MODULE 1
INTRODUCTION TO COMPUTER ARCHITECTURE AND ORGANIZATION

1.1 INTRODUCTION

- ***Computer Architecture*:**

  It refers to those **attributes of a system which are visible to the programmer**, or in other words, those **attributes that have a direct impact on the logical execution of a program**.

- ***Computer Organization*:**

  It refers to the **operational units and their interconnections** that realize the architectural specifications.

- **Examples of architecture** attributes include the **instruction set, the number of bits** to represent various data types (e.g.., numbers, and characters), **I/O mechanisms, and technique for addressing memory**.

- **Examples of organization** attributes include those hardware details transparent to the programmer, such as **control signals, interfaces between the computer and peripherals, and the memory technology used**.

- It is an architectural design issue whether a computer will have a multiply instruction.
- It is an organizational issue whether that instruction will be implemented by a special multiply unit or by a mechanism that makes repeated use of the add unit of the system.
- The organization decision may be based on the anticipated frequency of use of the multiply instruction, the relative speed of the two approaches, and the cost and physical size of a special multiply unit.
- Many computer manufacturers offer a family of computer model, all with the same architecture but with differences in organization. Consequently, the different models in the family have different price and performance characteristics.
- Furthermore, **an architecture may survive many years, but its organization changes with changing technology**.

- ***Example - IBM System/370 architecture***
- This architecture was first introduced in 1970 and included several models. The customer with modest requirements could buy a cheaper, slower model and, if demand increased, later upgrade to a more expensive, faster model without having to abandon software that had already been developed.
- Over the years, IBM has introduced many new models with improved technology to replace older models, offering the customer greater speed, lower cost, or both.
- These newer models retained the same architecture so that the customer's software investment was protected. Remarkably, the System/370 architecture, with a few enhancements, has survived to this day as the architecture of IBM's mainframe product line.

1.2.1 FUNCTION

- In general terms, there are four main functions of a computer:

1. *Data processing*
2. *Data storage*
3. *Data movement*
4. *Control*

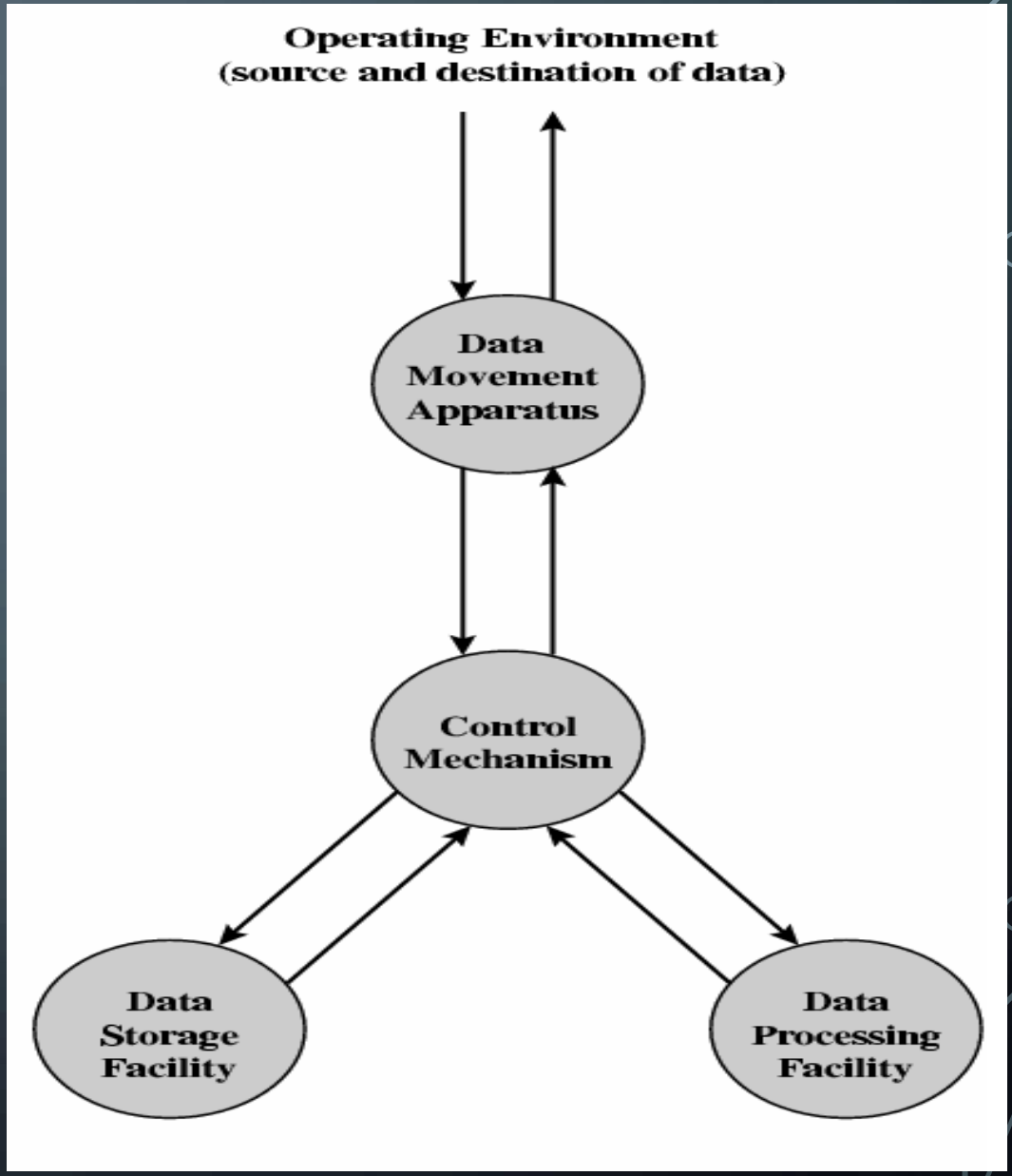


Figure 1.1 ***A functional view of the computer***

- The computer, of course, must be able to process data. The data may take a wide variety of forms, and the range of processing requirements is broad. However, we shall see that there are only a few fundamental methods or types of data processing.
- It is also essential that a computer stores data. Even if the computer is processing data on the fly (i.e., data come in and get processed, and the results go out immediately), the computer must temporarily store at least those pieces of data that are being worked on at any given moment. Thus, there is at least a short-term data storage function.
- The computer must be able to move data between itself and the outside world. The computer's operating environment consists of devices that serve as either sources or destinations of data. When data are received from or delivered to a device that is directly connected to the computer, the process is known as input-output (I/O), and the device is referred to as a peripheral. When data are moved over longer distances, to or from a remote device, the process is known as data communications.

- Finally, there must be control of these three functions. Ultimately, this control is exercised by the individual who provides the computer with instructions. Within the computer system, a control unit manages the computer's resources and orchestrates the performance of its functional parts in response to those instructions.

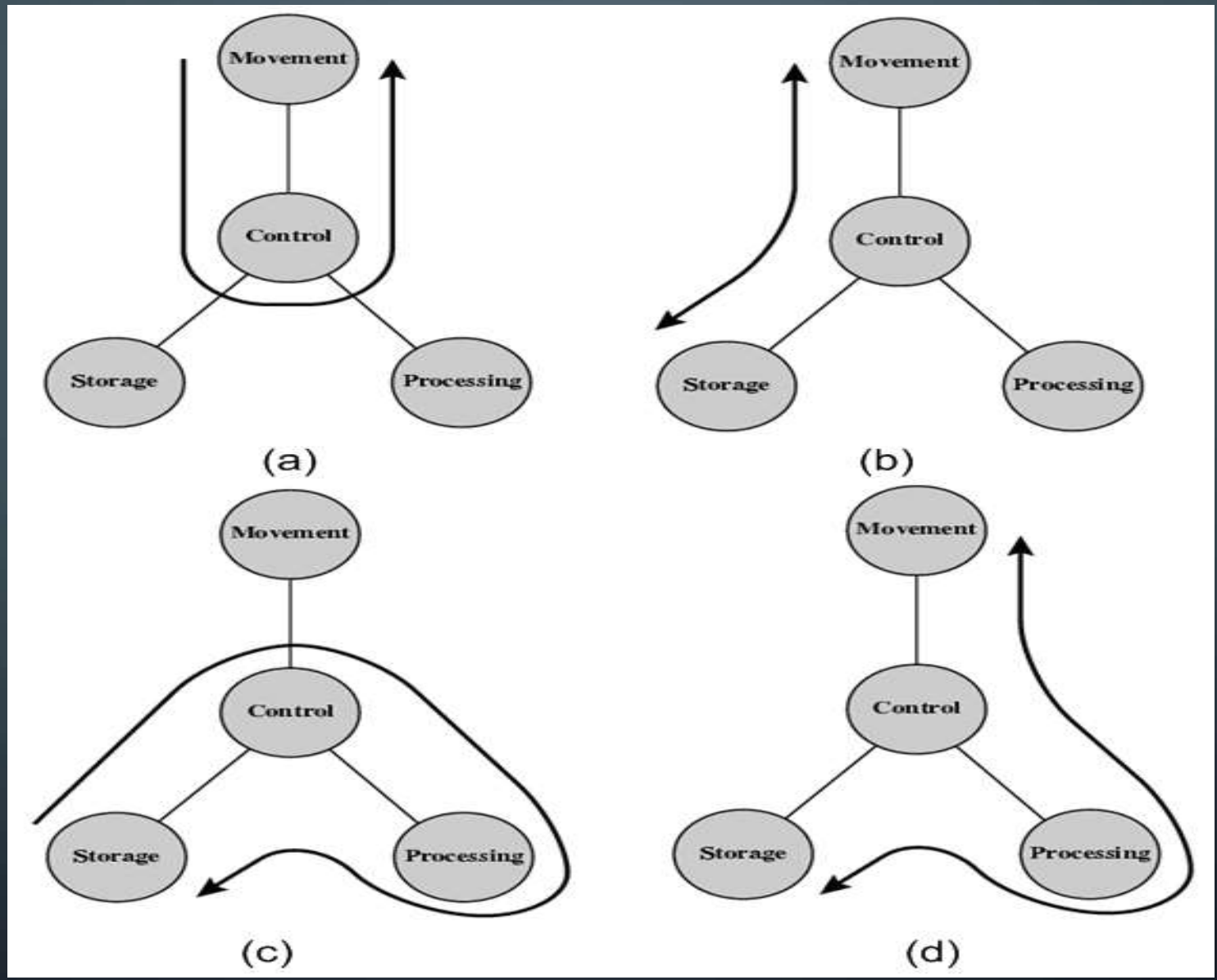


Figure 1.2 ***Possible computer operations***

1.2.2 STRUCTURE

- Figure 1.3 is the simplest possible depiction of a computer.
- The computer is an entity that interacts in some fashion with its external environment.
- In general, all of its linkages to the external environment can be classified as peripheral devices or communication lines.

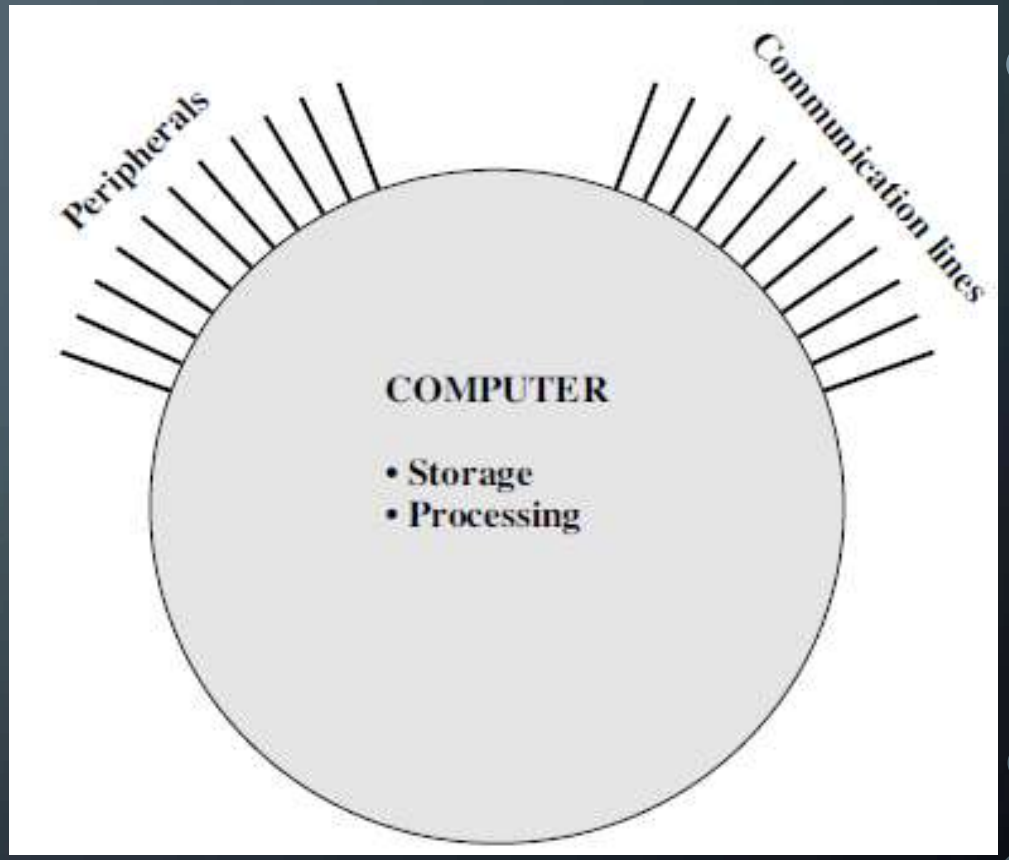


Figure 1.3: ***The Computer: top-level structure***

- But of greater concern is the internal structure of the computer itself, which is shown in Figure 1.4. There are four main structural components:

1. ***Central Processing Unit (CPU):*** Controls the operation of the computer and performs its data processing functions. Often simply referred to as processor.
2. ***Main Memory:*** Stores data.
3. ***I/O:*** Moves data between the computer and its external environment.
4. ***System Interconnection*:** Some mechanism that provides for communication among CPU, main memory, and I/O.

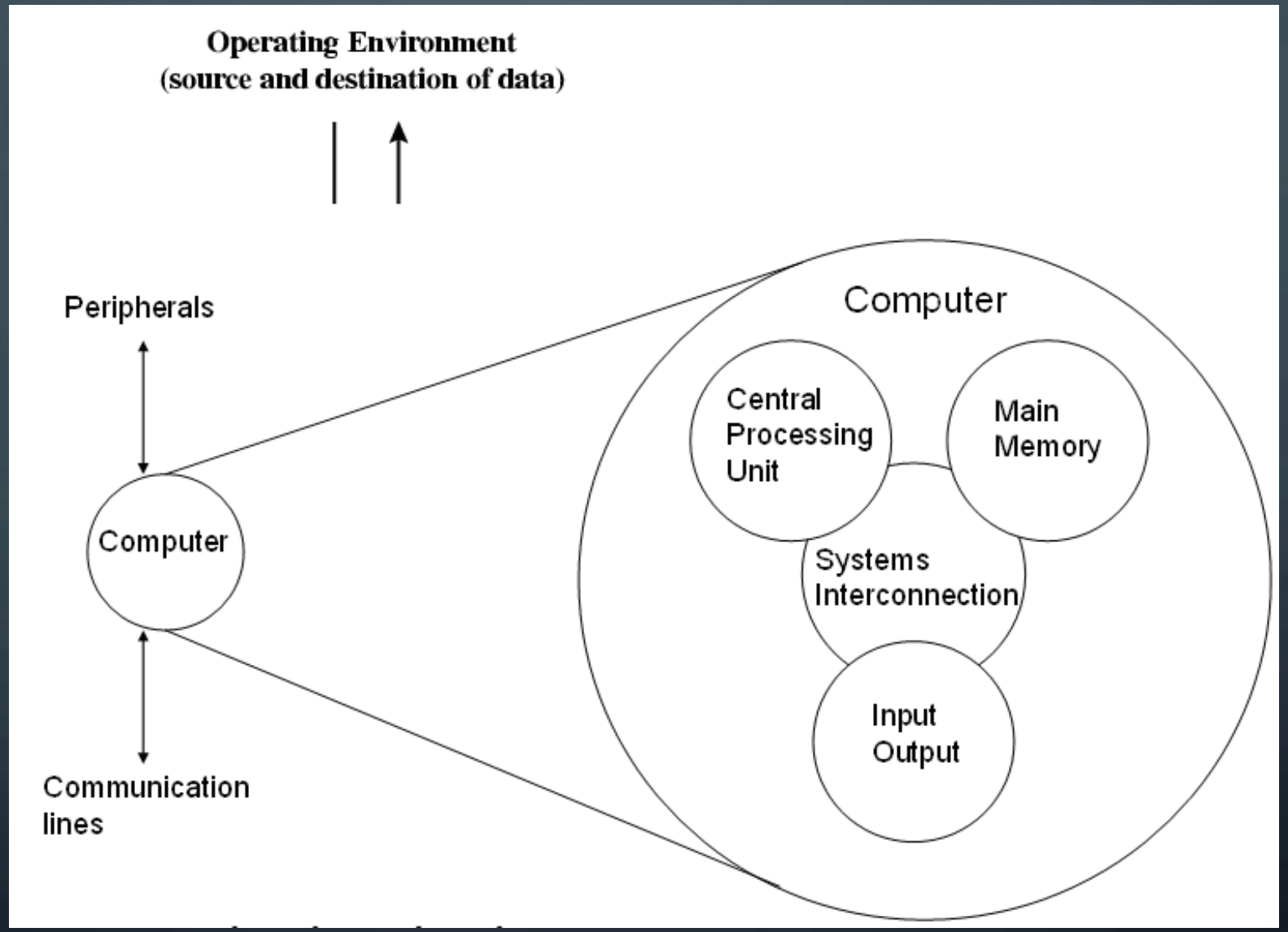


Figure 1.4: ***The computer: top-level structure***

- There may be one or more of each of the above components. Traditionally, there has been just a single CPU. However, in recent years, there has been increasing use of multiple processors, in a single system.
- For our purpose, the most interesting and in some ways the most complex component is the CPU; its structure is depicted in Figure 1.5.

- Its major structural components are as follows:
  1. ***Control Unit (CU):*** Controls the operation of the CPU and hence the computer.
  2. ***Arithmetic and Logic Unit (ALU):*** Performs computer's data processing functions.
  3. ***Register:*** Provides storage internal to the CPU.
  4. ***CPU Interconnection:*** Some mechanism that provides for communication among the control unit, ALU, and register.

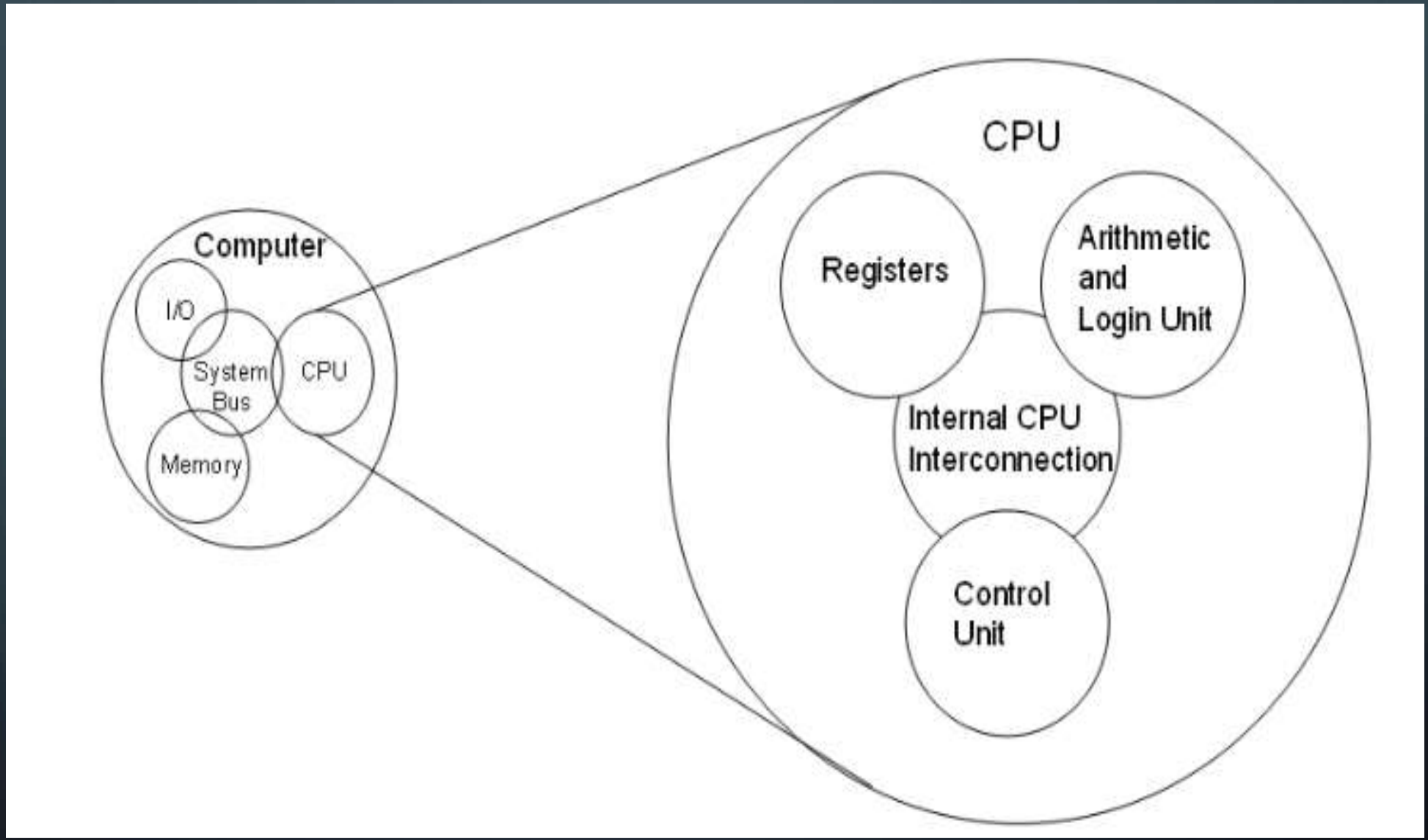


Figure 1.5***: The CPU***

- The Control Unit further comprises of
  1. ***Control Memory***
  2. ***Control Unit Register***
  3. ***Sequencing Logic.***

- The Control Memory stores the microinstructions, loads it into the control unit register and the sequencing logic gives these signals in a proper sequence to execute an instruction.

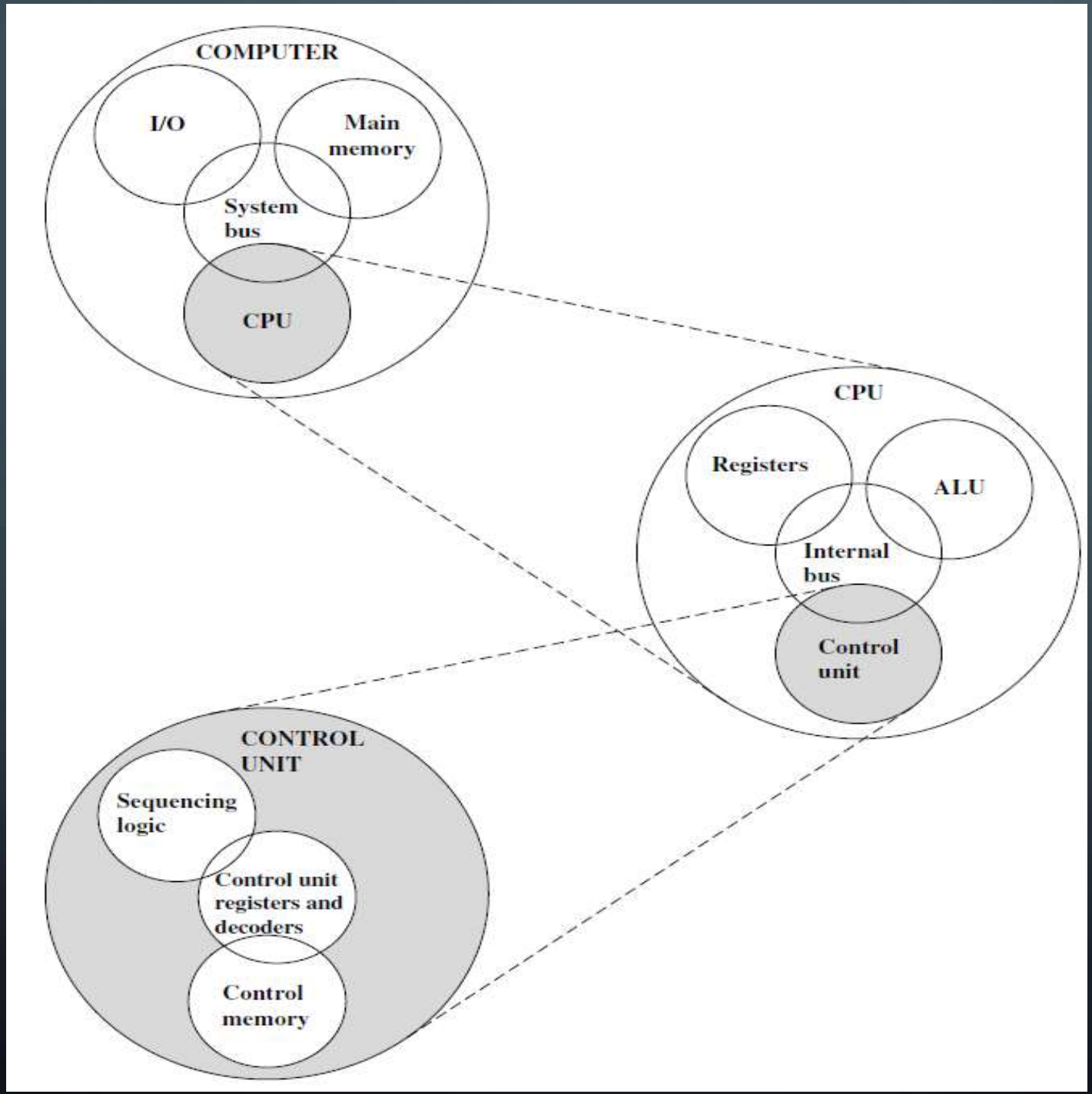
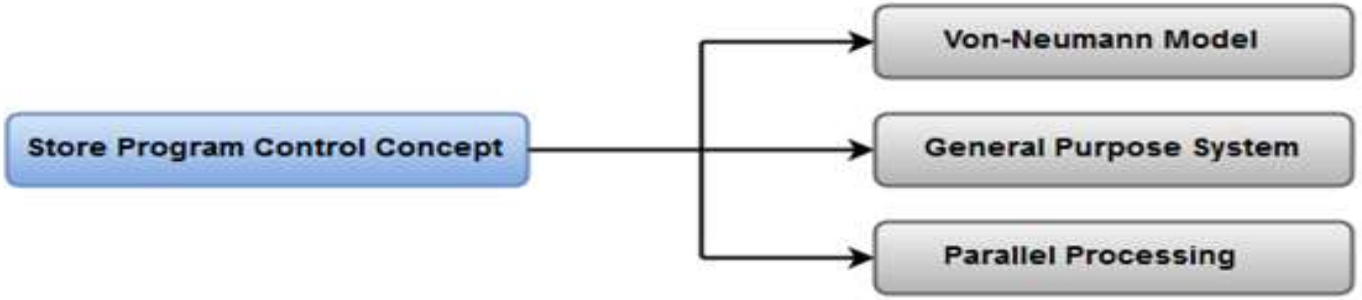


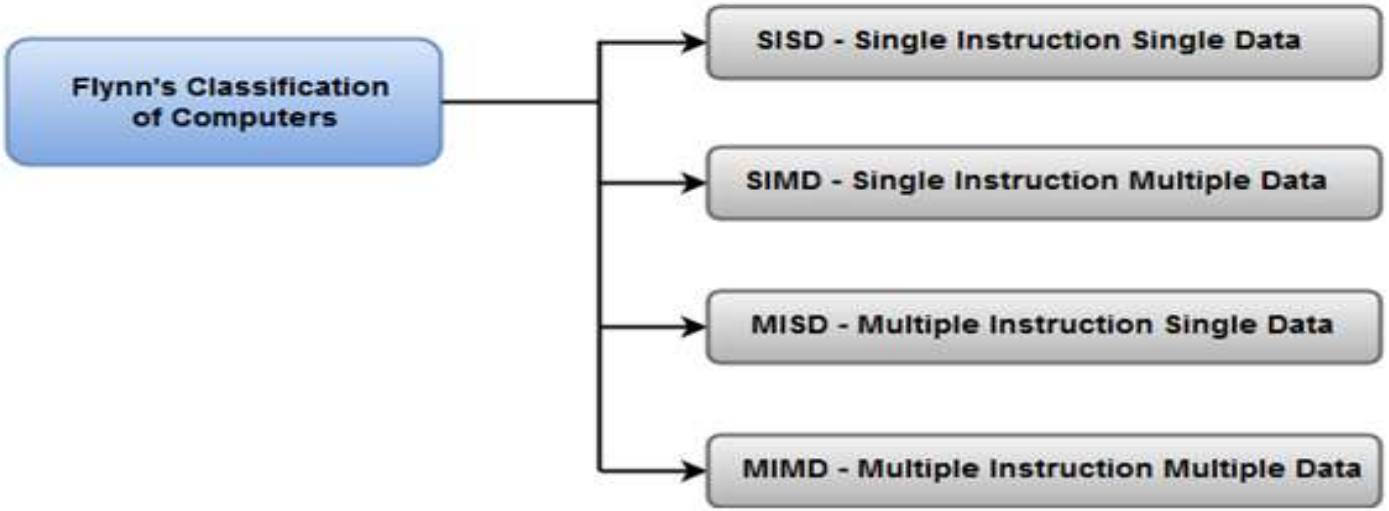
Figure 1.6: ***The Control Unit***

General System Architecture

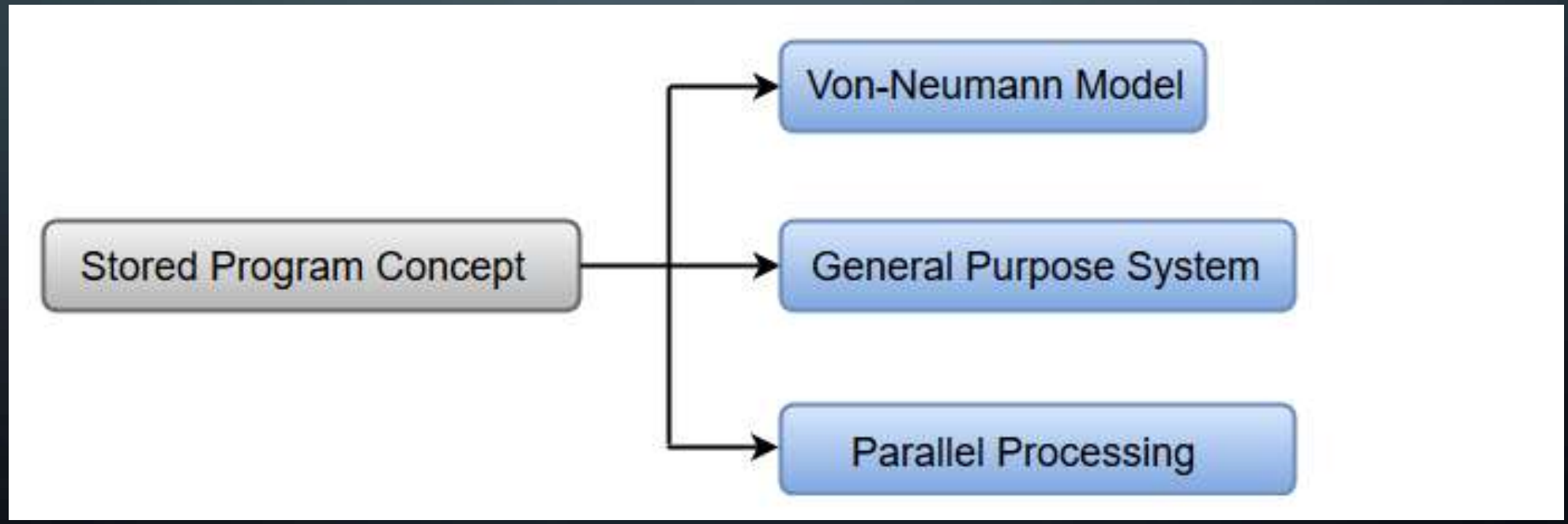
In Computer Architecture, the General System Architecture is divided into two major classification units.

1. Store Program Control Concept
2. Flynn's Classification of Computers





- The term **Stored Program Control Concept** refers to the storage of instructions in computer memory to enable it to perform a variety of tasks in sequence or intermittently.
- ***John von Neumann*** proposed that a program be electronically stored in the binary-number format in a memory device so that instructions could be modified by the computer as determined by intermediate computational results.



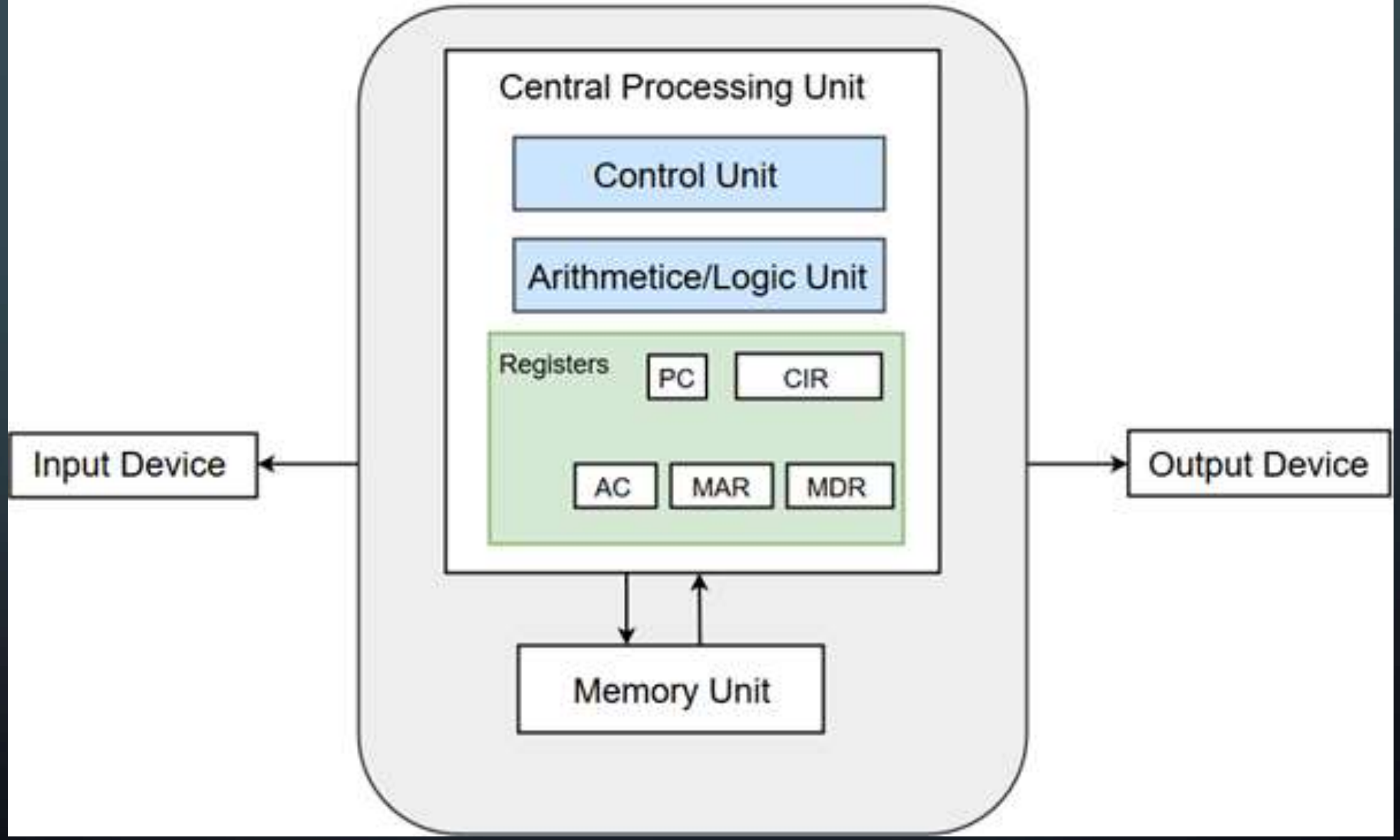
VON NUMEN MODEL

- It consists of a CU, ALU, Memory Unit, Registers & Inputs/Outputs.
- Von Neumann architecture is based on the stored-program computer concept, where *instruction data and program data are stored in the same memory*.

- Since instructions and data both are stored in the same memory, so *same buses are used to fetch instructions and data*. This means the *CPU cannot do both things together* (read the instruction and read/write data).

- Features of Von Numen System:
  - ***Uses a single processor***
  - ***Uses one memory for both instructions and data.***
  - ***Executes programs following the fetch-decode-execute cycle***

Von-Neumann Basic Structure:



- CU: Fetching next instructions and executing them.
- ALU: Arithmetic operations.
- Registers: Registers refer to high-speed storage areas in the CPU. The data processed by the CPU are fetched from the registers.

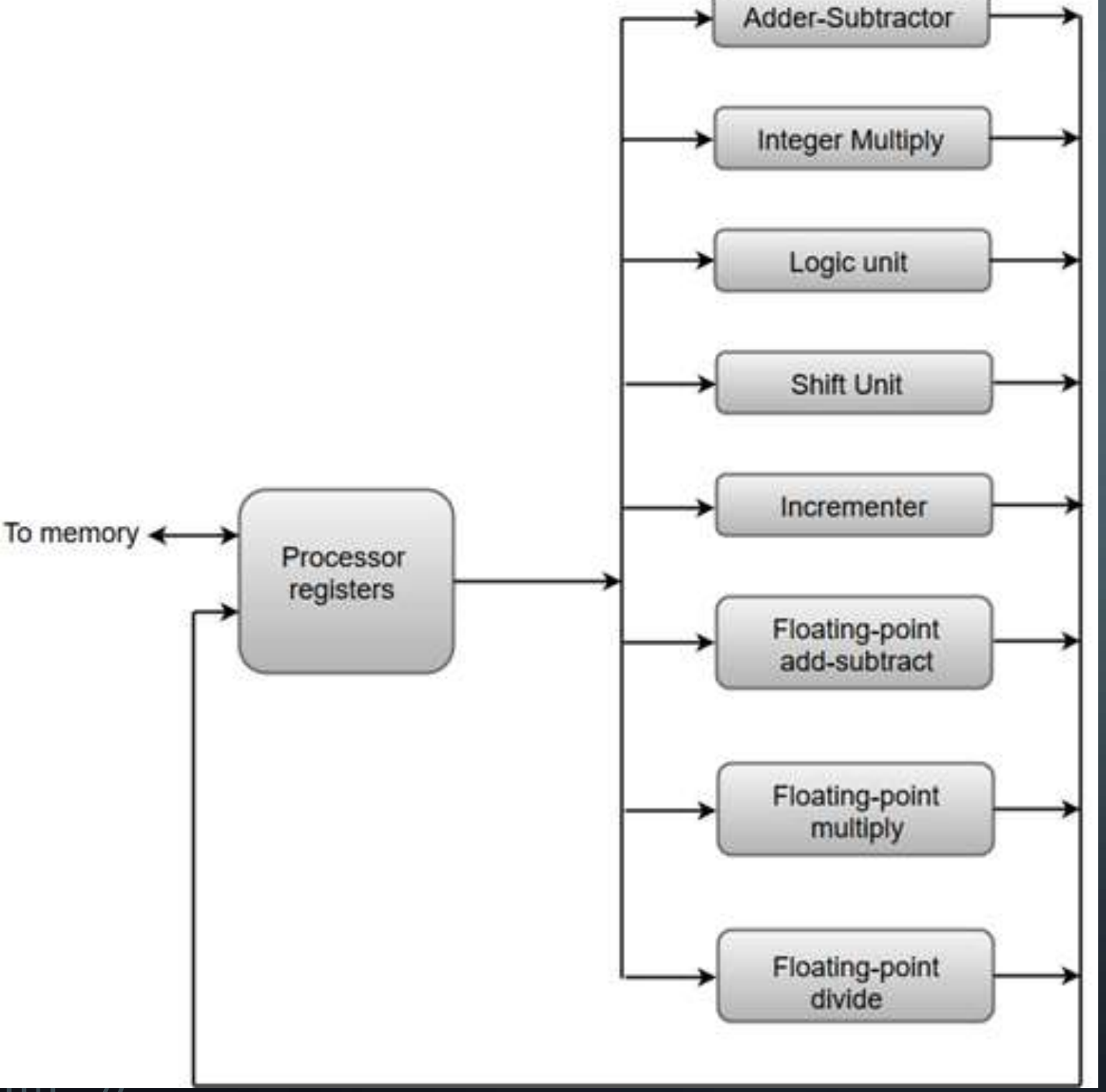
Registers	Description
MAR (Memory Address Register)	This register holds the memory location of the data that needs to be accessed.
MDR (Memory Data Register)	This register holds the data that is being transferred to or from memory.
AC (Accumulator)	This register holds the intermediate arithmetic and logic results.
PC (Program Counter)	This register contains the address of the next instruction to be executed.
CIR (Current Instruction Register)	This register contains the current instruction during processing.

PARALLEL PROCESSING

- Parallel processing can be described as a class of *techniques which enables the system to achieve simultaneous data-processing tasks to increase the computational speed of a computer system*.

- The primary purpose of parallel processing is to *enhance the computers processing capability and increase its throughput*, i.e. the amount of processing that can be accomplished during a given interval of time.

- A parallel processing system can be achieved by *having a multiplicity of functional units that perform identical or different operations simultaneously. The data can be distributed among various multiple functional units.*

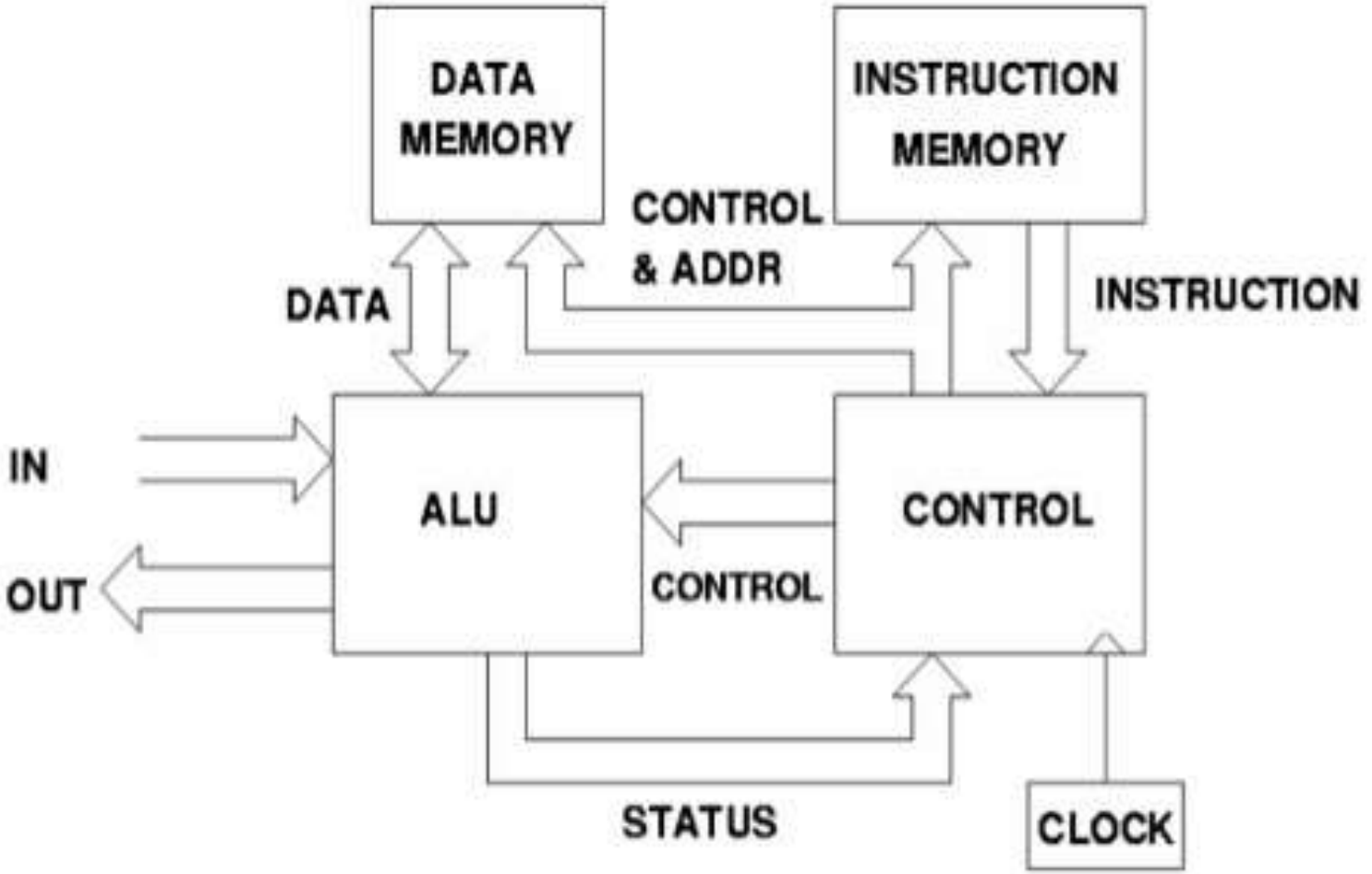


- The adder and integer multiplier performs the arithmetic operation with integer numbers.
- The floating-point operations are separated into three circuits operating in parallel.
- The logic, shift, and increment operations can be performed concurrently on different data.
- All units are independent of each other, so one number can be shifted while another number is being incremented.

Harvard Architecture

- Harvard Architecture is the computer architecture that *contains separate storage and separate buses* (signal path) *for instruction and data*.
- It was basically *developed to overcome the bottleneck of Von Neumann* Architecture.
- The main advantage of having separate buses for instruction and data is that the *CPU can access instructions and read/write data at the same time*.

Harvard Architecture

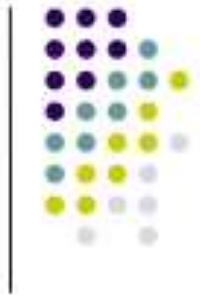


DATA REPRESENTATION AND ARITHMETIC ALGORITHMS

Register Organization

- CPU must have some working space (temporary storage) called **registers**.
- A computer system employs **a memory hierarchy**.
- At the **highest level** of hierarchy, **memory is faster, smaller and more expensive.**
- Within the CPU, there is **a set of registers** which can be treated as a memory in the **highest level of hierarchy**.

Register Organization



- The registers in the CPU can be categorized into two groups

1. User-visible registers:

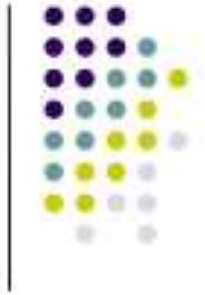
- These enables the machine - or assembly-language programmer to minimize main memory reference by optimizing use of registers.

2. Control and status registers:

- These are used by the control unit to control the operation of the CPU.
- Operating system programs may also use these in privileged mode to control the execution of program.

User-visible registers

- General Purpose
- Data
- Address
- Condition Codes



1. General Purpose Registers:

- Used for a variety of functions by the programmer.
- Sometimes used for holding **operands(data) of an instruction**.
- Sometimes used for **addressing functions** (e.g., register indirect, displacement).

2. Data registers:

- Used to hold **only data.**
- **Cannot** be employed in the calculation of an operand address.

3. Address registers:

- Used exclusively for the purpose of **addressing**.
- Examples include the following:

1. Segment pointer:

- In a machine with segment addressing, a segment register holds the **address of the base of the segment**.
- There may be multiple registers, one for the code segment and one for the data segment.

2. Index registers:

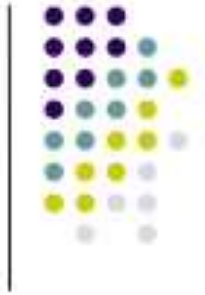
- These are used for **indexed addressing** and may be **auto indexed.**

3. Stack pointer:

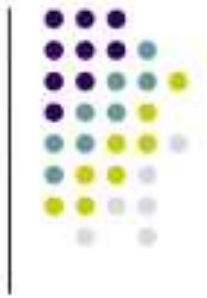
- A dedicated register that points to the top of the stack.
- Auto incremented or auto decremented using **PUSH** or **POP operation**

4. Condition Codes Registers:

- Sets of individual bits
  - e.g. result of last operation was zero
- Can be read (implicitly) by programs
  - e.g. Jump if zero
- Can not (usually) be set by programs



Control and status registers



- **Four registers are essential to instruction execution:**

1. Program Counter (PC):

- Contains the address of an instruction to be fetched.

2. Instruction Register (IR):

- Contains the instruction most recently fetched.

3. Memory Address Register (MAR):

- Contains the address of a location of main memory from where information has to be fetched or information has to be stored.

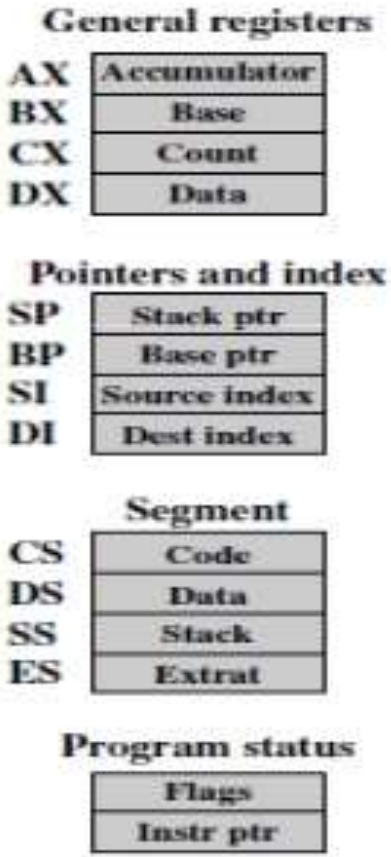
4. Memory Buffer Register (MBR):

- Contains a word of data to be written to memory or the word most recently read.

Control and status registers

- **Program Status Word (PSW)**
  - Condition code bits are collected into one or more registers, known as the **program status word (PSW),** that contains status information.
  - Common fields or flags include the following:
    - **Sign:** Contains the sign bit of the result of the last arithmetic operation.
    - **Zero:** Set when the result is zero.
    - **Carry:** Set if an operation resulted in a carry (addition) into or borrow (subtraction) out of a high order bit.
    - **Equal:** Set if a logical compare result is equal.
    - **Overflow:** Used to indicate arithmetic overflow.
    - **Interrupt enable/disable:** Used to enable or disable interrupts.

Register organization of INTEL 8086 processor



(b) 8086

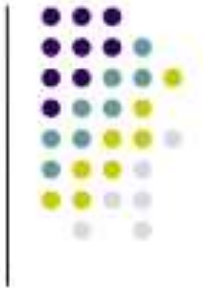
Register organization of INTEL 8086 processor

- **16-bit flags, Instruction Pointer**
- **General Registers, 16 bits**
  - AX – Accumulator, favored in calculations
  - BX – Base, normally holds an address of a variable or func
  - CX – Count, normally used for loops
  - DX – Data, normally used for multiply/divide
- **Segment, 16 bits**
  - SS – Stack, base segment of stack in memory
  - CS – Code, base location of code
  - DS – Data, base location of variable data
  - ES – Extra, additional location for memory data

Register organization of INTEL 8086 processor

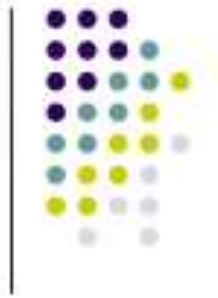
- **Index, 16 bits**
  - BP – Base Pointer, offset from SS for locating subroutines
  - SP – Stack Pointer, offset from SS for top of stack
  - SI – Source Index, used for copying data/strings
  - DI – Destination Index, used for copy data/strings

INSTRUCTION FORMAT



- The operation of the computer system are determined by **the instructions** executed by the central processing unit.
- These instructions are known as **machine instruction** and are in the form of **binary codes**.
- Each instruction of the CPU has specific **information field** which are required to execute it.
- These information field of instructions are called **elements of instruction**.

➢Elements of Instruction



1. **Operation Code:**

   Binary code that specifies which operation to be performed.

2. **Source operand address:**

   Specifies one or more source operands

3. **Destination operand address:**

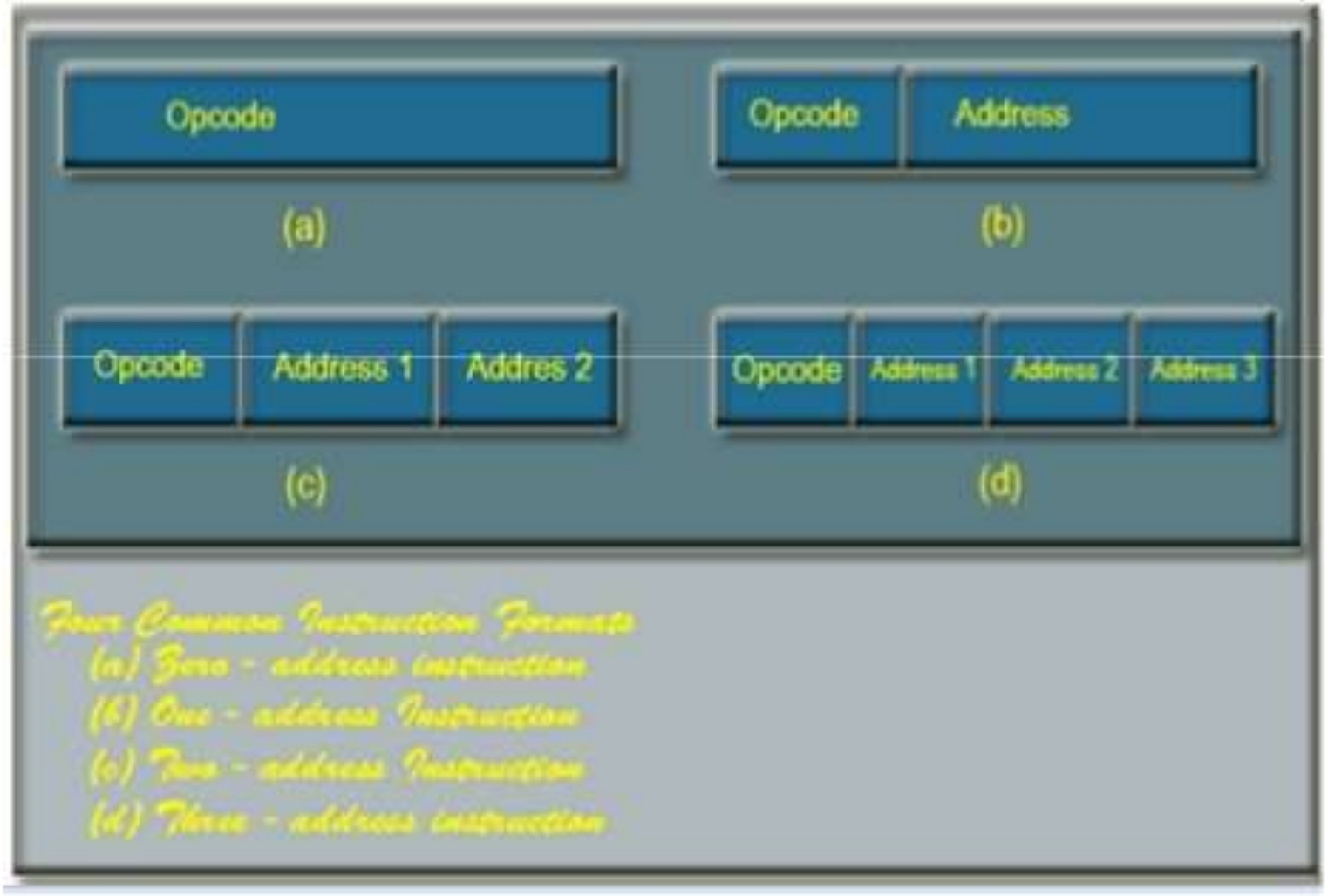
   The operation executed by the CPU may produce result which is stored in the destination address.

4. **Next instruction address:**

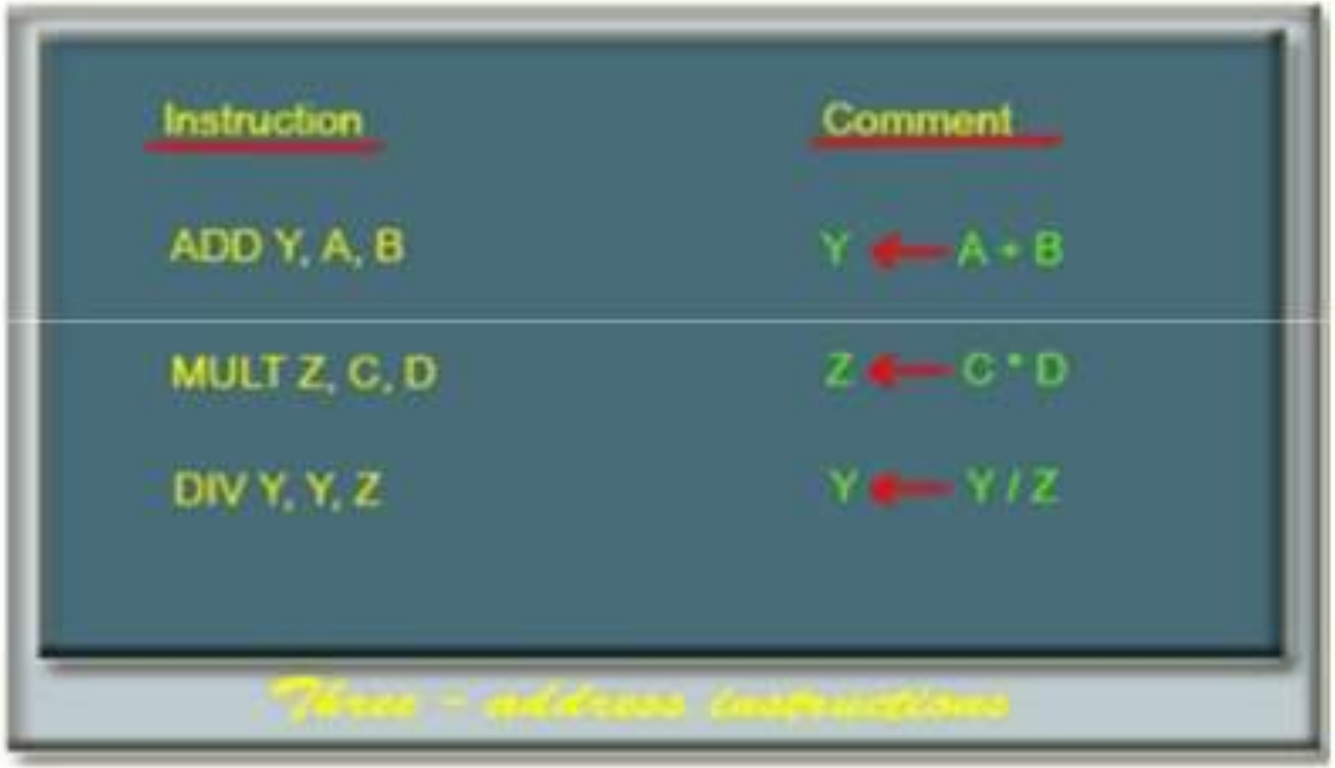
   Tells the CPU from where to fetch the next instruction after completion of execution of current instruction.

Representation of Instruction

➢Instruction Types According to Number of Addresses

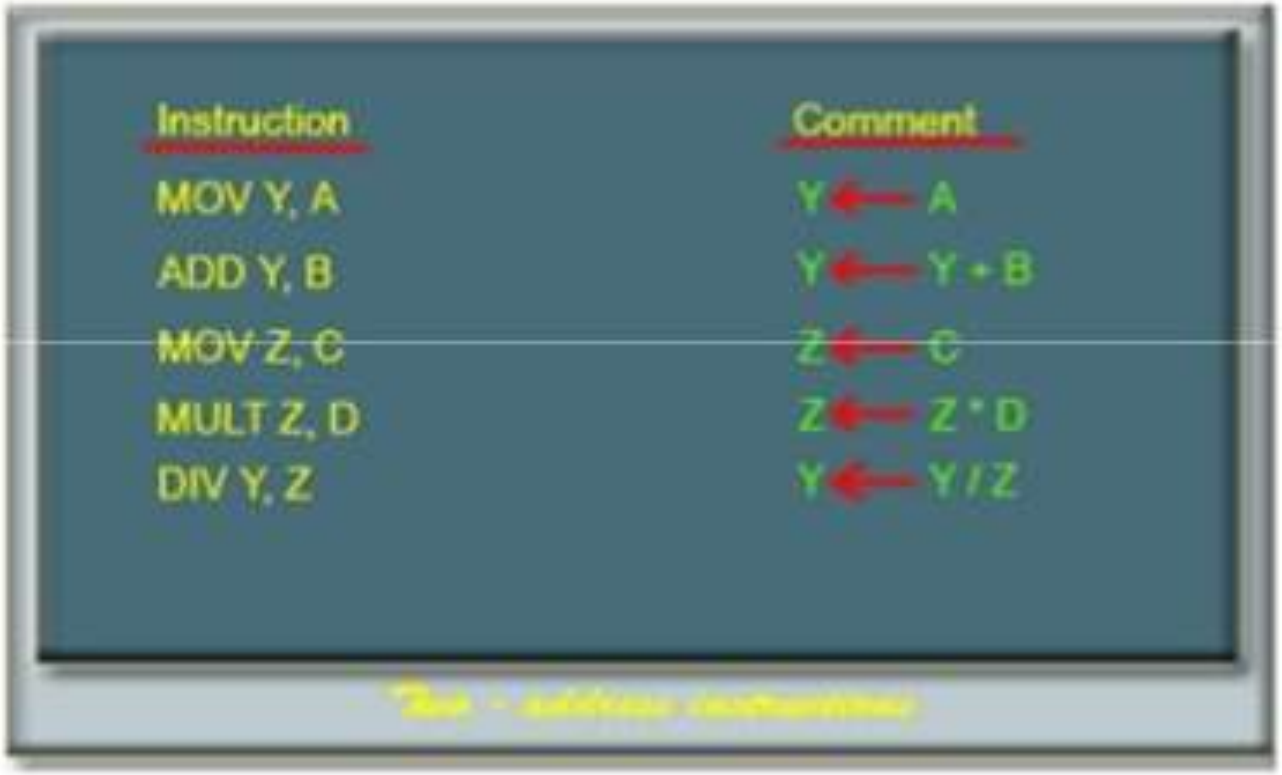


Three Address Instruction



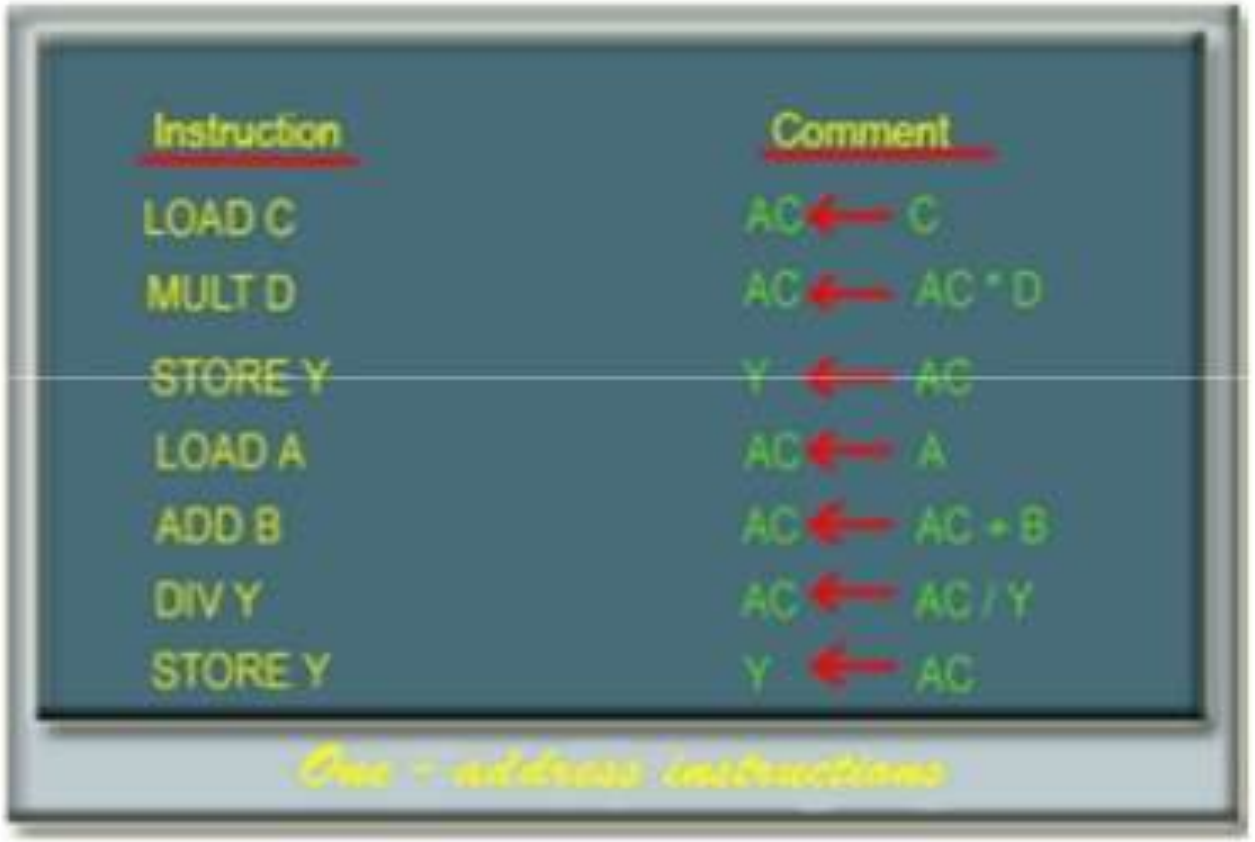
Three address instructions

Two Address Instruction



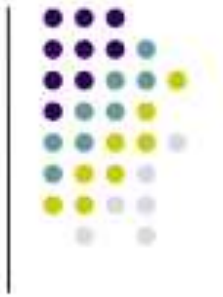
Two address instructions

One Address Instruction



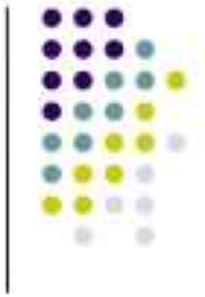
One address instructions

Zero Address Instruction



- The location of the operands are defined implicitly
- For implicit reference, a processor register is used and it is termed as accumulator(AC).
- E.g. CMA //complements the content of accumulator
- i.e. AC← $\overline{AC}$

➢Instruction Format Design Issues:



- An instruction consists of **an opcode and one or more operands**, implicitly or explicitly.
- Each explicit operand is referenced using one of **the addressing mode** that is available for that machine.
- **An instruction format is used to define the layout of the bits allocated to these elements of instructions.**
- Some of issues effecting instruction design are:
  1. **Instruction Length**
  2. **Allocation of bits for different fields in an instruction**
  3. **Variable length instruction**

1. Instruction Length

- A longer instruction means more time in fetching an instruction.
- For e.g. an instruction of length 32 bit on a machine with word size of 16 bit will need two memory fetch to bring the instruction.
- Programmer desires:
  - **More opcode and operands** in a instruction as it will reduce the program length.
  - **More addressing mode** for greater flexibility in accessing various types of data.

2. Allocation of Bits

- More opcodes obviously mean more bits in the opcode field.
- Factors which are considered for selection of addressing bits are:

A. Number of Addressing modes:

- More addressing modes, more bits will be needed.

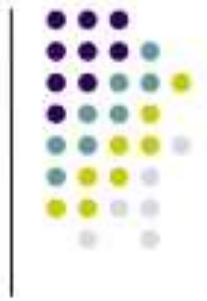
B. Number of Operands:

- More operands – more number of bits needed

C. Register versus memory:

- If more and more registers can be used for operand reference then the fewer bits are needed
- As number of register are far less than memory size.

2. Allocation of Bits



D. Number of Register Sets:

- Assume that A machine has 16 general purpose registers, a register address require 4 bits.
- However if these 16 registers are divided into two groups, then one of the 8 register of a group will need 3 bits for register addressing.

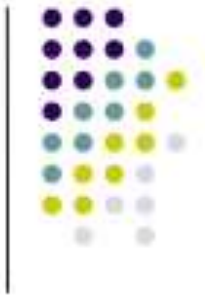
E. Address Range:

- The range of addresses that can be referenced is related to the number of address bits.
- With displacement addressing, the range is opened up to the length of the address register.

F. Address Granularity:

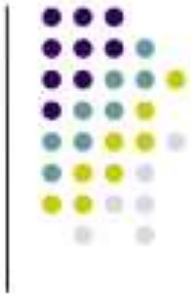
- In a system with 16- or 32-bit words, an address can reference a word or a byte at the designer's choice.

3. Variable length Instruction



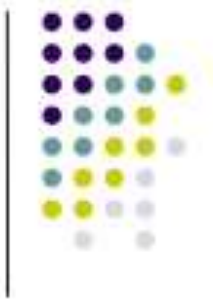
- Instead of looking for fixed length instruction format, designer may choose to provide a variety of instructions formats of different lengths.
- Addressing can be more flexible, with various combinations of register and memory references plus addressing modes.
- **Disadvantage:** an increase in the complexity of the CPU.

Concept of Program Execution



- The instructions constituting a program to be executed by a computer are loaded in sequential locations in its main memory.
- Processor fetches one instruction at a time and perform the operation specified.
- Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered.
- Processor keeps track of the address of the memory location containing the next instruction to be fetched using Program Counter (PC).
- Instruction Register (IR)

Executing an Instruction



1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are loaded into the IR (fetch phase).

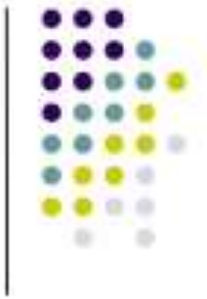
   IR ← [[PC]]

2. Assuming that the memory is byte addressable, increment the contents of the PC by 4 (fetch phase).

   PC ← [PC] + 4

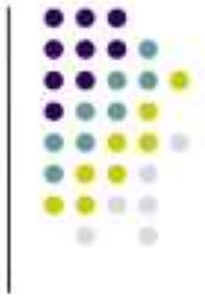
3. Carry out the actions specified by the instruction in the IR (execution phase).

Addressing Modes



- The term addressing mode refers to the mechanism employed for **specifying operands**.
- An operand can be specified as **part of the instruction or reference of the memory locations** can be given.
- An operand could also be **an address of CPU register**.
- The most common addressing techniques are:
  - **Immediate**
  - **Direct**
  - **Indirect**
  - **Register**
  - **Register Indirect**
  - **Displacement**
  - **Stack**

Addressing Modes



To explain the addressing modes, we use the following notation:

A=contents of an address field in the instruction that refers to a memory

R=contents of an address field in the instruction that refers to a register

EA=actual (effective) address of the location containing the referenced operand

(X)=contents of memory location X or register X

1. Immediate Addressing:

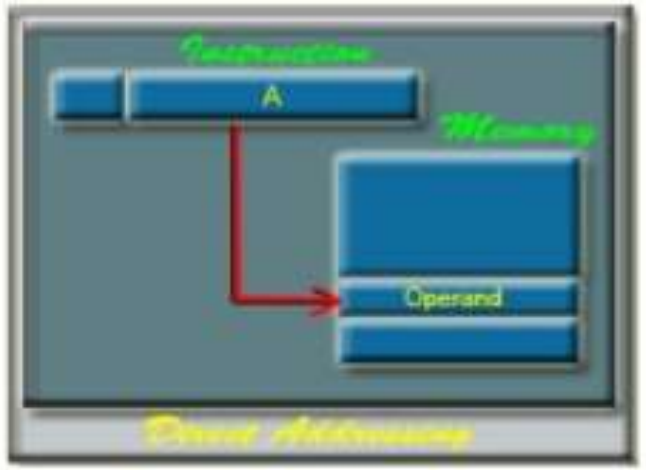
- The operand is actually present in the instruction
- OPERAND = A

- This mode can be used to define and use constants or set initial values of variables.
- The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand.
- e.g. **MOVE R0,300** — Immediate addressing

2. Direct Addressing

- The address field contains the effective address of the operand: EA= A
- It requires only one memory reference and no special calculation.
- Here, 'A' indicates the memory address field for the operand.
- e.g. MOVE R1, 1001

Direct addressing



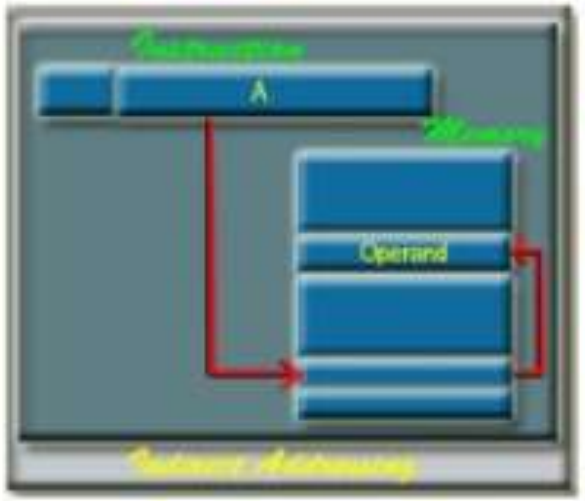
3. Indirect Addressing

- The effective address of the operand is stored in the memory and the instruction contains **the address of the memory containing the address of the data**.
- This is know as indirect addressing:

  EA = (A)

- Here 'A' indicates the memory address field of the required Operands.
- E.g. MOVE R0,(1000)

Indirect addressing



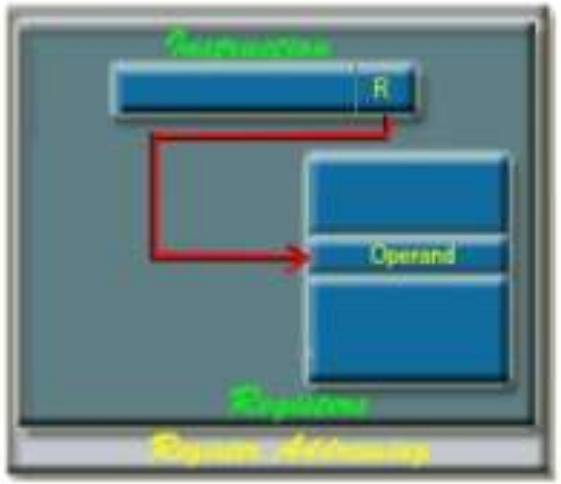
4. Register Addressing

- The instruction specifies the address of the register containing the operand.
- The instruction contains the name of the a CPU register.

  EA =R ← indicates a register where the operand is present.

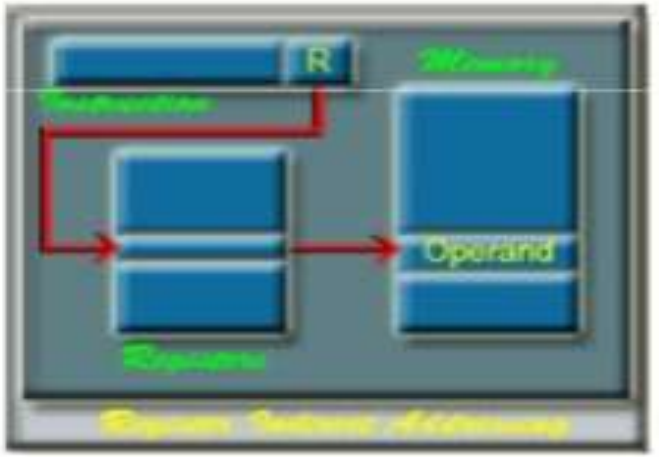
- E.g. MOVE R1, 1010

Register addressing



5. Register Indirect Addressing

- The effective address of the operand is stored in a register and instruction contains **the address of the register containing the address of the data.**
- $EA = (R)$
- Here 'R' indicates the memory address field of the required Operands.
- E.g. MOVE R0,(R1)



Register addressing

Register indirect addressing

6. Displacement Addressing

- A combination of both direct addressing and register indirect addressing modes.

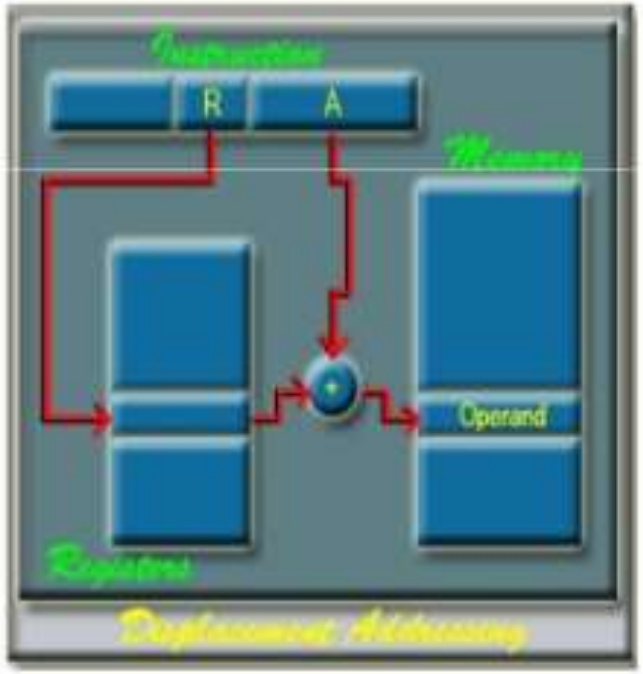
$$EA = A + (R)$$

- The value contained in one address field (value = A) is used directly. The other address field refers to a register whose contents are added to A to produce the effective address.

6. Displacement Addressing

Three of the most common use of displacement addressing are:

- ➔ **Relative addressing**
- ➔ **Base-register addressing**
- ➔ **Indexing**



Relative addressing

- For relative addressing, the implicitly referenced register is the program counter (PC).
- The current instruction address is added to the address field to produce the EA.
- Thus, the effective address is a displacement relative to the address of the instruction.
- e.g. 1001 JC X1

1050 X1: ADD R1,5

- X1=address of the target instruction-address of the current instruction
  =1050-1001=49

Base Register Addressing

- The **base register(reference register) contains a memory address**, and the **address field contains a displacement** from that base address specified by the base register.

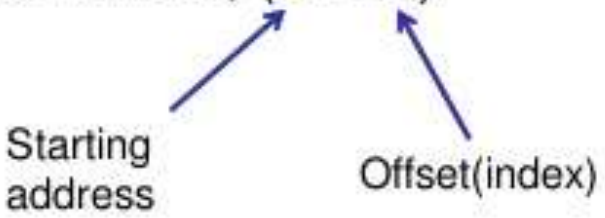
$$EA=A+(B)$$

Indexing or Indexed Addressing

- Used to access **elements an array** which are stored in consecutive location of memory.

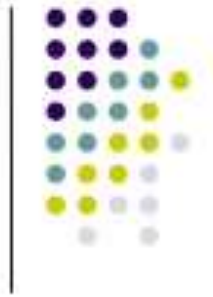
  EA = A+(R)

- Address field **A gives main memory address** and **R contains positive displacement** with respect to base address.
- The displacement can be specified either **directly** in the instruction or through another **registers**.
- E.g. MOVE R1, (BR+5)



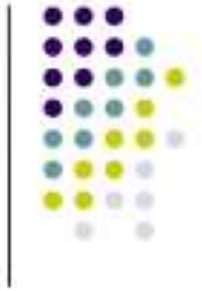
MOVE R0,(BR+R1)

Auto Indexing



- Generally **index register are used for iterative tasks**, it is typical that there is a need to increment or decrement the index register after each reference to it.
- Because this is such a common operation, some system will automatically do this as part of the same instruction cycle.
- This is known as **auto-indexing.**
- Two types of auto-indexing
  1. **auto-incrementing**
  2. **auto-decrementing.**

a. Auto Increment Mode



- If register R contains the address of the operand
- After accessing the operand, the contents of register R is incremented to point to the next item in the list.
- Auto-indexing using increment can be depicted as follows:

  EA = A + (R) or EA = (R)+

  (R) = (R) + 1
- E.g. MOVE R1,1010 /*starting Memory location 1010 is stored in R1*/
- ADD AC,(R1)+ /*contents of 1010 ML are added to AC and the contents of R1 is incremented by 1*/

b. Auto Decrement Mode

- The contents of register specified in the instruction are decremented and these contents are then used as the effective address of the operand.
- Auto-indexing using decrement can be depicted as follows:

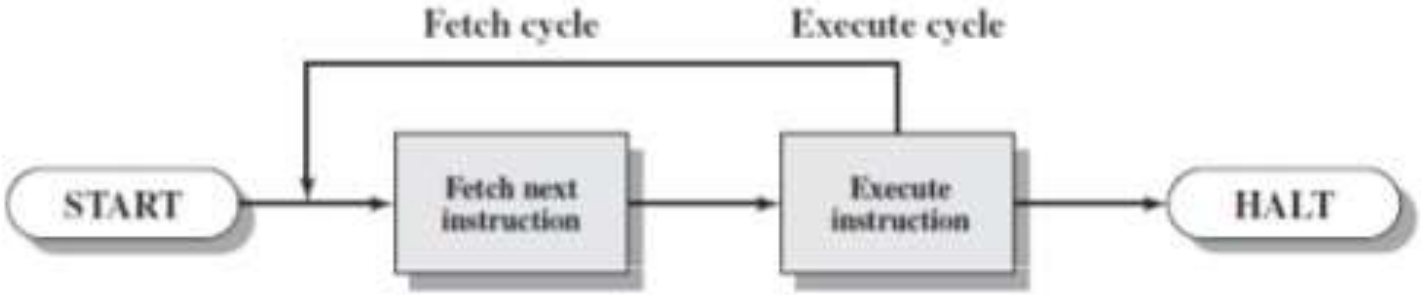
  $EA = A - (R)$ or $EA = -(R)$
  $(R) = (R) - 1$

- The contents of the register are to be decremented before used as the effective address.
- E.g. ADD R1,-(R2)

7. Stack Addressing

- A stack is a **linear array or list of locations**.
- Sometimes referred to as a **pushdown list** or **last-in-first-out queue**.
- Associated with the stack is a pointer whose value is the address of the **top of the stack**.
- The stack pointer is maintained in a register. Thus, references to stack locations in memory are in fact **register indirect addresses.**
- The stack mode of addressing is a form of implied addressing.
- E.g. **PUSH and POP**

Basic Instruction Cycle



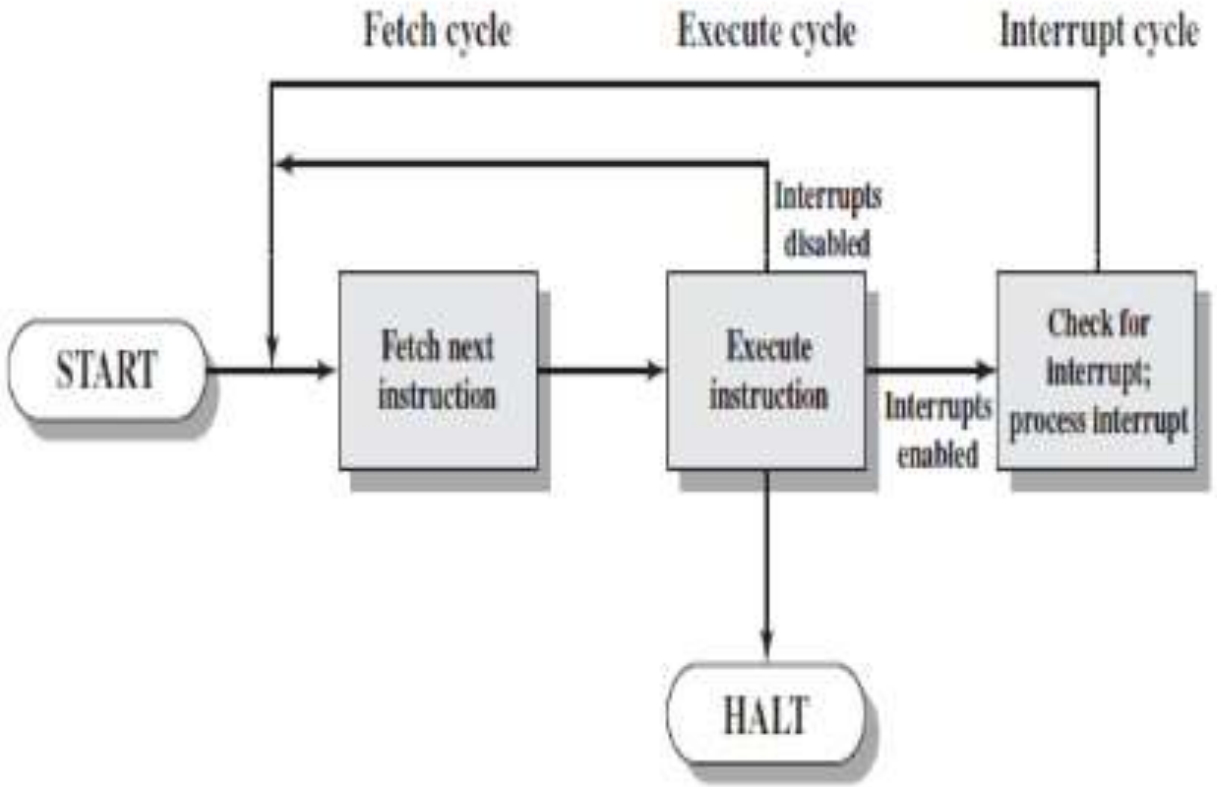
- **Fetch cycle** basically involves read the next instruction from the memory into the CPU and along with that update the contents of the program counter.
- In the **execution phase**, it interprets the opcode and perform the indicated operation.
- The instruction fetch and execution phase together known as **instruction cycle**.

Basic Instruction Cycle with Interrupt

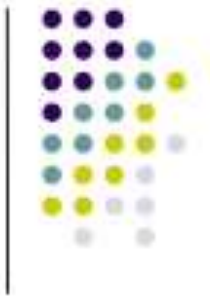
An instruction cycle includes the following sub cycles:

1. **Fetch:** Read the next instruction from memory into the processor.
2. **Execute:** Interpret the opcode and perform the indicated operation.
3. **Interrupt:** If interrupts are enabled and an interrupt has occurred, save the current process state and service the interrupt.

Basic Instruction Cycle with Interrupt

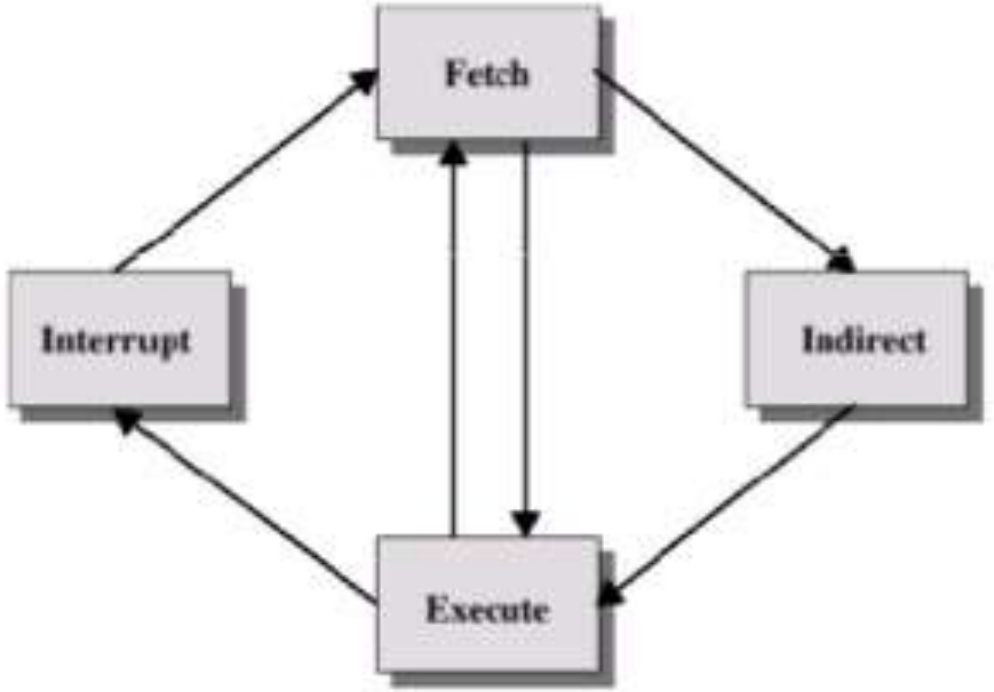


The Indirect Cycle

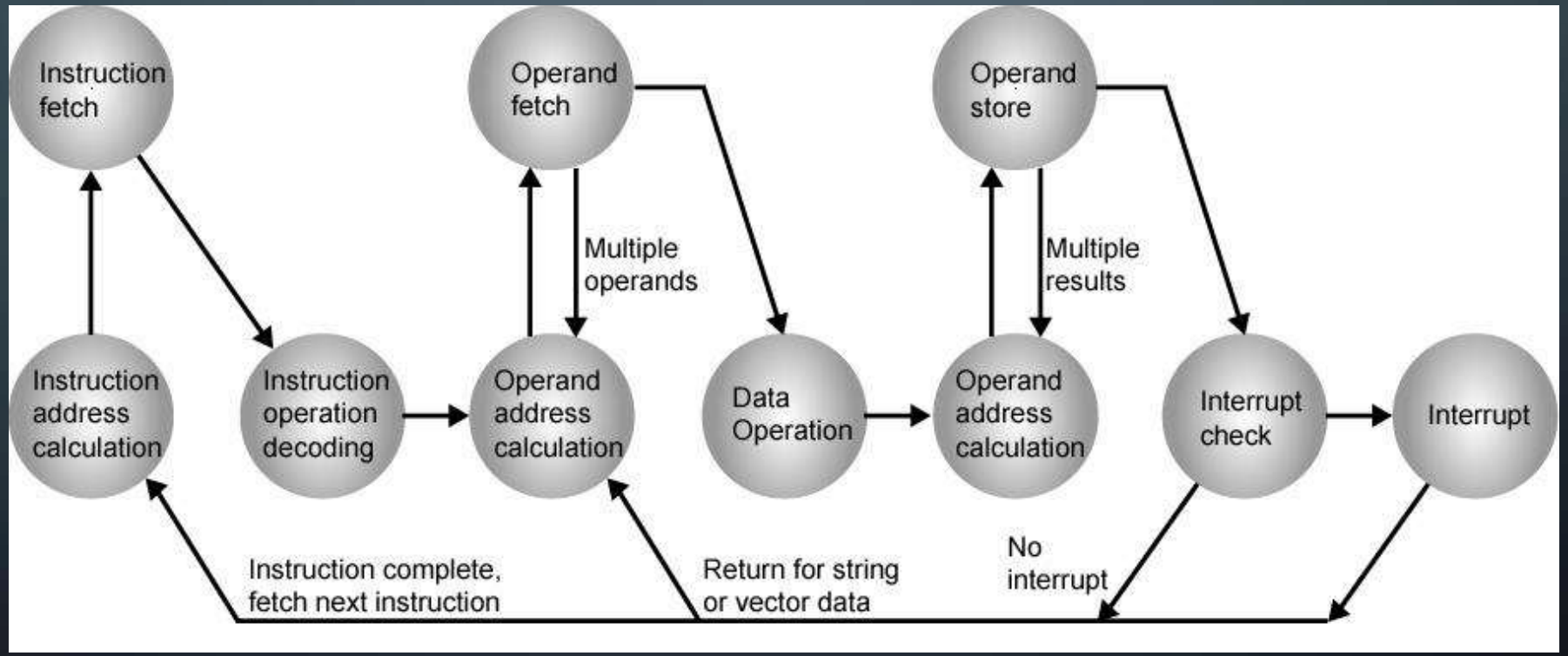


- The execution of an instruction may involve one or more operands in memory, each of which requires a memory access.
- Further, if indirect addressing is used, then additional memory accesses are required.
- For fetching the indirect addresses as one more instructions subcycle are required.
- After an instruction is fetched, it is examined to determine if any indirect addressing is involved. If so, the required operands are fetched using indirect addressing.

The Indirect Cycle

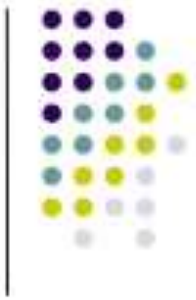


Instruction Cycle State Diagram



- **Instruction address calculation (iac):** Determine the address of the next instruction to be executed. Usually, this involves adding a fixed number to the address of the previous instruction.
- **Instruction fetch (if):** Read instruction from its memory location into the processor.
- **Instruction operation decoding (iod):** Analyze instruction to determine type of operation to be performed and operand(s) to be used.
- **Operand address calculation (oac):** If the operation involves reference to an operand in memory or available via I/O, then determine the address of the operand.
- **Operand fetch (of):** Fetch the operand from memory or read it in from I/O.
- **Data operation (do):** Perform the operation indicated in the instruction.
- **Operand store (os):** Write the result into memory or out to I/O.

Instruction Interpretation and Sequencing

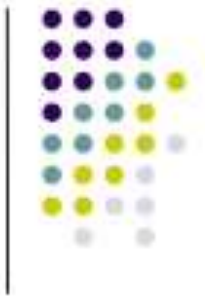


- Every processor has some basic type of instructions like data transfer instruction, arithmetic and logical instruction, branch instruction and so on.
- To perform a particular task on the computer it is programmers job **to select and write appropriate instructions one after the other**. This job of programmer is known as **instruction sequencing**.
- Two types:
  1. Straight line sequencing
  2. Branch instructon

1.Straight line sequencing

- Processor executes a program with the help of **Program Counter(PC)** which holds the address of the next instruction to be executed.
- To begin execution of a program, **the address of the its first instruction is placed into the PC**.
- The processor control circuit **fetches instruction from the memory address specified by the PC and executes instruction, one at a time**.
- At the same time **the content of PC is incremented** so as to point to the address of next instruction.
- This is called as **straight line sequencing**.

2. Branch Instruction



- After executing decision making instruction, processor have to follow one of the two program sequence.
- Branch instruction transfer the program control from one straight line sequence to another straight line sequence instruction.
- In branch instruction, the new address called **target address or branch target** is loaded into PC and instruction is fetched from the new address.

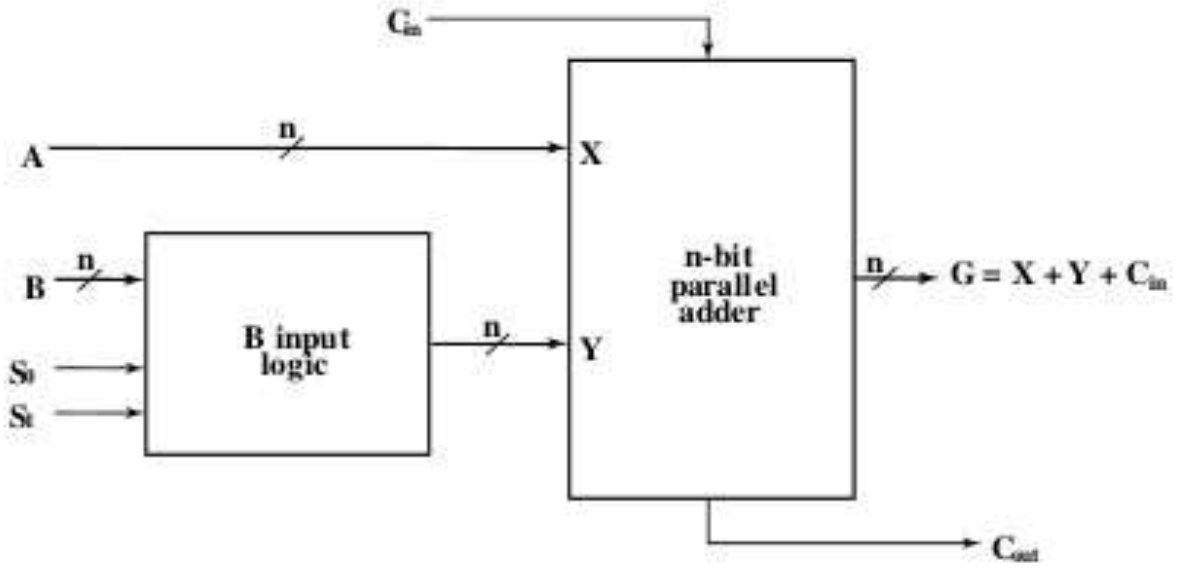
Arithmetic Logic Unit (ALU)

- In this and the next section, we deal with detailed design of typical ALUs and shifters
- Decompose the ALU into:
  - An arithmetic circuit
  - A logic circuit
  - A selector to pick between the two circuits
- Arithmetic circuit design
  - Decompose the arithmetic circuit into:
    - An n-bit parallel adder
    - A block of logic that selects four choices for the B input to the adder

Arithmetic Circuit Design

- There are only four functions of B to select as Y in G = A + Y:

	$C_{in} = 0$	Operation	$C_{in} = 1$	Operation
• All 0's	G = A	Transfer of A	G = A + 1	Increment
• B	G = A + B	Addition	G = A + B + 1	
• $\overline{B}$	G = A + $\overline{B}$	Subtraction 1C	G = A + $\overline{B}$ + 1	Subtraction 2C
• All 1's	G = A – 1	Decrement	G = A	Transfer of A

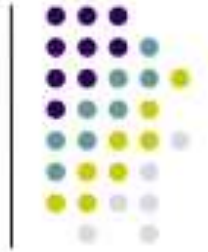


Logic Circuit

- The text gives a circuit implemented using a multiplexer plus gates implementing: AND, OR, XOR and NOT
- Here we custom design a circuit for bit G_i by beginning with a truth table organized as logic operation K-map and assigning (S1, S0) codes to AND, OR, etc.
- $G_i = S_0 \overline{A_i} B_i + \overline{S_1} A_i B_i + S_0 A_i \overline{B_i} + S_1 \overline{S_0} \overline{A_i}$
- Custom design better

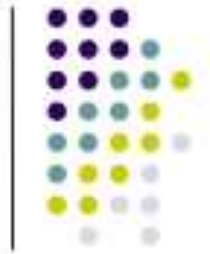
S_1S_0 / A_iB_i	AND 0 0	OR 0 1	XOR 1 1	NOT 1 0
0 0	0	0	0	1
0 1	0	1	1	1
1 1	1	1	0	0
1 0	0	1	1	0

What is Pipelining?



- A technique used in advanced microprocessors where the microprocessor begins executing a second instruction before the first has been completed.
- A Pipeline is a series of stages, where some work is done at each stage. The work is not finished until it has passed through all stages.
- With pipelining, the computer architecture allows the next instructions to be fetched while the processor is performing arithmetic operations, holding them in a buffer close to the processor until each instruction operation can performed.

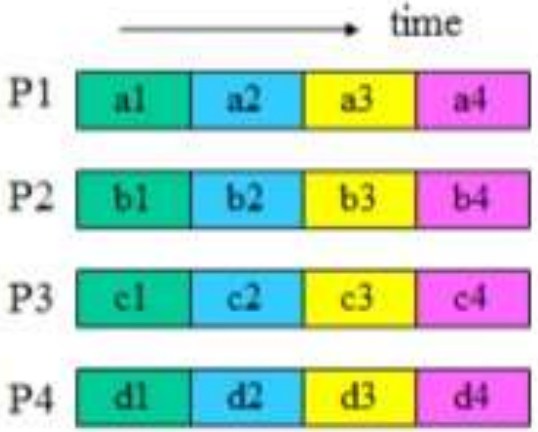
How Pipeline Works?



- The pipeline is divided into segments and each segment can execute its operation concurrently with the other segments. Once a segment completes an operation, it passes the result to the next segment in the pipeline and fetches the next operations from the preceding segment.

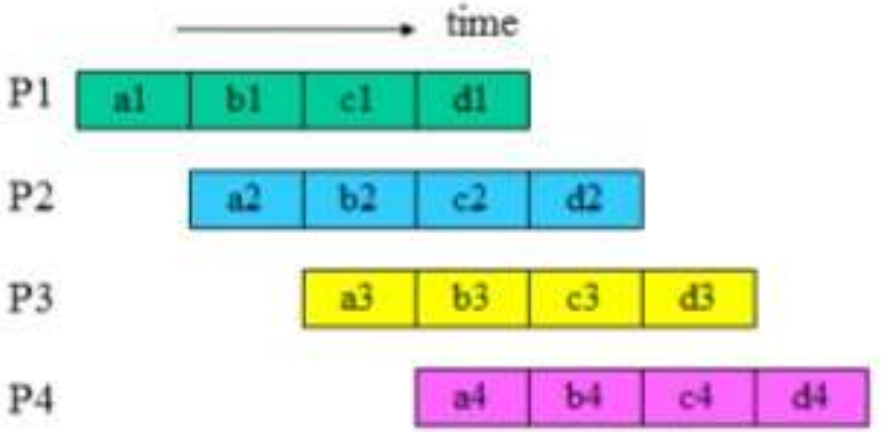
Basic Ideas

- Parallel processing



Less inter-processor communication
Complicated processor hardware

- Pipelined processing

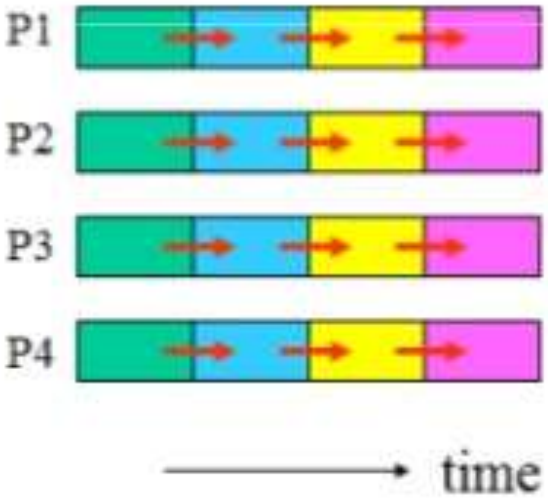


More inter-processor communication
Simpler processor hardware

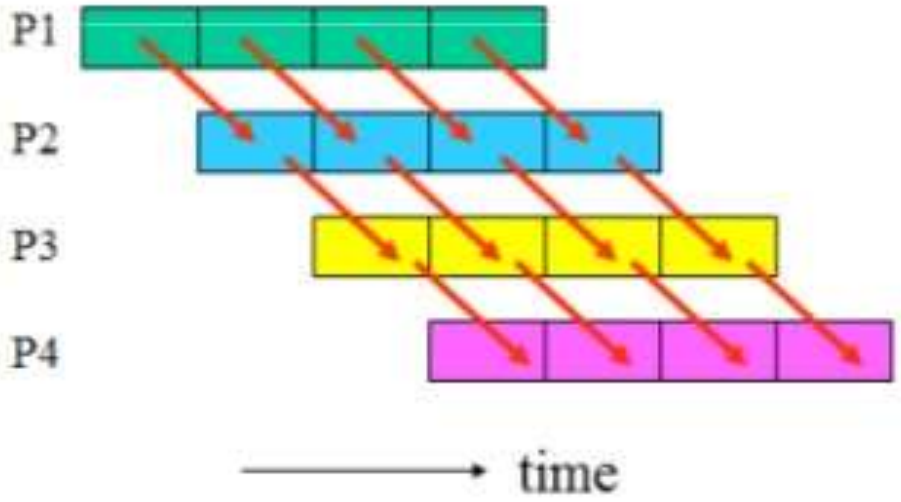
Colors: different types of operations performed
a, b, c, d: different data streams processed

Data Dependence

- Parallel processing requires NO data dependence between processors



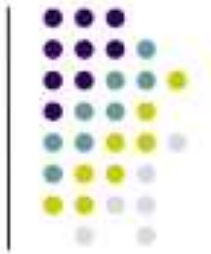
- Pipelined processing will involve inter-processor communication



Instruction Pipelining

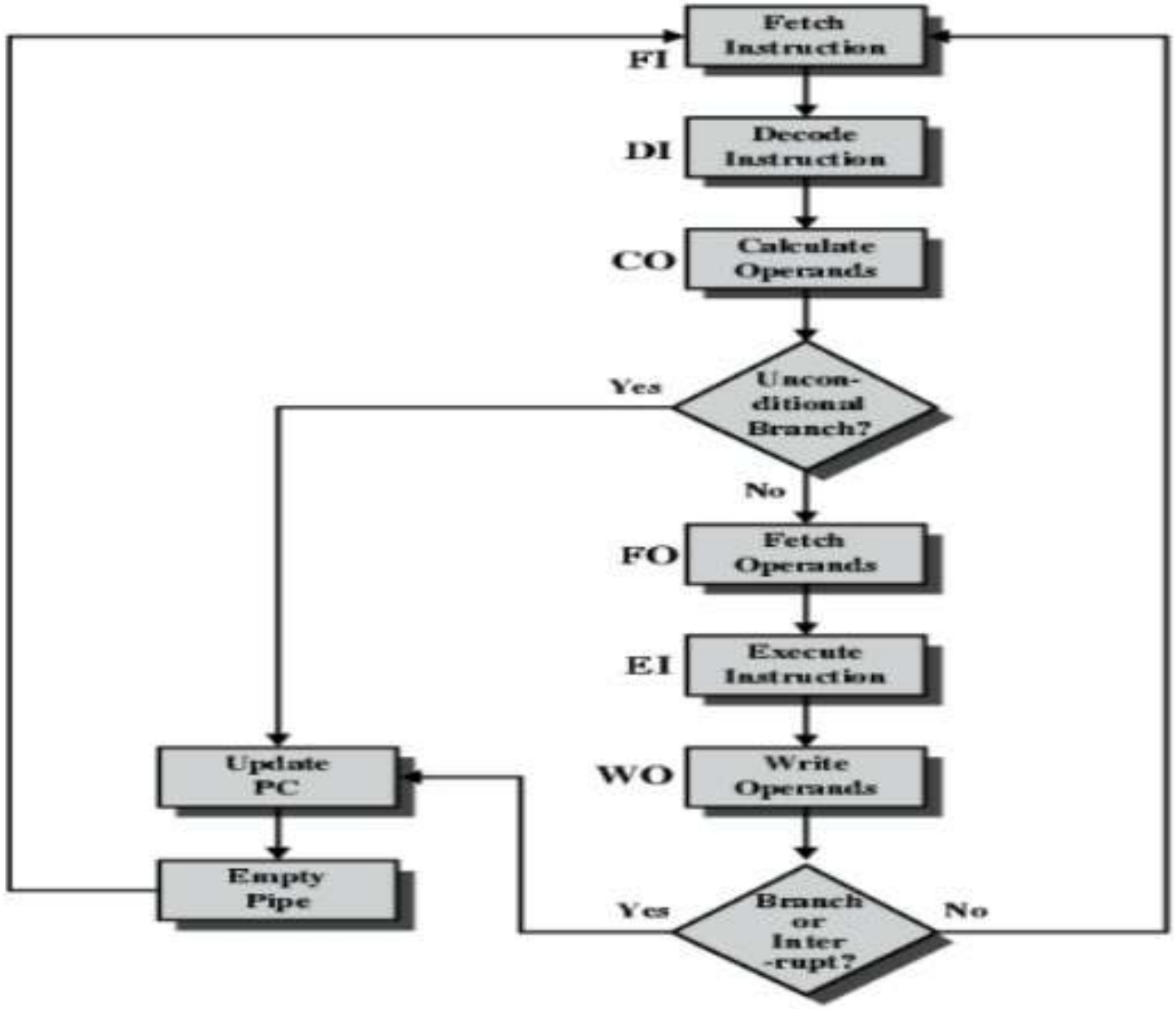
- Pipeline can also occur in instruction stream as with data stream
- Consecutive instructions are read from memory while previous instructions are executed in various pipeline stages.
- Difficulties
  - Different execution times for different pipeline stages
  - Some instructions may skip some of the stages. E.g. No need of effective address calculation in immediate or register addressing
  - Two or more stages require access of memory at same time. E.g. instruction fetch and operand fetch at same time

Pipeline Stages



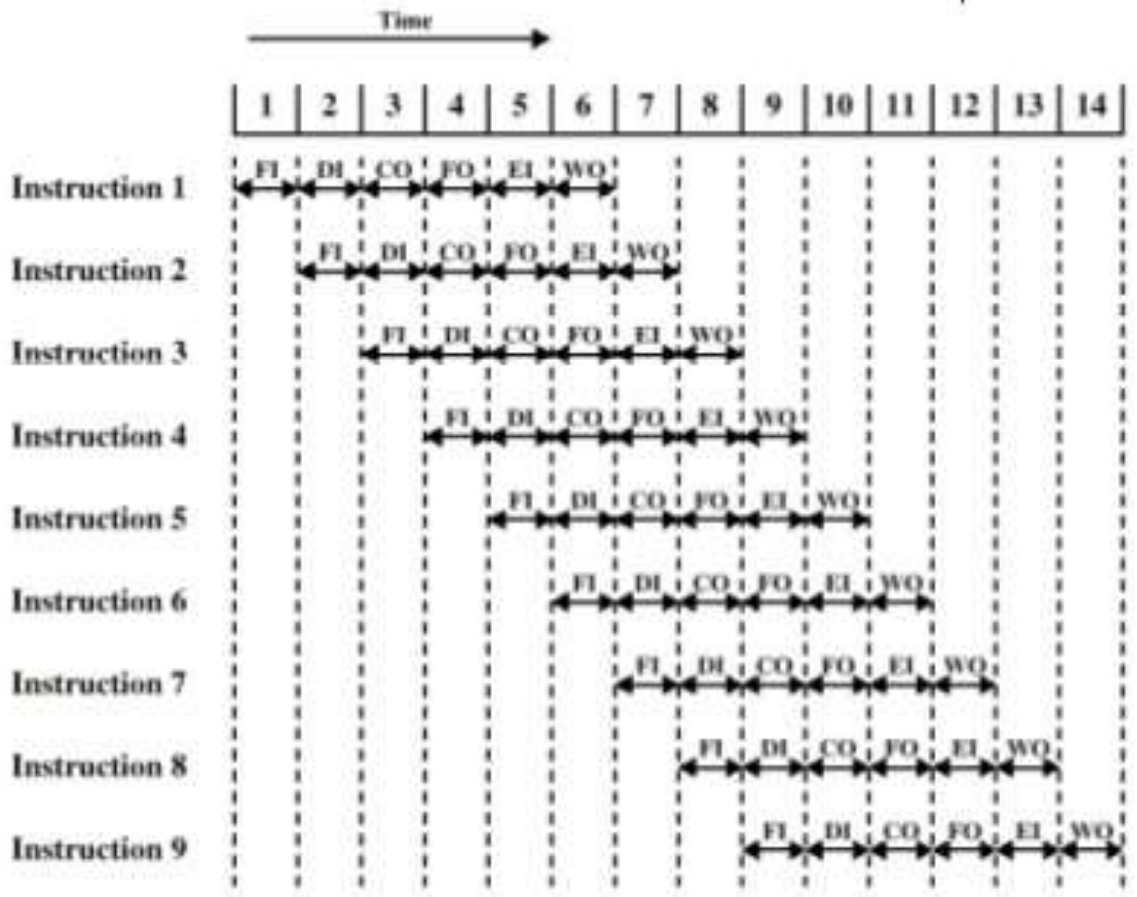
- Consider the following decomposition for processing the instructions
  - Fetch instruction (FI) – Read into a buffer
  - Decode instruction (DI) – Determine opcode, operands
  - Calculate operands (CO) – Indirect, Register indirect, etc.
  - Fetch operands (FO) – Fetch operands from memory
  - Execute instructions (EI)- Execute
  - Write operand (WO) – Store result if applicable
- Overlap these operations to make a 6 stage pipeline

Six-stage CPU instruction pipeline

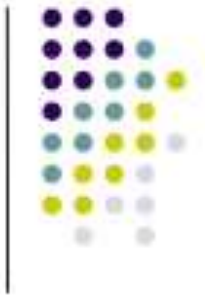


Timing of Instruction Pipeline - six stages

Reduction in instruction execution time from 54 time units to 14 time units



Pipeline Hazards



- **Structural Hazard or Resource Conflict** – Two instructions need to access the same resource at same time.
- **Data Hazard or Data Dependency** – An instruction uses the result of the previous instruction before it is ready.
  - A hazard occurs exactly when an instruction tries to read a register in its ID stage that an earlier instruction intends to write in its WO stage.
- **Control Hazard or Branch Difficulty** – The location of an instruction depends on previous instruction
  - Conditional branches break the pipeline
    - Stuff we fetched in advance is useless if we take the branch
- Pipeline implementation need to detect and resolve hazards.

•Structural Hazard (Resource conflict):

A resource hazard occurs when 2 or more instructions that are already in the pipeline need the same resources.

eg: x=y+z

The instructions required to perform this operation are:

1. R1←y
2. R2←R1+z
3. X←R2

(Diagram - self)

Data Hazard :

A data hazard occurs when an instruction depends upon the result from a previous instruction, but this result is not yet avail.

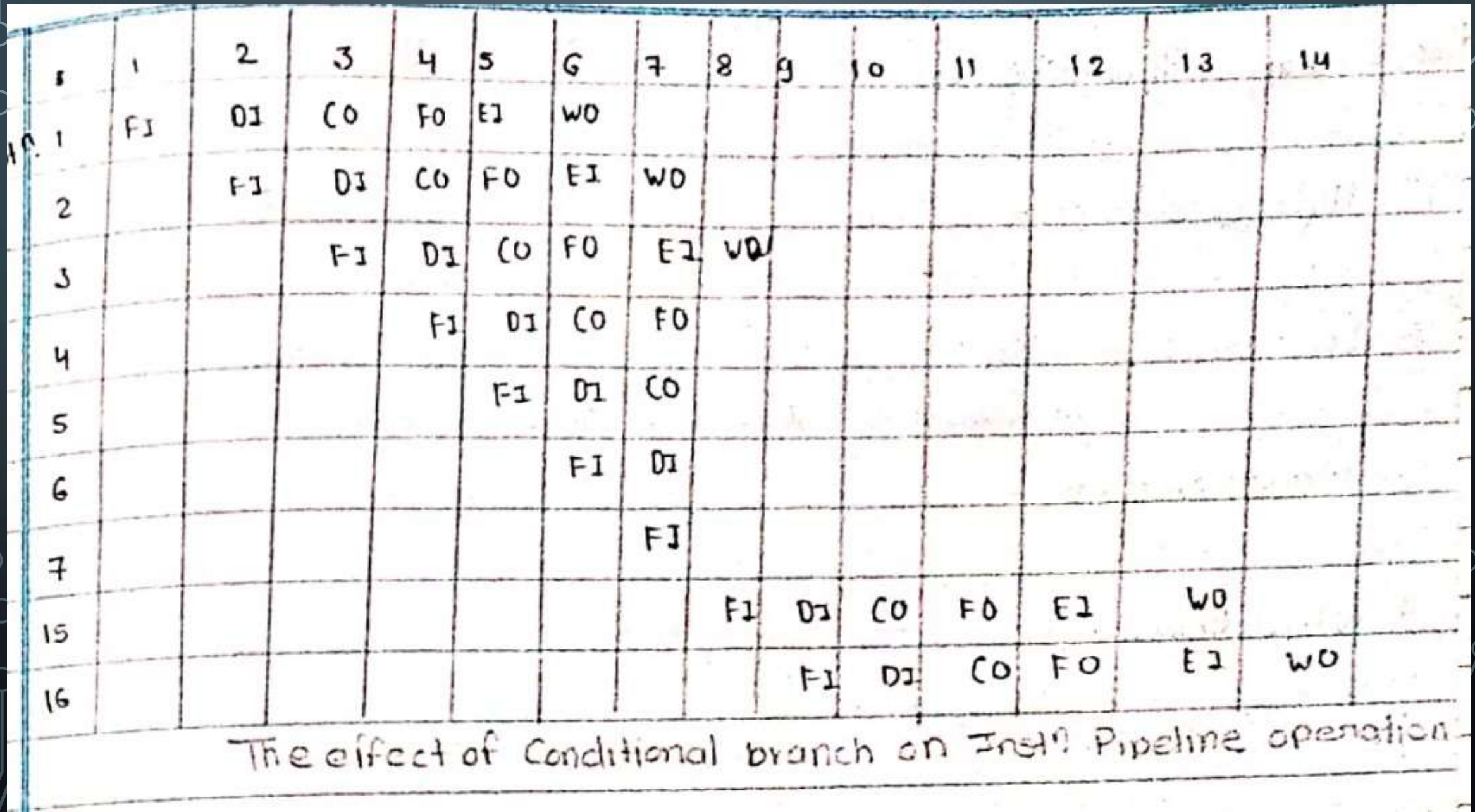
eg: ADD AX,BX

SUB CX,AX

(Diagram - self)

- 3 Types of Data hazard:
  1. RAW
  2. WAR
  3. WAW

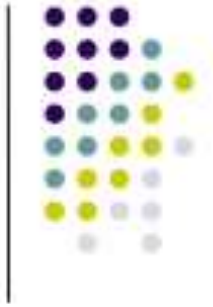
- Control (Branch) Hazard :



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
1	FI	DI	CO	FO	EI	WO									
2		FI	DI	CO	FO	EI	WO								
3			FI	DI	CO	FO	EI	WO							
4				FI	DI	CO	FO								
5					FI	DI	CO								
6						FI	DI								
7							FI								
15								FI	DI	CO	FO	EI	WO		
16									FI	DI	CO	FO	EI	WO	

The effect of Conditional branch on Instn Pipeline operation

Delayed Branch



- Delayed Branch – used with RISC machines
  - Requires some clever rearrangement of instructions
  - Burden on programmers but can increase performance
  - Most RISC machines: Doesn't flush the pipeline in case of a branch
  - Called the Delayed Branch
    - This means if we take a branch, we'll still continue to execute whatever is currently in the pipeline, at a minimum the next instruction
    - Benefit: Simplifies the hardware quite a bit
    - But we need to make sure it is safe to execute the remaining instructions in the pipeline
    - Simple solution to get same behavior as a flushed pipeline: Insert NOP – No Operation – instructions after a branch
      - Called the Delay Slot

Delayed Branch (cont.)

- **Normal vs Delayed Branch**

Address	Normal	Delayed
100	LOAD X,A	LOAD X,A
101	ADD 1,A	ADD 1,A
102	JUMP 105	JUMP 106
103	ADD A,B	NOOP
104	SUB C,B	ADD A,B
105	STORE A,Z	SUB C,B
106		STORE A,Z

- One delay slot - Next instruction is always in the pipeline. "Normal" path contains an implicit "NOP" instruction as the pipeline gets flushed. Delayed branch requires explicit NOP instruction placed in the code!

Delayed Branch (cont.)

- **Optimized Delayed Branch**
  - But we can optimize this code by rearrangement! Notice we always Add 1 to A so we can use this instruction to fill the delay slot

Address	Normal	Delayed	Optimized
100	LOAD X,A	LOAD X,A	LOAD X,A
101	ADD 1,A	ADD 1,A	JUMP 105
102	JUMP 105	JUMP 106	ADD 1,A
103	ADD A,B	NOOP	ADD A,B
104	SUB C,B	ADD A,B	SUB C,B
105	STORE A,Z	SUB C,B	STORE A,Z
106		STORE A,Z	

Branch Prediction

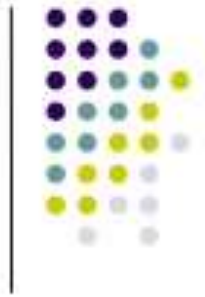
- Predict never taken
  - Assume that jump will not happen
  - Always fetch next instruction

- Predict always taken
  - Assume that jump will happen
  - Always fetch target instruction
  - Studies indicate branches are taken around 60% of the time in most programs

Branch Prediction (cont.)

- Predict by Opcode
  - Some types of branch instructions are more likely to result in a jump than others (e.g. LOOP vs. JUMP)
  - Can get up to 75% success
- Taken/Not taken switch – 1 bit branch predictor
  - Based on previous history
    - If a branch was taken last time, predict it will be taken again
    - If a branch was not taken last time, predict it will not be taken again
  - Good for loops
  - Could use a single bit to indicate history of the previous result
  - Need to somehow store this bit with each branch instruction
  - Could use more bits to remember a more elaborate history

Performance Measures



- The most important measure of the performance of a computer is how quickly it can execute program.
- The performance of a computer is affected by the design of its hardware, the complier and its machine language instructions, instruction set, implementation language etc.
- The computer user is always interested in reducing the execution time.
- The execution time is also referred as response time.
- Reduction in response time increases the throughput.
- Throughput: the total amount of work done in a given time.

Performance Measures

1. Clock Speed/Rate
2. MIPS: Million instruction per second
3. Speedup
4. Efficiency
5. Throughput Rate
6. Amdahl's Law

1. The System Clock

- Processor circuits are controlled by a timing signal called, a clock.
- The clock defines regular time intervals, called clock cycles.
- To execute a machine instruction, the processor divides the action to be performed into a sequence of basic steps, such that each step can be completed in one clock cycles.
- The constant cycle time(in nanoseconds) is denoted by t.
- The clock rate is given by f=1/t which is measured in cycle per second(CPS).

Therefore, T=IC*CPI*t

Where T= Total time req by the prog,

IC=No. of instructions

CPI= Cycles per instruction

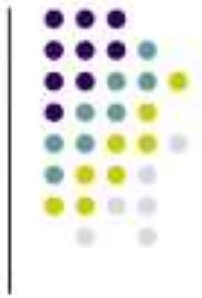
t= Cycle time in ns

2. Instruction Execution Rate

- **MIPS Rate:**
  - The rate at which instructions are executed, expressed as millions of instructions per second (MIPS).
  - MIPS rate in terms of the clock rate and CPI:

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

3. Speedup



- Increase in speed due to parallel system compared to uni-processor system

$$\textbf{Speedup } S(N) = \frac{\textbf{Serial Execution Time}}{\textbf{Parallel Execution Time}} = \frac{T(1)}{T(N)}$$

Where,
T(N) represents the execution time taken by program running on N processors.
and
T(1) represents time taken by best serial implementation of a program measured on one processor.

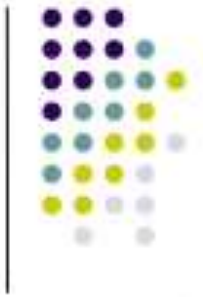
4. Efficiency

- The average contribution of the processors towards the global computation.
- It is defined as speedup divided by number of processors.

$$\textbf{Efficiency(N)} = \frac{\textbf{Speedup(N)}}{\textbf{N}}$$

Speedup(N) = speedup measured on N processors

5. Throughput ($_p$)



- Number of programs a system can execute per unit time

$$\omega_p = \frac{f}{I_c \times CPI}$$

$$\omega_p = \frac{\text{Number of programs}}{\text{Time in seconds}}$$

AMDAHL'S LAW

END