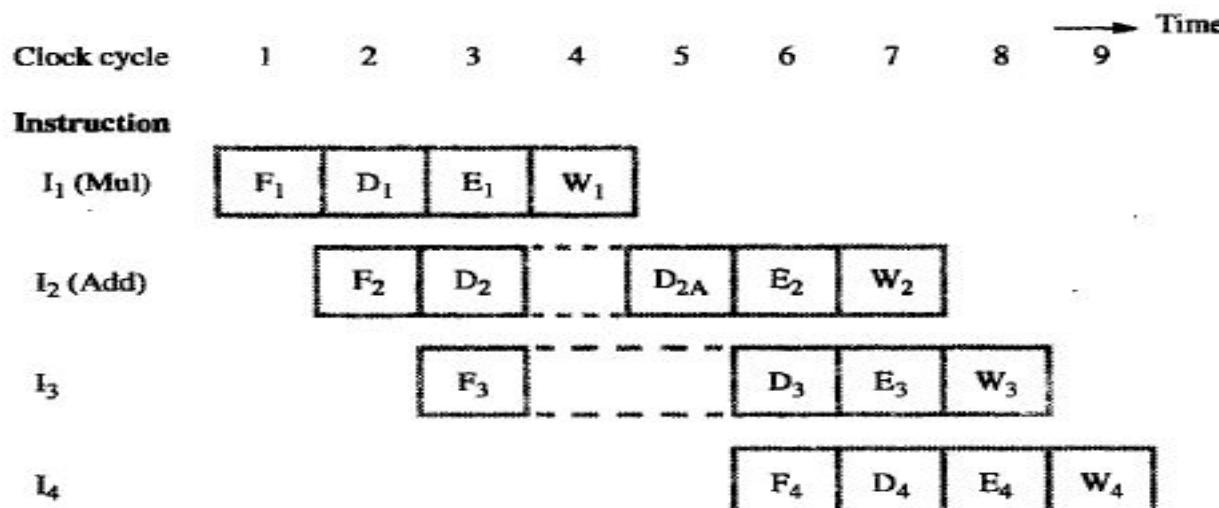
I1: $A=3+A$

I2: $B=4*A$

- When these instructions are executed in pipeline, the result generated by I1 may not be available to I2 , since execution of I2 begins before the completion of I1. **As a result, it gives the incorrect result.**
- In the above example the result of the I1 is taken as the Input to I2. So data dependency arises.
- The data dependency may also arises when the destination of one instruction is used as a source in the next instruction.
- Example:
- I1 : Mul R2,R3,R4 $R4= R2*R3$
- I2 : Add R5,R4,R6 $R6=R5+R4$

Contd...

- The result of Mul instruction is R4, which in turn is used as one of the two source operands of Add instruction.
- Assume that the execution unit of Mul instruction takes one clock cycle.
- In clock cycle3, when the decode unit of Add instruction realizes that R4 is used as the source operand, it cannot be completed the decoding until the write unit of Mul instruction has been completed.



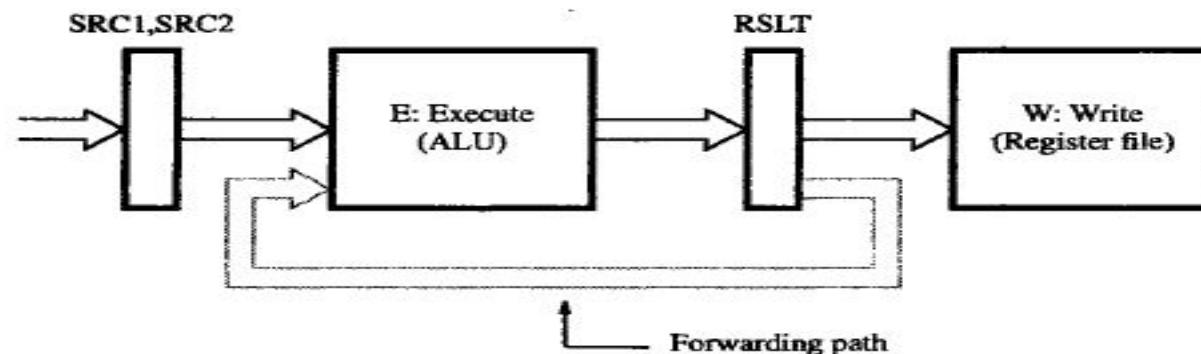
Pipeline stalled by data dependency between D₂ and W₁.

Contd...

- So the completion of D2 must be delayed to clock cycle 5 as shown in the above Figure as D2A.
- Instruction I3 is fetched in clock cycle 3, but its decoding is done in clock cycle 6. As shown in the above Figure the pipeline is stalled for two clock cycles.

Methods to overcome Data hazards (Operand Forwarding)

- Data hazards occur when instructions that exhibit data dependence
- In the previous example, instruction I2 is waiting for data to be written in the register by instruction I1.
- However, this data is available at the o/p of ALU once the execution of I1 gets over in the unit E1.
- Hence, the delay can be reduced or sometimes eliminated if we rearrange for the result of instruction I1 to be forwarded directly to the step E2.

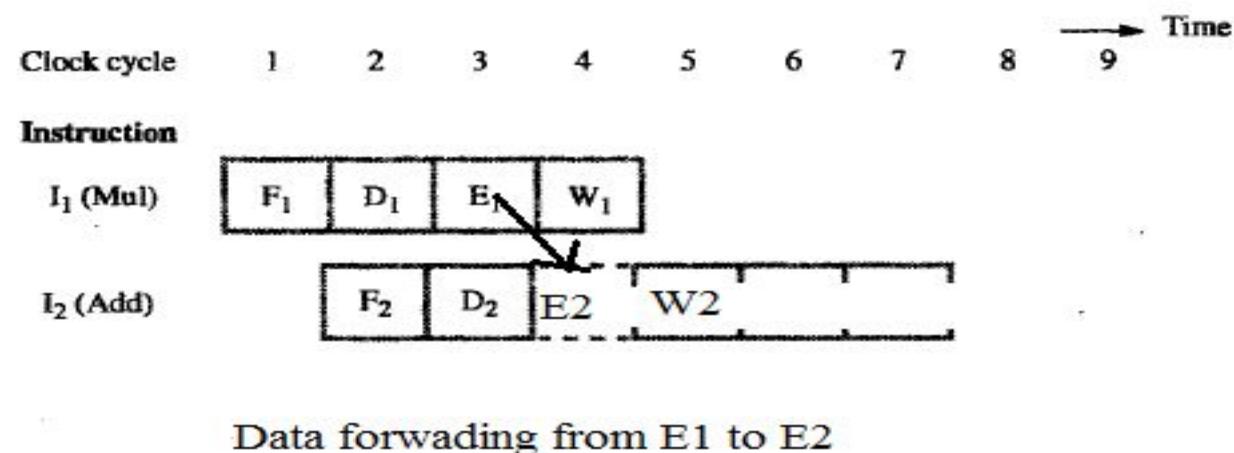


(b) Position of the source and result registers in the processor pipeline

Operand forwarding in a pipelined processor.

Contd...

- After decoding the instruction I2 in clock cycle 3 , the system identifies the data dependency and so a decision is made to use data forwarding.
- The operand not involved in the dependency is read and loaded in register SRC1 in clock cycle 3 .
- In the next clock cycle , the result produced by I1 is available in the register RSLT (because of data forwarding) can be used in step E2.
- Hence execution of I2 proceeds without interruption.



Handling Data hazards in Software

- Another approach of identifying data dependencies and dealing with them by using software.
 - In this case, the compiler can introduce 2 clock cycle delay needed between I1 and I2 by inserting NOP (No OPeration) instructions as follows.
-
- I1 : Mul R2,R3,R4 $R4 = R2 * R3$
 - NOP
 - NOP
 - I2 : Add R5,R4,R6 $R6 = R5 + R4$

What is implicit dependency and Side Effects of this?

- The data dependencies encountered in the previous example is explicit, easily detected and can be resolved.
- But this is not the case always, when a location other than one explicitly named in an instruction as a destination operand is affected, then the **instruction said to have side effect**.
- **Example1:**
- Stack instructions such as push and pop produce side effects because they implicitly use auto increment and auto decrement addressing modes.
- **Example2:**
- I1 : Add R1,R3
- I2 : Add WithCarry R2,R4

Contd...

- In example 2, an implicit dependency exists between these instructions through carry flag.
- The carry flag set by the first instruction is used in the second instruction which performs $R4=[R2]+[R4]+\text{carry}$.
- The instructions that have side effects give rise to multiple data dependencies which lead to the increase in both the hardware and software needed to resolve them.
- So, the instructions which are executed in pipeline manner should have few side effects.

Instruction Hazards

Whenever the stream of instructions supplied by the instruction fetch unit is interrupted, the pipeline stalls.

Unconditional Branches

- Sequence of instruction being executed in two stages pipeline instruction I1 to I3 are stored at consecutive memory address and instruction I2 is a branch instruction.
- If the branch is taken then the PC value is not known till the end of I2.
- Next three instructions are fetched even though they are not required
- Hence they have to be flushed after branch is taken and new set of instruction have to be fetched from the branch address

Unconditional Branches

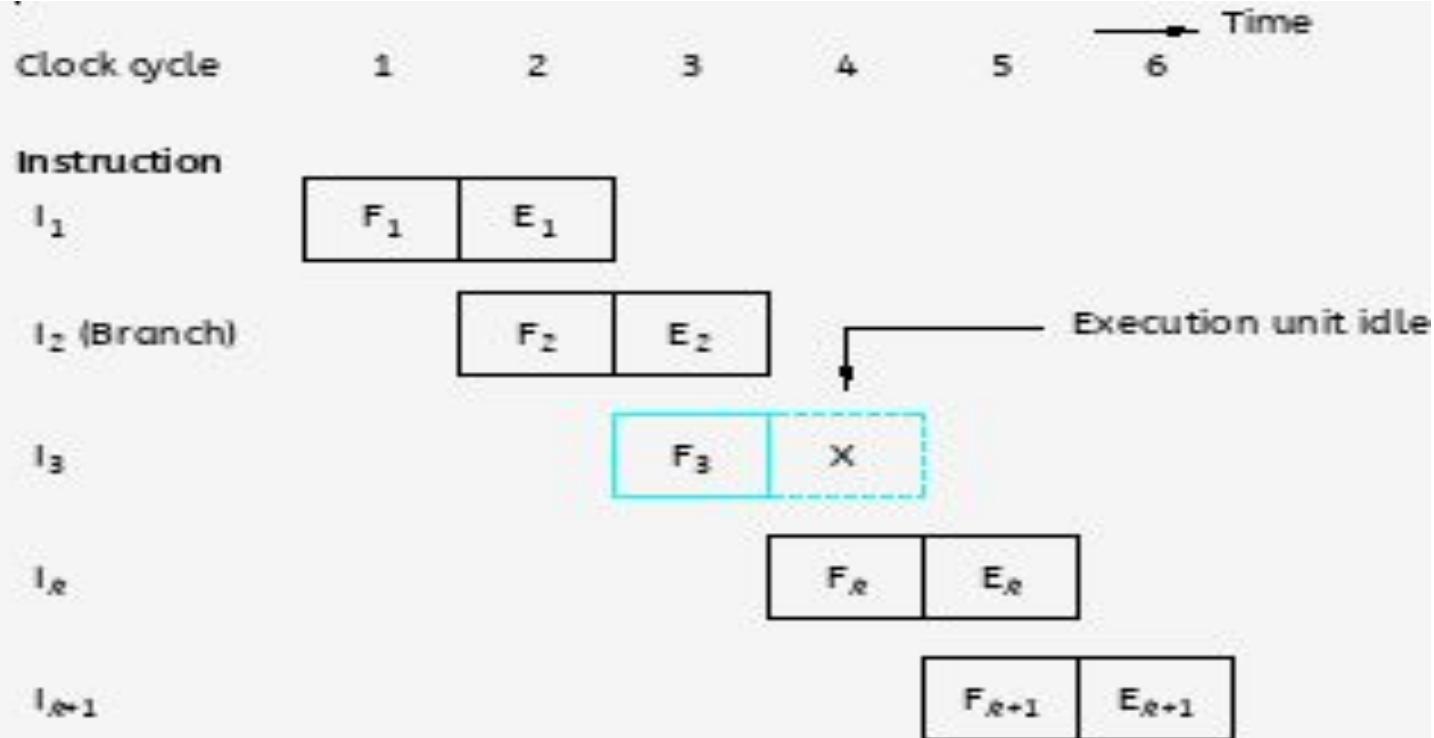


Figure 8.8. An idle cycle caused by a branch instruction.

Branch Timing

Branch penalty

The time lost as the result of branch instruction

Reducing the penalty

The branch penalties can be reduced by proper scheduling Through compiler techniques

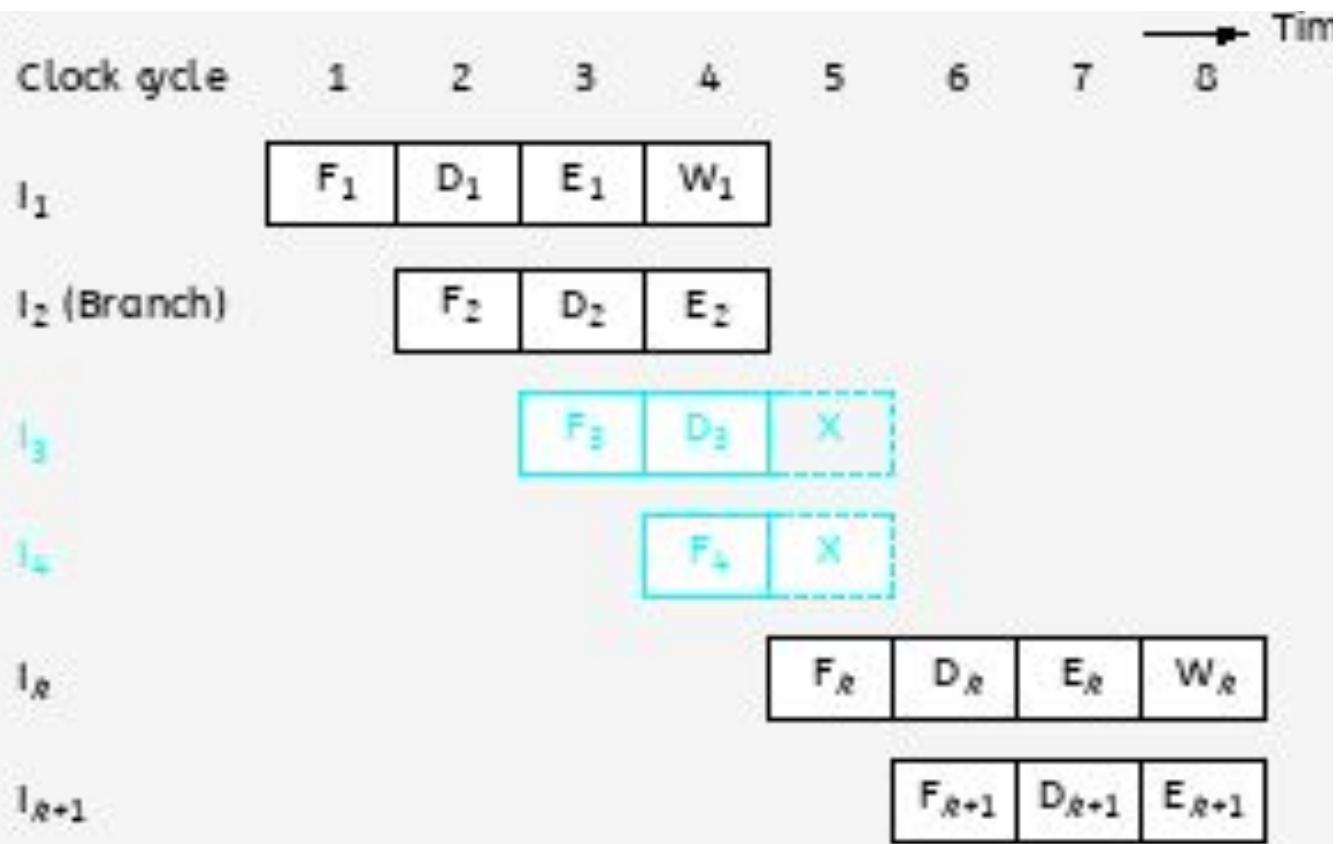
For longer pipeline, the branch penalty may be higher

Reducing the branch penalty requires branch address to be computed earlier in the pipeline

Instruction fetch unit has dedicated hardware

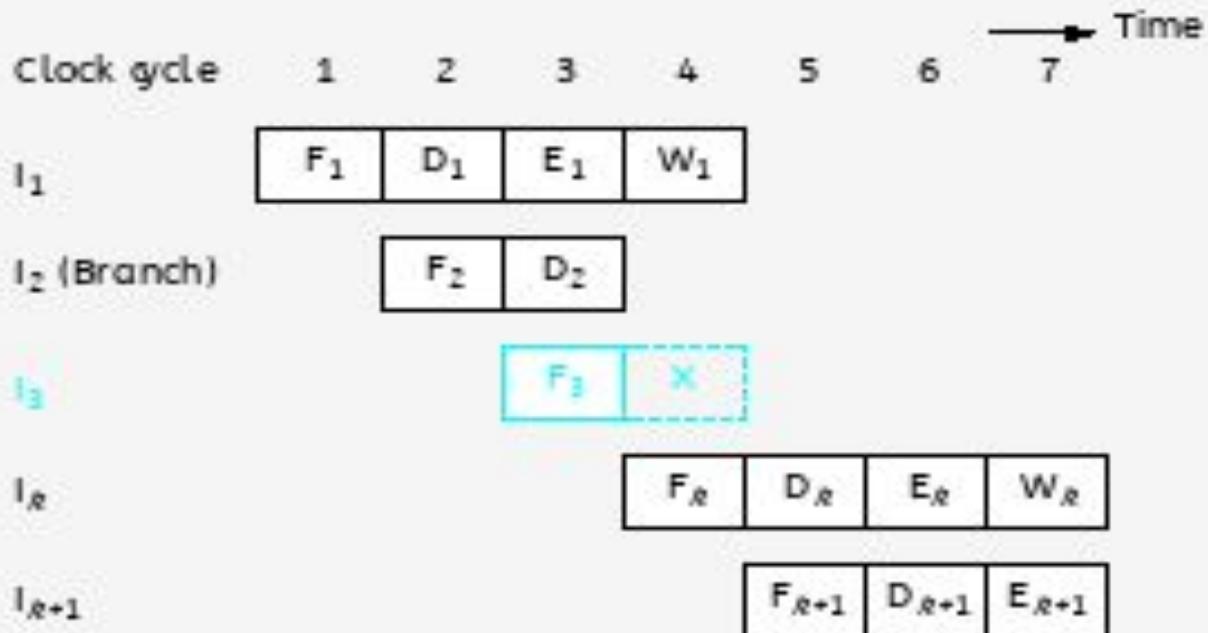
to identify a branch instruction and compute branch target address as quickly as possible after an instruction is fetched

Branch Timing



(a) Branch address computed in the stage

Branch Timing

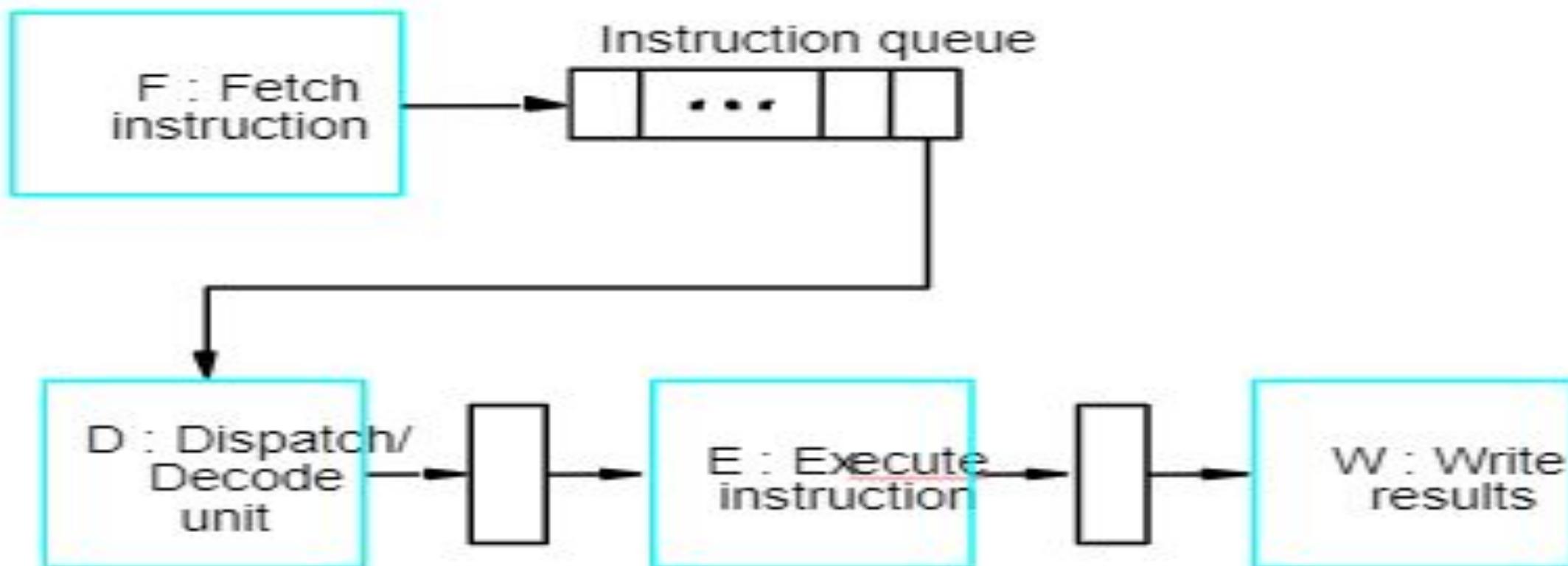


(b) Branch address computed in Decode stage

Figure 8.9. Branch timing.

Instruction Queue and Prefetching

Instruction fetch unit



Instruction Queue and Prefetching

Fetch Unit-Contain instruction queue to store the instruction before they are needed to avoid interruption

Dispatch unit-Takes instruction from the front of the queue and sends them to the execution unit, it also perform the decoding operation

Fetch unit keeps the instruction queue filled at all times, fetch instruction and add them to the queue.

If there is delay in fetching the instruction, the dispatch unit continues to issue the instruction from the instruction queue

Conditional Branches

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 represent about 20% of the dynamic instruction count of most programs.

Delayed Branch

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 place useful instructions in these slots.

The effectiveness of the delayed branch approach depends on how often it is possible to reorder instructions.

Delayed Branch

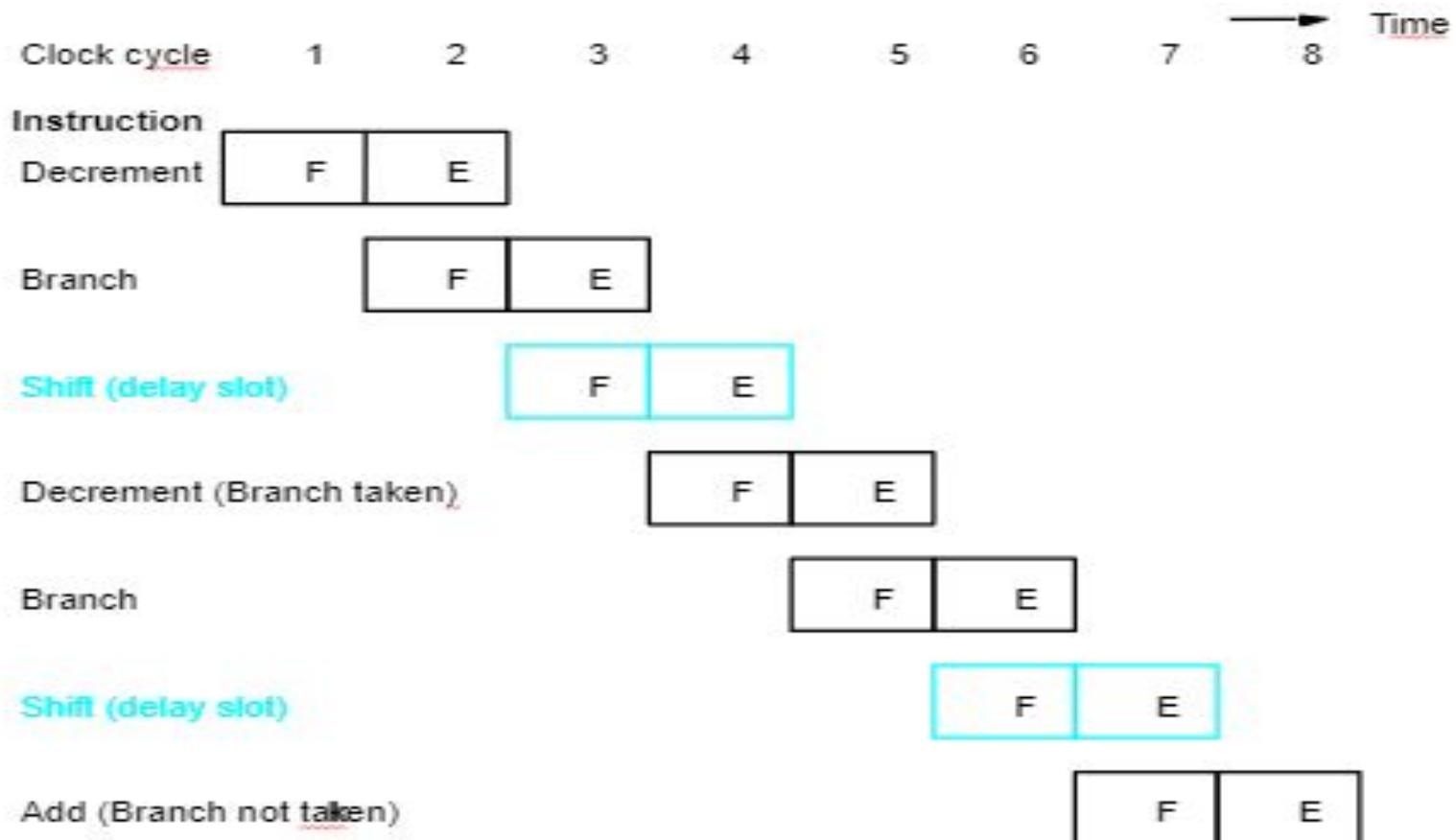
LOOP	Shift_left	R1
	Decrement	R2
	Branch=0	LOOP
NEXT	Add	R1,R3

(a) Original program loop

LOOP	Decrement	R2
	Branch=0	LOOP
	Shift_left	R1
NEXT	Add	R1,R3

(b) Reordered instructions

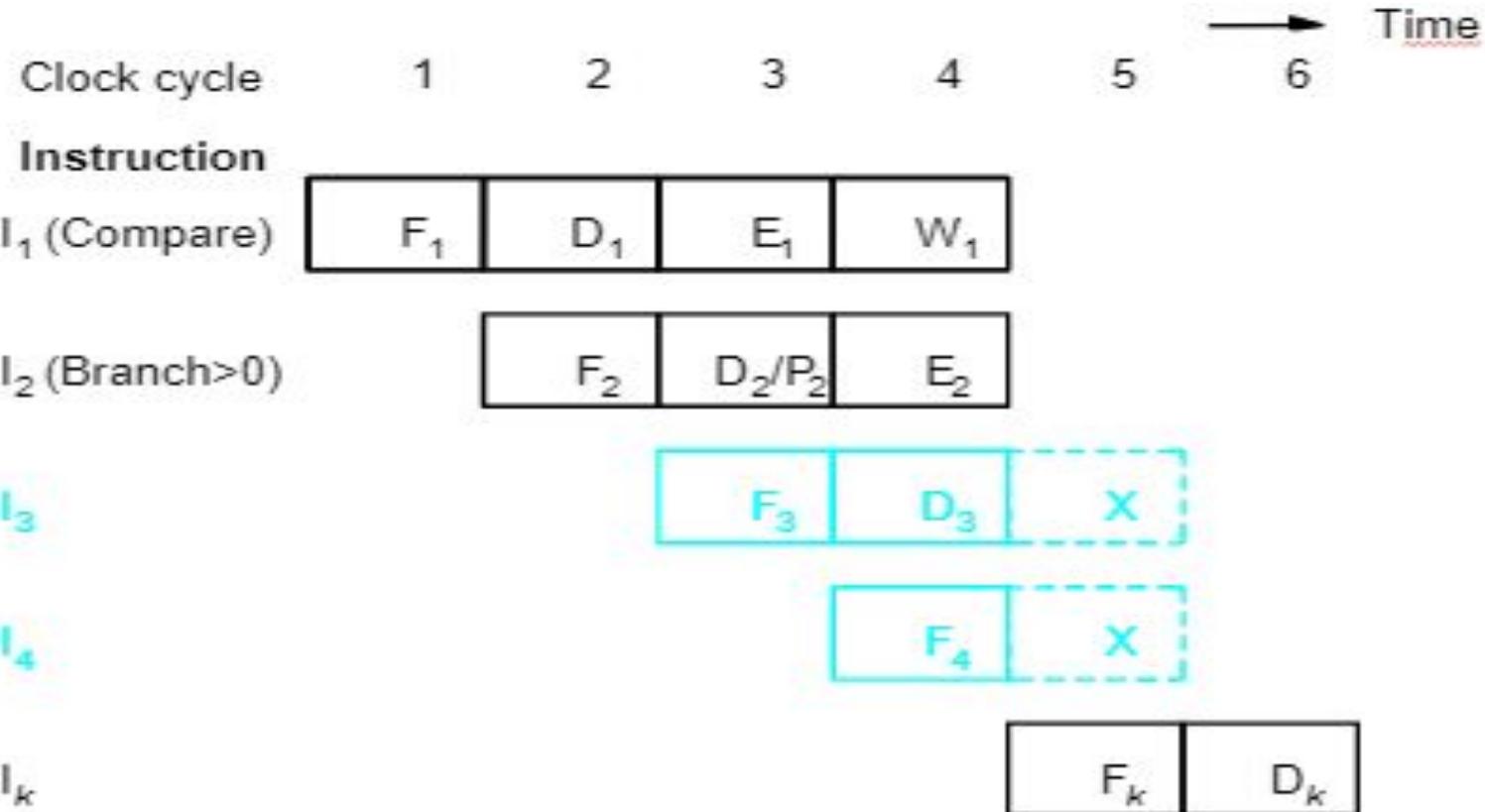
Delayed Branch



Branch Prediction

- To predict whether or not a particular branch will be taken.
- Simplest form: assume branch will not take place and continue to fetch instructions in sequential address order.
- Until the branch is evaluated, instruction execution along the predicted path must be done on a speculative basis.
- Speculative execution: instructions are executed before the processor is certain that they are in the correct execution sequence.
- Need to be careful so that no processor registers or memory locations are updated until it is confirmed that these instructions should indeed be executed.

Incorrectly Predicted Branch



Branch Prediction

- . Better performance can be achieved if we arrange for some branch instructions to be predicted as taken and others as not taken.
- . Use hardware to observe whether the target address is lower or higher than that of the branch instruction.
- . Let compiler include a branch prediction bit.
- . So far the branch prediction decision is always the same every time a given instruction is executed – static branch prediction.

Branch Prediction

- . Static Prediction
- . Dynamic branch Prediction

Static Prediction

- . Prediction is carried out by compiler and it is static because the prediction is already known before the program is executed

Dynamic Branch Prediction

- . Dynamic prediction in which the prediction decision may change depending on the execution history

Branch Prediction Algorithm

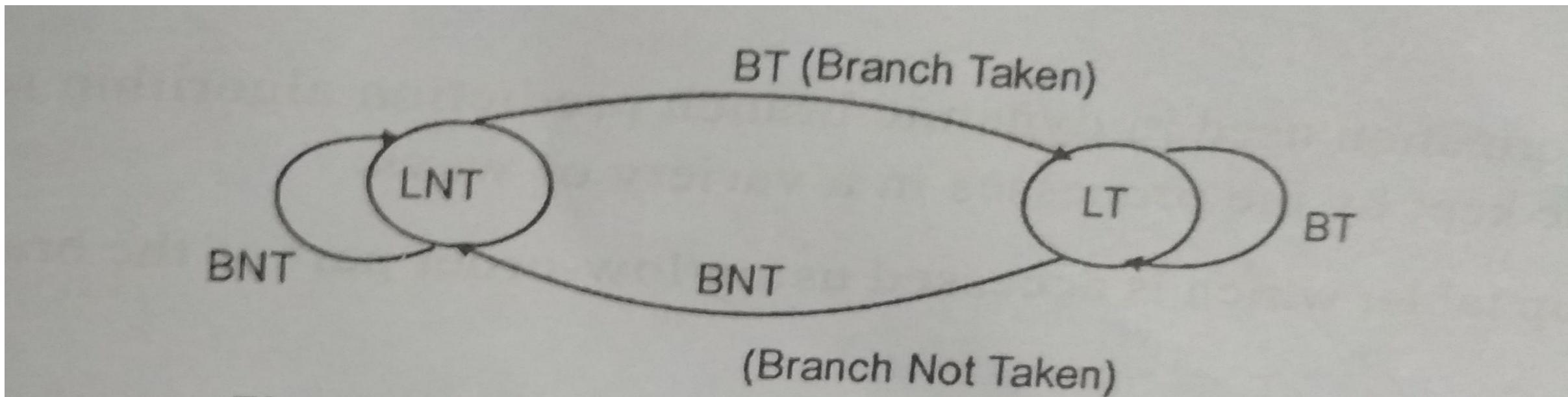
If the branch taken recently, the next time if the same branch is executed, it is likely that the branch is taken

State 1: LT : Branch is likely to be taken

State 2: LNT : Branch is likely not to be taken

1. If the branch is taken, the machine moves to LT. otherwise it remains in state LNT.
2. The branch is predicted as taken if the corresponding state machine is in state LT, otherwise it is predicted as not taken

Branch Prediction Algorithm



4 State Algorithm

ST-Strongly likely to be taken

LT-Likely to be taken

LNT-Likely not to be taken

SNT-Strongly not to be taken

Step 1: Assume that the algorithm is initially set to LNT

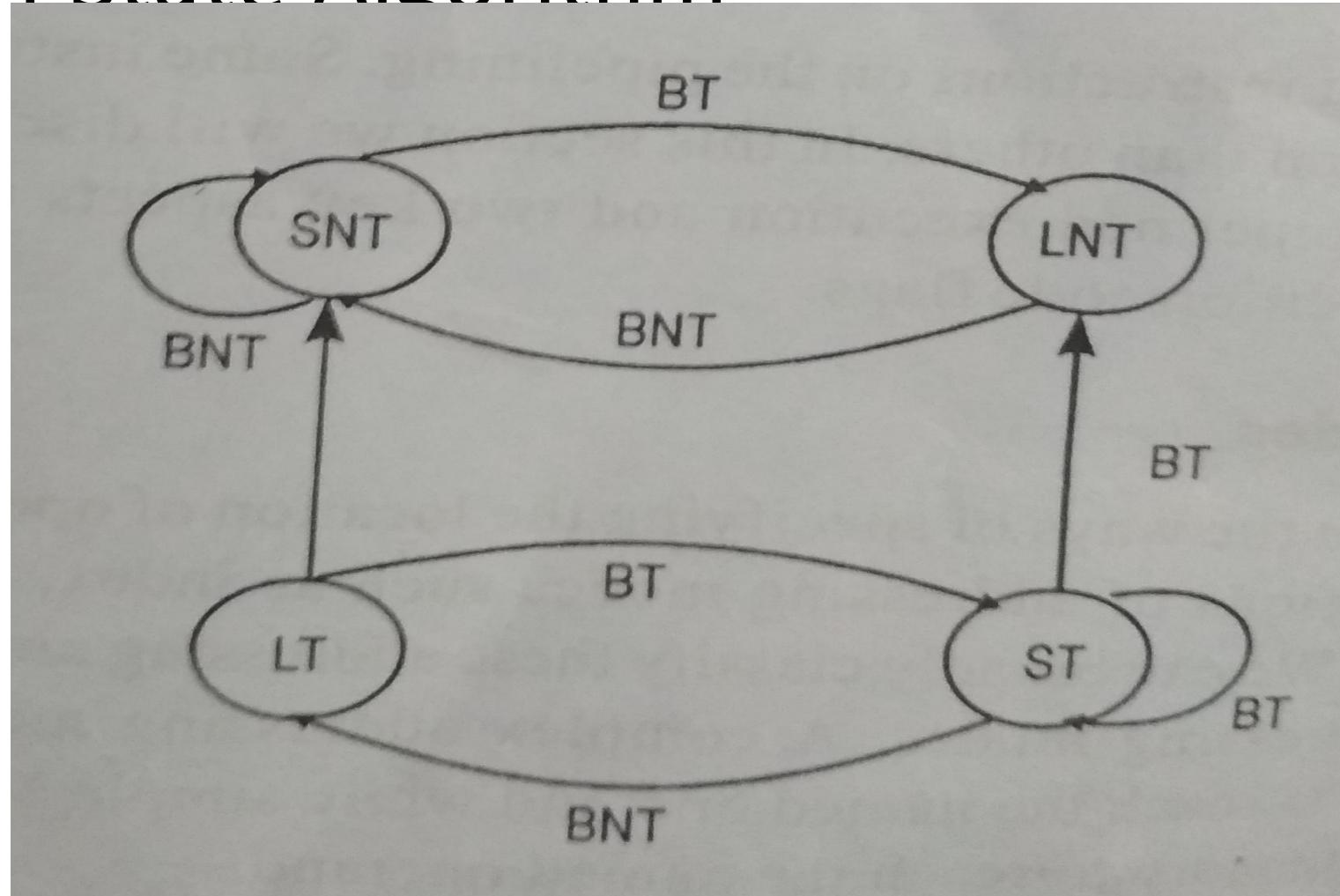
Step 2: If the branch is actually taken changes to ST, otherwise it is changed to SNT.

Step 3: when the branch instruction is encountered, the branch will be taken if the state is either LT or ST and begins to fetch instruction at branch target address, otherwise it continues to fetch the instruction in sequential manner

4 State Algorithm

- . 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

4 State Algorithm





SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

CONTROL HAZARDS

CONTROL HAZARDS

- A Control Hazard occurs if there is a control instruction(e.g.BEQ) because the program counter(PC) following the control instruction is not known until the control instruction is computes if the branch should be taken or not.
- If branch taken,then need to zap/flush instructions.
- There is a performance penalty for branches:
- Need to stall,then may ned to zap(flush)
- Subsequent instructions that have already been fetched.

Contd...

Control Hazards

- Instructions are fetched in stage 1(IF)
- Branch and jump decisions occur in stage 3(EX)
 - i.e. Next PC is not known until 2 cycles after branch/jump
- Can optimize and move branch and jump decision to stage 2
 - i.e. Next PC is not known until 1 cycles after branch/jump

Contd...

Stall(+Zap)

- Prevent PC update
- Clear IF/ID pipeline register
 - Instruction just fetched might be wrong one, so convert to nop
- Allow branch to continue into EX stage
- We can reduce cost of a control hazard by moving branch decision and calculation from EX stage to ID stage. This reduces the cost from flushing two instructions to only flushing one.

Contd...

- ISA says N instructions after branch/jump always executed
 - MIPS has 1 branch delay slot
 - i.e. Whether branch taken or not, instruction following branch is always executed
- Delay Slots can potentially increase performance due to control hazards by putting a useful instruction in the delay slot since the instruction in the delay slot will always be executed.
- Require software (compiler) to make use of delay slot.
- Put NOP in delay slot if not able to put useful instruction in delay slot.

Contd...

- Pipelining and branching don't get along
- Transfer of control (jumps, procedure call/returns, successful branches) cause control hazards
- When a branch is known to succeed, at the Mem stage (but could be done one stage earlier), there are instructions in the pipeline in stages before Mem that
 - need to be converted into “no-op”
 - and we need to start fetching the correct instructions by using the right PC

Contd...

- Branches(conditional, unconditional, call-return)
- Interrupts : asynchronous event(e.g.,I/O)
 - Occurrence of an event checked at every cycle
 - If an interrupt has been raised, don't fetch next instruction,
- flush the pipe, handle the interrupt
- Exceptions(e.g., arithmetic overflow, page fault etc.,)
 - Program and data dependent(repeatable), hence "synchronous"

RESOLVING CONTROL HAZARDS

- Detecting a potential control hazard is easy
 - Look at the opcode
- We must insure that the state of the program is not changed until the outcome of the branch is known.
Possibilities are:
 - Stall as soon as opcode is detected(cost 3 bubbles; same type of logic as for the load stall but for 3 cycles instead of 1)
 - Assume that branch won't be taken(cost only if branch is taken)
 - Use some predictive techniques

INFLUENCE ON INSTRUCTION SETS

OVERVIEW

- Some instructions are much better suited to pipeline execution than others.
- Addressing modes
- Conditional code flags

ADDRESSING MODES

- Addressing modes include simple ones and complex ones.
- 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:
 - Side effects
 - The extent to which complex addressing modes cause the pipeline to stall
 - Whether a given mode is likely to be used by compilers

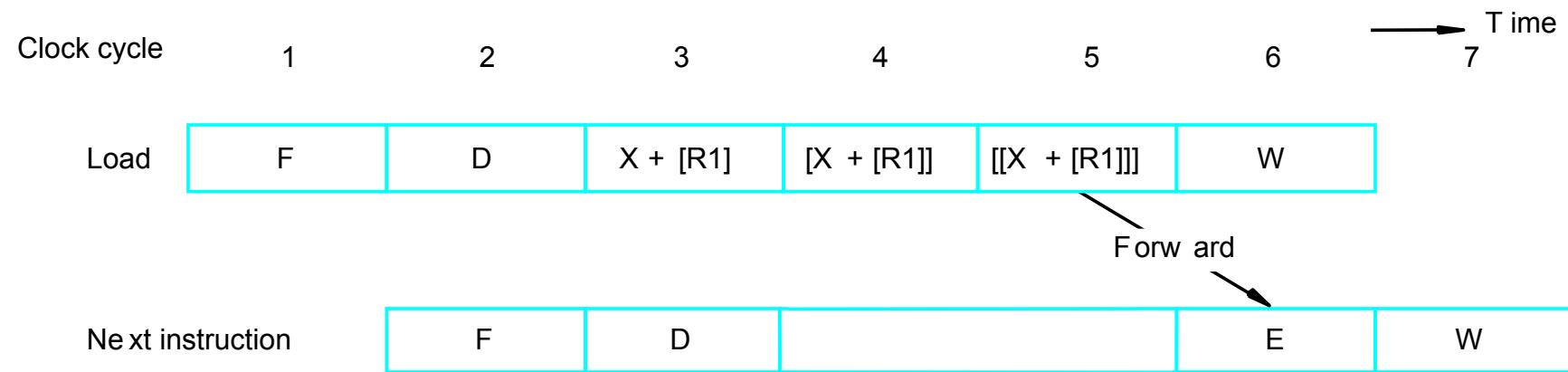
RECALL

Load X(R1), R2

Load (R1), R2

COMPLEX ADDRESSING MODE

Load $(X(R1))$, R2



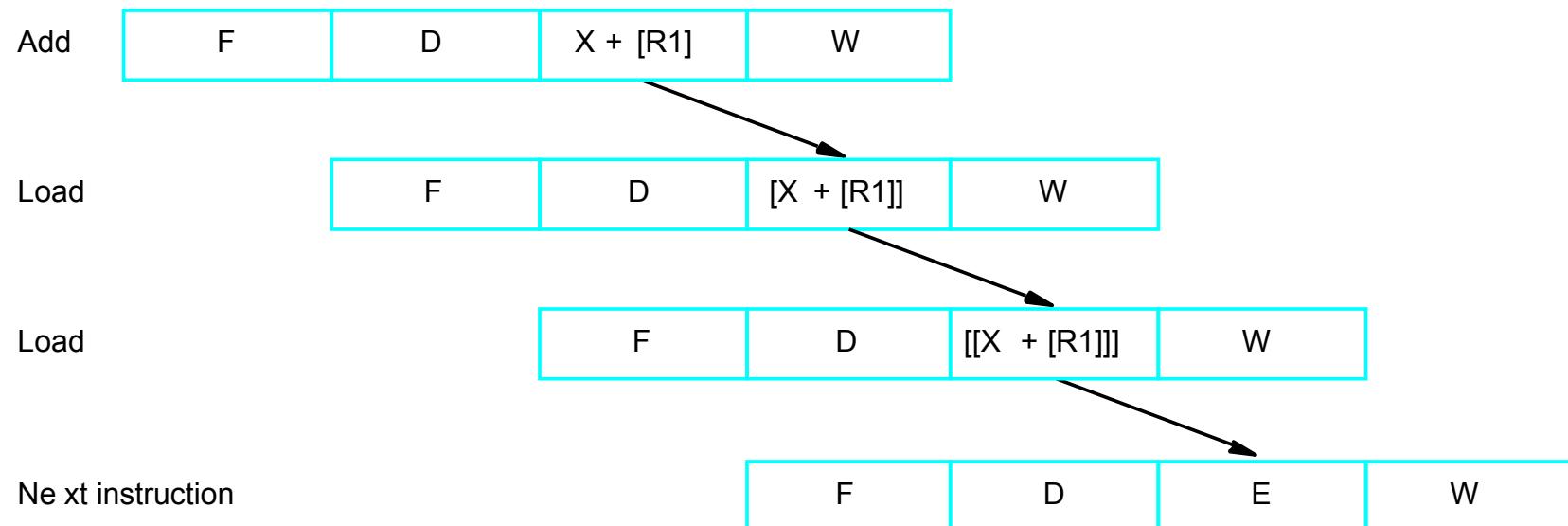
(a) Complex addressing mode

SIMPLE ADDRESSING MODE

Add #X, R1, R2

Load (R2), R2

Load (R2), R2



(b) Simple addressing mode

ADDRESSING MODES

- In a pipelined processor, complex addressing modes do not necessarily lead to faster execution.
- Advantage: reducing the number of instructions / program space
- Disadvantage: cause pipeline to stall / more hardware to decode / not convenient for compiler to work with
- Conclusion: complex addressing modes are not suitable for pipelined execution.

ADDRESSING MODES

- Good addressing modes should have:
 - Access to an operand does not require more than one access to the memory
 - Only load and store instruction access memory operands
 - The addressing modes used do not have side effects
- Register, register indirect, index

CONDITIONAL CODES

- If an optimizing compiler attempts to reorder instruction to avoid stalling the pipeline when branches or data dependencies between successive instructions occur, it 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.

CONDITIONAL CODES

Add R1,R2

Compare R3,R4

Branch=0 . . .

a) A program fragment

Compare R3,R4

Add R1,R2

Branch=0 . . .

b) Instructions reordered

Instruction reordering

CONDITIONAL CODES

Two conclusion:

- To provide flexibility in reordering instructions, the condition-code flags should be affected by as few instruction as possible.
- The compiler should be able to specify in which instructions of a program the condition codes are affected and in which they are not.

SRM
**INSTITUTE OF SCIENCE AND
TECHNOLOGY**
KATTANKULATHUR, Chennai.

**18CSC203J - COA
PARALLELISM**

Course Learning Rationale (CLR)

CLR-4 : *Study about parallel processing and performance considerations.*

Course Learning Outcome (CLO)

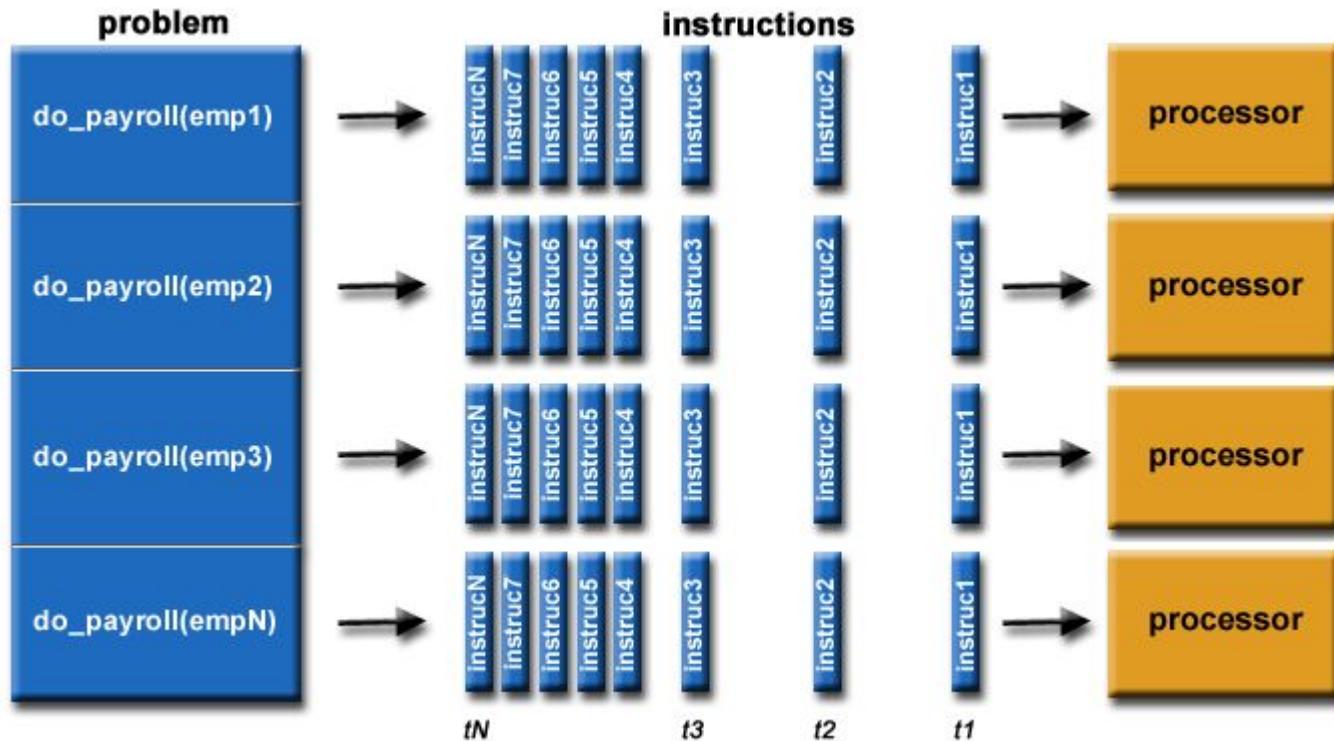
CLO-4 : *Analyze concepts of parallelism and multi-core processors*

Contents

- Parallelism
- Need for Parallelism
- Types of Parallelism
- Applications of Parallelism
- Parallelism in Software
 - Instruction Level Parallelism
 - Data Level Parallelism
- Challenges in Parallelism
- Architecture of Parallel System
 - Flynn's Classification
 - SISD , SIMD
 - MISD, MIMD
- Hardware Multi Threading
 - Coarse Grain Parallelism
 - Fine Grain Parallelism
- Uni-Processor and Multi Processor
- Multi-Core Processor
- Memory in Multi-Processor System
- Cache Coherency in Multi-Processor System
- MESI Protocol for Multi-Processor System

Parallelism

- Executing two or more operations at the same time is known as parallelism.



Parallel Processing

- Parallel processing improves the performance by executing two or more instructions simultaneously
- A *parallel computer* is a set of processors that are able to work cooperatively to solve a computational problem.
- On a single processor may have Two or more ALUs which work concurrently to increase throughput
- The system may have two or more processors operating concurrently (Eg: i3,i5,i7)

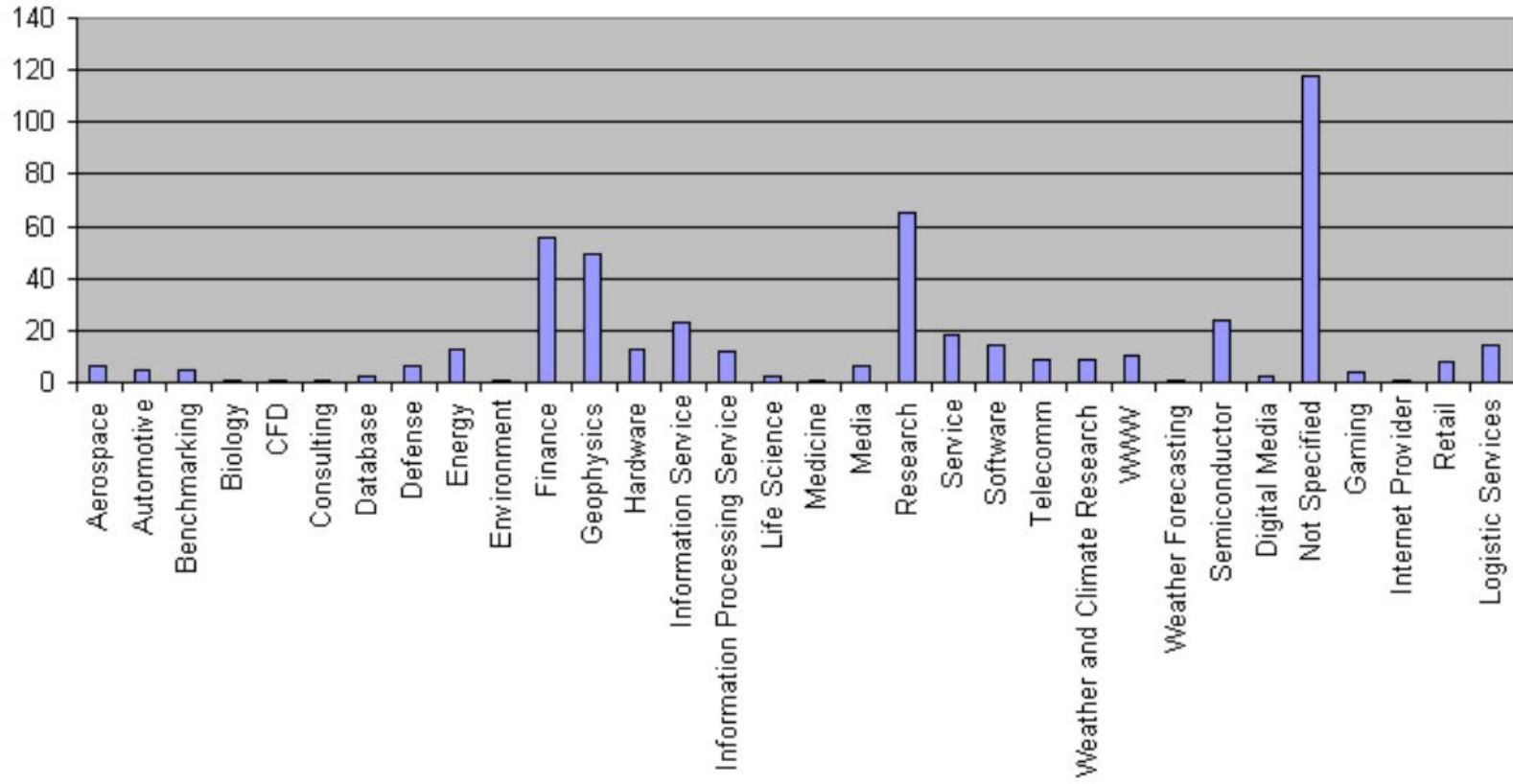
Goals of Parallelism

- To increase the computational speed (ie) to reduce the amount of time that you need to wait for a problem to be solved
- To increase throughput (ie) the number of processes completed on unit time
- To improve the performance of the computer for a given clock speed
- To solve bigger problems that might not fit in the limited memory of a single CPU
- To take advantage of non-local resources when the local resources are finite

Applications of Parallelism

- Numeric Weather Prediction
- Socio Economics
- Finite Element Analysis
- Artificial Intelligence and Automation
- Genetic Engineering
- Weapon Research and Defense
- Medical Applications
- Remote Sensing Applications

Applications of Parallelism



Types of Parallelism

1. Hardware Parallelism
2. Software Parallelism

- Hardware Parallelism :

It is based on the processor multiplicity and machine architecture.

One way to characterize the parallelism in a processor is by the number of instruction issues per machine cycle. If a processor sends k instructions per machine cycle, then it is called a k -issue processor

Software Parallelism

- Determined by the dependency among the sub tasks
- The flow control graph used to represent the degree of parallelism in the program
- The flow graphs shows the possible ways in which the sub programs can be executed in parallel.
- Software parallelism can be exhibited by the algorithm, programming pattern (threads) and optimization on compilers
- Parallelism in a program varies during the execution period .
- It limits the sustained performance of the processor.

Example

Example:

$$A = L1 * L2 + L3 * L4$$

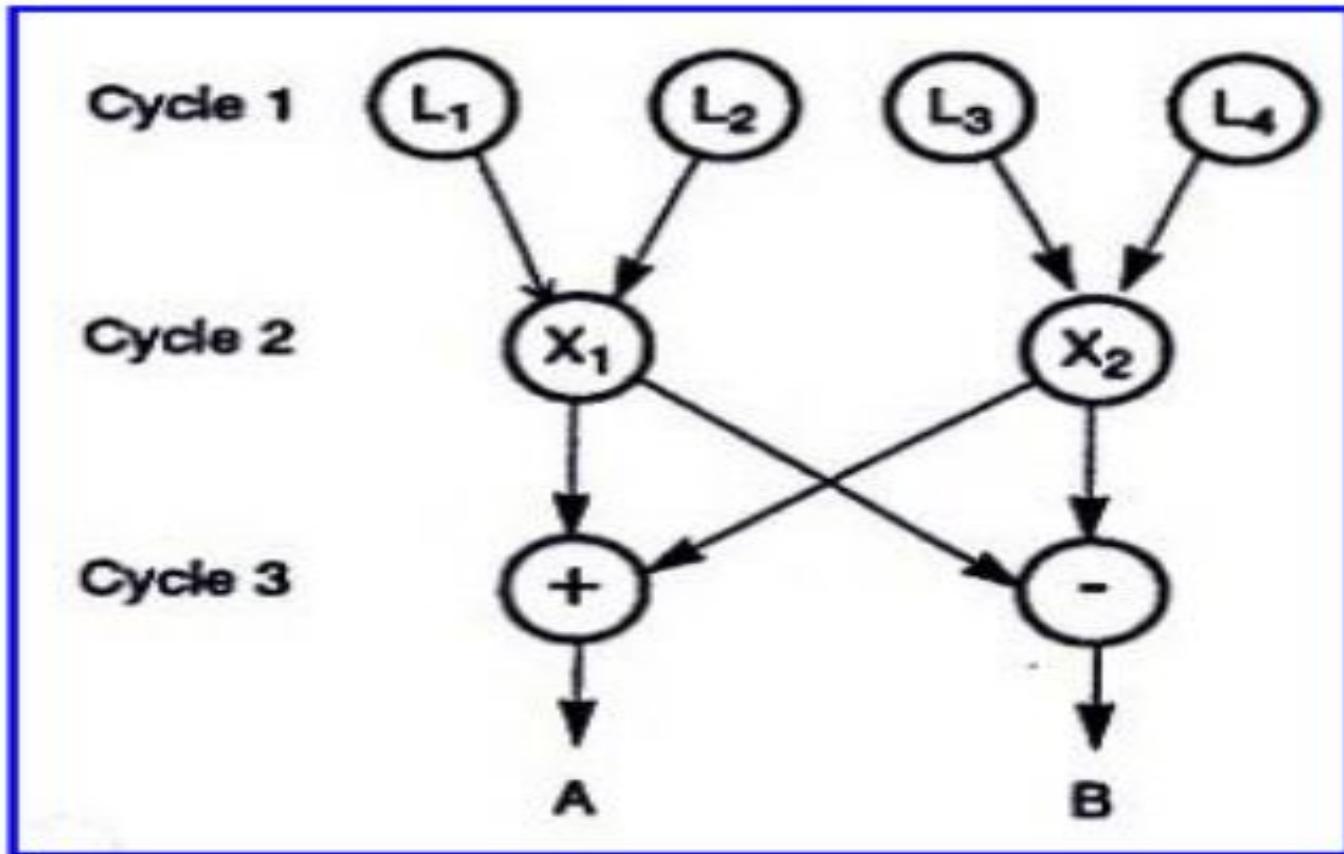
$$B = L1 * L2 - L3 * L4$$

Software Parallelism:

There are 8 instructions

- FOUR Load instructions(L1,L2,L3&L4).
- TWO Multiply instructions(X1&X2)
- ONE Add instruction(+)
- ONE Subtract instruction(-)

The parallelism varies from 4 to 2 in three cycles.



$$\text{Average S/W Parallelism} = \frac{8 \text{ cycles}}{3 \text{ cycles}} = \frac{8}{3} = 2.67$$

Hardware Parallelism - Example

Parallel Execution:

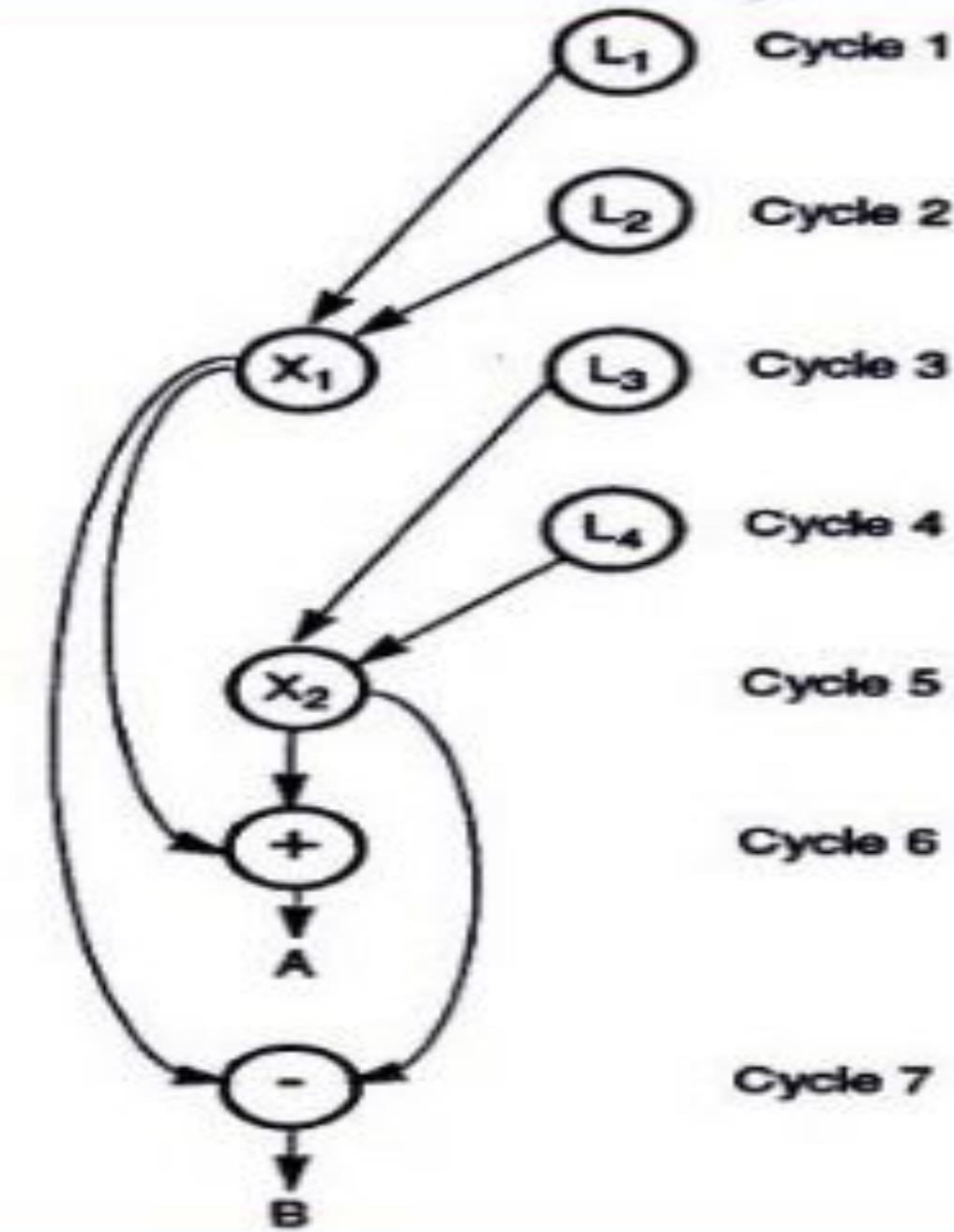
Using TWO-issue processor:

The processor can execute one memory access(Load / Store) and one arithmetic operation(multiply/add/subtract) simultaneously.

The program must execute in 7 cycles.

The H/w parallelism average is $8/7=1.14$.

It is clear from this example that there is a mismatch between the s/w and h/w parallelism.



Software Parallelism - Types

Instruction level parallelism

Task-level parallelism

Data parallelism

Transaction level parallelism

Instruction Level Parallelism

- Instruction level Parallelism (ILP) is a measure of how many operations (instructions) can be performed in parallel at the same time in a computer program.
- Parallel instructions are set of instructions that do not depend on each other for execution
- ILP allows the compiler and processor to overlap the execution of multiple instructions or even to change the order in which instructions are executed.

Eg. Instruction Level Parallelism

Consider the following example

1. $x = a + b$
2. $y = c - d$
3. $z = x * y$

Operation depends on the results of 1 & 2

So ‘Z ‘ cannot be calculated until X & Y are calculated

But 1 & 2 do not depend on any other. So they can be computed simultaneously.

- If we assume that each operation can be completed in one unit of time then these 3 operations can be completed in 2 units of time .
- ILP factor is $3/2=1.5$ which is greater than without ILP.
- A superscalar CPU architecture implements ILP inside a single processor which allows faster CPU throughput at the same clock rate.

Instruction-level parallelism (ILP)

Executes the multiple instructions on same instruction streams simultaneously.

These are generated and managed by either hardware (superscalar) or by compiler (VLIW) – Limited in practice by data and control dependences

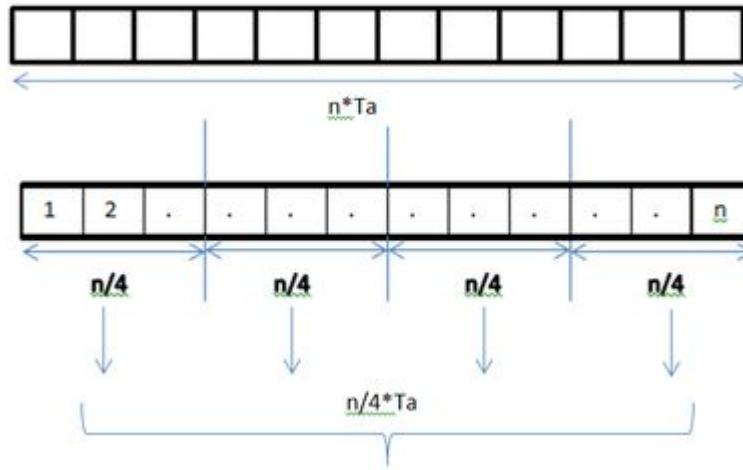
Data-Level Parallelism (DLP)

- **Data parallelism** is parallelization across multiple processors in **parallel computing** environments.
- It focuses on distributing the **data** across different nodes, which operate on the **data** in **parallel**.
- Instructions from a single stream operate concurrently on several data
- Limited by non-regular data manipulation patterns and by memory bandwidth

DLP - example

- Let us assume we want to sum all the elements of the given array of size n and the time for a single addition operation is T_a time units.
- In the case of sequential execution, the time taken by the process will be $n*T_a$ time unit
- if we execute this job as a data parallel job on 4 processors the time taken would reduce to $(n/4)*T_a +$ merging overhead time units.

DLP in Adding elements of array



DLP in matrix multiplication

$$\begin{array}{c}
 \left(\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 1 & 3 & 2 \end{array} \right) \times \left(\begin{array}{cc} 10 & 11 \\ 7 & 5 \\ 2 & 4 \end{array} \right) = \left(\begin{array}{cc} 1*10+2*7+3*2 & 1*11+2*5+3*4 \\ 4*10+5*7+6*2 & 4*11+5*5+6*4 \\ 1*10+3*7+2*2 & 1*11+3*5+2*4 \end{array} \right)
 \end{array}$$

3 x 3 3 x 2 3 x 2

— $O(m*n*k)$ when executed in parallel using $m*k$ processors.

- The locality of data references plays an important part in evaluating the performance of a data parallel programming model.
- Locality of data depends on the memory accesses performed by the program as well as the size of the cache.

Thread-Level or Task-Level Parallelism (TLP)

- Multiple program/Task threads are created for same application and executed simultaneously
- These task threads are created and managed by compiler and hardware, however the program threads are created by programmer and managed by compiler and hardware during run time.
- User rarely suffers by communication/synchronization overheads

Challenges under Parallel Processing

- The Hardware Model
- Limitations on the Window Size and Maximum Issue Count
- The Effects of Realistic Branch and Jump Prediction
- The Effects of Finite Registers
- The Effects of Imperfect Alias Analysis

Flynn's Classification

- Was proposed by researcher Michael J. Flynn in 1966.
- It is the most commonly accepted taxonomy of computer organization.
- In this classification, computers are classified by whether it processes a single instruction at a time or multiple instructions simultaneously, and whether it operates on one or multiple data sets.

Flynn's Classification

- This taxonomy distinguishes multi-processor computer architectures according two independent dimensions
 - Instruction stream
 - Data stream.
- An instruction stream is sequence of instructions executed by machine.
- A data stream is a sequence of data including input, partial or temporary results used by instruction stream.
- Each of these dimensions can have only one of two possible states: **Single or Multiple**.
- Flynn's classification depends on the distinction between the performance of control unit and the data processing unit rather than its operational and structural interconnections.

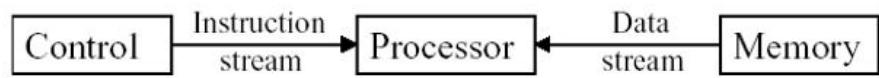
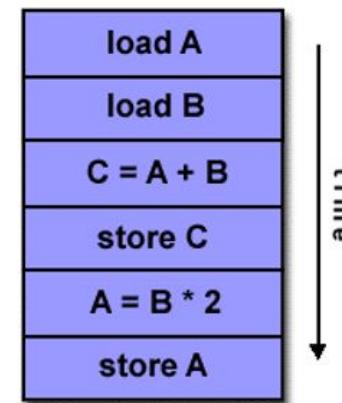
Flynn's Classification

- Four category of Flynn classification

		Single	DATA STREAM	Multiple
		Single	SISD	SIMD
		Multiple	MISD	MIMD
INSTRUCTION STREAM				

SISD

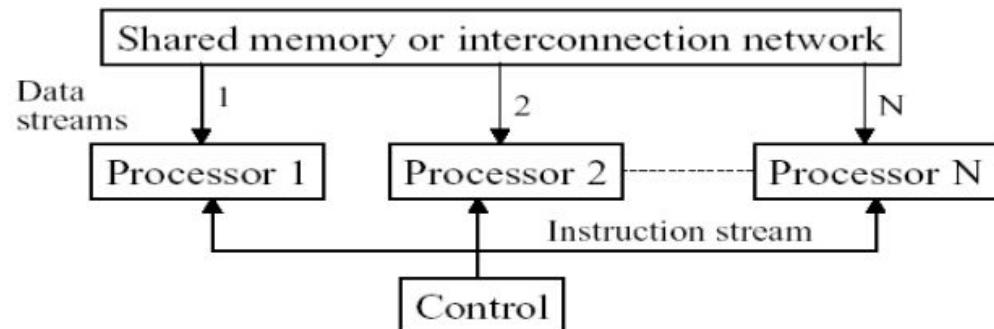
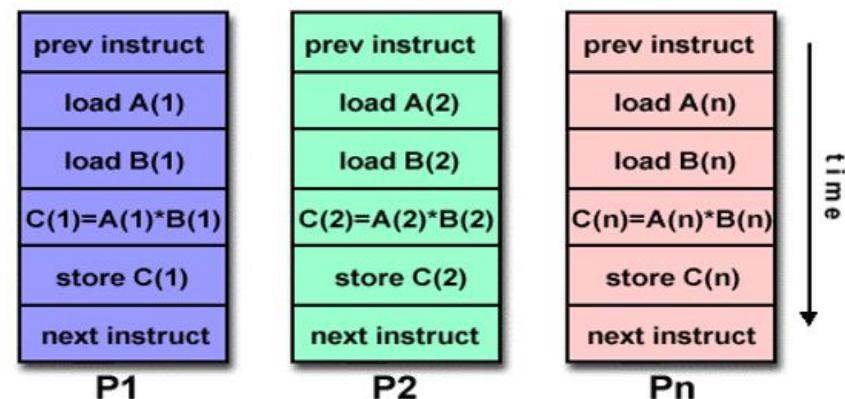
- They are also called scalar processor i.e., one instruction at a time and each instruction have only one set of operands.
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle.
- Single data: only one data stream is being used as input during any one clock cycle.
- Deterministic execution.
- Instructions are executed sequentially.
- SISD computer having one control unit, one processor unit and single memory unit.



SIMD

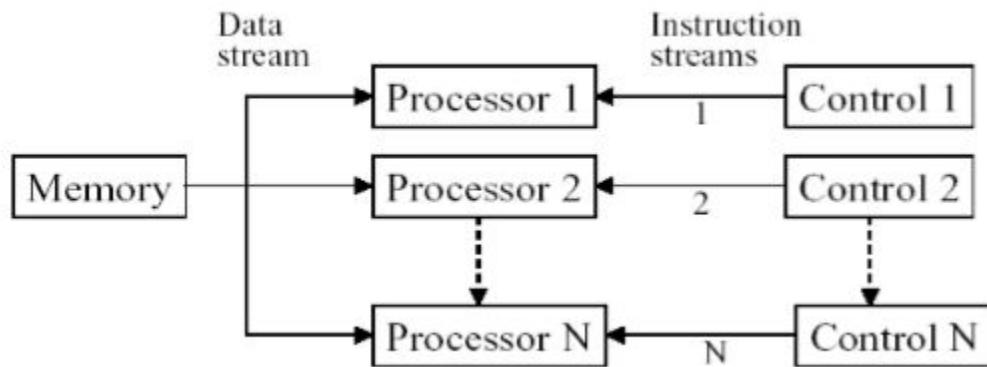
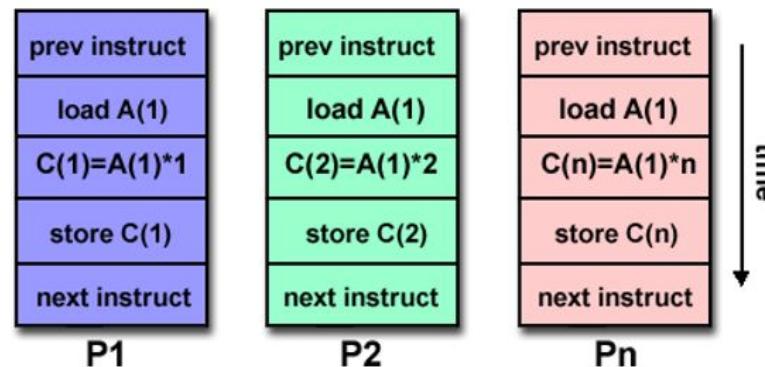
- A type of parallel computer.
- Single instruction: All processing units execute the same instruction issued by the control unit at any given clock cycle .
- Multiple data: Each processing unit can operate on a different data element as shown if figure below the processor are connected to shared memory or interconnection network providing multiple data to processing unit

- single instruction is executed by different processing unit on different set of data



- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction.
- A single data stream is forwarded to different processing unit which are connected to different control unit and execute instruction given to it by control unit to which it is attached.

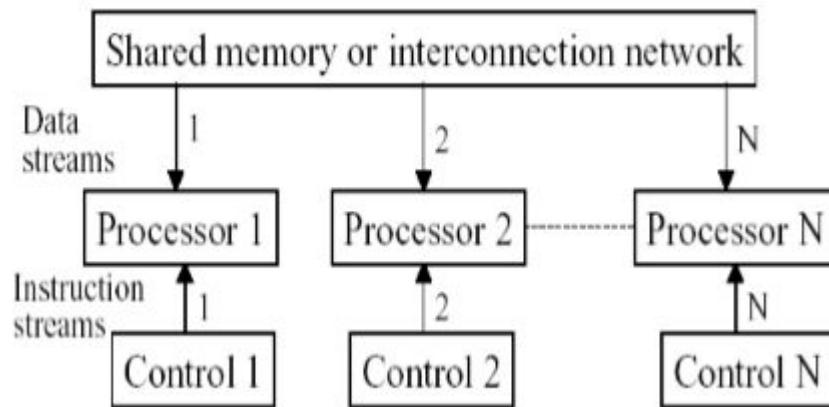
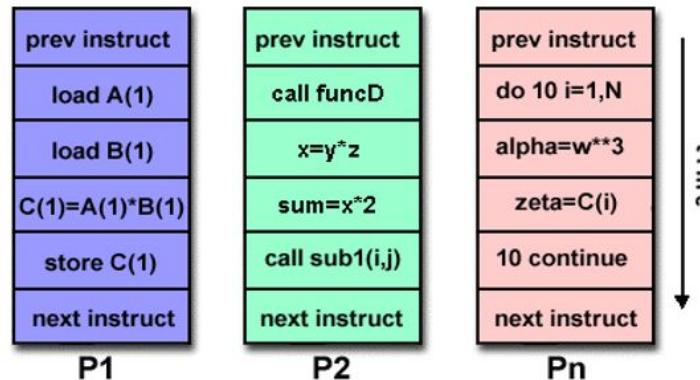
- same data flow through a linear array of processors executing different instruction streams



MIMD

- Multiple Instruction: every processor may be executing a different instruction stream.
- Multiple Data: every processor may be working with a different data stream.
- Execution can be synchronous or asynchronous, deterministic or nondeterministic

- Different processor each processing different task.

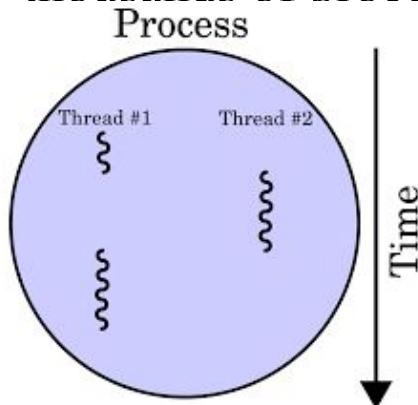


Thread and Multithreads

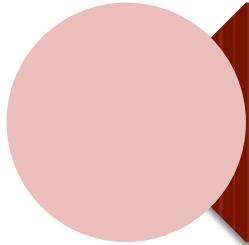
- It is an instruction stream with state (Register/Memory)
- The Thread state is called as thread context (Register state Register context)
- It may belongs to same process or different processes
- Threads in the same program share the same address space
- Under Multi threading/ Multitasking when new thread need execution, processor saves the older thread context and takes up the new thread context.

Hardware Multithreading

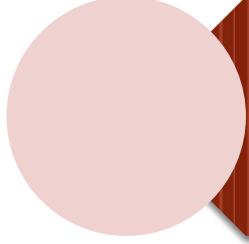
- Hardware multithreading allows multiple threads to share **the functional units of a single processor** in an overlapping fashion to try to utilize the hardware resources efficiently.
- To permit this sharing, the processor must duplicate the independent state of each thread. It Increases the utilization of a processor
- For example, each thread would have a separate copy of register file and program counter. The memory itself can be shared through virtual memory mechanisms, which already support multi-programming.
- In addition, hardware must supportability to change to a different thread relatively quickly. In particular, a thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles while a thread switch can be instantaneous.



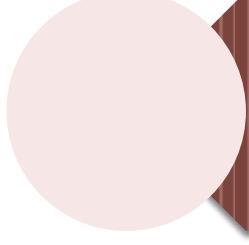
Multithreading Types



Fine Grained



Course Grained



Simultaneous

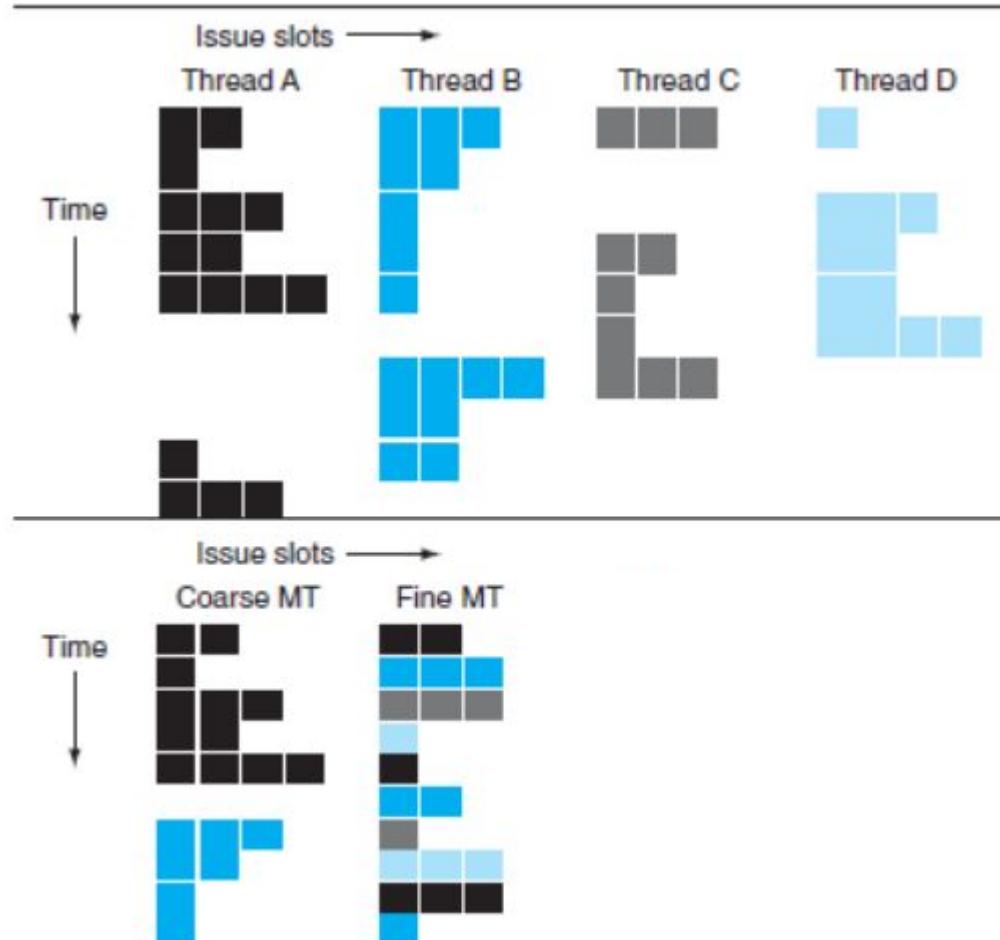
Fine Grained Multi Threading

- Fine-grained multithreading switches between threads on each instruction, resulting in interleaved execution of multiple threads.
- This interleaving is often done in around-robin fashion, skipping any threads that are stalled at that clock cycle.
- To make fine-grained multithreading practical, processor must be able to switch threads on every clock cycle.
- One advantage of fine-grained multithreading is that it can hide throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls.
- The primary disadvantage of fine-grained multithreading is that it slows down execution of individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

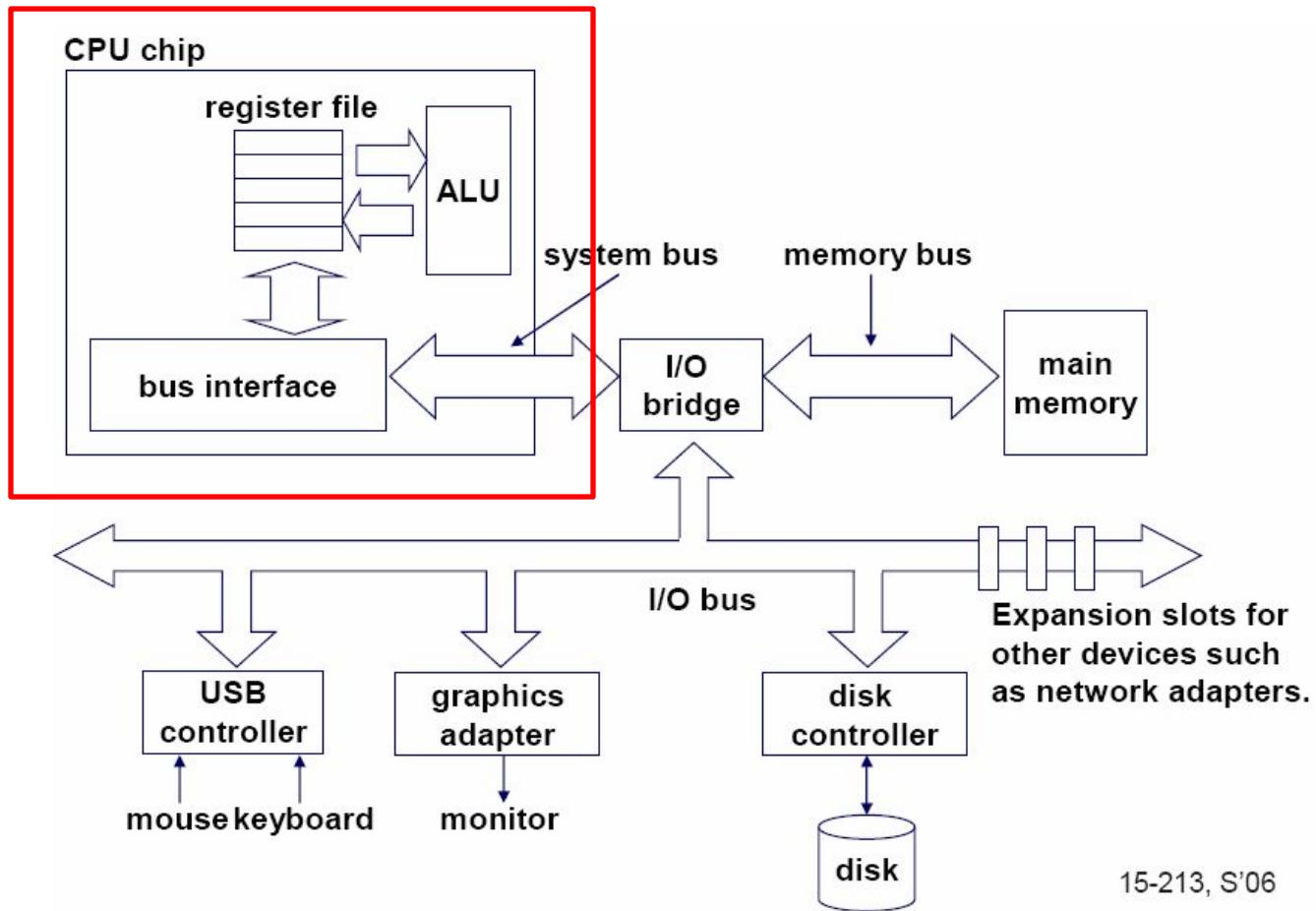
Coarse Grained Multi Threading

- Coarse-grained multithreading was invented as an alternative to fine-grained multithreading.
- Coarse-grained multithreading switches threads only on costly stalls, such as last-level cache misses.
- This change relieves need to have thread switching be extremely fast and is much less likely to slow down execution of an individual thread, since instructions from or threads will only be issued when a thread encounters a costly stall.
- **Drawback:** it is limited in its ability to overcome throughput losses, especially from shorter stalls.
- This limitation arises from pipeline start-up costs of coarse-grained multithreading. Because a processor with coarse-grained multithreading issues instructions from a single thread, when a stall occurs, pipeline must be emptied or frozen.
- The new thread that begins executing after stall must fill pipeline before instructions will be able to complete. Due to start-up overhead, coarse-grained multithreading is much more useful for reducing penalty of high-cost stalls, where pipeline refill is negligible compared to stall time.

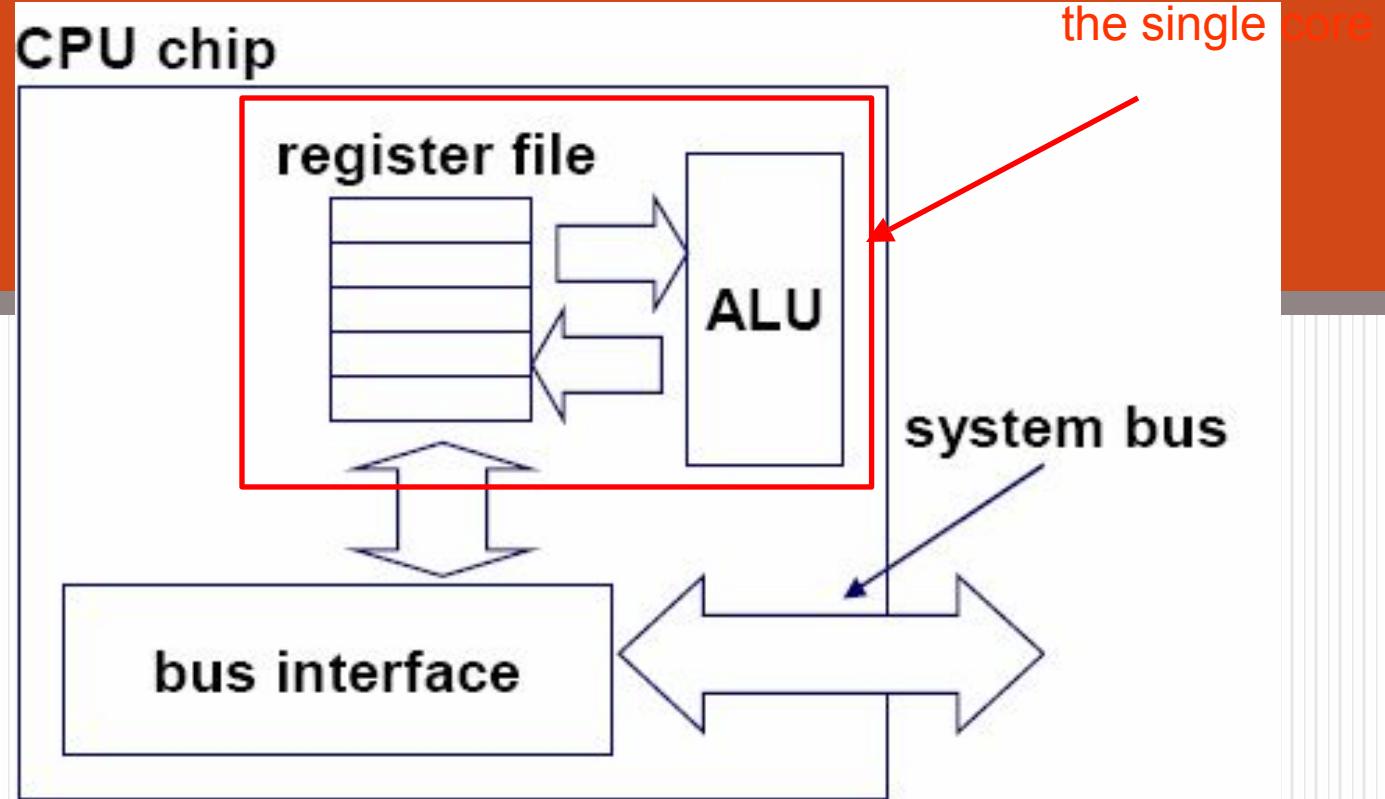
Comparison



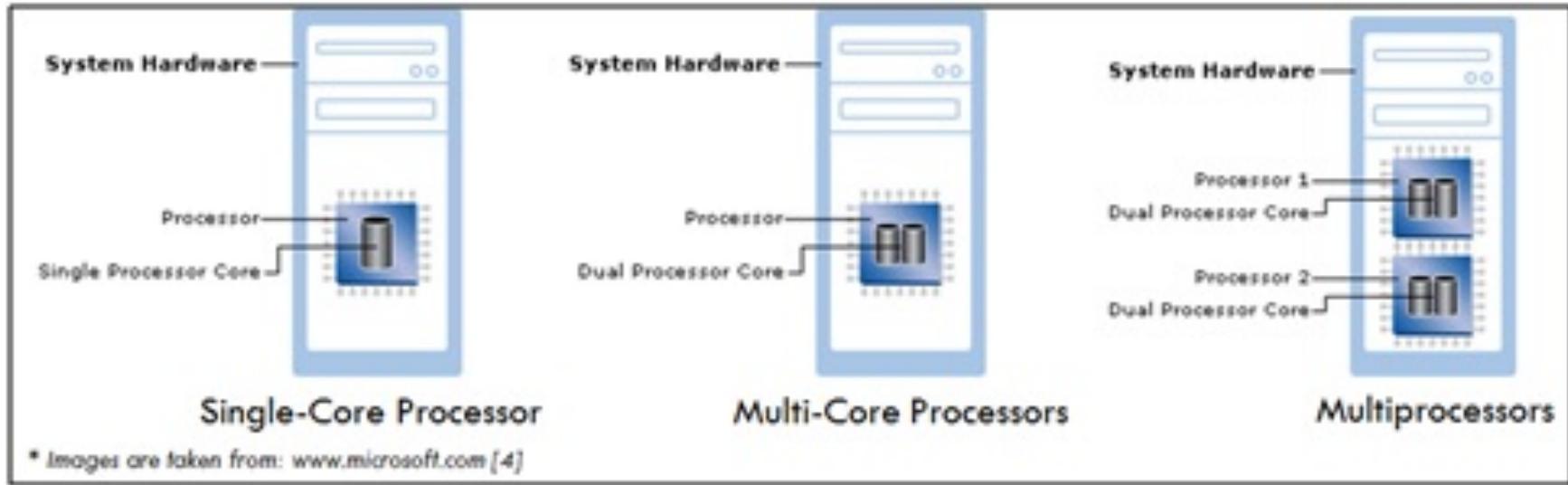
Single-core computer



Single-core CPU chip

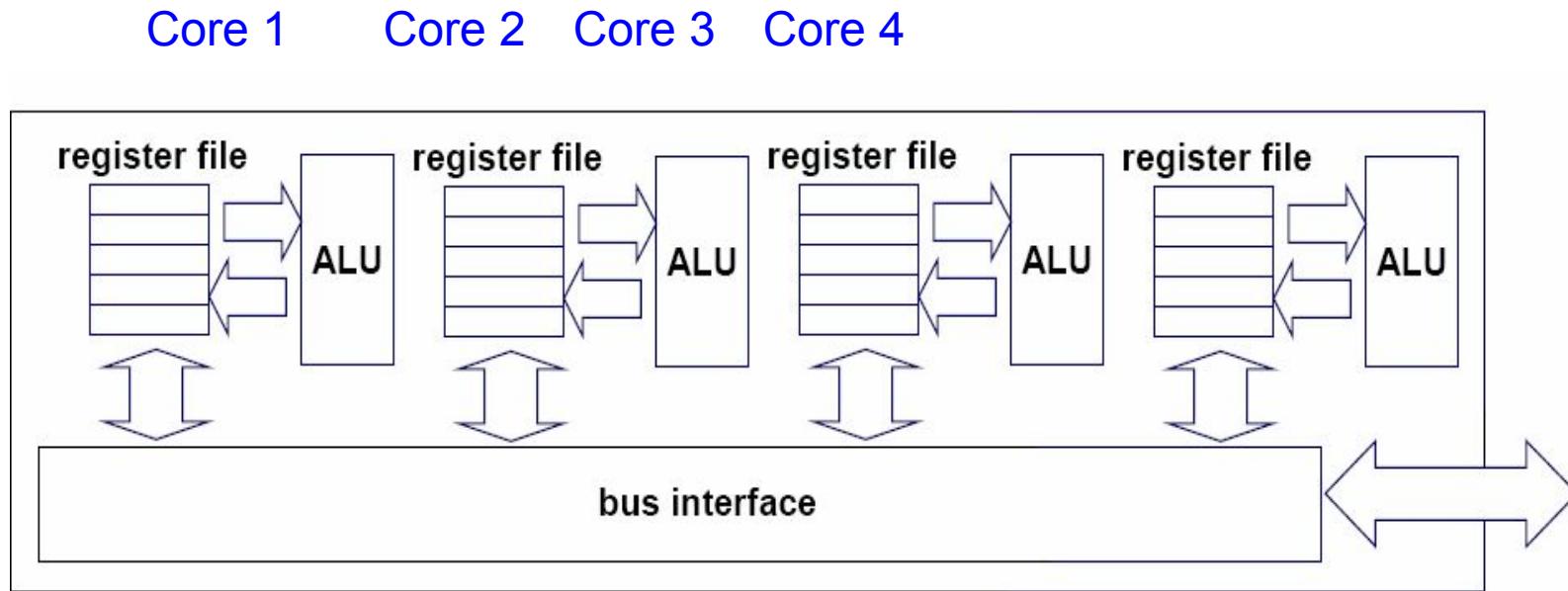


Single Vs Multicore Processors



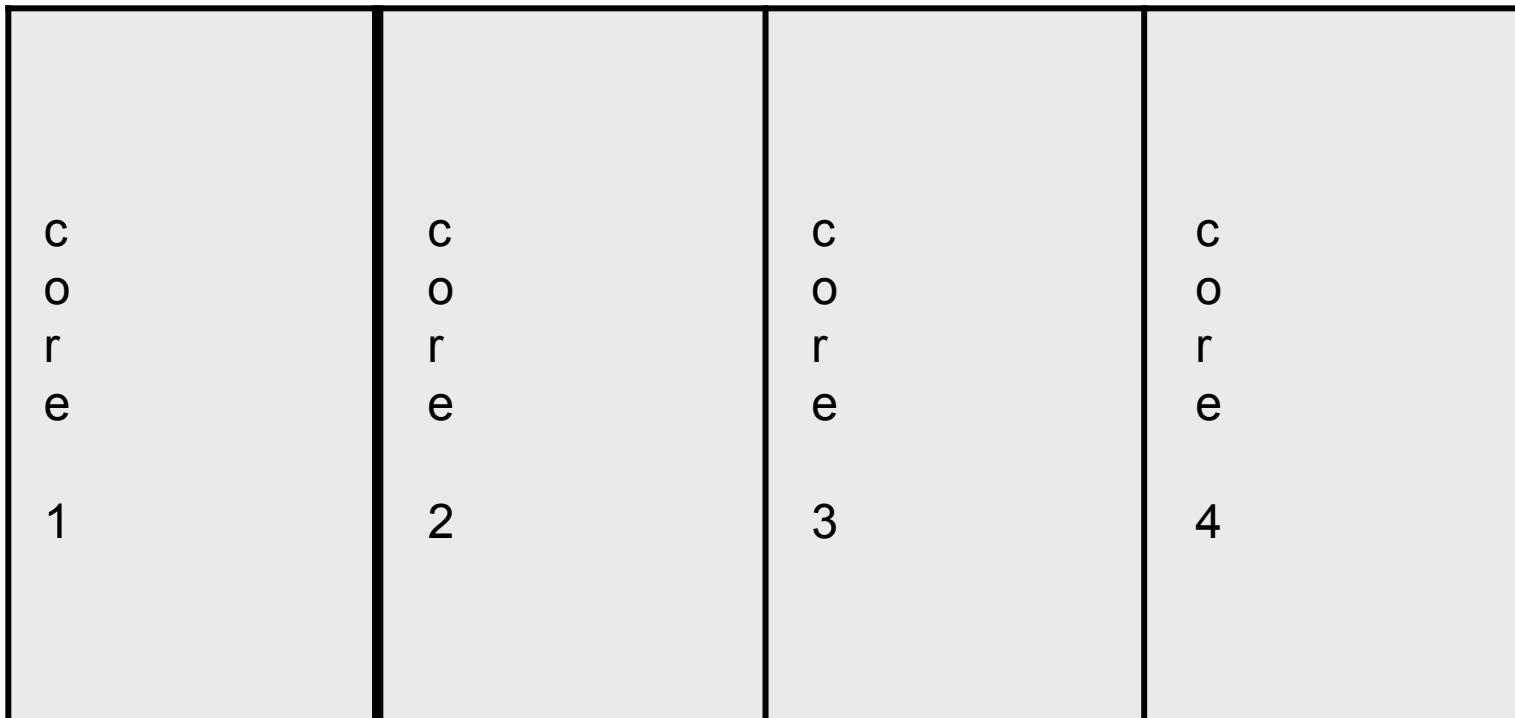
Multi-core architectures

- Replicate multiple processor cores on a single die.



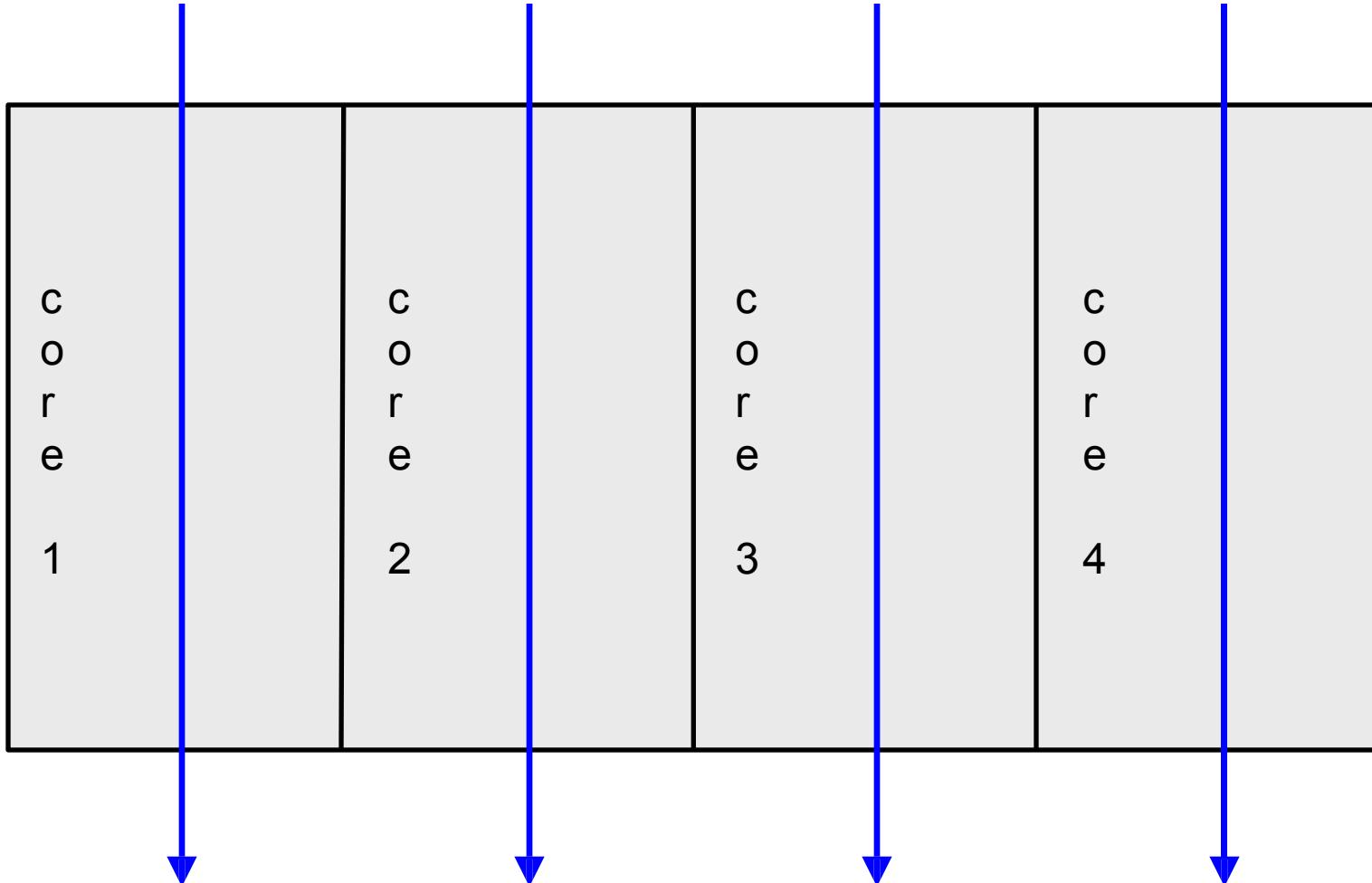
Multi-core CPU chip

- The cores fit on a single processor socket
- Also called CMP (Chip Multi-Processor)

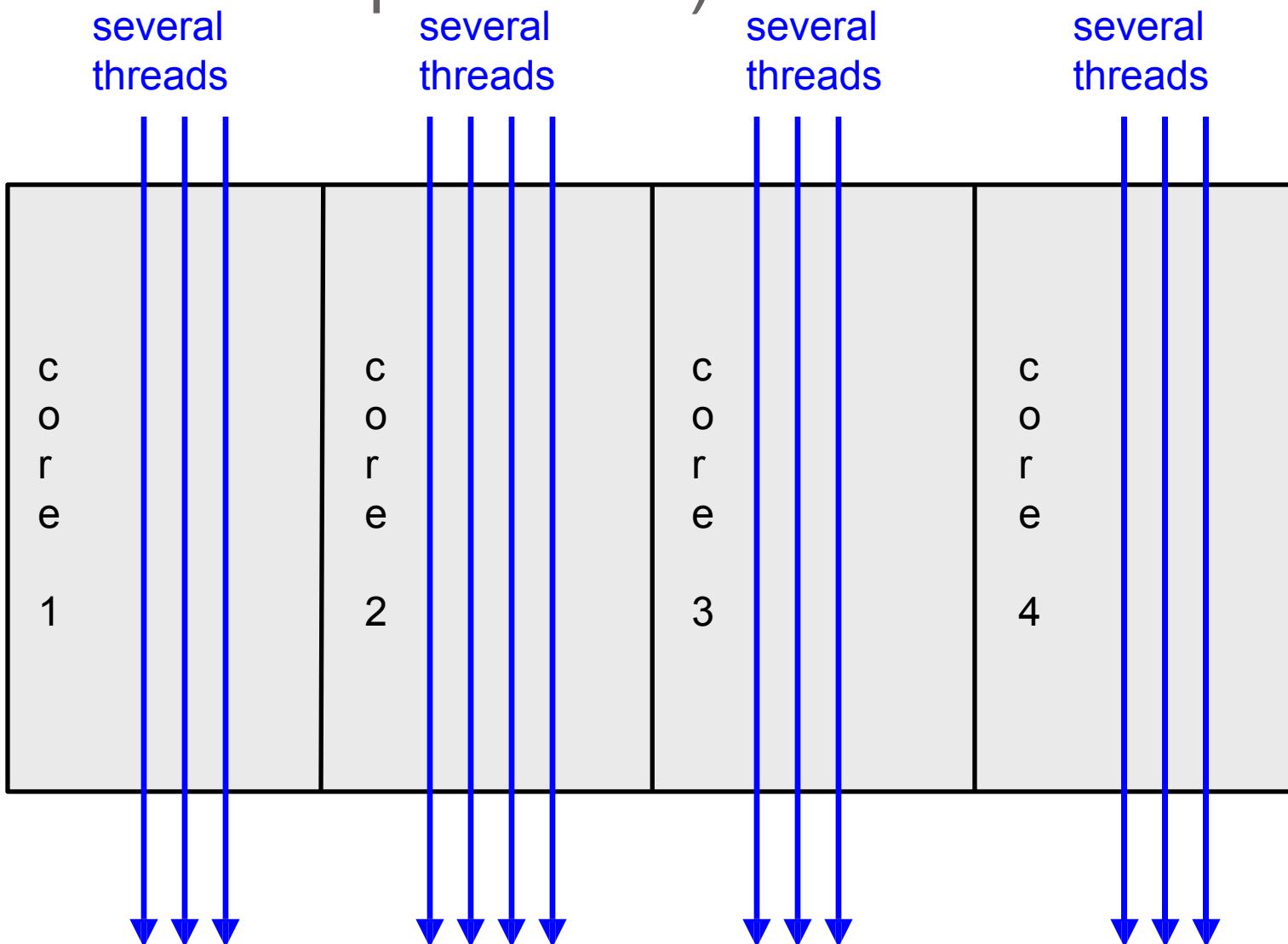


The cores run in parallel

thread 1 thread 2 thread 3 thread 4



Within each core, threads are time-sliced (just like on a uniprocessor)



Memory in Multiprocessor System

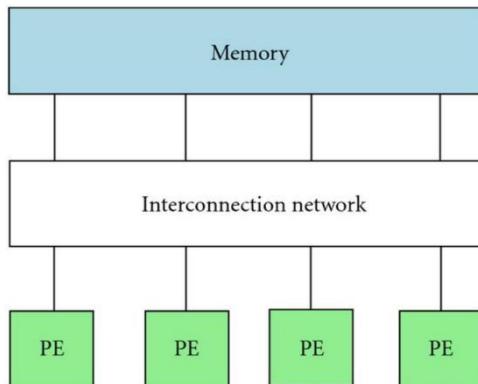
Architectures

Shared common
memory

Distributed memory.
(Unshared)

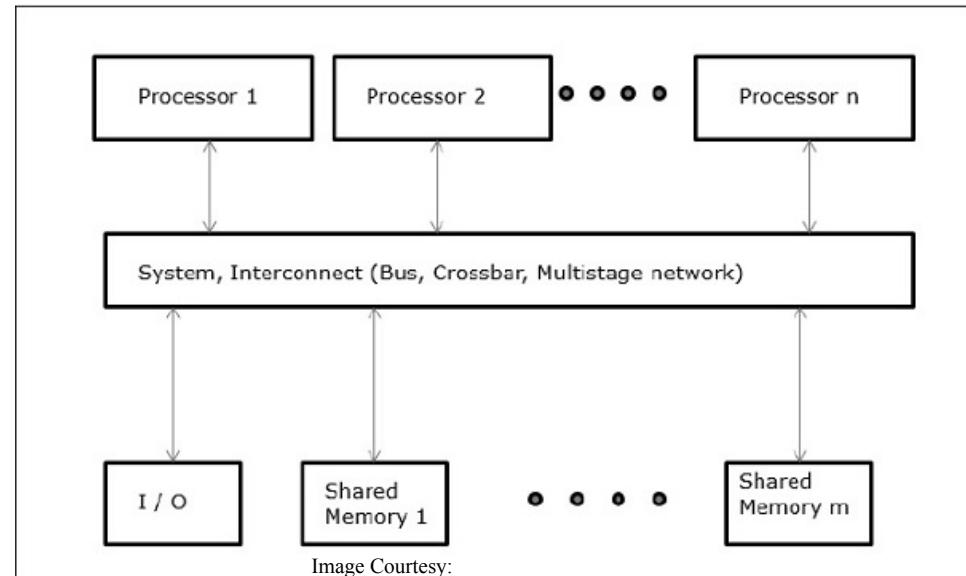
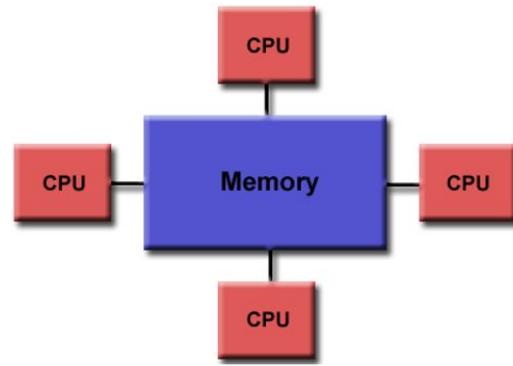
Shared Memory Multiprocessors

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: **UMA** , **NUMA** and **COMA**.



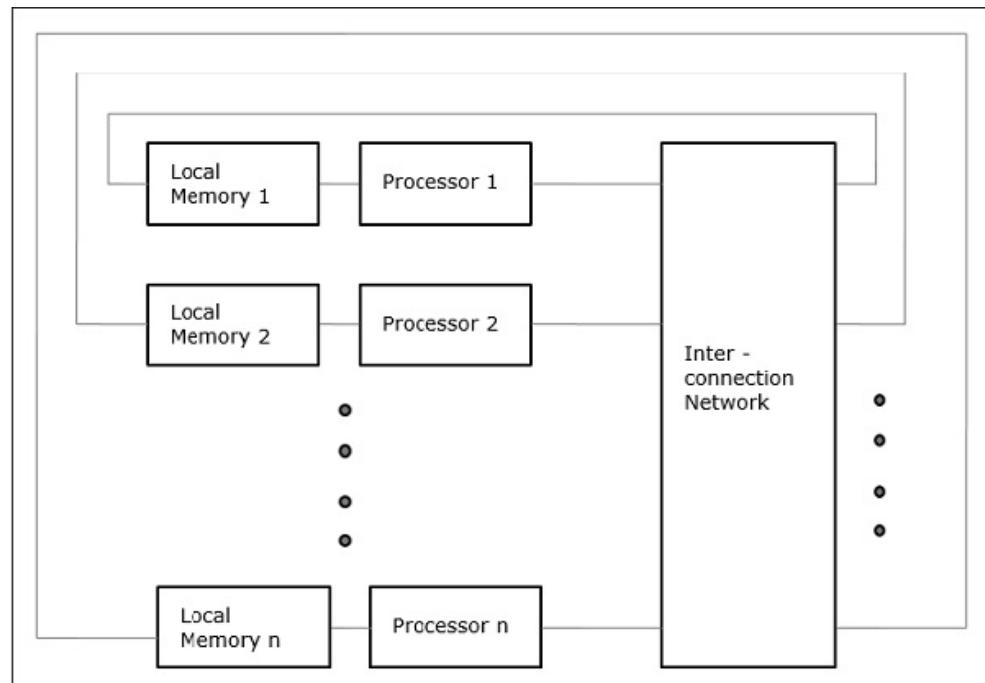
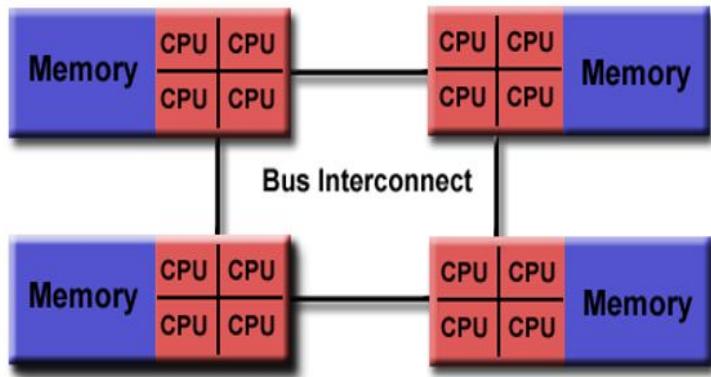
Uniform Memory Access (UMA)

- Most commonly represented today by Symmetric Multiprocessor (SMP) machines.
- Identical processors .
- Equal access and access times to memory .
- Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.



Non-Uniform Memory Access (NUMA)

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories .
- Memory access across link is slower
- If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA



COMA Model

- It is a special case of NUMA machine in which the distributed main memories are converted to caches. All caches form a global address space and there is no memory hierarchy at each processor node.

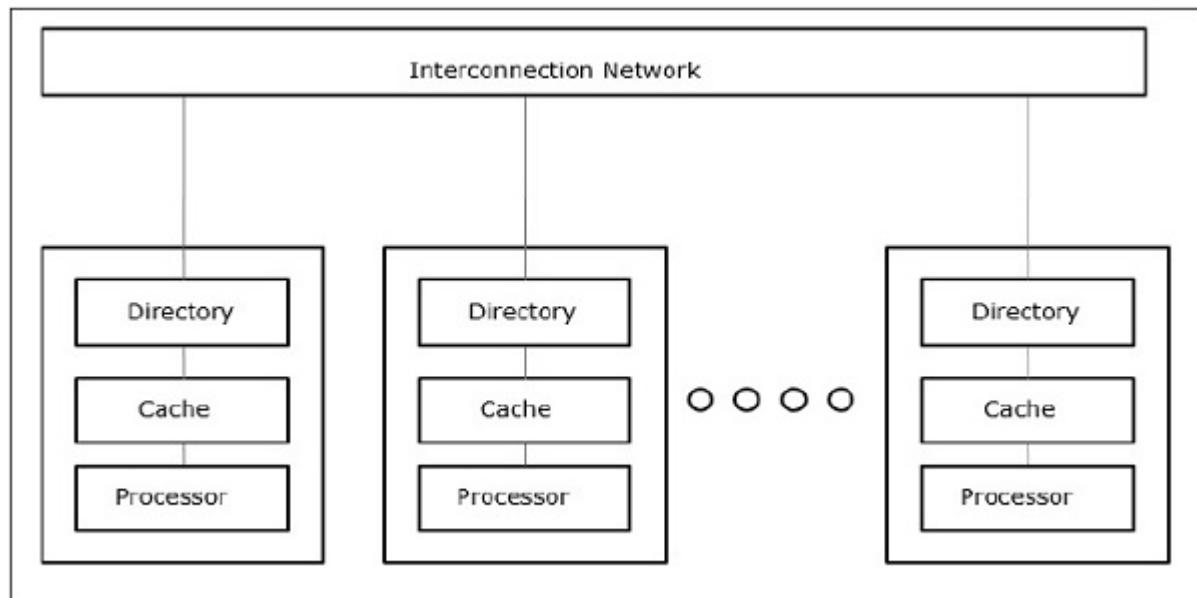
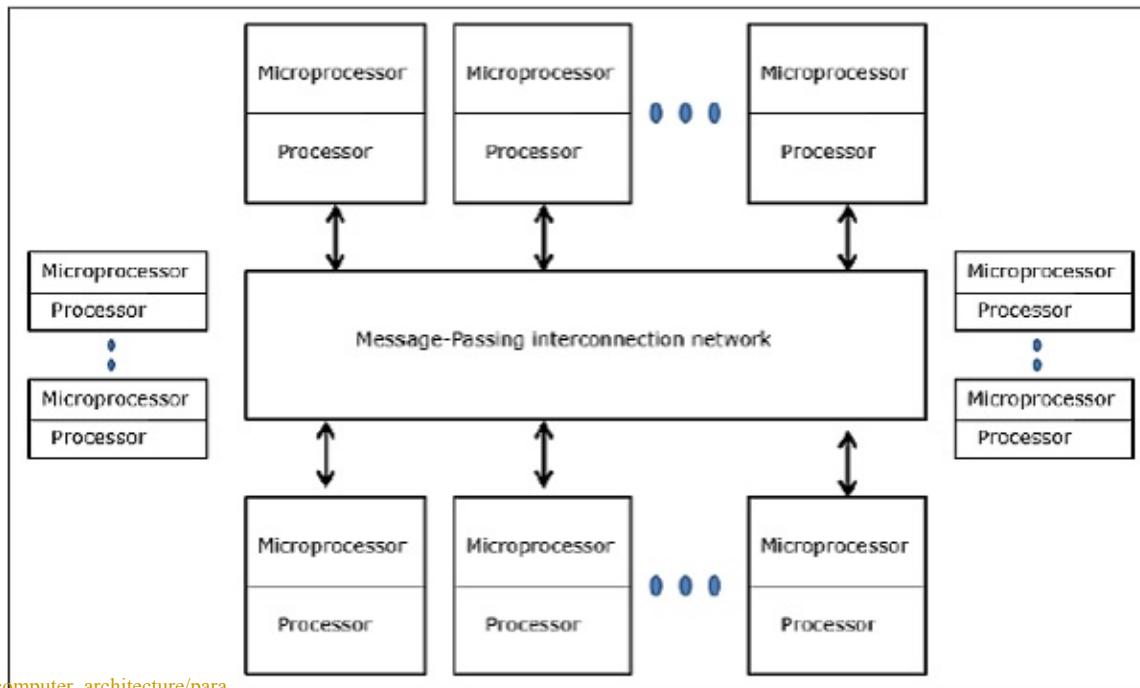


Image Courtesy:

https://www.tutorialspoint.com/parallel_computer_architecture/parallel_computer_architecture_models.htm

Distributed Memory Systems

- Distributed memory systems each processor has its own memory and it require a communication network to connect inter-processor memory.
- Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently.



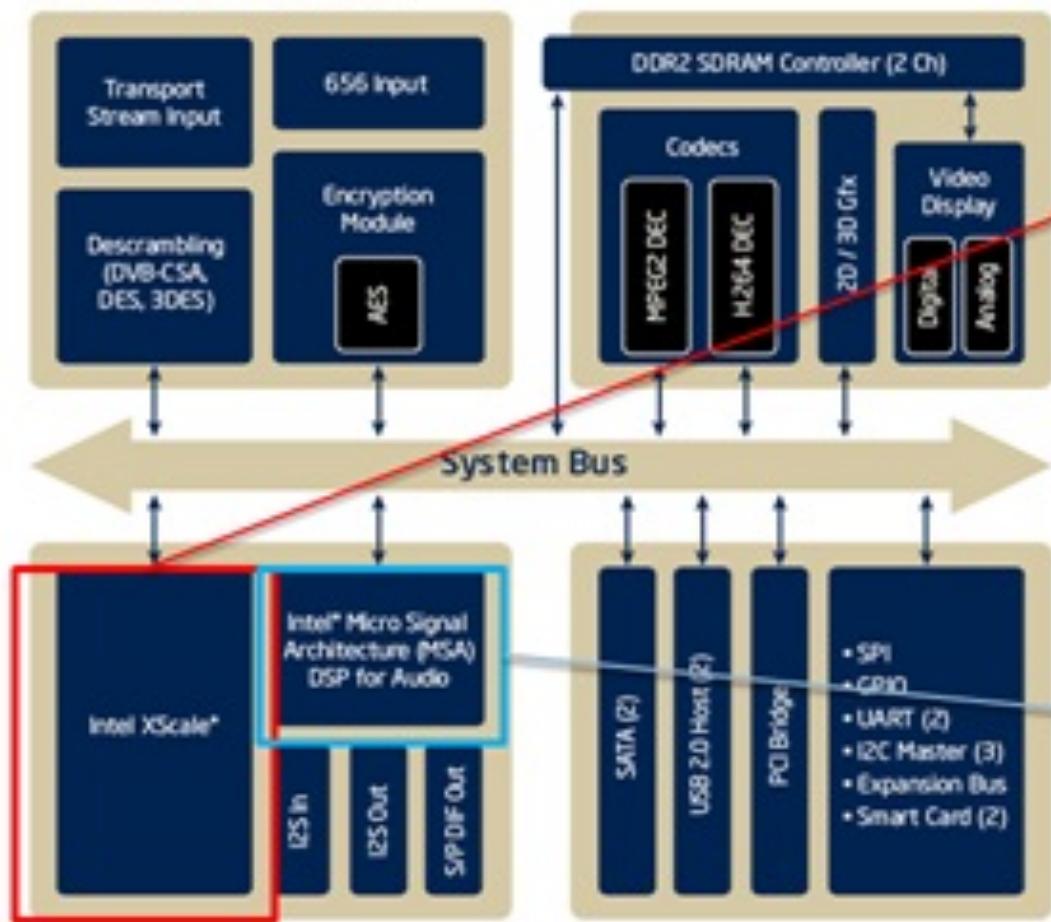
- Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated.
- Synchronization between tasks is likewise the programmer's responsibility

- Cache Coherency in Multiprocessor Systems

Examples of Multi-Core Processors

- Homogeneous Multi-Core Processor
 - Contains identical processor cores that support the same instruction set architecture (ISA)
- Heterogeneous Multi-Core Processor
 - Consist of non-identical processor cores that support different ISA
 - For example: **Intel CE 2110** Media that consists of an **Intel Xscale processor** core and an **Intel Micro Signal Architecture (MSA) DSP core**.
 - Each processor usually has own specialty

Intel CE 2110 Media



Intel Xscale:

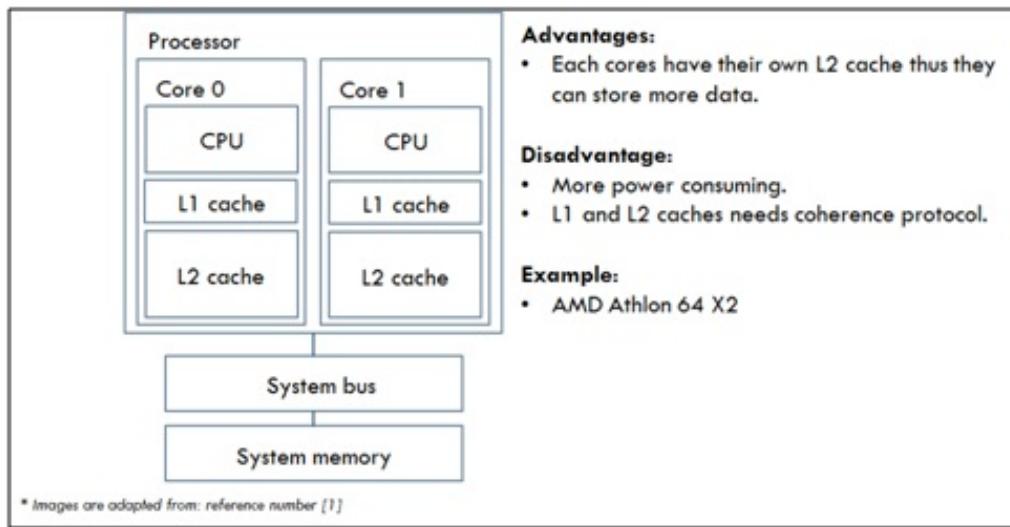
- General Tasks

Intel Micro Signal Architecture (MSA) [8]

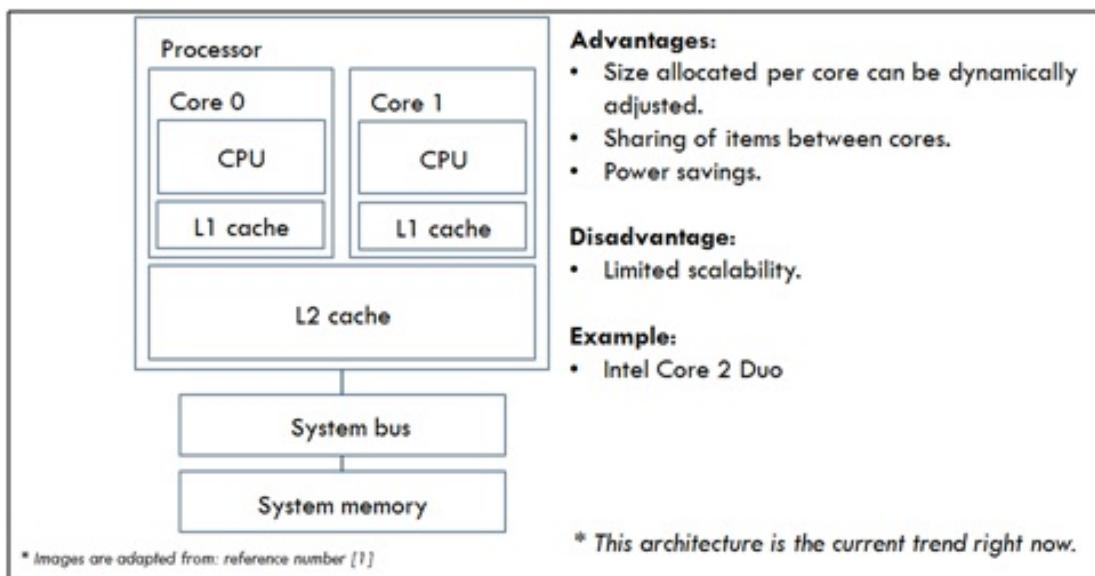
- Audio codecs
- PowerVR® 2D/3D graphics accelerator
- Hardware accelerators for encryption and decryption
- Comprehensive peripheral interfaces

* Images are taken from: www.linuxforddevices.com [7]

- Further it contains Separate L2 cache [1]



- Shared L2 cache



● **Advantages**

Having a multi-core processor in a computer means that it will work faster for certain programs.

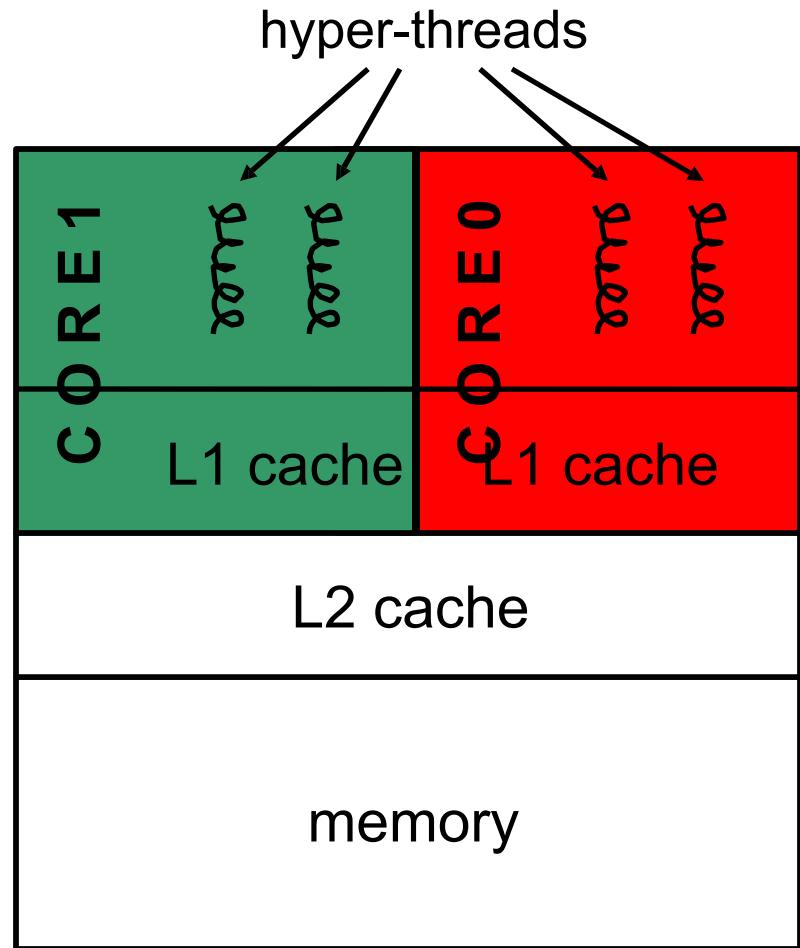
- The computer may not get as hot when it is turned on.
- The computer needs less power because it can turn off some sections if they aren't needed.
- More features can be added to the computer.
- The signals between different CPUs travel shorter distances, therefore they degrade less.

● **Disadvantages**

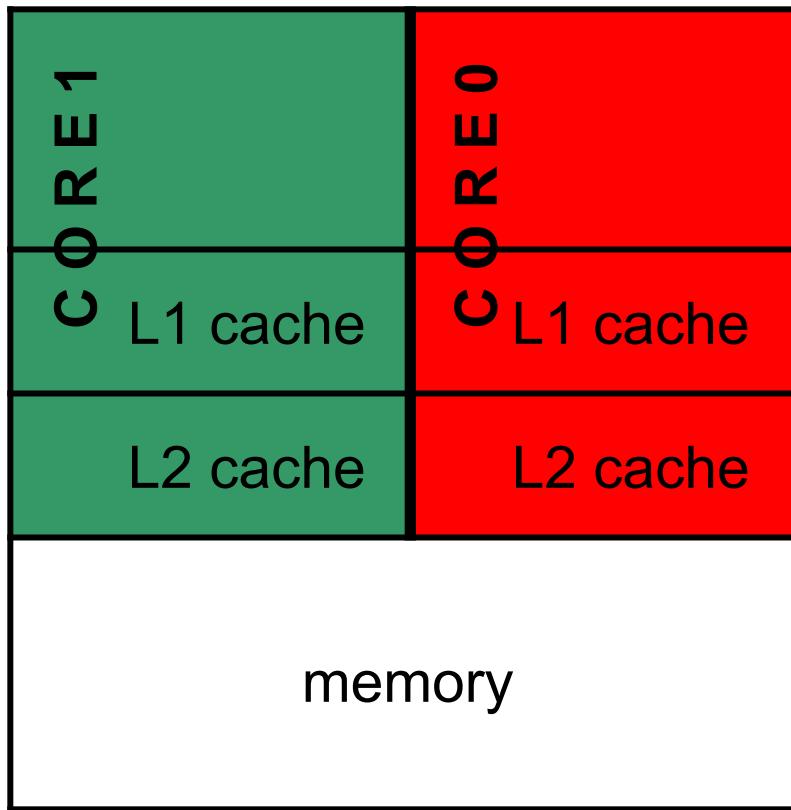
- They do not work at twice the speed as a normal processor. They get only 60-80% more speed.
- The speed that the computer works at depends on what the user is doing with it.
- They cost more than single core processors.
- They are more difficult to manage thermally than lower-density single-core processors.
- Not all operating systems support more than one core.
- Operating systems compiled for a multi-core processor will run slightly slower on a single-core processor.

“Fish” machines

- Dual-core Intel Xeon processors
- Each core is hyper-threaded
- Private L1 caches
- Shared L2 caches

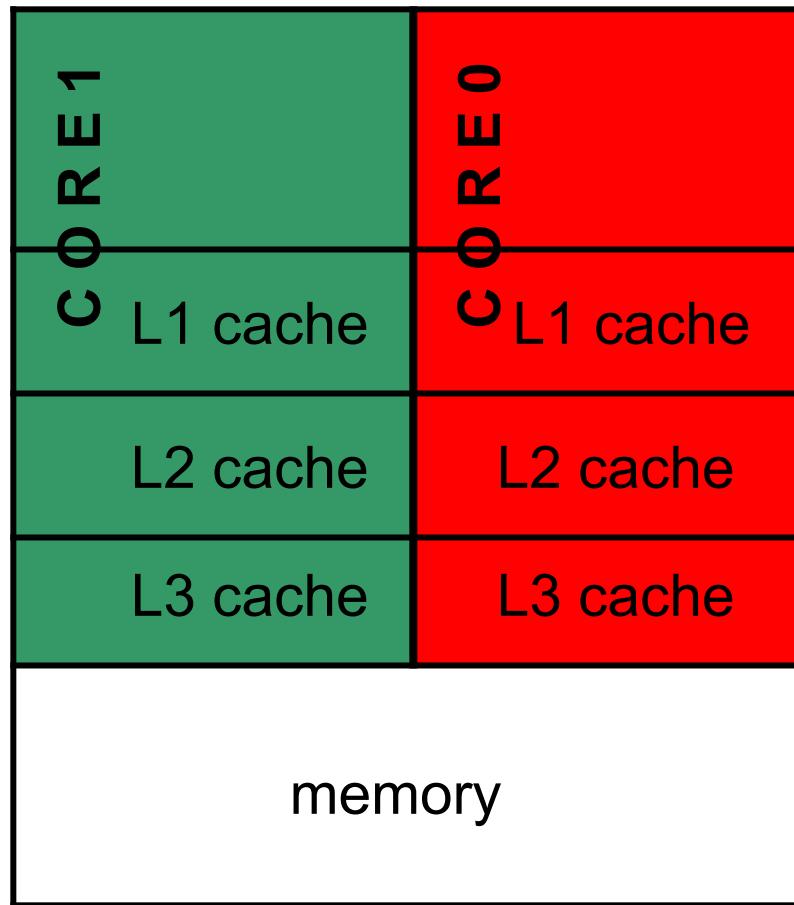


Designs with private L2 caches



Both L1 and L2 are private

Examples: AMD Opteron,
AMD Athlon, Intel Pentium D



A design with L3 caches

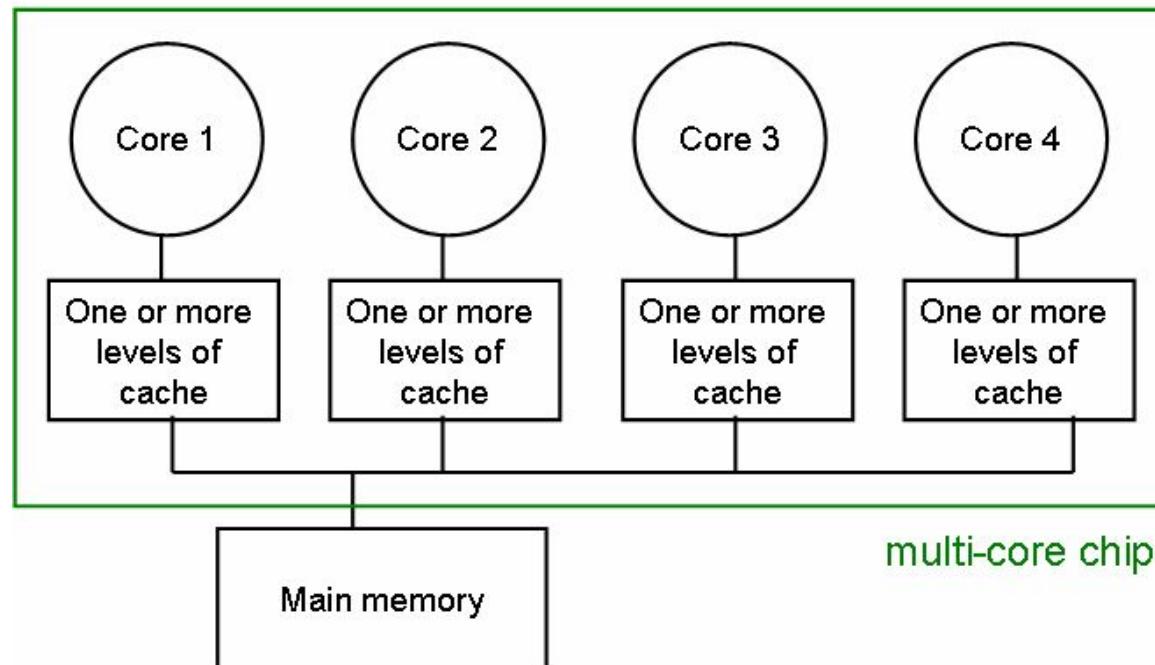
Example: Intel Itanium 2 32

Private vs Shared Caches

- Advantages of private:
 - They are closer to core, so faster access
 - Reduces contention
- Advantages of shared:
 - Threads on different cores can share the same cache data
 - More cache space available if a single (or a few) high-performance thread runs on the system

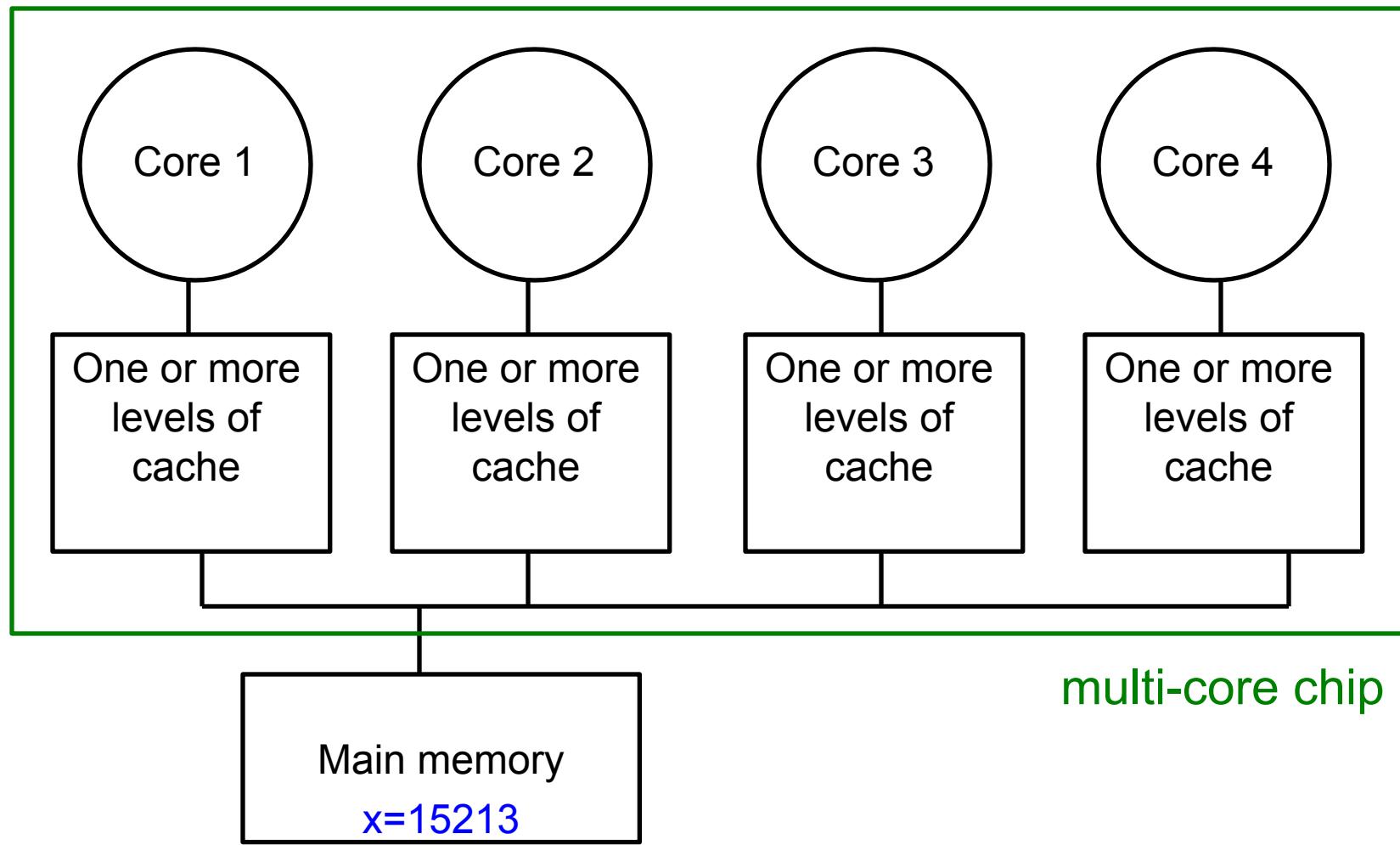
The cache coherence problem

- Since we have private caches:
How to keep the data consistent across caches?
- Each core should perceive the memory as a monolithic array, shared by all the cores



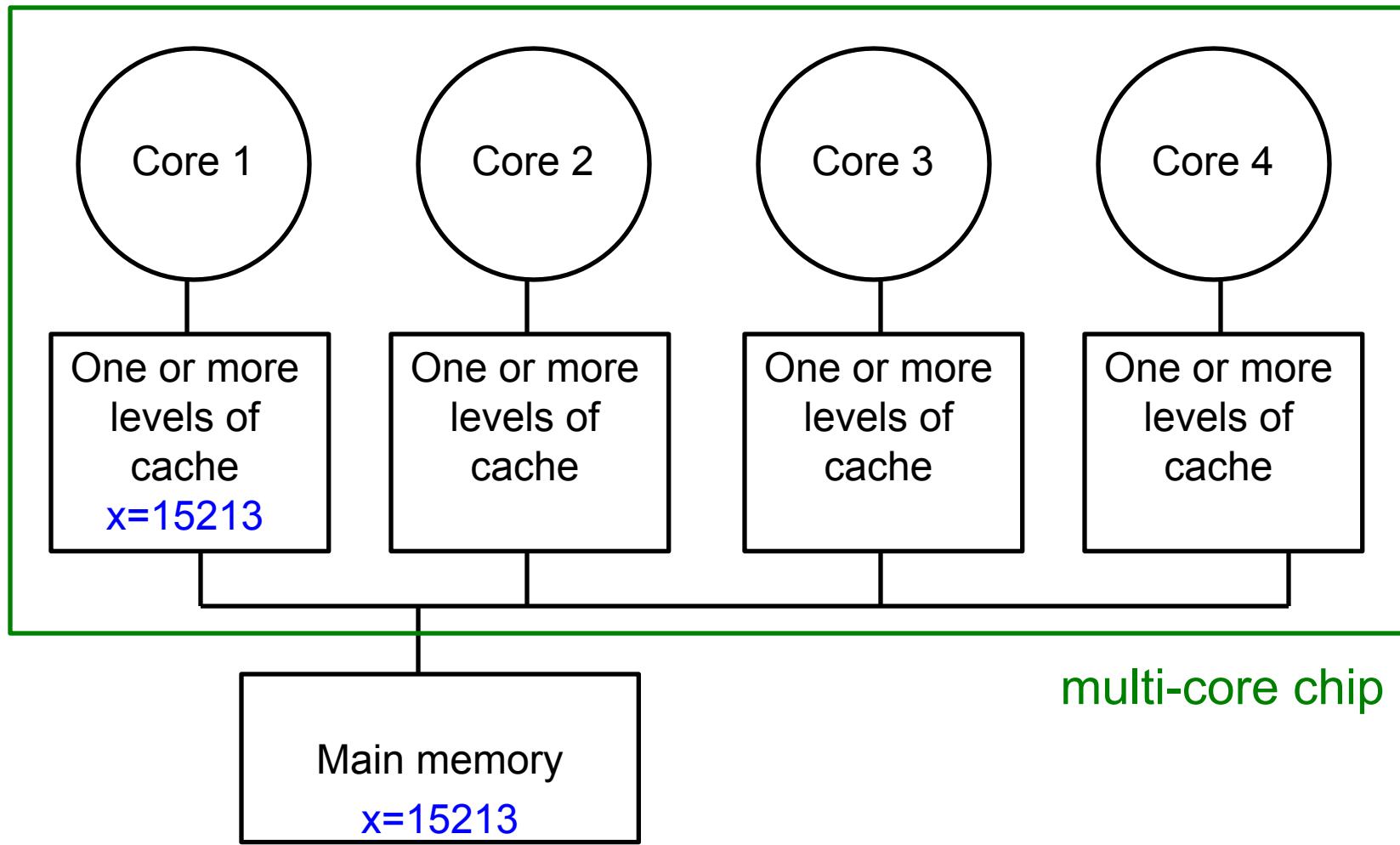
The cache coherence problem

Suppose variable x initially contains 15213



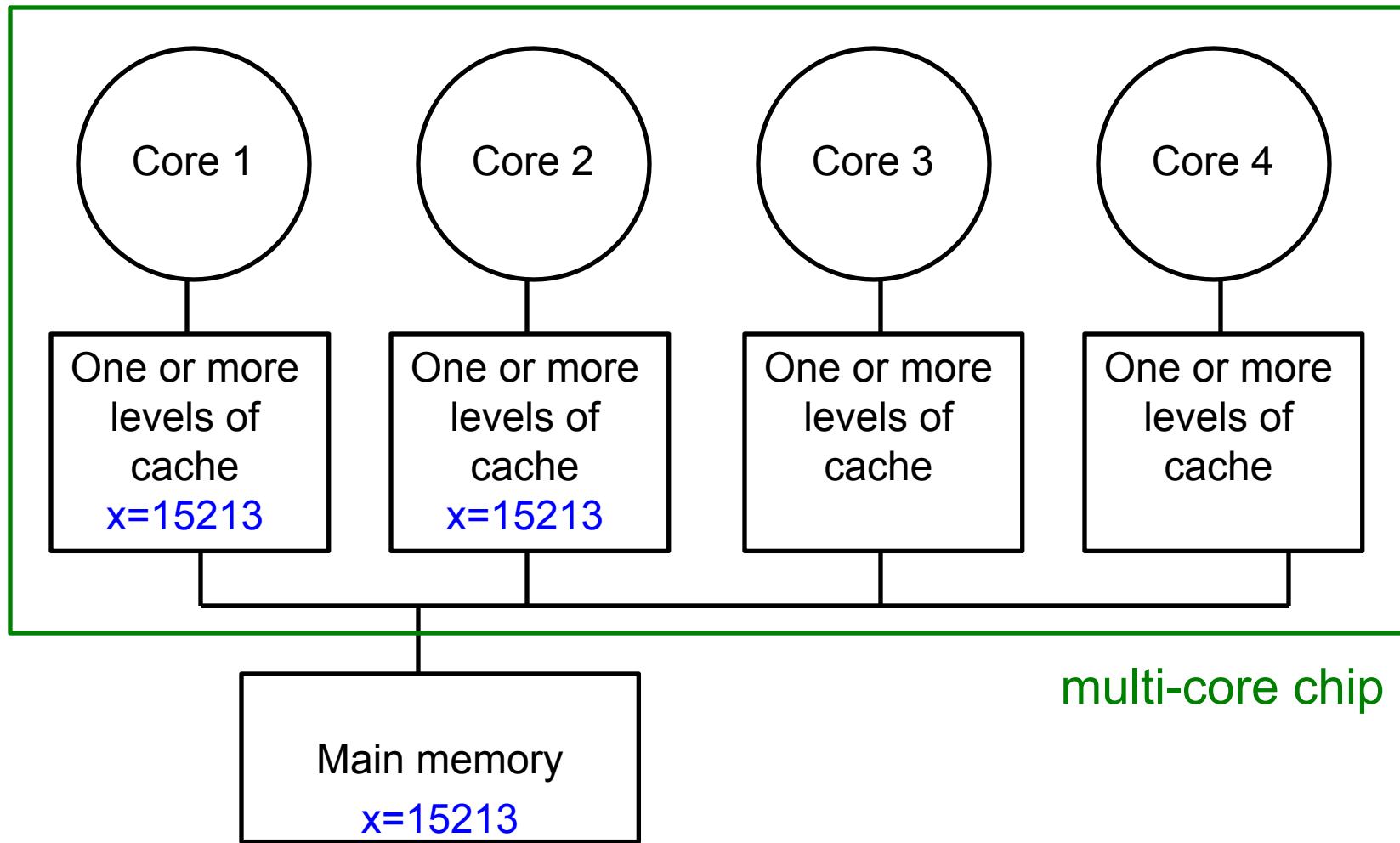
The cache coherence problem

Core 1 reads x



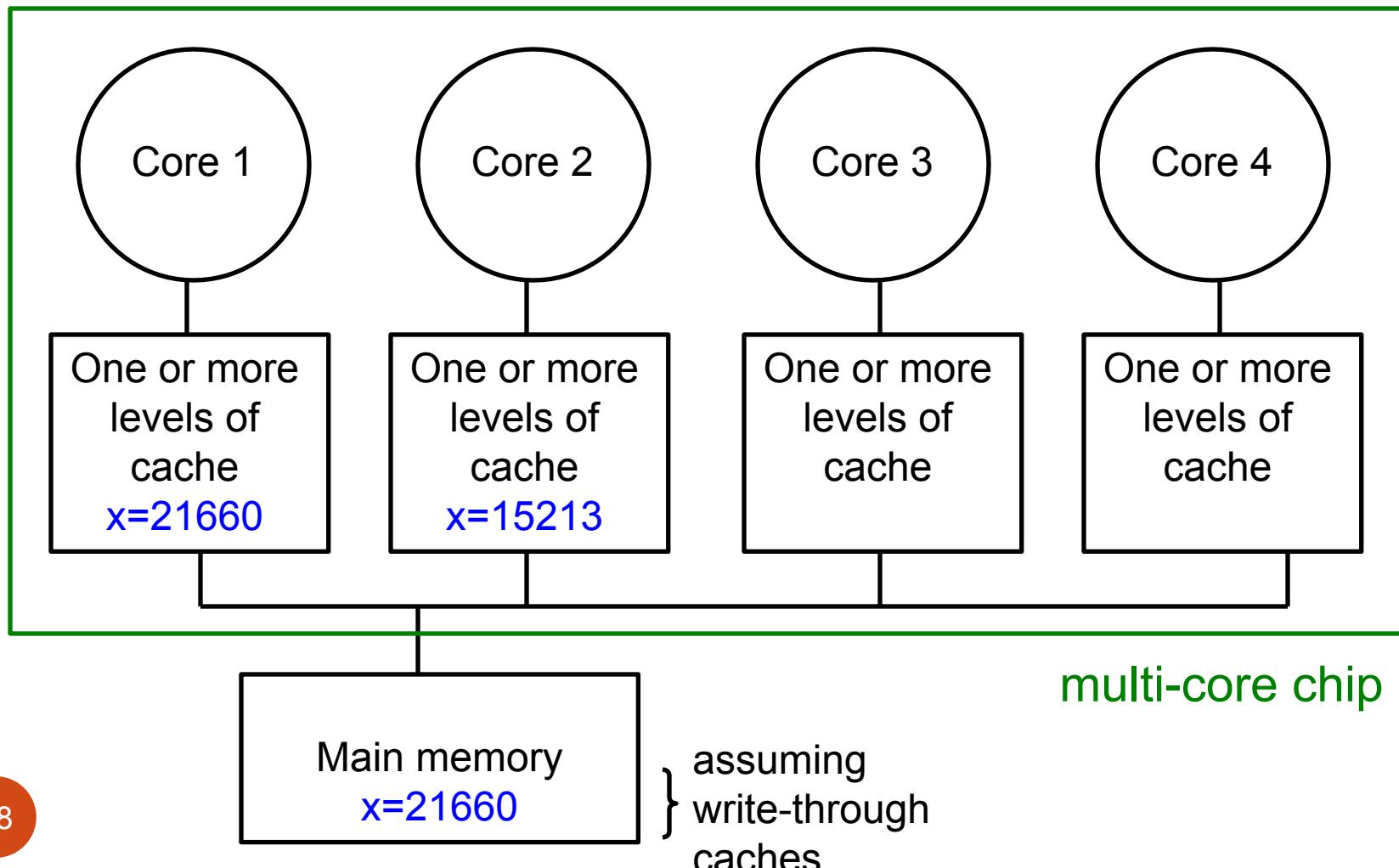
The cache coherence problem

Core 2 reads x



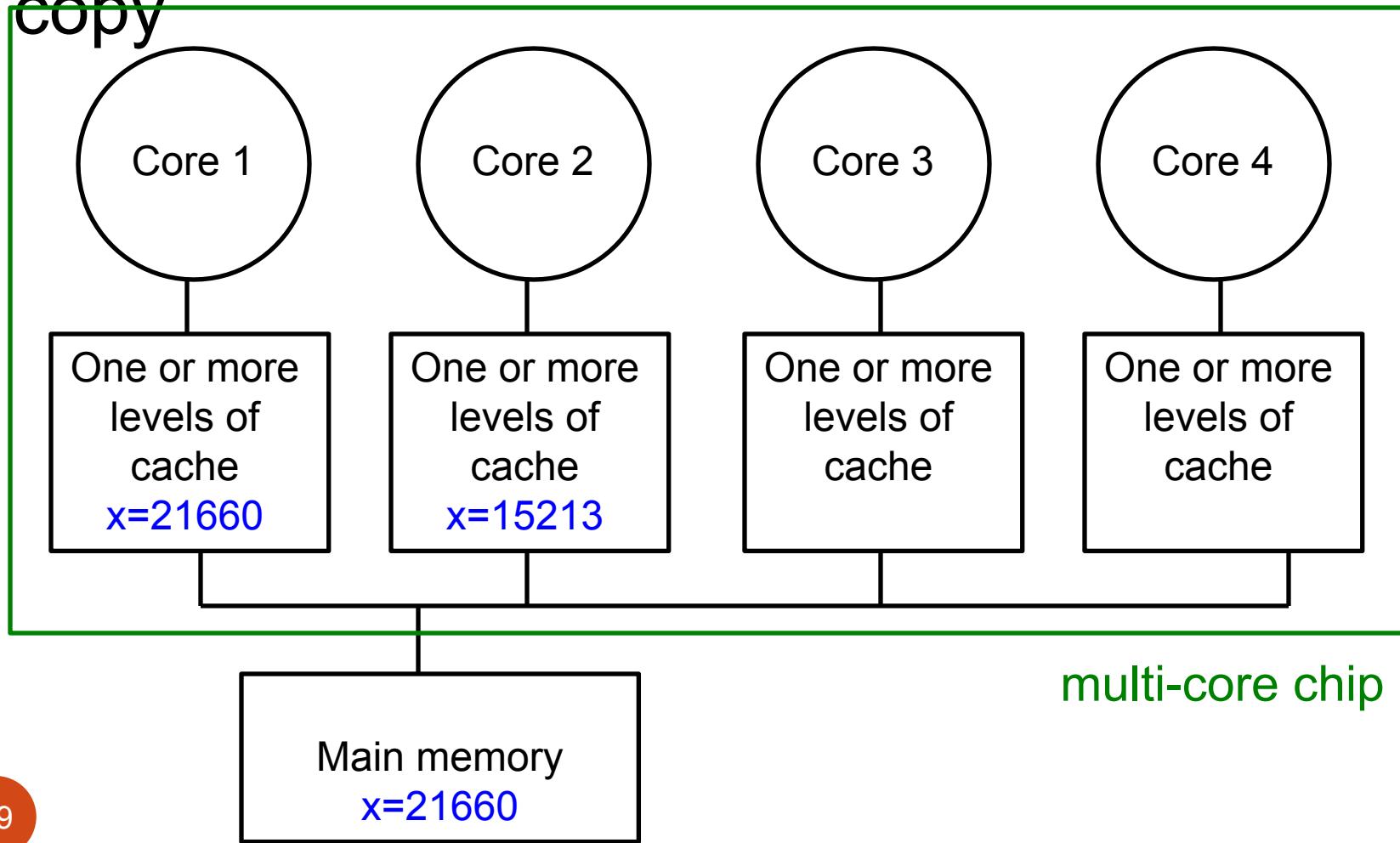
The cache coherence problem

Core 1 writes to x , setting it to 21660



The cache coherence problem

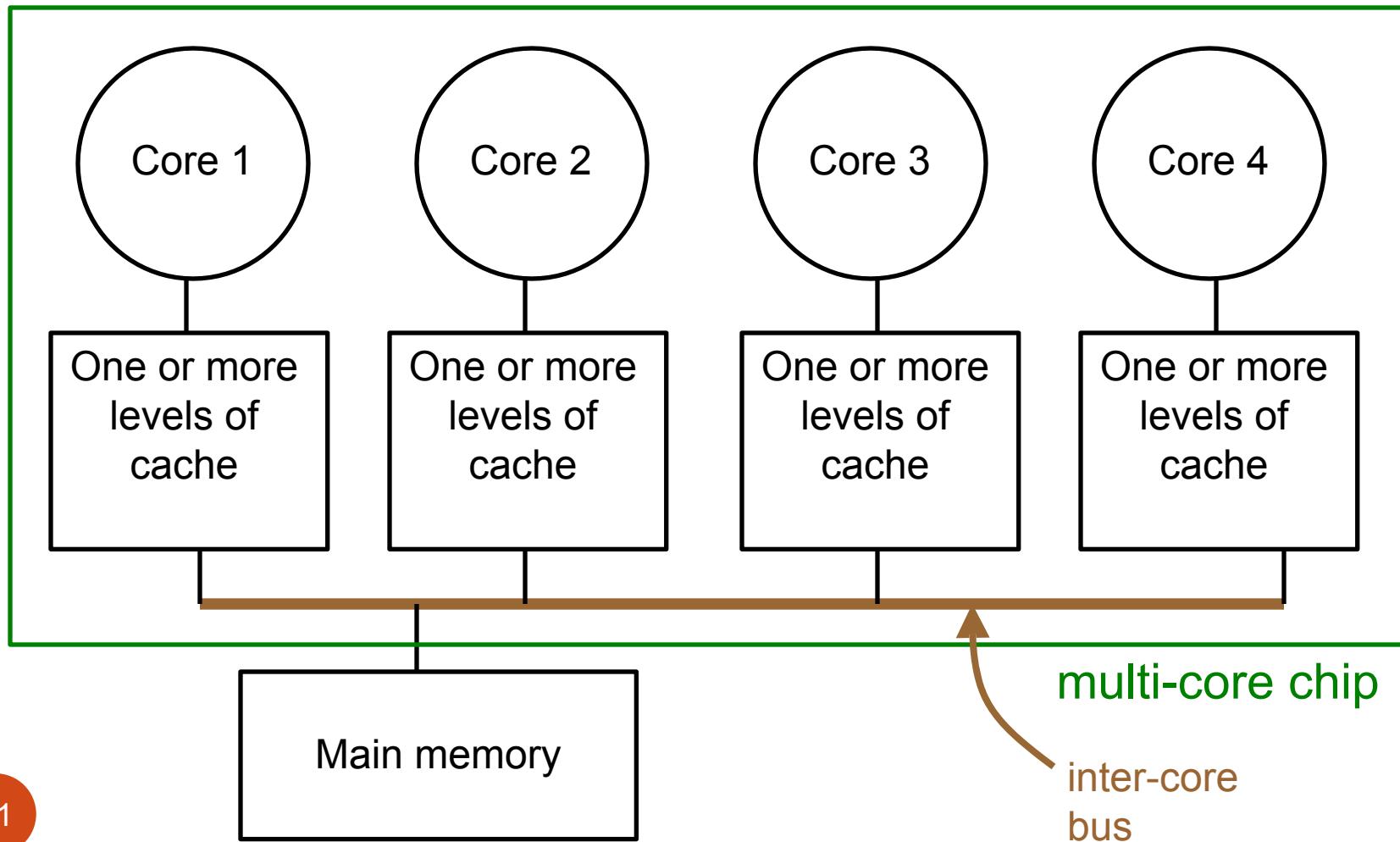
Core 2 attempts to read x... gets a stale copy



Solutions for cache coherence

- This is a general problem with multiprocessors, not limited just to multi-core
- There exist many solution algorithms, coherence protocols, etc.
- A simple solution:
invalidation-based protocol with snooping

Inter-core bus

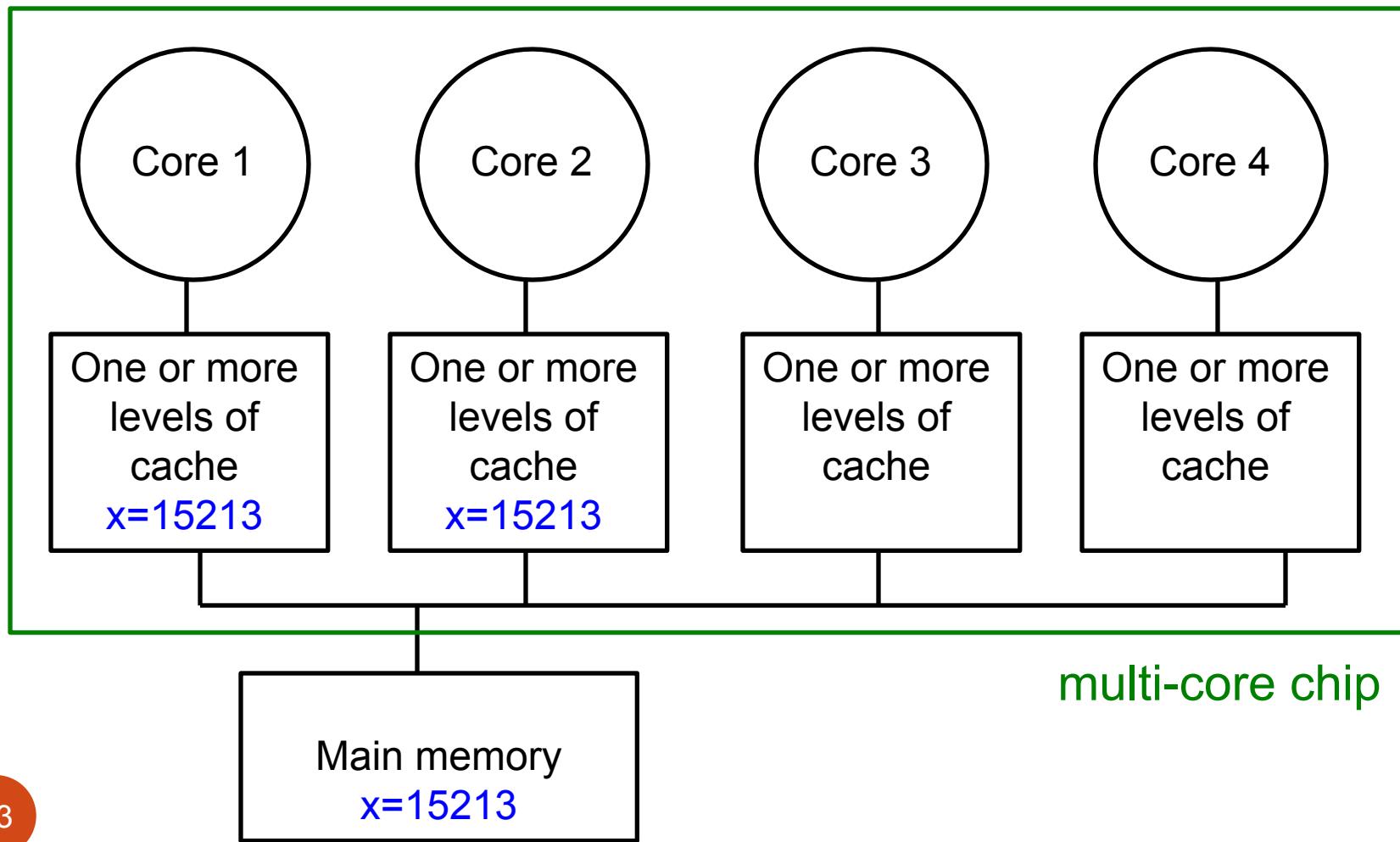


Invalidation protocol with snooping

- Invalidation:
If a core writes to a data item, all other copies of this data item in other caches are *invalidated*
- Snooping:
All cores continuously “snoop” (monitor) the bus connecting the cores.

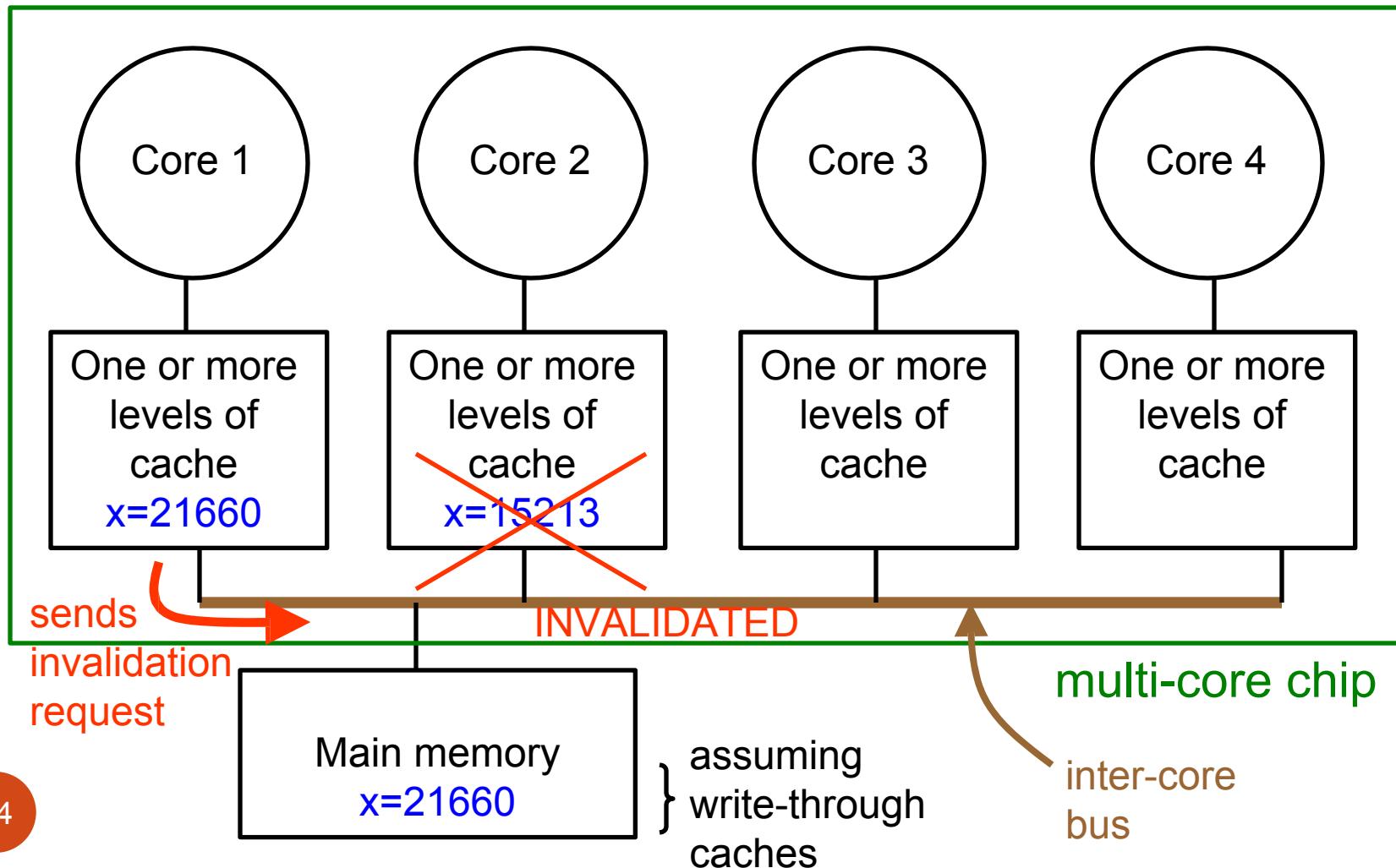
The cache coherence problem

Revisited: Cores 1 and 2 have both read x



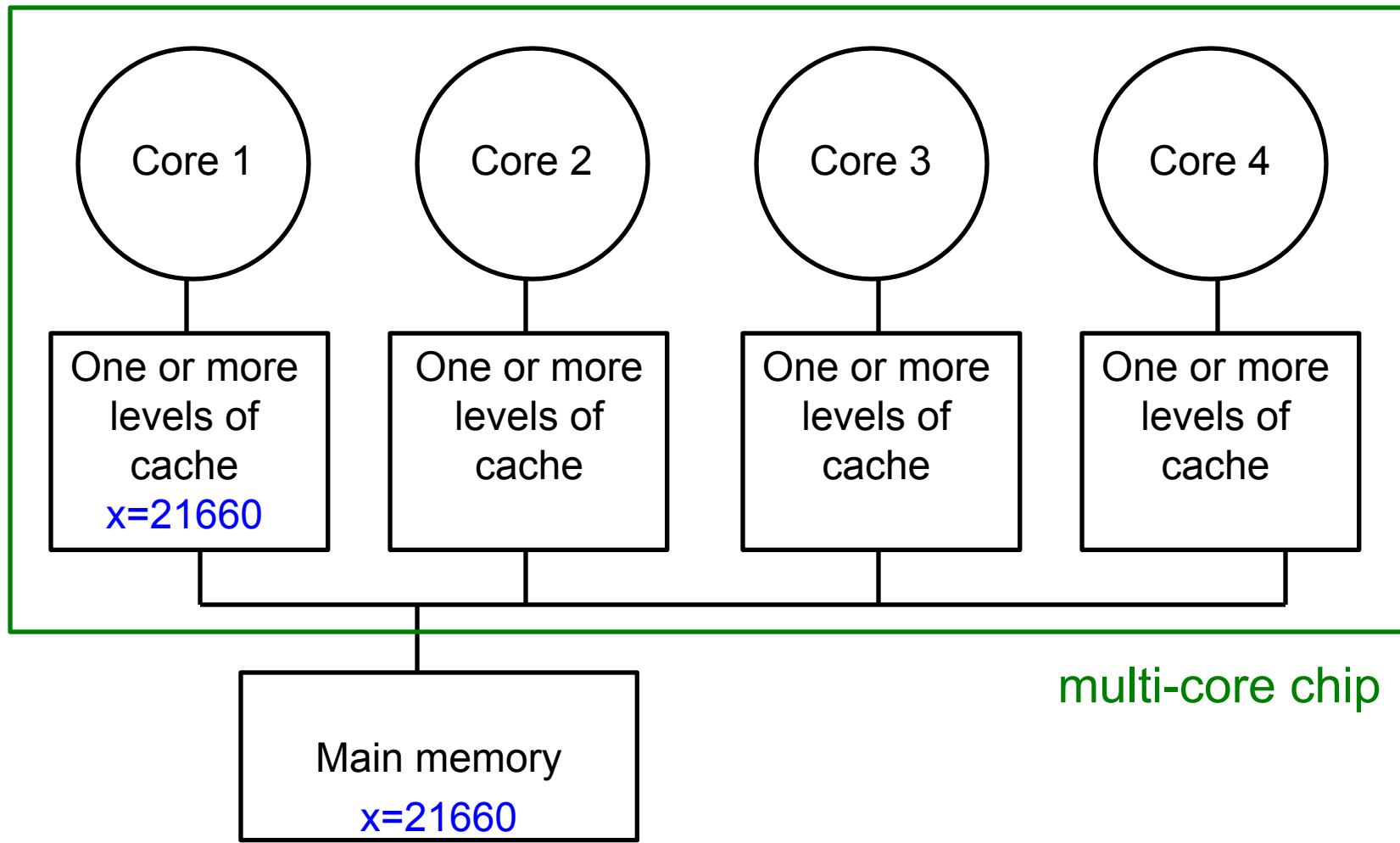
The cache coherence problem

Core 1 writes to x , setting it to 21660



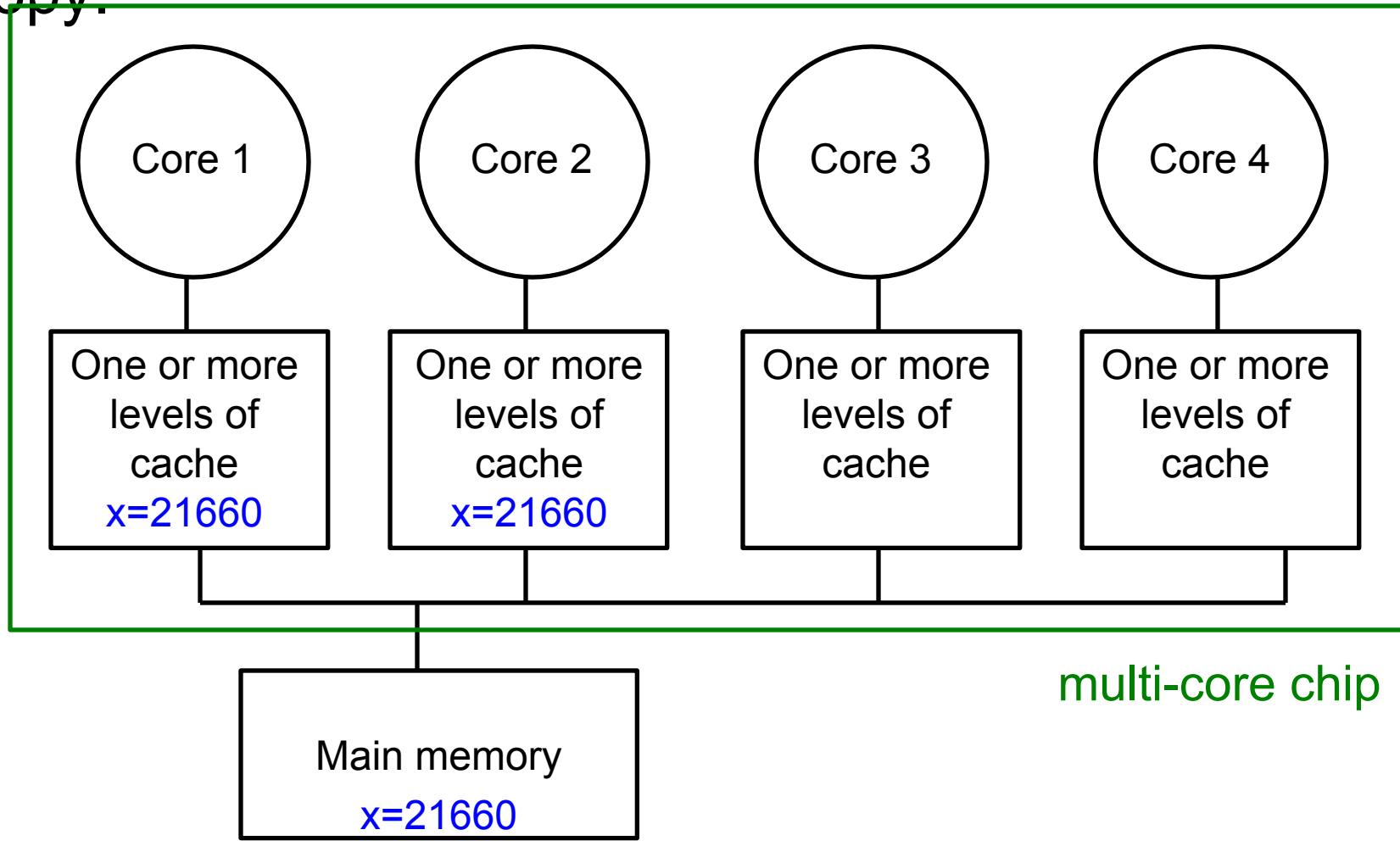
The cache coherence problem

After invalidation:



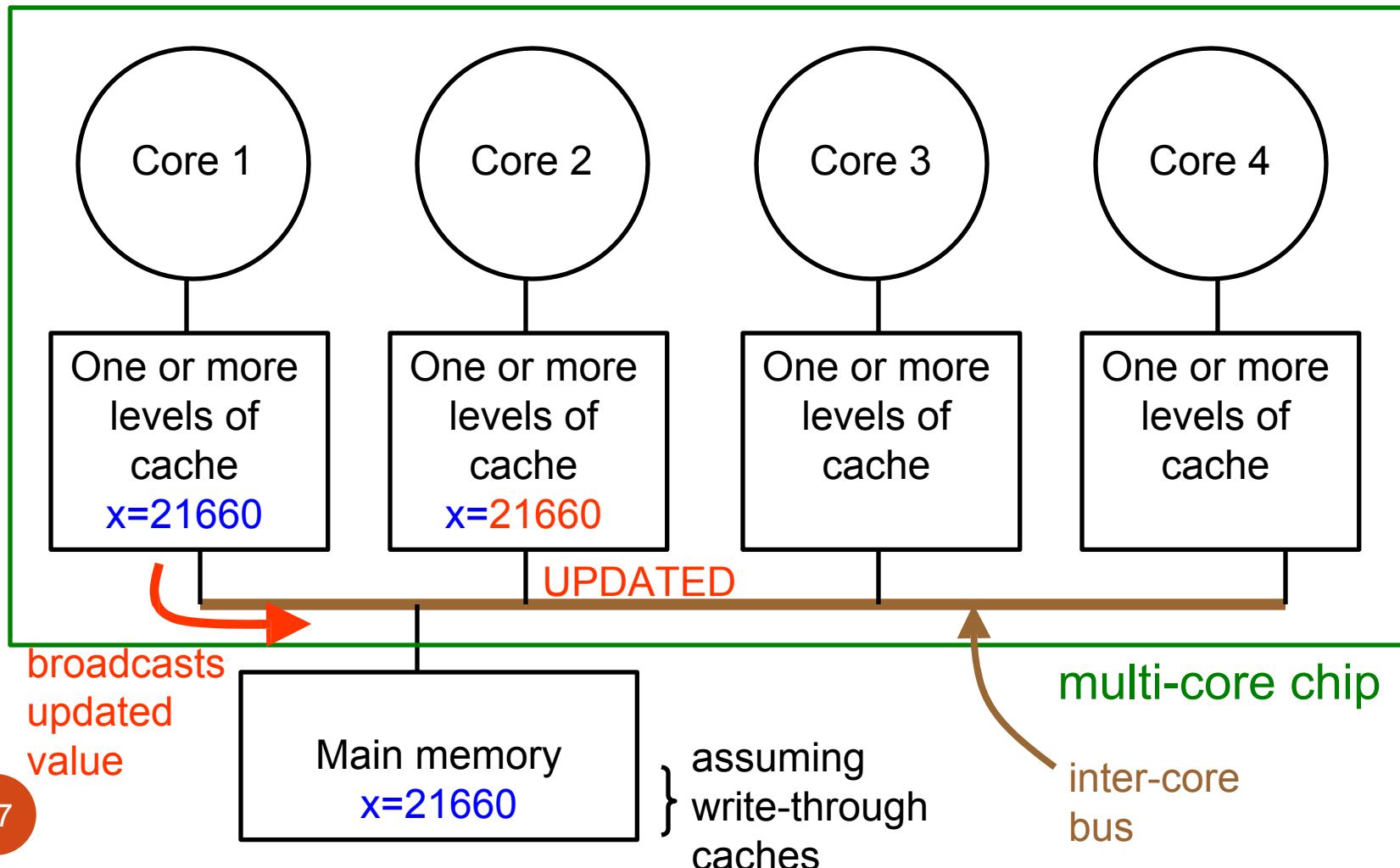
The cache coherence problem

Core 2 reads x. Cache misses, and loads the new copy.



Alternative to invalidate protocol: update protocol

Core 1 writes $x=21660$:



Invalidation vs update

- Multiple writes to the same location
 - invalidation: only the first time
 - update: must broadcast each write
(which includes new variable value)
- Invalidation generally performs better: it generates less bus traffic

Invalidation protocols

- This was just the basic invalidation protocol
- More sophisticated protocols use extra cache state bits
- MSI, MESI
(Modified, Exclusive, Shared, Invalid)

MESI Protocol

For Multiprocessor Systems

- The MESI protocol is one implementation of enforcing data integrity among caches sharing data; the write once write policy is a method to keep the protocol performing efficiently by avoiding superfluous data traffic on the system bus

MESI Protocol (1)

- A practical multiprocessor invalidate protocol which attempts to minimize bus usage.
- Allows usage of a ‘write back’ scheme - i.e. main memory not updated until ‘dirty’ cache line is displaced
- Extension of usual cache tags, i.e. invalid tag and ‘dirty’ tag in normal write back cache.

MESI Protocol (2)

Any cache line can be in one of 4 states (2 bits)

- **Modified** - cache line has been modified, is different from main memory - is the only cached copy. (multiprocessor ‘dirty’)
- **Exclusive** - cache line is the same as main memory and is the only cached copy
- **Shared** - Same as main memory but copies may exist in other caches.
- **Invalid** - Line data is not valid (as in simple cache)

MESI Protocol (3)

- Cache line changes state as a function of memory access events.
- Event may be either
 - Due to local processor activity (i.e. cache access)
 - Due to bus activity - as a result of snooping
- Cache line has its own state affected only if address matches

MESI Protocol (4)

- Operation can be described informally by looking at action in local processor
 - Read Hit
 - Read Miss
 - Write Hit
 - Write Miss
- More formally by state transition diagram

MESI Local Read Hit

- Line must be in one of MES
- This must be correct local value (if M it must have been modified locally)
- Simply return value
- No state change

MESI Local Read Miss (1)

- No other copy in caches
 - Processor makes bus request to memory
 - Value read to local cache, marked E
- One cache has E copy
 - Processor makes bus request to memory
 - Snooping cache puts copy value on the bus
 - Memory access is abandoned
 - Local processor caches value
 - Both lines set to S

MESI Local Read Miss (2)

- Several caches have S copy
 - Processor makes bus request to memory
 - One cache puts copy value on the bus (arbitrated)
 - Memory access is abandoned
 - Local processor caches value
 - Local copy set to S
 - Other copies remain S

MESI Local Read Miss (3)

- One cache has M copy
 - Processor makes bus request to memory
 - Snooping cache puts copy value on the bus
 - Memory access is abandoned
 - Local processor caches value
 - Local copy tagged S
 - **Source (M) value copied back to memory**
 - Source value M -> S

MESI Local Write Hit (1)

Line must be one of MES

- M
 - line is exclusive and already ‘dirty’
 - Update local cache value
 - no state change
- E
 - Update local cache value
 - State E -> M

MESI Local Write Hit (2)

- S
 - Processor broadcasts an invalidate on bus
 - Snooping processors with S copy change S->I
 - Local cache value is updated
 - Local state change S->M

MESI Local Write Miss (1)

Detailed action depends on copies in other processors

- No other copies
 - Value read from memory to local cache (?)
 - Value updated
 - Local copy state set to M

MESI Local Write Miss (2)

- Other copies, either one in state E or more in state S
- Value read from memory to local cache - bus transaction marked RWITM (read with intent to modify)
- Snooping processors see this and set their copy state to I
- Local copy updated & state set to M

MESI Local Write Miss (3)

Another copy in state M

- Processor issues bus transaction marked RWITM
- Snooping processor sees this
 - Blocks RWITM request
 - Takes control of bus
 - Writes back its copy to memory
 - Sets its copy state to I

MESI Local Write Miss (4)

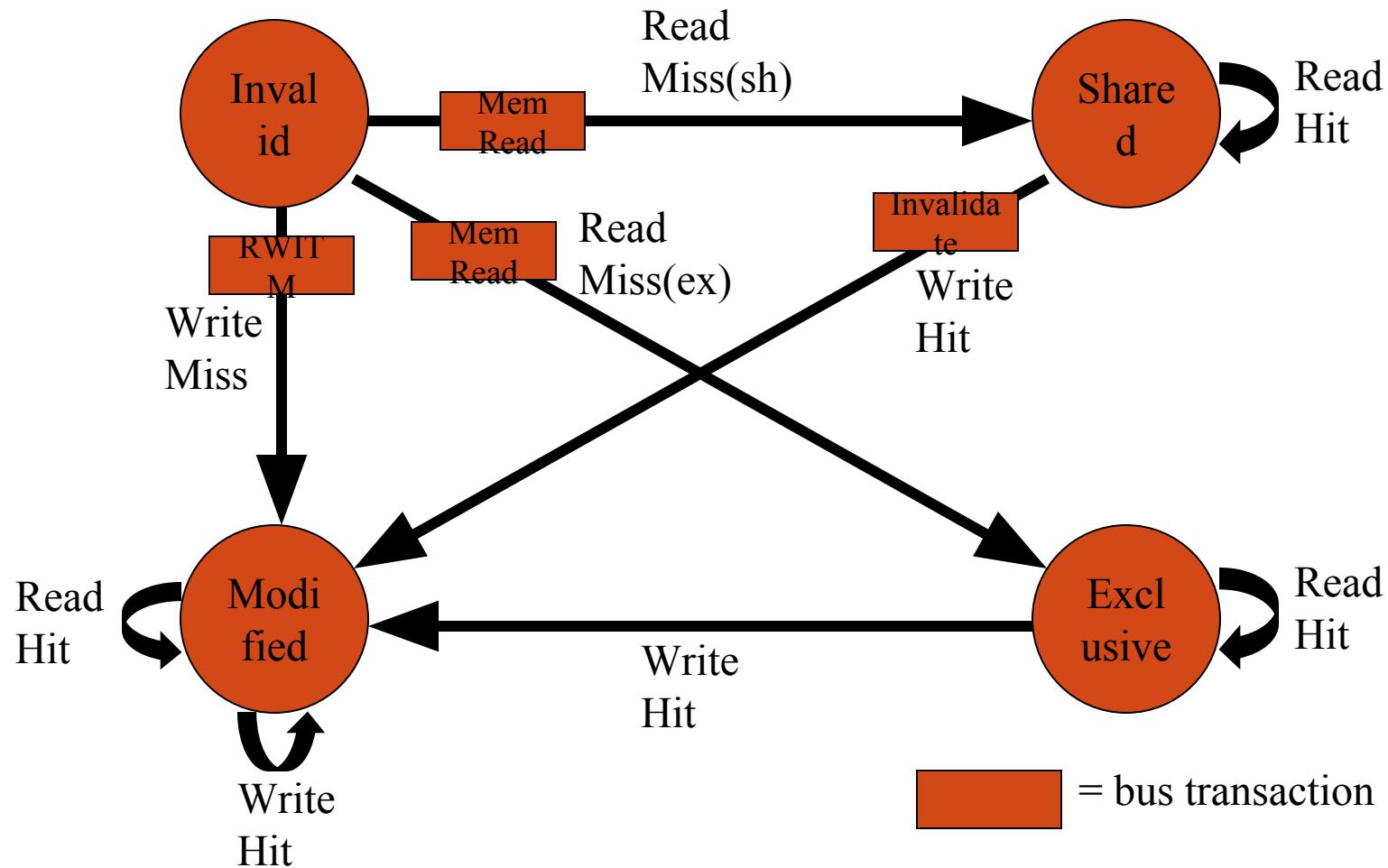
Another copy in state M (continued)

- Original local processor re-issues RWITM request
- Is now simple no-copy case
 - Value read from memory to local cache
 - Local copy value updated
 - Local copy state set to M

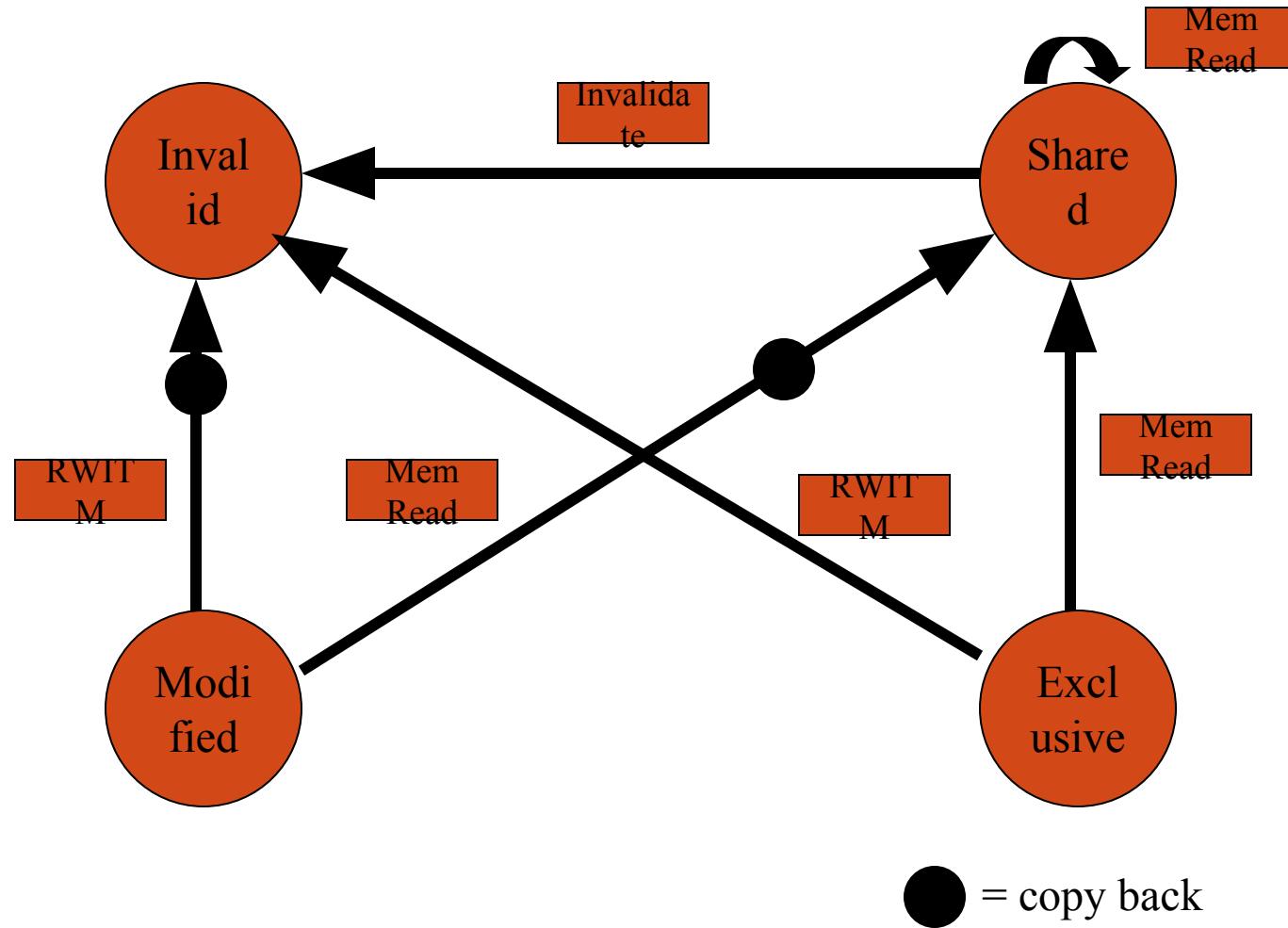
Putting it all together

- All of this information can be described compactly using a state transition diagram
- Diagram shows what happens to a cache line in a processor as a result of
 - memory accesses made by that processor (read hit/miss, write hit/miss)
 - memory accesses made by other processors that result in bus transactions observed by this snoopy cache (Mem read, RWITM, Invalidate)

MESI – locally initiated accesses



MESI – remotely initiated accesses



MESI notes

- There are minor variations (particularly to do with write miss)
- Normal ‘write back’ when cache line is evicted is done if line state is M
- Multi-level caches
 - If caches are inclusive, only the lowest level cache needs to snoop on the bus

THANK YOU

UNIT-5:

The Memory

System



FUNDAMENTAL CONCEPTS

Table of Contents



- Memory Systems – Basic Concepts
- Memory Hierarchy
- Memory technologies
- RAM, Semi Conductors RAM
- ROM Types
- Speed, Size and Cost
- Cache Memory
- Mapping Functions
- Replacement Algorithms
- Virtual Memory
- Performance Considerations of various memories
- Input and Output Organization
- Need of Input and Output Devices
- Memory and Program Mapped IO
- Interrupts – Hardware, Enabling and Disabling Interrupts
- Handling Multiple Devices



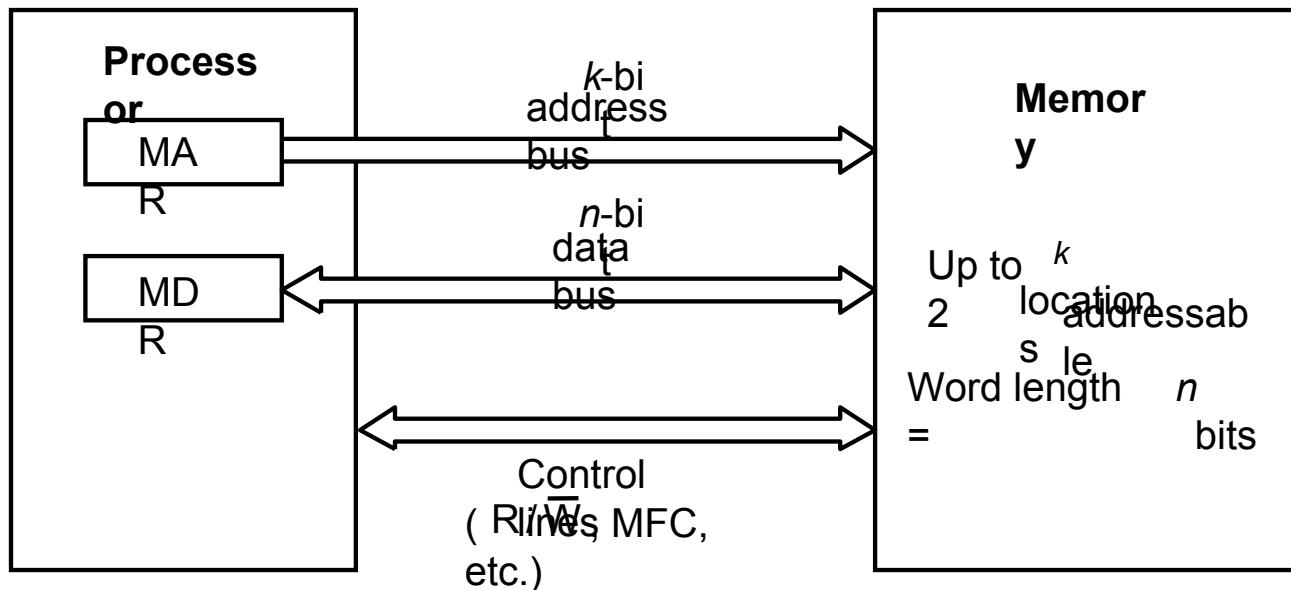
Memory Basic Concepts



Maximum size of the Main Memory

byte-addressable

CPU-Main Memory Connection



Memory Basic Concepts(Contd.,)



- Measures for the speed of a memory:
 - Memory access time.
 - Memory cycle time.
- An important design issue is to provide a computer system with as large and fast a memory as possible, within a given cost target.
- Several techniques to increase the effective size and speed of the memory:
 - Cache memory (to increase the effective speed).
 - Virtual memory (to increase the effective size).





The Memory System

SEMICONDUCTOR RAM MEMORIES

Internal Organization of Memory Chips



Each memory cell can hold one bit of information.

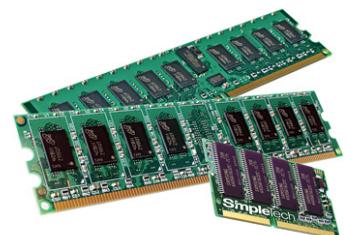
Memory cells are organized in the form of an array.

One row is one memory word.

All cells of a row are connected to a common line, known as the “word line”.

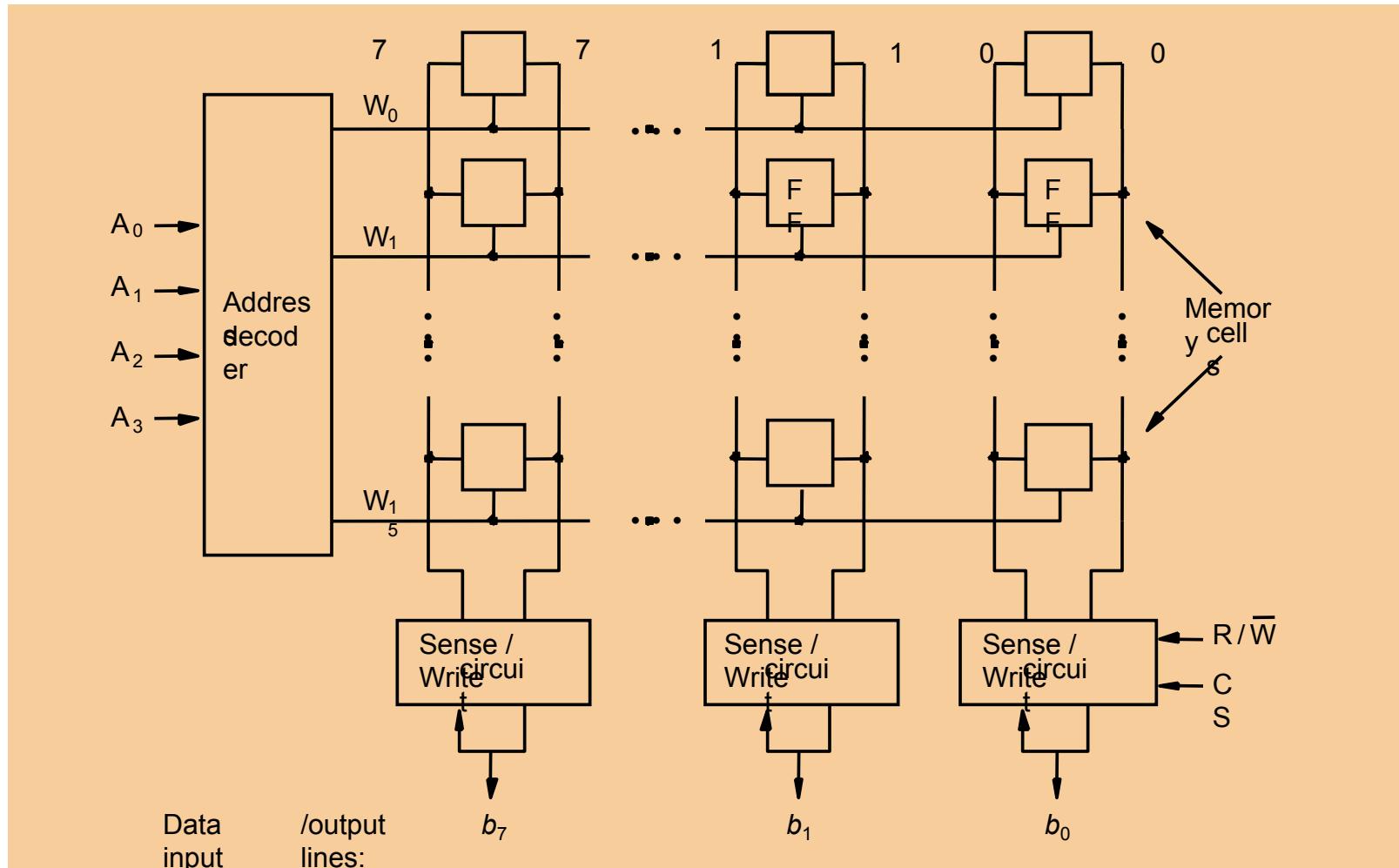
Word line is connected to the address decoder.

Sense/write circuits are connected to the data input/output lines of the memory chip.





Internal Organization of Memory Chips (Contd.,)





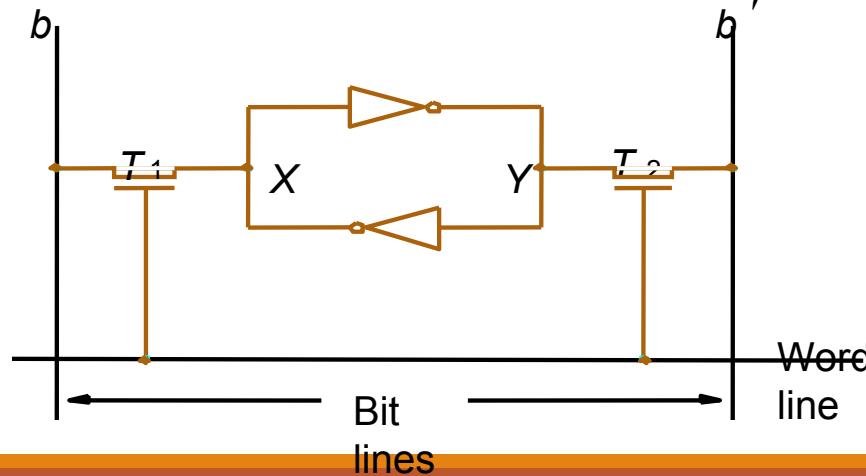
SRAM Cell

Two transistor inverters are cross connected to implement a basic flip-flop.

The cell is connected to one word line and two bits lines by transistors T1 and T2

When word line is at ground level, the transistors are turned off and the latch retains its state

Read operation: In order to read state of SRAM cell, the word line is activated to close switches T1 and T2. Sense/Write circuits at the bottom monitor the state of b and b'





Asynchronous DRAMs

Static RAMs (SRAMs):

- Consist of circuits that are capable of retaining their state as long as the power is applied.
- Volatile memories, because their contents are lost when power is interrupted.
- Access times of static RAMs are in the range of few nanoseconds.
- However, the cost is usually high.

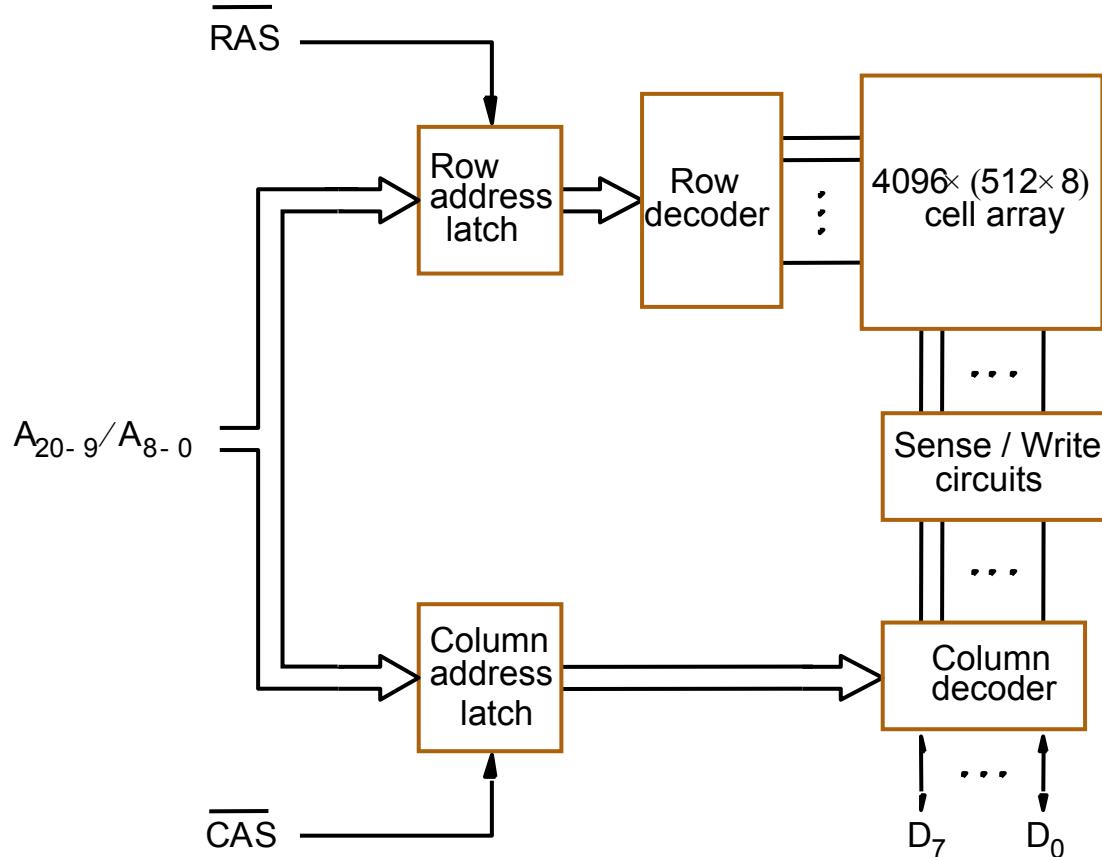
Dynamic RAMs (DRAMs):

- Do not retain their state indefinitely.
- Contents must be periodically refreshed.
- Contents may be refreshed while accessing them for reading.





Asynchronous DRAMs



Each row can store 512 bytes. 12 bits to select a row, and 9 bits to select a group in a row. Total of 21 bits.

- First apply the row address, RAS signal latches the row address. Then apply the column address, CAS signal latches the address.
- Timing of the memory unit is controlled by a specialized unit which generates RAS and CAS.
- This is asynchronous DRAM





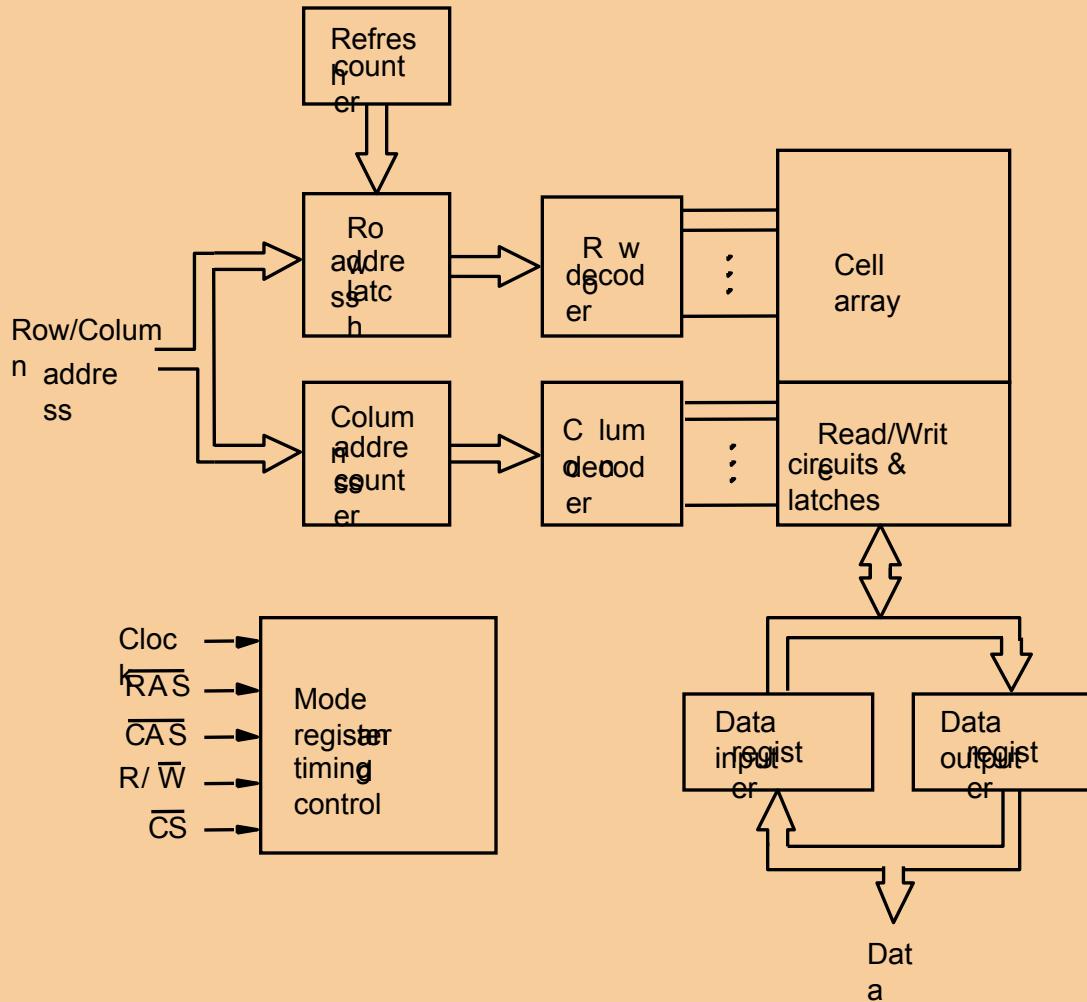
Fast Page Mode

- Suppose if we want to access the consecutive bytes in the selected row.
- This can be done without having to reselect the row.
 - Add a latch at the output of the sense circuits in each row.
 - All the latches are loaded when the row is selected.
 - Different column addresses can be applied to select and place different bytes on the data lines.
- Consecutive sequence of column addresses can be applied under the control signal CAS, without reselecting the row.
 - Allows a block of data to be transferred at a much faster rate than random accesses.
 - A small collection/group of bytes is usually referred to as a block.
- This transfer capability is referred to as the fast page mode feature.





Synchronous DRAMs



- Operation is directly synchronized with processor clock signal.
- The outputs of the sense circuits are connected to a latch.
- During a Read operation, the contents of the cells in a row are loaded onto the latches.
- During a refresh operation, the contents of the cells are refreshed without changing the contents of the latches.
- Data held in the latches correspond to the selected columns are transferred to the output.
- For a burst mode of operation, successive columns are selected using column address counter and clock. CAS signal need not be generated externally. A new data is placed during raising edge of the clock

Latency, Bandwidth, and DDRSDRAMs



Memory latency is the time it takes to transfer a word of data to or from memory

Memory bandwidth is the number of bits or bytes that can be transferred in one second.

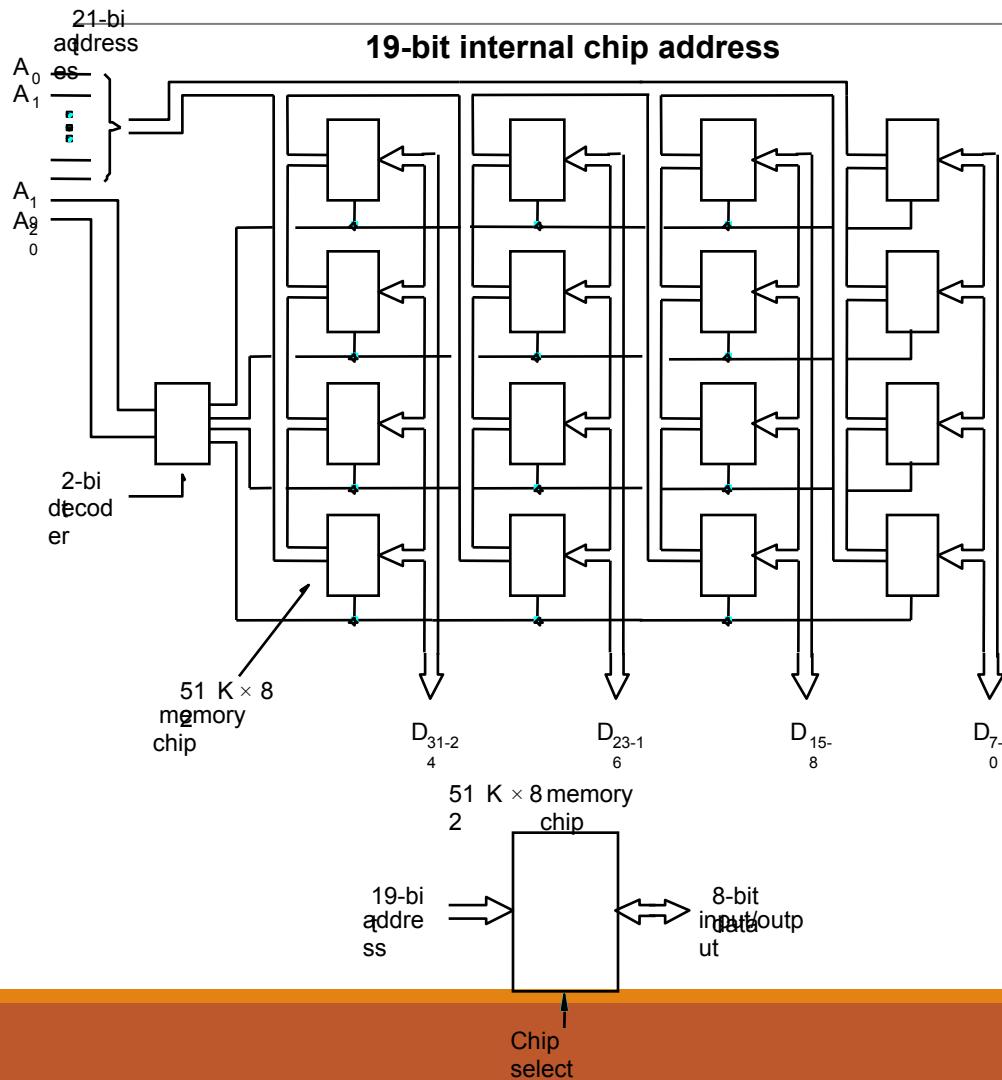
DDRSDRAMs-Double-Data-Rate-SDRAM

- Cell array is organized in two banks





Static Memories



Implement a memory unit of 2M words of 32 bits each.

Use 512x8 static memory chips.
Each column consists of 4 chips.
Each chip implements one byte position.

A chip is selected by setting its chip select control line to 1.
Selected chip places its data on the data output line, outputs of other chips are in high impedance state.
21 bits to address a 32-bit word.
High order 2 bits are needed to select the row, by activating the four Chip Select signals.
19 bits are used to access specific byte locations inside the selected chip.



Dynamic Memories

- Large dynamic memory systems can be implemented using DRAM chips in a similar way to static memory systems.
- Placing large memory systems directly on the motherboard will occupy a large amount of space.
- Packaging considerations have led to the development of larger memory units known as **SIMMs (Single In-line Memory Modules)** and **DIMMs (Dual In-line Memory Modules)**.
- Memory modules are an assembly of memory chips on a small board that plugs vertically onto a single socket on the motherboard.





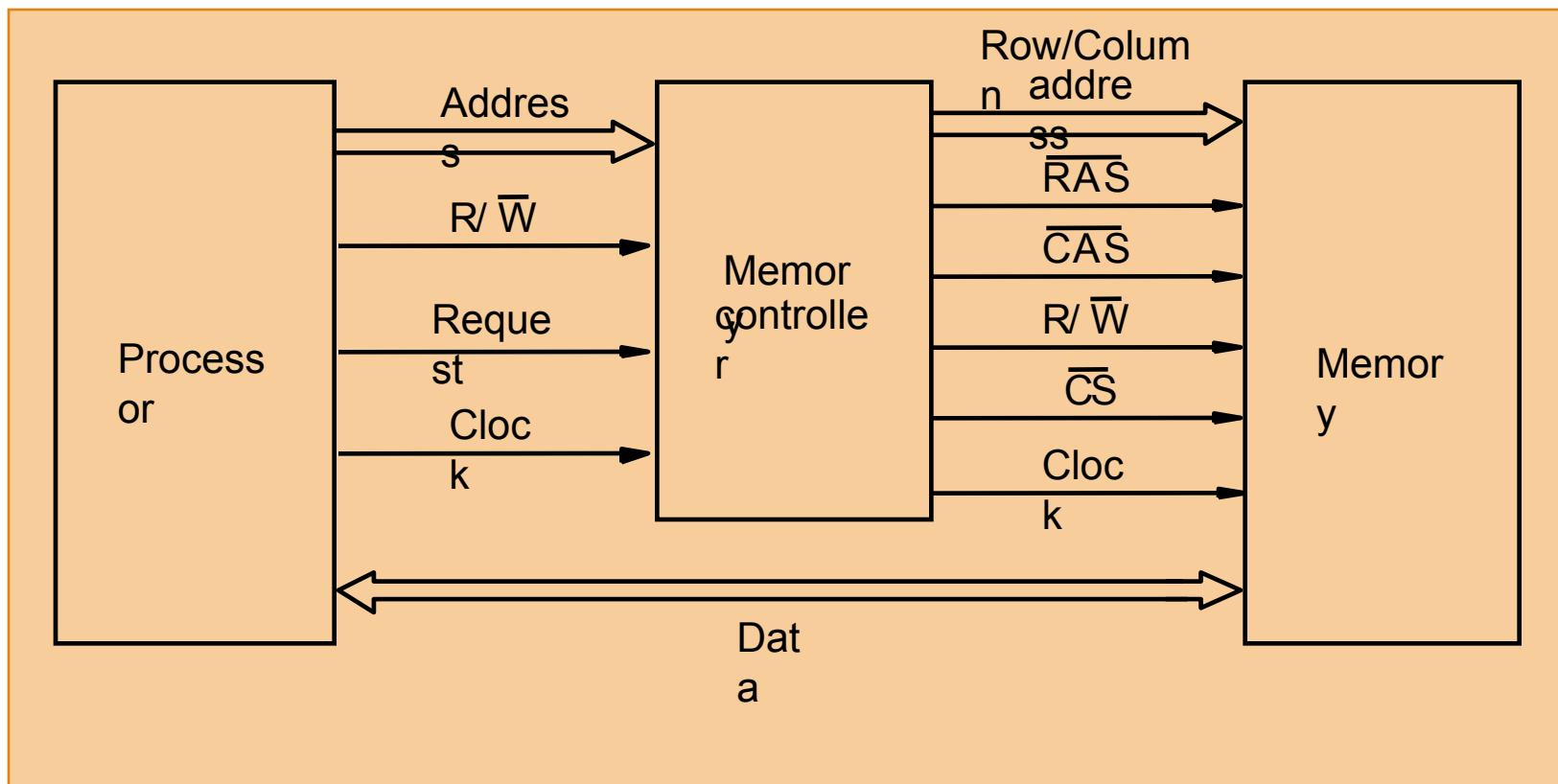
Memory Controller

- Recall that in a dynamic memory chip, to reduce the number of pins, multiplexed addresses are used.
- Address is divided into two parts:
 - High-order address bits select a row in the array.
 - They are provided first, and latched using RAS signal.
 - Low-order address bits select a column in the row.
 - They are provided later, and latched using CAS signal.
- However, a **processor issues all address bits at the same time.**
- In order to achieve the **multiplexing**, **memory controller circuit is inserted between the processor and memory.**





Memory Controller (contd..)





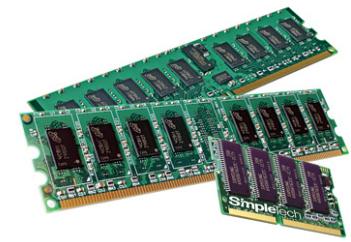
The Memory System

READ-ONLY MEMORIES (ROMS)



Read-Only Memories (ROMs)

- SRAM and SDRAM chips are volatile:
 - Lose the contents when the power is turned off.
- Many applications need memory devices to retain contents after the power is turned off.
 - For example, computer is turned on, the operating system must be loaded from the disk into the memory.
 - Store instructions which would load the OS from the disk.
 - Need to store these instructions so that they will not be lost after the power is turned off.
 - We need to store the instructions into a non-volatile memory.
- Non-volatile memory is read in the same manner as volatile memory.
 - Separate writing process is needed to place information in this memory.
 - Normal operation involves only reading of data, this type of memory is called Read-Only memory (ROM).





Read-Only Memories (Contd.,)

- **Read-Only Memory:**
 - Data are written into a ROM when it is manufactured.
- **Programmable Read-Only Memory (PROM):**
 - Allow the data to be loaded by a user.
 - Process of inserting the data is irreversible.
 - Storing information specific to a user in a ROM is expensive.
 - Providing programming capability to a user may be better.
- **Erasable Programmable Read-Only Memory (EPROM):**
 - Stored data to be erased and new data to be loaded.
 - Flexibility, useful during the development phase of digital systems.
 - Erasable, reprogrammable ROM.
 - Erasure requires exposing the ROM to UV light.





Read-Only Memories (Contd.,)

- Electrically Erasable Programmable Read-Only Memory (EEPROM):
 - To erase the contents of EPROMs, they have to be exposed to ultraviolet light.
 - Physically removed from the circuit.
 - EEPROMs the contents can be stored and erased electrically.
- Flash memory:
 - Has similar approach to EEPROM.
 - Read the contents of a single cell, but write the contents of an entire block of cells.





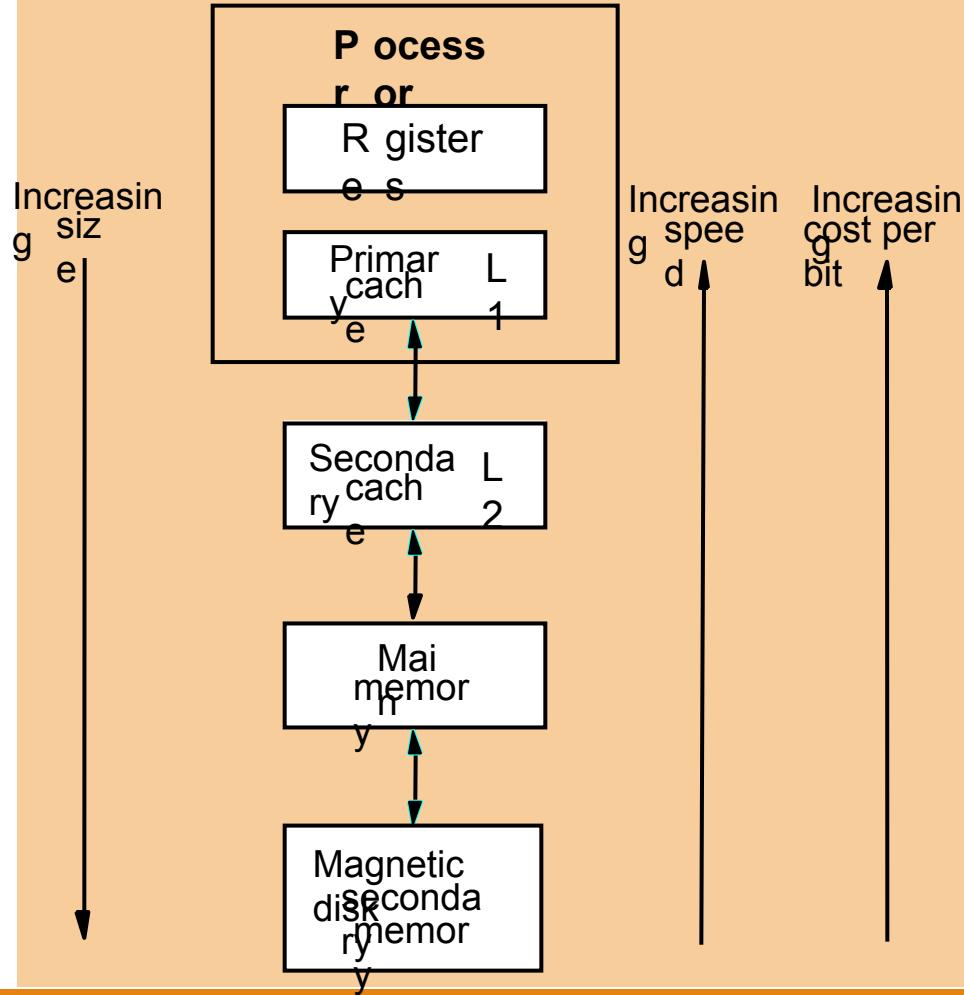
Speed, Size, and Cost

- A big challenge in the design of a computer system is to provide a sufficiently large memory, with a reasonable speed at an affordable cost.
- **Static RAM:**
 - Very fast, but expensive, because a basic SRAM cell has a complex circuit making it impossible to pack a large number of cells onto a single chip.
- **Dynamic RAM:**
 - Simpler basic cell circuit, hence are much less expensive, but significantly slower than SRAMs.
- **Magnetic disks:**
 - Storage provided by DRAMs is higher than SRAMs, but is still less than what is necessary.
 - Secondary storage such as magnetic disks provide a large amount of storage, but is much slower than DRAMs.





Memory Hierarchy



- Fastest access is to the data held in processor registers. Registers are at the top of the memory hierarchy.
- Relatively small amount of memory that can be implemented on the processor chip. This is processor cache.
- Two levels of cache. Level 1 (L1) cache is on the processor chip. Level 2 (L2) cache is in between main memory and processor.
- Next level is main memory, implemented as SIMMs. Much larger, but much slower than cache memory.
- Next level is magnetic disks. Huge amount of inexpensive storage.
- Speed of memory access is critical, the idea is to bring instructions and data that will be used in the near future as close to the processor as possible.



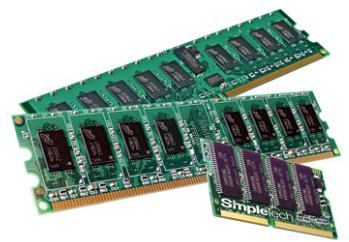
The Memory System

CACHE MEMORIES



Cache Memory

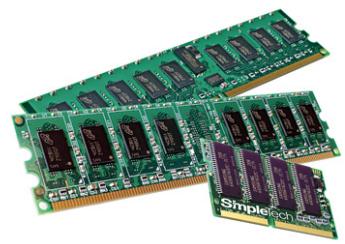
- Processor is much faster than the main memory.
- Speed of the main memory cannot be increased beyond a certain point.
- Cache memory is an architectural arrangement which makes the main memory appear faster to the processor than it really is.
- Cache memory is based on the property of computer programs known as "locality of reference".





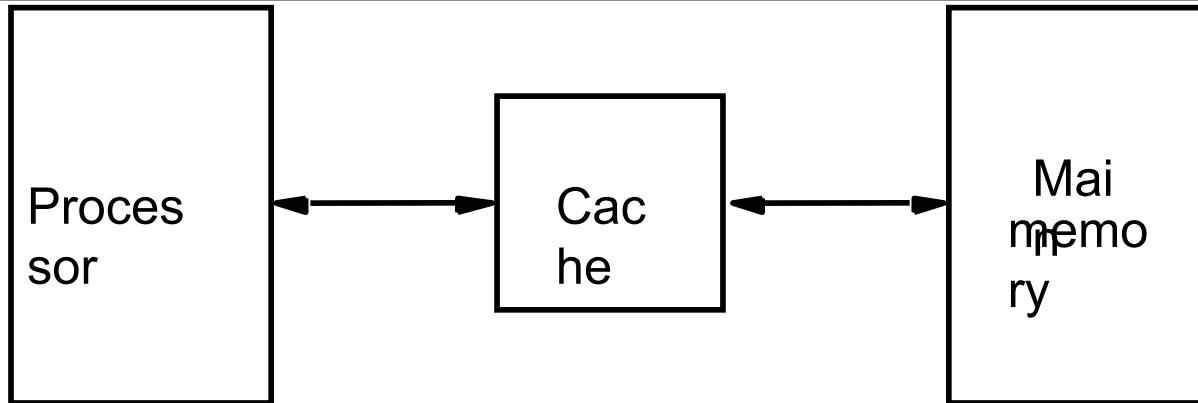
Locality of Reference

- Analysis of programs indicates that many instructions in localized areas of a program are executed repeatedly during some period of time, while the others are accessed relatively less frequently.
- Temporal locality of reference:
 - Recently executed instruction is likely to be executed again very soon.
- Spatial locality of reference:
 - Instructions with addresses close to a recently instruction are likely to be executed soon.





Cache memories



- Processor issues a Read request, a block of words is transferred from the main memory to the cache, one word at a time.
- Subsequent references to the data in this block of words are found in the cache.
- At any given time, only some blocks in the main memory are held in the cache. Which blocks in the main memory are in the cache is determined by a "mapping function".
- When the cache is full, and a block of words needs to be transferred from the main memory, some block of words in the cache must be replaced. This is determined by a "replacement algorithm".





Cache Hit

- Existence of a cache is transparent to the processor. The processor issues Read and Write requests in the same manner.
- If the data is in the cache it is called a Read or Write hit.
- **Read hit:**
 - The data is obtained from the cache.
- **Write hit:**
 - Cache has a replica of the contents of the main memory.
 - Contents of the cache and the main memory may be updated simultaneously. This is the write-through protocol.
 - Update the contents of the cache, and mark it as updated by setting a bit known as the dirty bit or modified bit. The contents of the main memory are updated when this block is replaced. This is write-back or copy-back protocol.





Cache Miss

- If the data is not present in the cache, then a Read miss or Write miss occurs.
- **Read miss:**
 - Block of words containing this requested word is transferred from the memory.
 - After the block is transferred, the desired word is forwarded to the processor.
 - The desired word may also be forwarded to the processor as soon as it is transferred without waiting for the entire block to be transferred. This is called load-through or early-restart.
- **Write-miss:**
 - Write-through protocol is used, then the contents of the main memory are updated directly.
 - If write-back protocol is used, the block containing the addressed word is first brought into the cache. The desired word is overwritten with new information.





Cache Coherence Problem

- A bit called **“valid bit”** is provided for each block.
- If the block contains valid data, then the bit is set to 1, else it is 0.
- Valid bits are set to 0, when the power is just turned on.
- When a block is loaded into the cache for the first time, the valid bit is set to 1.
- Data transfers between main memory and disk occur directly bypassing the cache.
- When the data on a disk changes, the main memory block is also updated.
- However, if the data is also resident in the cache, then the valid bit is set to 0.
- What happens if the data in the disk and main memory changes and the write-back protocol is being used?
- In this case, the data in the cache may also have changed and is indicated by the dirty bit.
- The copies of the data in the cache, and the main memory are different. This is called the **cache coherence problem**.
- One option is to force a write-back before the main memory is updated from the disk.





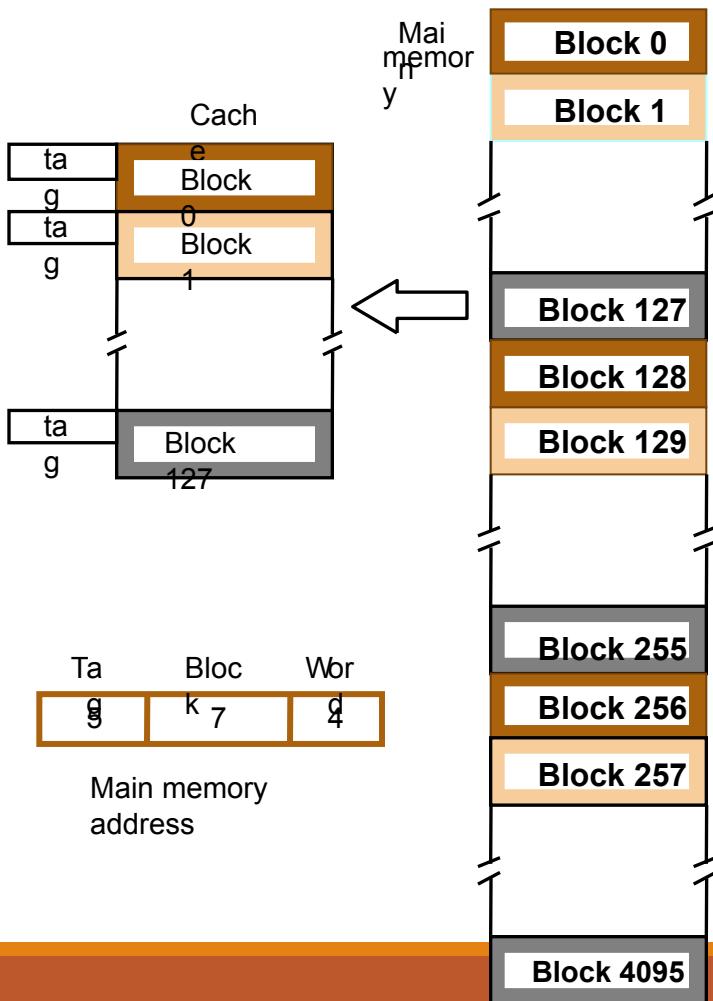
Mapping Functions

- **Mapping functions** determine how memory blocks are placed in the cache.
- A simple processor example:
 - Cache consisting of 128 blocks of 16 words each.
 - Total size of cache is 2048 (2K) words.
 - Main memory is addressable by a 16-bit address.
 - Main memory has 64K words.
 - Main memory has 4K blocks of 16 words each.
- Three mapping functions:
 - Direct mapping
 - Associative mapping
 - Set-associative mapping.





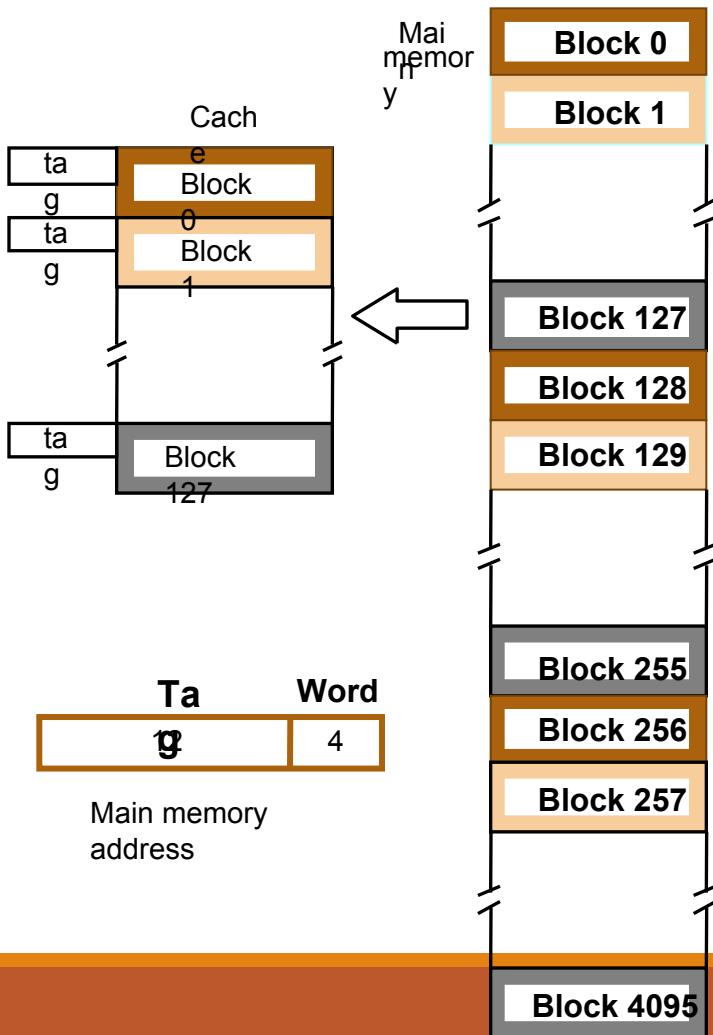
Direct Mapping



- Block j of the main memory maps to $j \bmod 128$ of the cache. 0 maps to 0, 129 maps to 1.
- More than one memory block is mapped onto the same position in the cache.
- May lead to contention for cache blocks even if the cache is not full.
- Resolve the contention by allowing new block to replace the old block, leading to a trivial replacement algorithm.
- Memory address is divided into three fields:
 - Low order 4 bits determine one of the 16 words in a block.
 - When a new block is brought into the cache, the next 7 bits determine which cache block this new block is placed in.
 - High order 5 bits determine which of the possible 32 blocks is currently present in the cache. These are tag bits.
- Simple to implement but not very flexible.



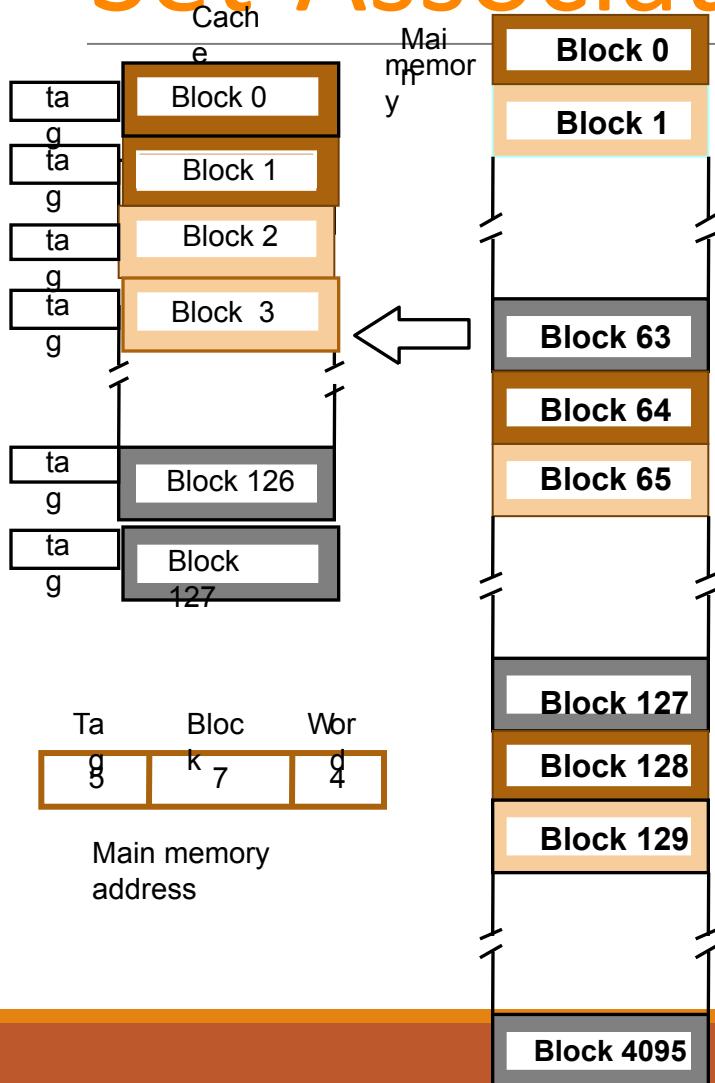
Associative Mapping



- Main memory block can be placed into any cache position.
- Memory address is divided into two fields:
 - Low order 4 bits identify the word within a block.
 - High order 12 bits or tag bits identify a memory block when it is resident in the cache.
- Flexible, and uses cache space efficiently.
- Replacement algorithms can be used to replace an existing block in the cache when the cache is full.
- Cost is higher than direct-mapped cache because of the need to search all 128 patterns to determine whether a given block is in the cache.



Set-Associative mapping



Blocks of cache are grouped into sets.

Mapping function allows a block of the main memory to reside in any block of a specific set.

Divide the cache into 64 sets, with two blocks per set. Memory block 0, 64, 128 etc. map to block 0, and they can occupy either of the two positions.

Memory address is divided into three fields:

- 6 bit field determines the set number.
- High order 6 bit fields are compared to the tag fields of the two blocks in a set.

Set-associative mapping combination of direct and associative mapping.

Number of blocks per set is a design parameter.

- One extreme is to have all the blocks in one set, requiring no set bits (*fully associative mapping*).
- Other extreme is to have one block per set, is the same as *direct mapping*.



Replacement Algorithms

- For direct mapping where there is only one possible line for a block of memory, no replacement algorithm is needed.
- For associative and set associative mapping, however, an algorithm is needed.
- For maximum speed, this algorithm is implemented in the hardware. Four of the most common algorithms are:
 1. **Least Recently Used**:- This replaces the candidate line in cache memory that has been there the longest with no reference to it.
 2. **First In First Out**:- This replaces the candidate line in the cache that has been there the longest.
 3. **Least Frequently Used**:- This replaces the candidate line in the cache that has had the fewest references.
 4. **Random Replacement**:- This algorithm randomly chooses a line to be replaced from among the candidate lines. This yields only slightly inferior performance than other algorithms.

FIFO (First In First Out)

- Pages in main memory are kept in a list
- First in first out is very easy to implement
- The FIFO algorithm select the page for replacement that has been in memory the longest time

FIFO Example

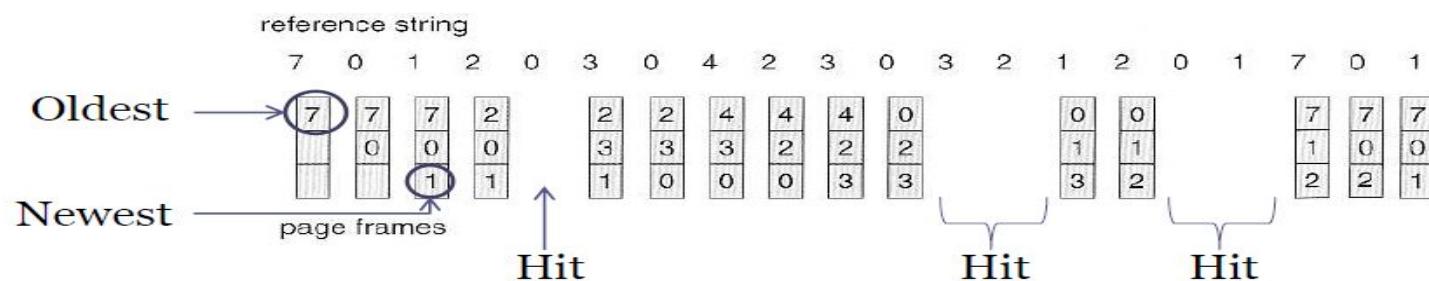


Fig: FIFO example

FIFO (First In First Out)

- Advantages:
 - FIFO is easy to understand.
 - It is very easy to implement.
- Disadvantages:
 - The oldest block in memory may be often used.

LRU (Least Recently Used)

- The least recently used page replacement algorithm keeps track page uses over a short period of time.

LRU Example

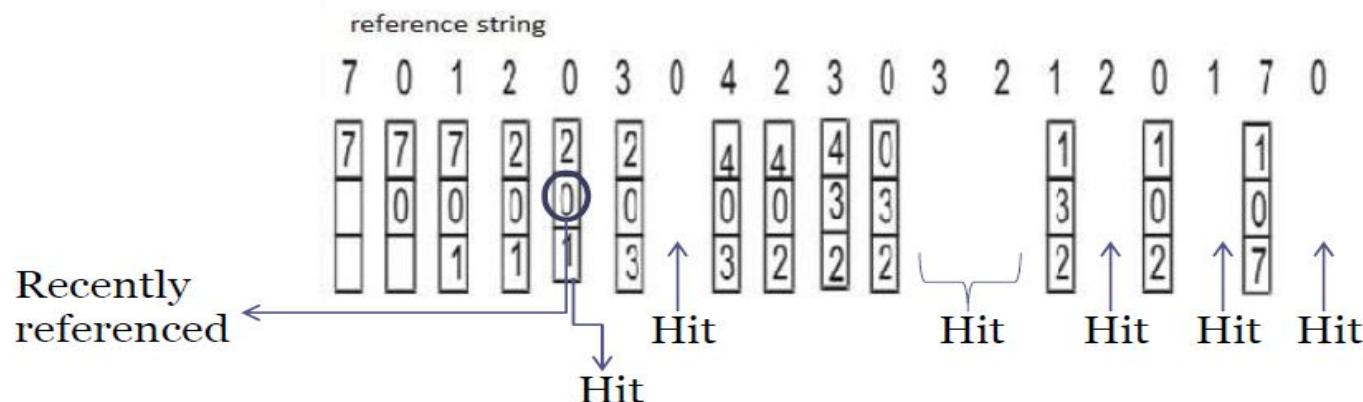


Fig: LRU example

LRU (Least Recently Used)

- Advantages:
 - LRU page replacement algorithm is quiet efficient.
- Disadvantages:
 - Implementation difficult. This algorithm requires keeping track of what was used when, which is expensive if one wants to make sure
 - the algorithm always discards the least recently used item.

Comparison of Clock with FIFO and LRU

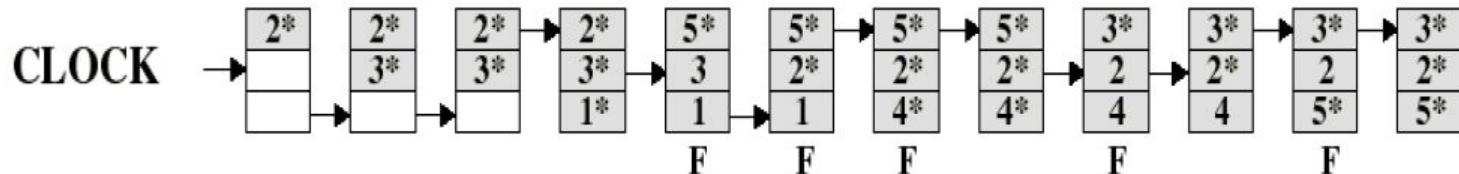
Comparison of Clock with FIFO and LRU (1)

Page address

stream

LRU	2	3	2	1	5	2	4	5	3	2	5	2
	2	2	2	2	2	2	2	2	3	3	3	3
	3	3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
					F	F	F	F	F	F	F	F

FIFO	2	2	2	2	5	5	5	5	3	3	3	3
	2	2	2	2	5	5	5	5	3	3	3	3
	3	3	3	3	3	2	2	2	2	5	5	5
				1	1	1	4	4	4	4	4	4
					F	F	F	F	F	F	F	F



LFU (Least Frequently Used)

- The Least-Frequently-Used (LFU) Replacement technique replaces the least-frequently block in use when an eviction must take place.
- Software counter associated with each block, initially zero is required in this algorithm.
- The operating system checks all the blocks in the cache at each clock interrupt.
- The R bit, which is '0' or '1', is added to the counter for each block. Consequently, the counters are an effort to keep track of the frequency of referencing each block.
- When a block must be replaced, the block that has the lowest counter is selected for the replacement.

LFU (Least Frequently Used)

Reference string:

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1
0	0	0	0	0	0	0	3	3	3	3	3	3	3	0	0	0	0	0
	1	1	1	1	3	3	3	2	2	2	2	2	2	2	2	7	7	7
F	F	F	F		F		F	F	F		F		F		F			

Number of page faults = 12.

Number of page hits= 8.

LFU (Least Frequently Used)

- Advantages:
 - Frequently used block will stay longer than (fifo)
- Disadvantages:
 - Older blocks are less likely to be removed , even if they are on longer frequently used because this algorithm never forgets anything.
 - Newer blocks are more likely to be replaced even if they are frequently used.
 - Captures only frequency factor.

Random Replacement

- When we need to evict a page, choose one randomly
- Advantage:
 - Extremely simple
- Disadvantages:
 - Can easily make "bad" choices by swapping out pages right before they are needed.



The Memory System

PERFORMANCE CONSIDERATIONS



Performance Considerations

A key design objective of a computer system is to achieve the best possible performance at the lowest possible cost.

- Price/performance ratio is a common measure of success.

Performance of a processor depends on:

- How fast machine instructions can be brought into the processor for execution.
- How fast the instructions can be executed.





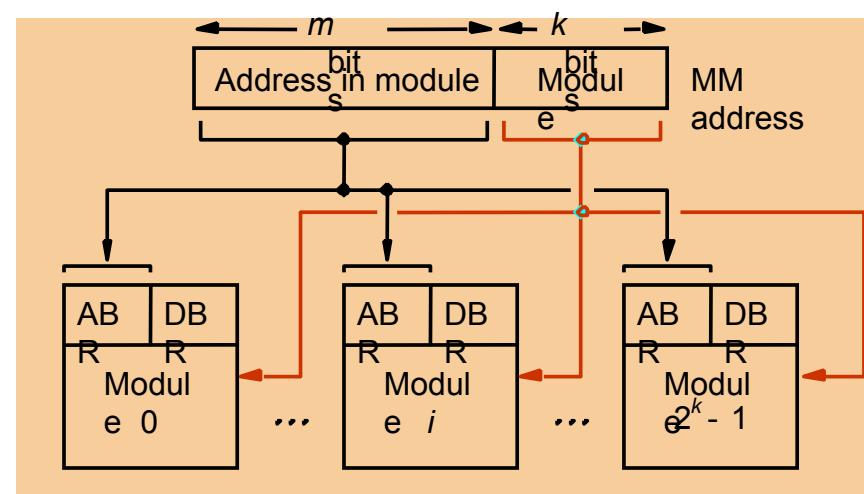
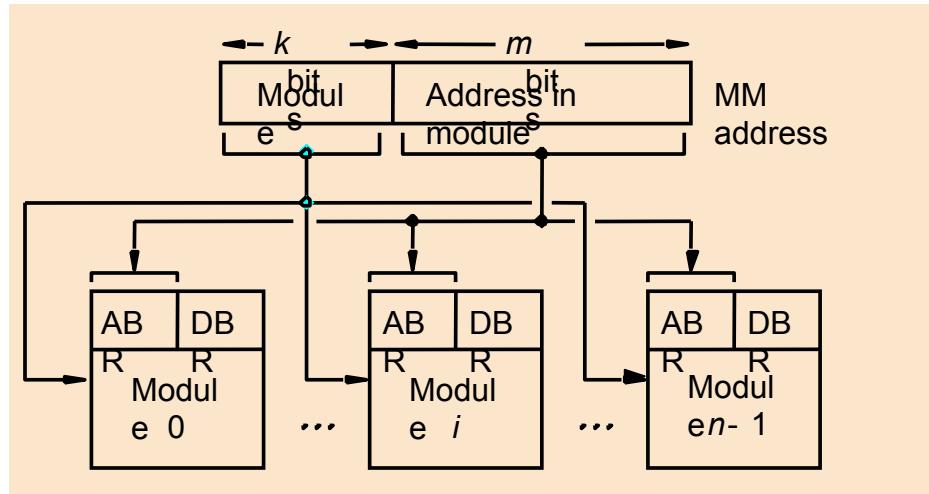
Interleaving

- Divides the memory system into a number of memory modules. Each module has its own address buffer register (ABR) and data buffer register (DBR).
- Arranges addressing so that successive words in the address space are placed in different modules.
- When requests for memory access involve consecutive addresses, the access will be to different modules.
- Since parallel access to these modules is possible, the average rate of fetching words from the Main Memory can be increased.





Methods of Address Layouts



- Consecutive words are placed in a module.
- High-order k bits of a memory address determine the module.
- Low-order m bits of a memory address determine the word within a module.
- When a block of words is transferred from main memory to cache, only one module is busy at a time.

- Consecutive words are located in consecutive modules.
- Consecutive addresses can be located in consecutive modules.
- While transferring a block of data, several memory modules can be kept busy at the same time.





Hit Rate and Miss Penalty

Hit rate

Miss penalty

Hit rate can be improved by increasing block size, while keeping cache size constant

Block sizes that are neither very small nor very large give best results.

Miss penalty can be reduced if load-through approach is used when loading new blocks into cache.





Caches on the Processor Chip

In high performance processors 2 levels of caches are normally used.

Avg access time in a system with 2 levels of caches is

$$T_{ave} = h_1c_1 + (1-h_1)h_2c_2 + (1-h_1)(1-h_2)M$$





Other Performance Enhancements

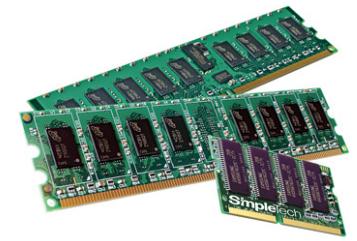
Write buffer

■ Write-through:

- *Each write operation involves writing to the main memory.*
- *If the processor has to wait for the write operation to be complete, it slows down the processor.*
- *Processor does not depend on the results of the write operation.*
- *Write buffer can be included for temporary storage of write requests.*
- *Processor places each write request into the buffer and continues execution.*
- *If a subsequent Read request references data which is still in the write buffer, then this data is referenced in the write buffer.*

■ Write-back:

- *Block is written back to the main memory when it is replaced.*
- *If the processor waits for this write to complete, before reading the new block, it is slowed down.*
- *Fast write buffer can hold the block to be written, and the new block can be read first.*





Other Performance Enhancements (Contd.,)

Prefetching

- New data are brought into the processor when they are first needed.
- Processor has to wait before the data transfer is complete.
- Prefetch the data into the cache before they are actually needed, or a before a Read miss occurs.
- Prefetching can be accomplished through software by including a special instruction in the machine language of the processor.
 - **Inclusion of prefetch instructions increases the length of the programs.**
- Prefetching can also be accomplished using hardware:
 - **Circuitry that attempts to discover patterns in memory references and then prefetches according to this pattern.**





Other Performance Enhancements (Contd.,)

Lockup-Free Cache

- Prefetching scheme does not work if it stops other accesses to the cache until the prefetch is completed.
- A cache of this type is said to be “locked” while it services a miss.
- Cache structure which supports multiple outstanding misses is called a lockup free cache.
- Since only one miss can be serviced at a time, a lockup free cache must include circuits that keep track of all the outstanding misses.
- Special registers may hold the necessary information about these misses.





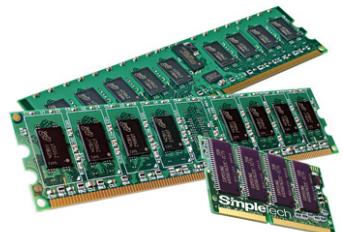
The Memory System

VIRTUAL MEMORY



Virtual Memory

- Recall that an important challenge in the design of a computer system is to provide a large, fast memory system at an affordable cost.
- Architectural solutions to increase the effective speed and size of the memory system.
- Cache memories were developed to increase the effective speed of the memory system.
- Virtual memory is an architectural solution to increase the effective size of the memory system.





Virtual Memory (contd..)

- Recall that the addressable memory space depends on the number of address bits in a computer.
 - For example, if a computer issues 32-bit addresses, the addressable memory space is 4G bytes.
- Physical main memory in a computer is generally not as large as the entire possible addressable space.
 - Physical memory typically ranges from a few hundred megabytes to 1G bytes.
- Large programs that cannot fit completely into the main memory have their parts stored on secondary storage devices such as magnetic disks.
 - Pieces of programs must be transferred to the main memory from secondary storage before they can be executed.





Virtual Memory (contd..)

- When a new piece of a program is to be transferred to the main memory, and the main memory is full, then some other piece in the main memory must be replaced.
 - Recall this is very similar to what we studied in case of cache memories.
- Operating system automatically transfers data between the main memory and secondary storage.
 - Application programmer need not be concerned with this transfer.
 - Also, application programmer does not need to be aware of the limitations imposed by the available physical memory.





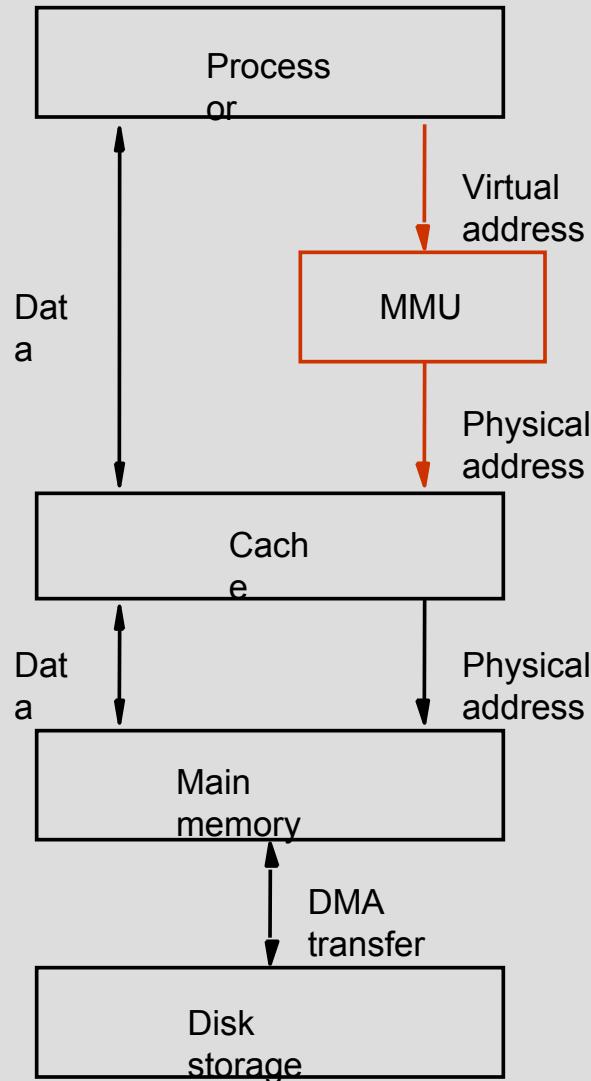
Virtual Memory (contd..)

- Techniques that automatically move program and data between main memory and secondary storage when they are required for execution are called virtual-memory techniques.
- Programs and processors reference an instruction or data independent of the size of the main memory.
- Processor issues binary addresses for instructions and data **called logical or virtual addresses**.
- Virtual addresses are translated into physical addresses by a combination of hardware and software subsystems.
 - If virtual address refers to a part of the program that is currently in the main memory, it is accessed immediately.
 - If the address refers to a part of the program that is not currently in the main memory, it is first transferred to the main memory before it can be used.





Virtual Memory Organization



- *Memory management unit (MMU) translates virtual addresses into physical addresses.*
- *If the desired data or instructions are in the main memory they are fetched as described previously.*
- *If the desired data or instructions are not in the main memory, they must be transferred from secondary storage to the main memory.*
- *MMU causes the operating system to bring the data from the secondary storage into the main memory.*



Address Translation

- Assume that program and data are composed of fixed-length units called pages.
- A page consists of a block of words that occupy contiguous locations in the main memory.
- Page is a basic unit of information that is transferred between secondary storage and main memory.
- Size of a page commonly ranges from 2K to 16K bytes.
 - Pages should not be too small, because the access time of a secondary storage device is much larger than the main memory.
 - Pages should not be too large, else a large portion of the page may not be used, and it will occupy valuable space in the main memory.





Address Translation (contd..)

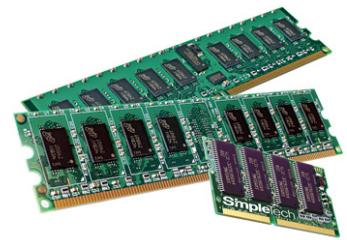
Concepts of virtual memory are similar to the concepts of cache memory.

Cache memory:

- Introduced to bridge the speed gap between the processor and the main memory.
- Implemented in hardware.

Virtual memory:

- Introduced to bridge the speed gap between the main memory and secondary storage.
- Implemented in part by software.





Address Translation (contd..)

- Each virtual or logical address generated by a processor is interpreted as a virtual page number (high-order bits) plus an offset (low-order bits) that specifies the location of a particular byte within that page.
- Information about the main memory location of each page is kept in the page table.
 - Main memory address where the page is stored.
 - Current status of the page.
- Area of the main memory that can hold a page is called as page frame.
- Starting address of the page table is kept in a page table base register.



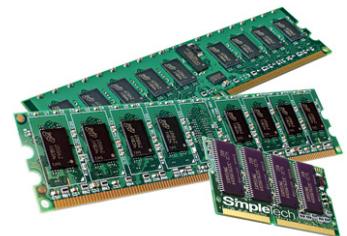


Address Translation (contd..)

Virtual page number generated by the processor is added to the contents of the page table base register.

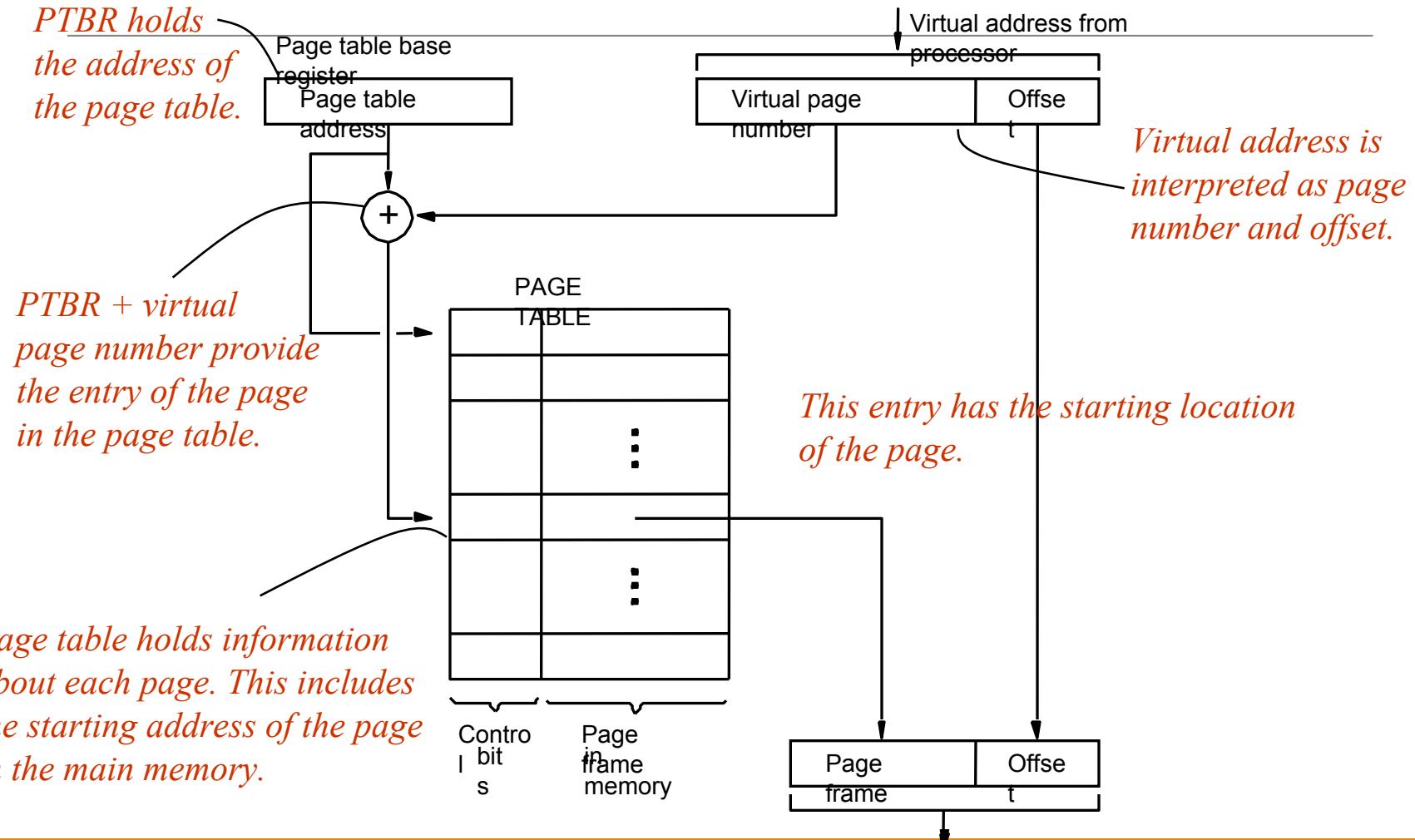
- This provides the address of the corresponding entry in the page table.

The contents of this location in the page table give the starting address of the page if the page is currently in the main memory.





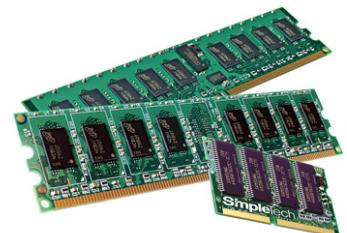
Address Translation (contd.)





Address Translation (contd..)

- Page table entry for a page also includes some control bits which describe the status of the page while it is in the main memory.
- One bit indicates the validity of the page.
 - Indicates whether the page is actually loaded into the main memory.
 - Allows the operating system to invalidate the page without actually removing it.
- One bit indicates whether the page has been modified during its residency in the main memory.
 - This bit determines whether the page should be written back to the disk when it is removed from the main memory.
 - Similar to the dirty or modified bit in case of cache memory.



Address Translation (contd..)



Other control bits for various other types of restrictions that may be imposed.

- For example, a program may only have read permission for a page, but not write or modify permissions.



Address Translation (contd..)



- Where should the page table be located?
- Recall that the page table is used by the MMU for every read and write access to the memory.
 - Ideal location for the page table is within the MMU.
- Page table is quite large.
- MMU is implemented as part of the processor chip.
- Impossible to include a complete page table on the chip.
- Page table is kept in the main memory.
- A copy of a small portion of the page table can be accommodated within the MMU.
 - Portion consists of page table entries that correspond to the most recently accessed pages.



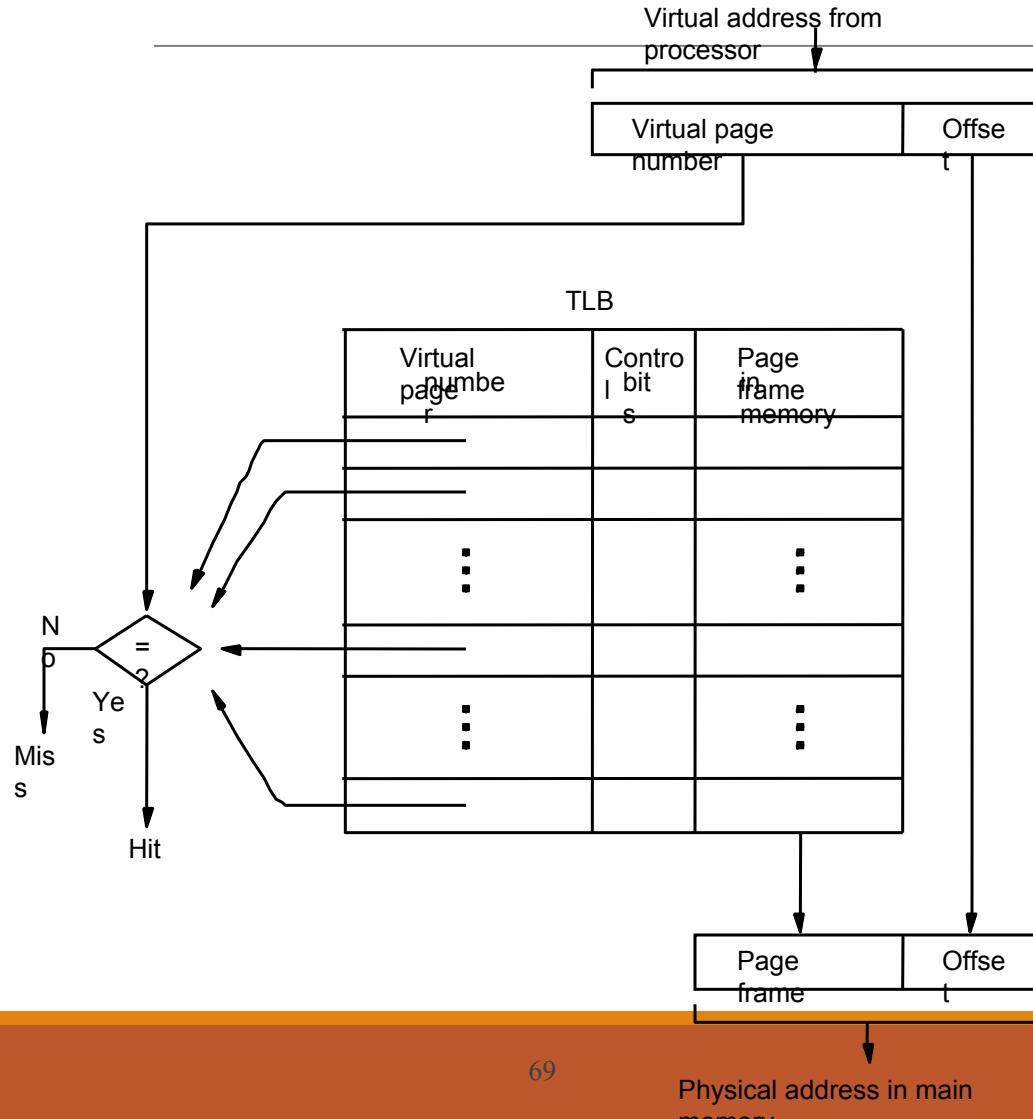


Address Translation (contd..)

- A small cache called as Translation Lookaside Buffer (TLB) is included in the MMU.
 - TLB holds page table entries of the most recently accessed pages.
- Recall that cache memory holds most recently accessed blocks from the main memory.
 - Operation of the TLB and page table in the main memory is similar to the operation of the cache and main memory.
- Page table entry for a page includes:
 - Address of the page frame where the page resides in the main memory.
 - Some control bits.
- In addition to the above for each page, TLB must hold the virtual page number for each page.



Address Translation (contd..)



Associative-mapped TLB

High-order bits of the virtual address generated by the processor select the virtual page.

These bits are compared to the virtual page numbers in the TLB.

If there is a match, a hit occurs and the corresponding address of the page frame is read.

If there is no match, a miss occurs and the page table within the main memory must be consulted.

Set-associative mapped TLBs are found in commercial processors.

Address Translation (contd..)



- How to keep the entries of the TLB coherent with the contents of the page table in the main memory?
- Operating system may change the contents of the page table in the main memory.
 - Simultaneously it must also invalidate the corresponding entries in the TLB.
- A control bit is provided in the TLB to invalidate an entry.
- If an entry is invalidated, then the TLB gets the information for that entry from the page table.
 - Follows the same process that it would follow if the entry is not found in the TLB or if a “miss” occurs.



Address Translation (contd..)



- What happens if a program generates an access to a page that is not in the main memory?
- In this case, a page fault is said to occur.
 - Whole page must be brought into the main memory from the disk, before the execution can proceed.
- Upon detecting a page fault by the MMU, following actions occur:
 - MMU asks the operating system to intervene by raising an exception.
 - Processing of the active task which caused the page fault is interrupted.
 - Control is transferred to the operating system.
 - Operating system copies the requested page from secondary storage to the main memory.
 - Once the page is copied, control is returned to the task which was interrupted.



Address Translation (contd..)



Servicing of a page fault requires transferring the requested page from secondary storage to the main memory.

This transfer may incur a long delay.

While the page is being transferred the operating system may:

- Suspend the execution of the task that caused the page fault.
- Begin execution of another task whose pages are in the main memory.

Enables efficient use of the processor.



Address Translation (contd..)



How to ensure that the interrupted task can continue correctly when it resumes execution?

There are two possibilities:

- Execution of the interrupted task must continue from the point where it was interrupted.
- The instruction must be restarted.

Which specific option is followed depends on the design of the processor.



Address Translation (contd..)



- When a new page is to be brought into the main memory from secondary storage, the main memory may be full.
 - Some page from the main memory must be replaced with this new page.
- How to choose which page to replace?
 - This is similar to the replacement that occurs when the cache is full.
 - The principle of locality of reference (?) can also be applied here.
 - A replacement strategy similar to LRU can be applied.
- Since the size of the main memory is relatively larger compared to cache, a relatively large amount of programs and data can be held in the main memory.
 - Minimizes the frequency of transfers between secondary storage and main memory.



Address Translation (contd..)



- A page may be modified during its residency in the main memory.
- When should the page be written back to the secondary storage?
- Recall that we encountered a similar problem in the context of cache and main memory:
 - Write-through protocol(?)
 - Write-back protocol(?)
- Write-through protocol cannot be used, since it will incur a long delay each time a small amount of data is written to the disk.



Input and Output Organization: Data Transfer Techniques

- The I/O subsystem of a computer provides an efficient mode of communication between the central system and the outside environment.
- It handles all the input-output operations of the computer system.



Input and Output Organization: Data Transfer Techniques

□ Peripheral Devices:

- Input or output devices that are connected to computer are called **peripheral devices**.
- These devices are designed to read information into or out of the memory unit upon command from the CPU and are considered to be the part of computer system. These devices are also called **peripherals**.

For example: Keyboards, display units and printers are common peripheral devices.

□ There are three types of peripherals:

- **Input peripherals** : Allows user input, from the outside world to the computer.
Example: Keyboard, Mouse etc.
- **Output peripherals**: Allows information output, from the computer to the outside world. Example: Printer, Monitor etc.
- **Input-Output peripherals**: Allows both input (from outside world) as well as, output(from computer to the outside world). Example: Touchscreen



Input and Output Organization: Data Transfer Techniques

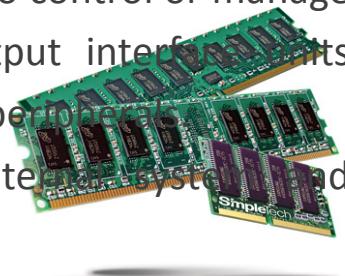
□ Interfaces:

Interface is a shared boundary between two separate components of the computer system which can be used to attach two or more components to the system for communication purposes.

- There are two types of interface:
 - CPU Interface
 - I/O Interface

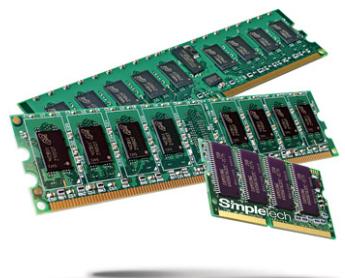
□ Input-Output Interface:

- Peripherals connected to a computer need special communication links for interfacing with CPU.
- It is a special hardware component between the CPU and peripherals to control or manage the input-output transfers. These components are called input-output interfaces because they provide communication links between processor bus and peripheral devices.
- They provide a method for transferring information between internal system and input-output devices.



Modes of I/O Data Transfer

- Data transfer between the central unit and I/O devices can be handled in generally three types of modes which are given below:
 - **Programmed I/O**
 - **Interrupt Initiated I/O**
 - **Direct Memory Access**



Programmed I/O

- Programmed I/O instructions are the result of I/O instructions written in computer program. Each data item transfer is initiated by the instruction in the program.

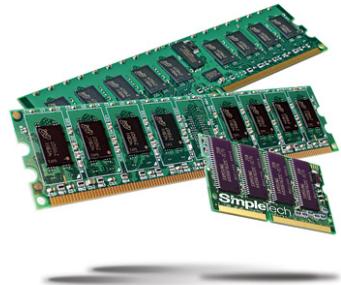
- Usually the program controls data transfer to and from CPU and peripheral. Transferring data under programmed I/O requires constant monitoring of the peripherals by the CPU.



Interrupt Initiated I/O

- In the programmed I/O method the CPU stays in the program loop until the I/O unit indicates that it is ready for data transfer. This is time consuming process because it keeps the processor busy needlessly.

- This problem can be overcome by using interrupt initiated I/O. In this when the interface determines that the peripheral is ready for data transfer, it generates an interrupt. After receiving the interrupt signal, the CPU stops the task which it is processing and service the I/O transfer and then returns back to its previous processing task.

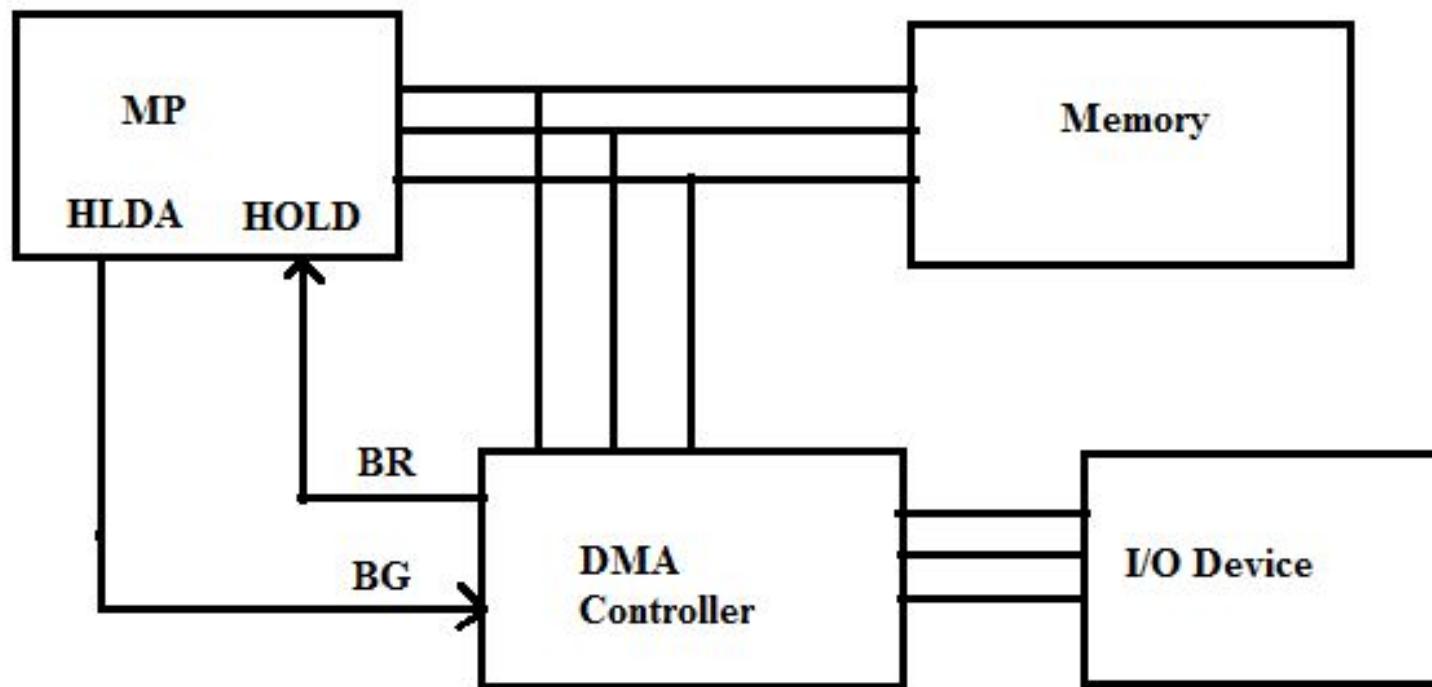


Direct Memory Access (DMA)

- Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This technique is known as **DMA**.
- In this, the interface transfer data to and from the memory through memory bus. A DMA controller manages to transfer data between peripherals and memory unit.
- Many hardware systems use DMA such as disk drive controllers, graphic cards, network cards and sound cards etc. It is also used for intra chip data transfer in multicore processors. In DMA, CPU would initiate the transfer, do other operations while the transfer is in progress and receive an interrupt from the DMA controller when the transfer has been completed.



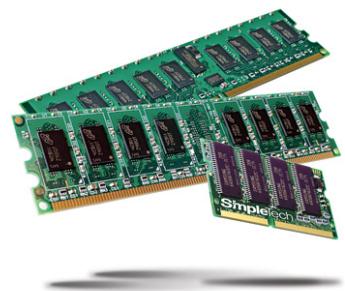
Direct Memory Access



Input/Output Processor

- An **Input-Output Processor (IOP)** is a processor with direct memory access capability. In this, the computer system is divided into a memory unit and number of processors.

- Each IOP controls and manage the input-output tasks. The IOP is similar to CPU except that it handles only the details of I/O processing. The IOP can fetch and execute its own instructions. These IOP instructions are designed to manage I/O transfers only.



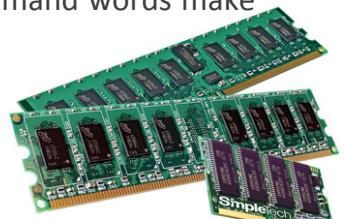
Input/Output Processor

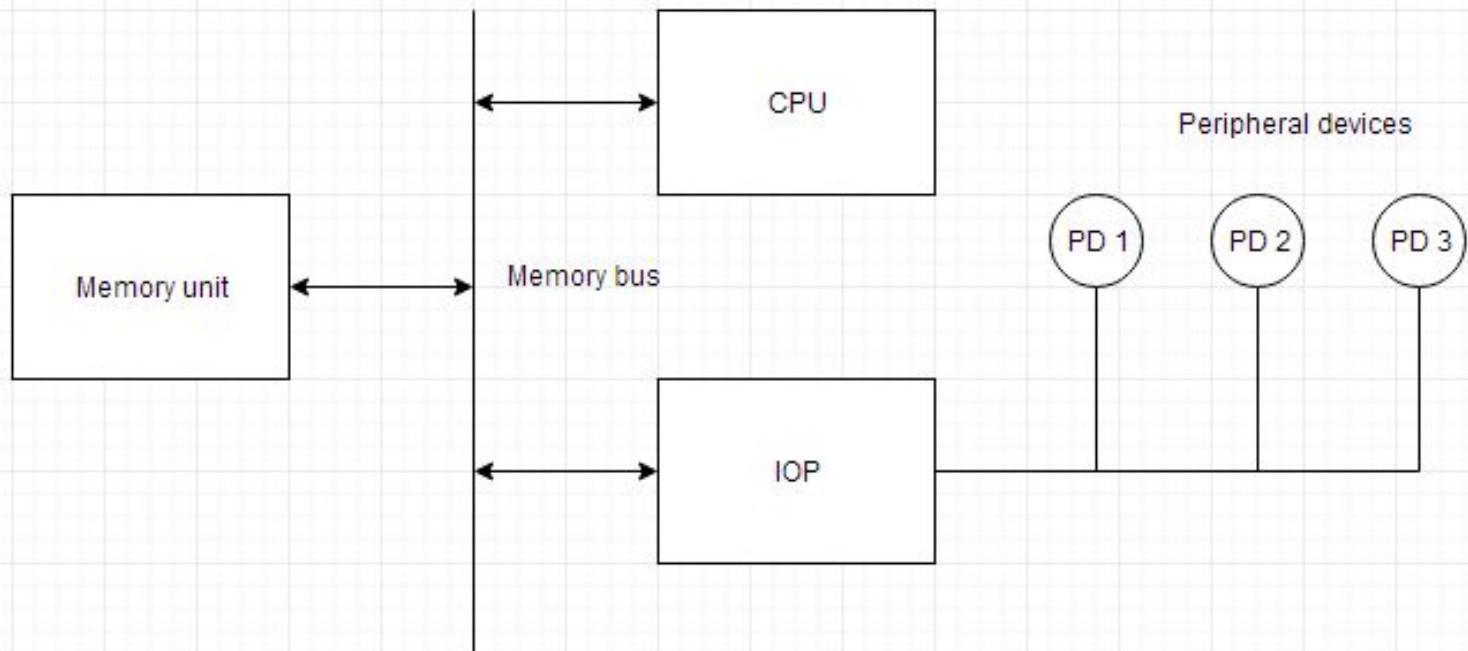
- Below is a block diagram of a computer along with various I/O Processors. The memory unit occupies the central position and can communicate with each processor.
- The CPU processes the data required for solving the computational tasks. The IOP provides a path for transfer of data between peripherals and memory. The CPU assigns the task of initiating the I/O program.
- The IOP operates independent from CPU and transfer data between peripherals and memory.



Input/Output Processor

- ❑ The communication between the IOP and the devices is similar to the program control method of transfer. And the communication with the memory is similar to the direct memory access method.
- ❑ In large scale computers, each processor is independent of other processors and any processor can initiate the operation.
- ❑ The CPU can act as master and the IOP act as slave processor. The CPU assigns the task of initiating operations but it is the IOP, who executes the instructions, and not the CPU. CPU instructions provide operations to start an I/O transfer. The IOP asks for CPU through interrupt.
- ❑ Instructions that are read from memory by an IOP are also called commands to distinguish them from instructions that are read by CPU. Commands are prepared by programmers and are stored in memory. Command words make the program for IOP. CPU informs the IOP where to find the commands in memory.





Input/Output Processor



Need for Input/Output Processor

- Input Output Interface provides a method for transferring information between internal storage and external I/O devices.
- Peripherals connected to a computer need special communication links for interfacing them with the central processing unit.
- The purpose of communication link is to resolve the differences that exist between the central computer and each peripheral.



Need for Input/Output Processor

- The Major Differences are:-
 - Peripherals are electromechanically and electromagnetic devices and CPU and memory are electronic devices. Therefore, a conversion of signal values may be needed.
 - The data transfer rate of peripherals is usually slower than the transfer rate of CPU and consequently, a synchronization mechanism may be needed.
 - Data codes and formats in the peripherals differ from the word format in the CPU and memory.
 - The operating modes of peripherals are different from each other and must be controlled so as not to disturb the operation of other peripherals connected to the CPU.



The Memory System

MEMORY MANAGEMENT

Memory management

Operating system is concerned with transferring programs and data between secondary storage and main memory.

Operating system needs memory routines in addition to the other routines.

Operating system routines are assembled into a virtual address space called system space.

System space is separate from the space in which user application programs reside.

- This is user space.

Virtual address space is divided into one system space + several user spaces.





Memory management (contd..)

- Recall that the Memory Management Unit (MMU) translates logical or virtual addresses into physical addresses.
- MMU uses the contents of the page table base register to determine the address of the page table to be used in the translation.
 - Changing the contents of the page table base register can enable us to use a different page table, and switch from one space to another.
- At any given time, the page table base register can point to one page table.
 - Thus, only one page table can be used in the translation process at a given time.
 - Pages belonging to only one space are accessible at any given time.





Memory management (contd..)

- When multiple, independent user programs coexist in the main memory, how to ensure that one program does not modify/destroy the contents of the other?
- Processor usually has two states of operation:
 - Supervisor state.
 - User state.
- Supervisor state:
 - Operating system routines are executed.
- User state:
 - User programs are executed.
 - Certain privileged instructions cannot be executed in user state.
 - These privileged instructions include the ones which change page table base register.
 - Prevents one user from accessing the space of other users





Memory Mapped I/O

There is no specific input or output instructions

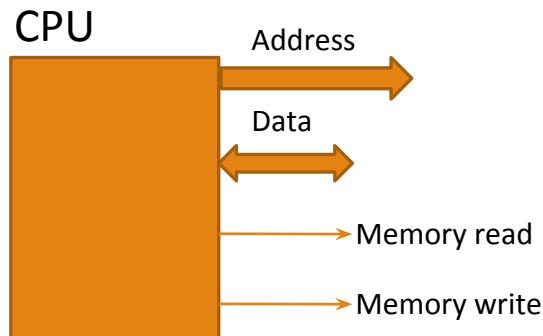
The CPU can manipulate I/O data residing in interface registers with the same instructions that are used to manipulate memory words.

Each interface is organized as set of registers(read & write in normal address space).

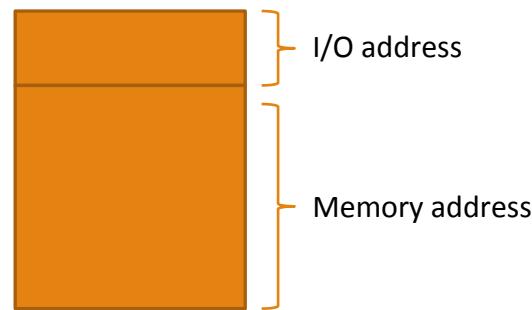
Memory mapped I/O can use memory type instructions to access I/O data.

It allows the computer to use the same instructions for either i/o transfer or for memory transfers.

The advantage is that the load and store instructions used for reading and writing from memory can be used to input and output data from I/O registers.



a) CPU Signals



b) Address space division





Difference between Memory mapped I/O and I/O mapped I/O

	Memory Mapped Input/Output	Input/Output Mapped Input/Output
1.	Each port is treated as a memory location.	Each port is treated as an independent unit.
2.	CPU's memory address space is divided between memory and input/output ports.	Separate address spaces for memory and input/output ports.
3.	Single instruction can transfer data between memory and port.	Two instruction are necessary to transfer data between memory and port.
4.	Data transfer is by means of instruction like MOVE.	Each port can be accessed by means of IN or OUT instructions.





Program Controlled I/O

Program controlled I/O is one in which the processor repeatedly checks a status flag to achieve the required synchronization between processor & I/O device.

The processor polls the device.

It is useful in small low speed systems where hardware cost must be minimized.

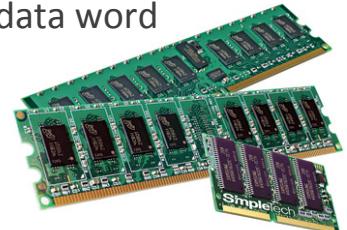
It requires that all input/output operators be executed under the direct control of the CPU.

The transfer is between CPU registers(accumulator) and a buffer register connected to the input/output device.

The i/o device does not have direct access to main memory.

A data transfer from an input/output device to main memory requires the execution of several instructions by the CPU, including an input instruction to transfer a word from the input/output device to the CPU and a store instruction to transfer a word from CPU to main memory.

One or more additional instructions may be needed for address communication and data word counting.



Typical Program Control Instructions



Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare	CMP
Test(by ADDing)	TST





Interrupts

A Suspension of a process such as the execution of a computer program, caused by an event external to that process, and performed in such a way that the process can be resumed. A way to improve processor utilization.

Need For Interrupts?

The OS is a reactive program

1. When you give some input
2. It will perform computations
3. Produces output BUT
4. Meanwhile you can interact with the system by interrupting the running process
or
5. You can stop and start another process.

This reactivity is due to interrupts

Modern Operating Systems Are Interrupt driven





Interrupt Hardware

I/O device request an interrupt by activating a bus line called interrupt-request.

A single interrupt request line may be used to serve n devices.

All devices are connected to interrupt request line via switches to ground.

13-Nov-15 (11)

Interrupt Hardware

- Need to be able to handle an unknown number of possible interrupting devices
- INTR = INTR1 or INTR2 or ... INTR n

The diagram illustrates an equivalent circuit for an open-drain bus. On the left, a box labeled "Processor" has an output line labeled "INTR" with a speaker icon. This line connects to a horizontal bus line. Along this bus line, there are multiple input ports, each consisting of a small square symbol followed by a line labeled "INTR1", "INTR2", "...", and "INTRn". Each port is connected to the bus line via a switch-like symbol (an open rectangle with a diagonal line). The other end of each switch is connected to ground. A pull-up resistor, represented by a zigzag line, is connected between the bus line and a power source labeled "V_{dd}".

Figure 4.6. An equivalent circuit for an open-drain bus used to implement a common interrupt-request line.





Interrupt Hardware Cont....

To request an interrupt, a device closes its associated switch.

If all interrupt-request signals INTR1 to INTRn are inactive, that is, if all switches are open, the voltage on the interrupt request line will equal to Vdd.

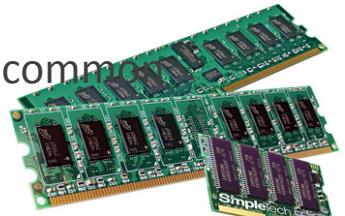
This is an inactivate state of the line.

When a device requests an interrupt by closing its switch, the voltage on the line drops to 0, causing the interrupt- request signal INTR received by the processor to go to 1.

If closing of one (or) more switches that cause the line value to drop to 0, the value of logical OR of the request from individual devices, that is

$$\text{INTR} = \text{INTR1} + \text{INTR2} + \text{INTR3} \dots \dots \dots$$

Use the complement form of INTR to name of the interrupt signal on the common line because this signal is active in the low voltage state





Enabling and Disabling Interrupts

A processor has the facility to enable and disable interrupts as desired.

When a device request the interrupt during the processor service for another interrupt, the result cause the processor enter into the infinite loop.

This can be handled by the following 2 ways:

- The processor ignore the interrupt request line(INTR) until the Interrupt Service Routine(ISR) is completed.
- This can be done by using interrupt-Disable as first instruction and interrupt-Enable as the last instruction.





Enabling and Disabling Interrupts Cont...

The second option is processor automatically disable interrupts before starting the execution of the ISR.

The status register PS stored in the stack with PC value.

The processor set this register bit 1 when the interrupt accept and when a return instruction is executed, the contents of the PS are cleared (0)and stored in the stack again.





Handling Multiple Devices

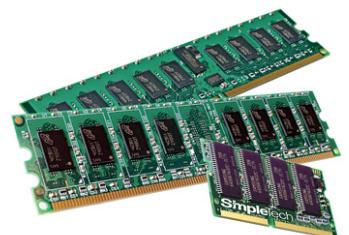
When the number of devices initiating interrupts.

For example, device X may request an interrupt while an interrupt caused by device Y is being serviced.

Hence all the device using the common interrupt line.

Additional information require to identify the device that activated the request.

When the two devices activated the line at the same time, we must break up the tie and chose one the device request among two. Some scheme should be used by the processor.





Handling Multiple Devices

Contd...

1. Polling Scheme

The device that raises the interrupt will set one of the bit (IRQ) in status register to the processor will poll the devices to find which raised an interrupt first.

Disadvantage:

Time spend in interrogating the IRQ bits of the devices that may not be requesting any service.

2. Vectored Interrupts

To reduce the time involved in the polling scheme, a device requesting an interrupt may identify itself directly to the processor. A device can send a special code to the processor over the bus. The code is used to identify the device. If the interrupt produces a CALL to a predetermined memory location, which is the starting address of ISR, then that address is called vectored address and such interrupts are called vectored interrupts.



Handling Multiple Devices

Contd...



3. Interrupt priority

When a interrupt arrives from one (or) more devices simultaneously, the processor has to decide which request should be serviced first.

- The processor takes this decision with the help of interrupt priorities.
- The processor accepts interrupt request having highest priority.
- Each request assign a different priority level.
- The request received from the interrupt request line are sent to a priority arbitration circuit in the processor.
- The request is accepted only if it has a higher priority level than that currently assigned to the processor.

4. Controlling device request

The processor allow only the input / output devices requested(interrupt), that are being used by a given program.

- Other devices should not be allowed to generate interrupt requests even though they are ready to transfer the data.
- Hence, we need a mechanism in the interface circuits of individual devices to control whether the device is allowed to generate an interrupt request.

Two mechanism for control request:

1. One is at the device end- interrupt enable bit in the control register(IRQ).
2. Processor end- enable bit in the program status register(PS) or priority structure determine whether a given interrupt request will be accepted.



Thank You

