VIT-AP UNIVERSITY

Computer Architecture

ECE 2015

DR. M. S. ELLISON

ASSOCIATE PROFESSOR

SENSE

Contact Details

Email: ellison.mathe@vitap.ac.in

Phone: +91-9491902516

Cabin: CB-206

Course Pre-requisites

Digital logic design

Chap. 1: Digital Logic Circuits

- Logic Gates, • Boolean Algebra
- K-Map Simplification, • Combinational Circuits
- Flip-Flops, • Sequential Circuits

Chap. 2: Digital Components

- Integrated Circuits, • Decoders, • Multiplexers
- Registers, • Shift Registers, • Binary Counters
- Memory Unit

Chap. 3: Data Representation

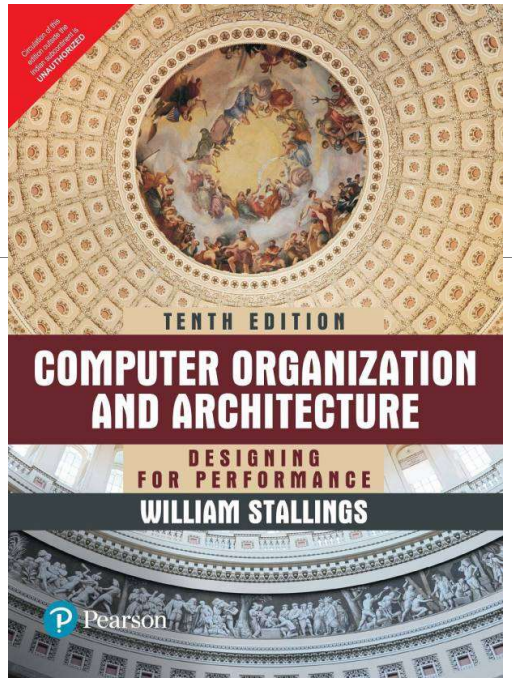
- Data Types • Complements • Fixed Point Representation
- Floating Point Representation
- Other Binary Codes

Please refer M. Morris Mano, Computer System Architecture, Pearson Education, Third Edition

Objectives:	1. To familiarize students about hardware design including logic design, basic structure and behavior of the various functional modules of the computer and how they interact to provide the **processing needs** of the user. 2. To obtain a working knowledge of **assembly language**
Expected Outcome:	On completion of the course, students will have the ability to 1. Understand the overview of basic computer architecture. 2. Learn basic computer logic: adders, multipliers, ALU, and memory. 3. Learn basic assembly language programming, basic Instruction Set Architecture (ISA), and the design of single cycle CPU 4. Understand Parallel processors, RISC and CISC architecture.
Mode of Evaluation	Practice/Quiz Tests-20%, Continuous Assessment Tests-60%, Practical Assesment-20% Quiz Test 20% Continuous Assessment Test-1 20% Continuous Assessment Test-2 20% Continuous Assessment Test-3 20% Practical Assessment (Mini Project) 20%
Open Hours	**Will be displayed**

Text Book:

William Stallings, Computer Organization and Architecture: Designing for Performance, Pearson Education, Tenth Edition,2013

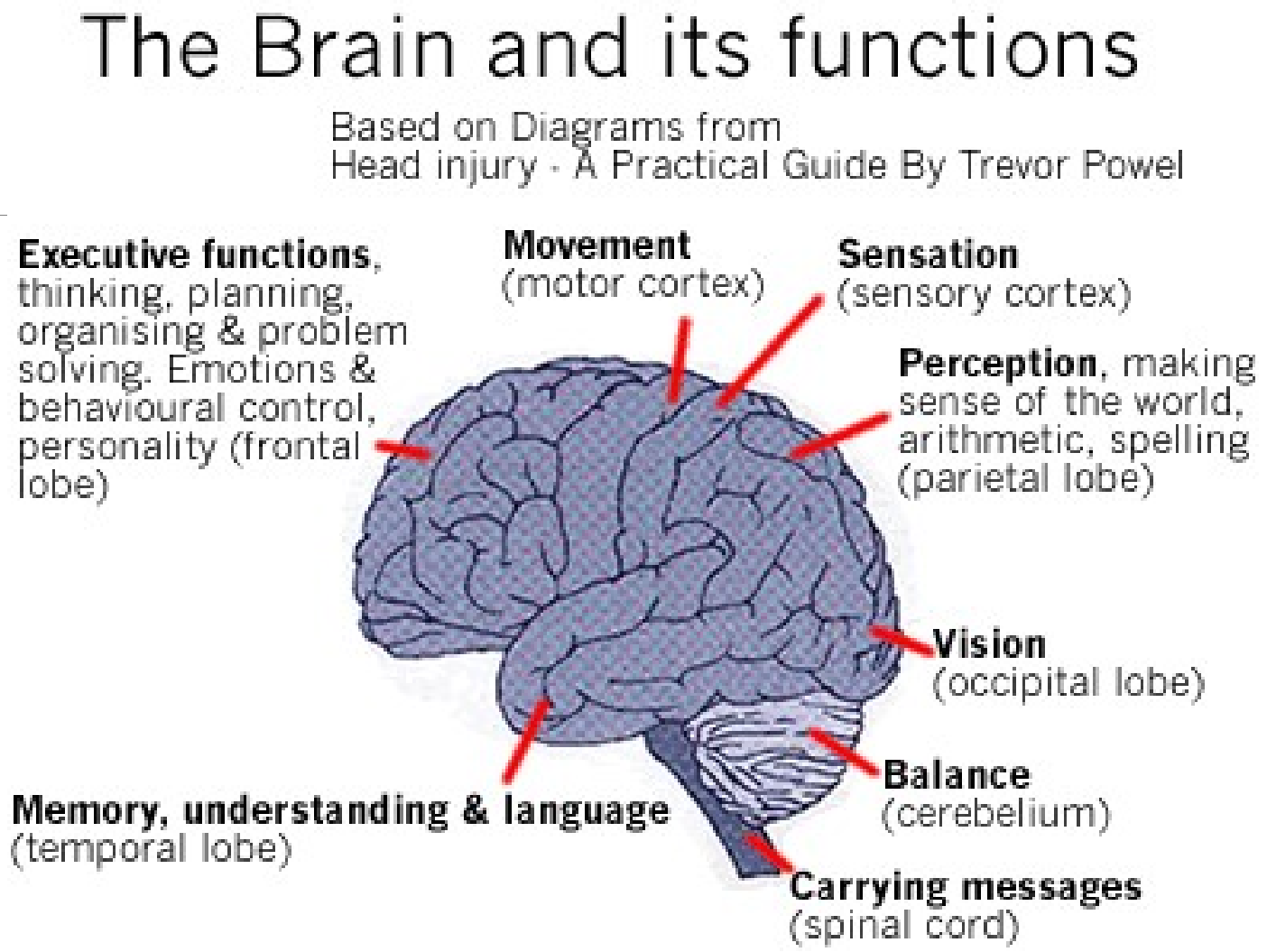


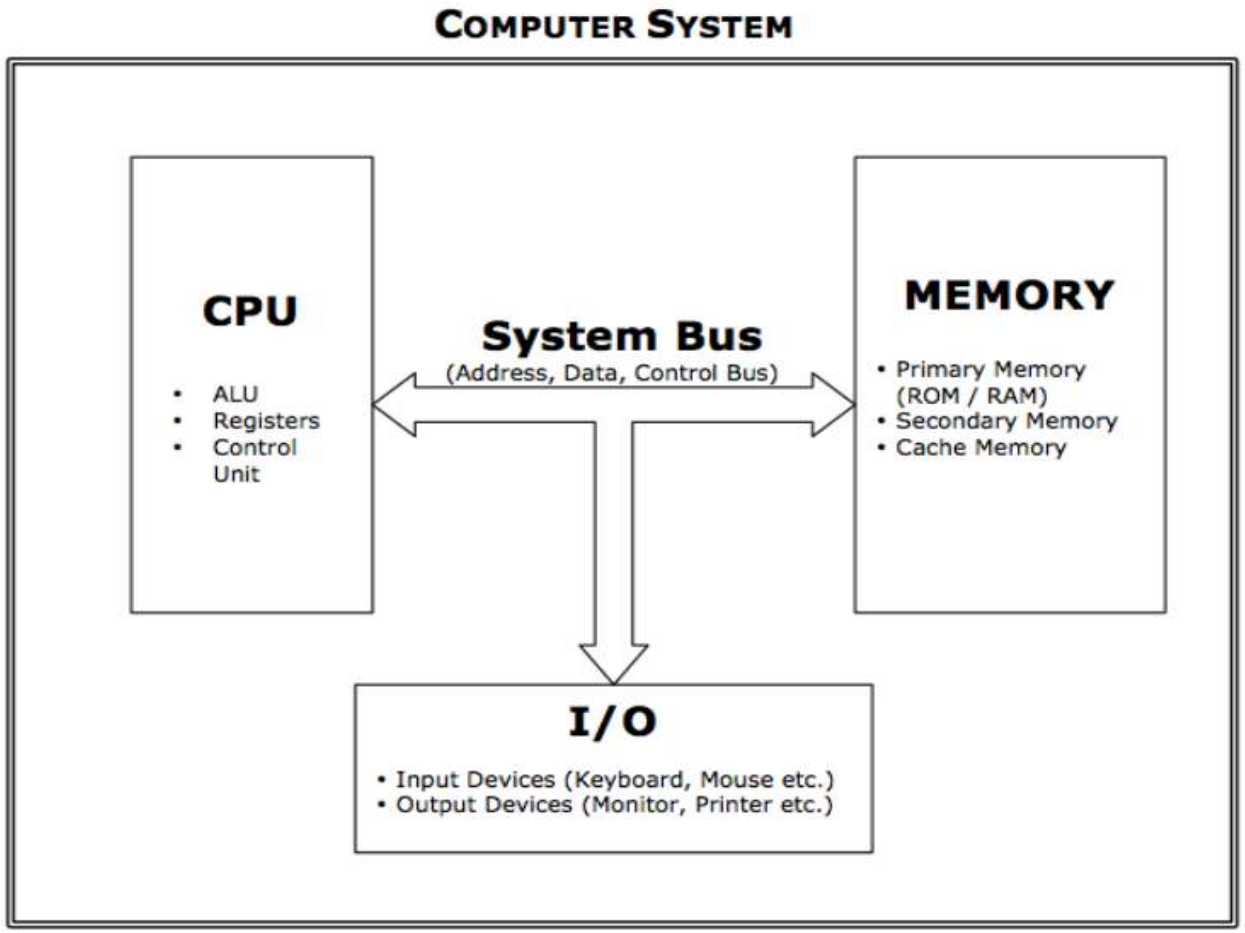
Reference Books:

1. M. Morris Mano, Rajib Mall, Computer System Architecture, Pearson Education Third Edition,2017.
2. Carl Hamacher, Zvonkovranesic, Safwat Zaky , Computer Organization, McGraw Hill, Fifth Edition,2011.

	COs Mapping with POs and PEOs	
	Course Outcome Statement	**PO's / PEO's**
CO1	Apply different formats of data representation and number systems.	PO1, PO2, PO3
CO2	Assemble a simple computer with hardware design including data format, instruction format, instruction set, addressing modes, and bus structure.	PO1, PO2, PO3
CO3	Understand the hierarchy of Memory and cache memory mapping techniques	PO1, PO2, PO3, PO5
CO4	Design and analyse Arithmetic/Logic unit, control unit, data, instruction and address flow.	PO1,PO2,PO3, PO5
CO5	Design simple assembly language programs that make appropriate use of a registers and memory.	PO1, PO2, PO3
		TOTAL HOURS OF INSTRUCTIONS: 60

Brain


The Brain and its functions
Based on Diagrams from
Head injury - A Practical Guide By Trevor Powel
Executive functions, thinking, planning, organising & problem solving. Emotions & behavioural control, personality (frontal lobe)
Movement (motor cortex)
Sensation (sensory cortex)
Perception, making sense of the world, arithmetic, spelling (parietal lobe)
Vision (occipital lobe)
Balance (cerebelium)
Carrying messages (spinal cord)
Memory, understanding & language (temporal lobe)


COMPUTER SYSTEM
CPU
ALU
Registers
Control Unit
System Bus
(Address, Data, Control Bus)
MEMORY
Primary Memory (ROM / RAM)
Secondary Memory
Cache Memory
I/O
Input Devices (Keyboard, Mouse etc.)
Output Devices (Monitor, Printer etc.)

A computer system mainly consists of The CPU, Memory, I/O devices and the System bus used for interconnections

CPU

1) The Central Processing Unit is the **most important part** of the Computer.
2) It is also called the **microprocessor** or simply the **processor**.
3) It consists of the ALU, Registers, Control Unit etc.
4) All **programs are executed** in the CPU.
5) A program is a **set of instructions** stored in the memory.
6) The main function of the CPU is to **fetch, decode and execute** these instructions.
7) Instructions are **fetched from the memory** using the various **buses**.
8) Thereafter they are **decoded by the Control Unit** to analyze the Opcode.
9) Finally the instruction is **executed** to perform the **desired operation**.
10) This **execution** mainly involves the **ALU** and the **internal registers** of the processor.

MEMORY

1) The memory is used to **store information**.
2) It mainly stores **programs and data**.
3) Memory has various **locations**.
4) Each location is identified by its own **unique address and contains some data**.
5) The most basic form of memory is called **Primary Memory**, which consists of **RAM and ROM**.
6) Then there are **secondary** storage devices such as **hard disk**.
7) There are **portable** storage devices like **CD, DVD, Pen drives** etc.

I/O

1) I/O devices are used for the **flow of information in and out** of the computer system.
2) **Input** devices such as **keyboard, mouse**, etc. are used to provide inputs into the computer.
3) They are used to **enter programs and data**.
4) **Output** devices such as **monitor and printer** are used to **generate results**.
5) Some devices such as a **touch-screen** can be used for **both input and output**.

System Bus

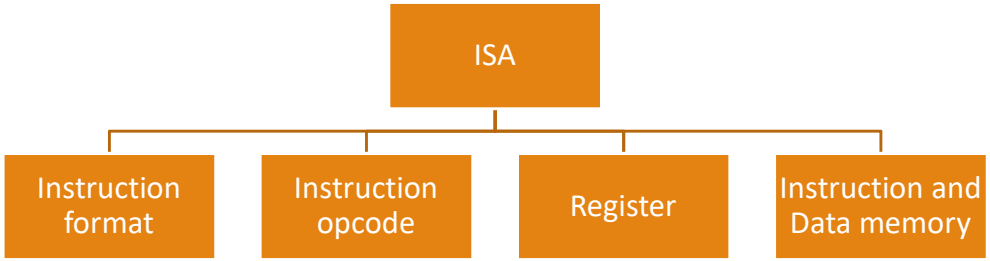
1) A bus is a set of **interconnecting lines used to carry information**.
2) **Size** of a bus means its **number of lines**.
3) An 8-bit bus has eight lines carrying **one bit each**.
4) There are three types of buses.
5) **Address Bus: It carries the address for the operation.**
   During any operation, the address bus identifies the location where the operation is performed.
   The size of the address bus determines the amount of Primary Memory that can be connected.
   Example: If address bus is 16-bit, we can connect 2^{16} = 64 KB Memory.
   Bigger the address bus, bigger is the memory.
6) **Data Bus: It carries data to and from the processor.**
   The size of data bus determines how much data can be transferred in one operation (cycle).
   Bigger the data bus, faster the processor, as it can transfer more data in one cycle.
7) **Control Bus: It Carries control signals like RD, WR etc.**
   These signals determine the kind of operation that will be performed on the system bus.

Architecture & Organization

Architecture is those attributes visible to the programmer

- Instruction set, number of bits used for data representation, I/O mechanisms, addressing techniques.
- e.g. Is there a multiply instruction?

Computer Architecture is also referred as Instruction set architecture (ISA) which has an algorithm to control various components.



Organization is how features are implemented by interconnecting the operational units to realize the specific architectural specifications.

- Control signals, interfaces, memory technology.
- e.g. Is there a hardware multiply unit or is it done by repeated addition?

All Intel x86 family share the same basic architecture

The IBM System/370 family share the same basic architecture

This gives code compatibility

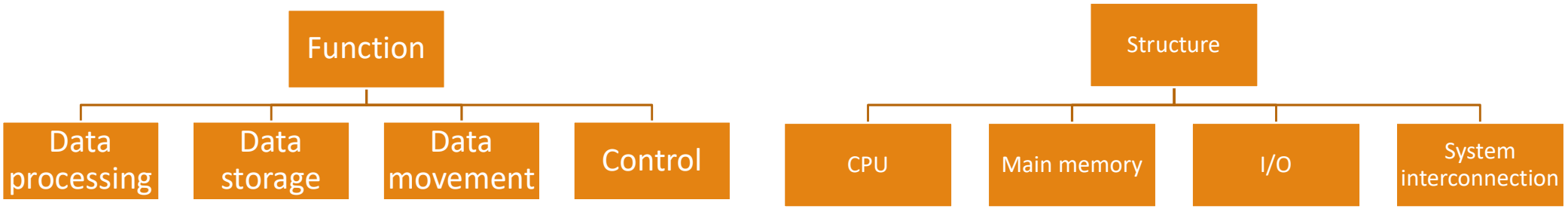
Organization differs between different versions

Ex: Pipelining

Structure & Function

Structure is the way in which components relate to each other

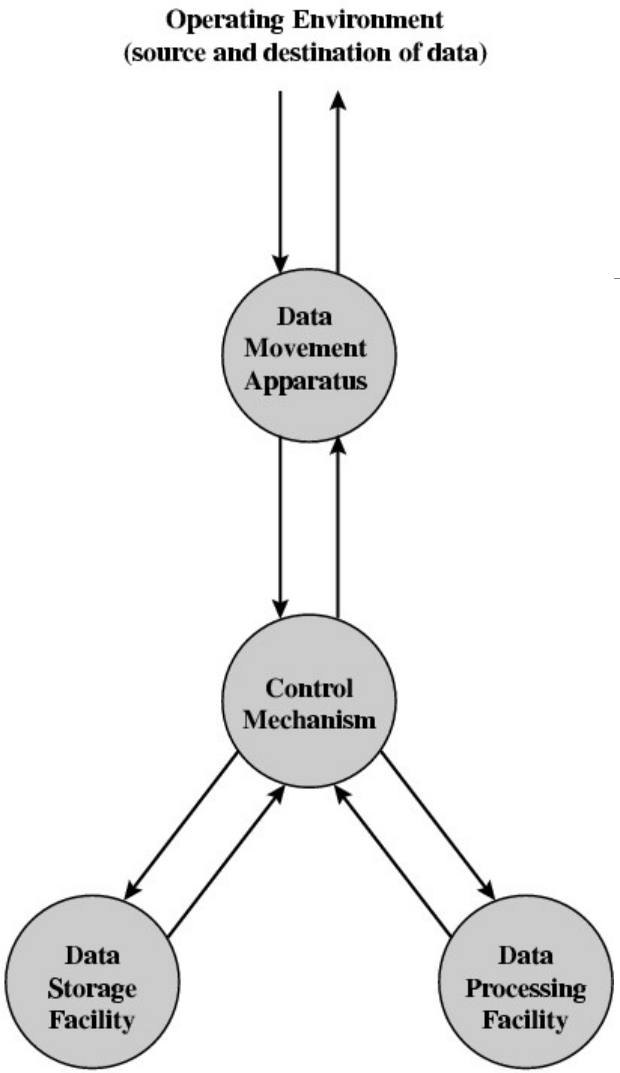
Function is the operation of individual components as part of the structure



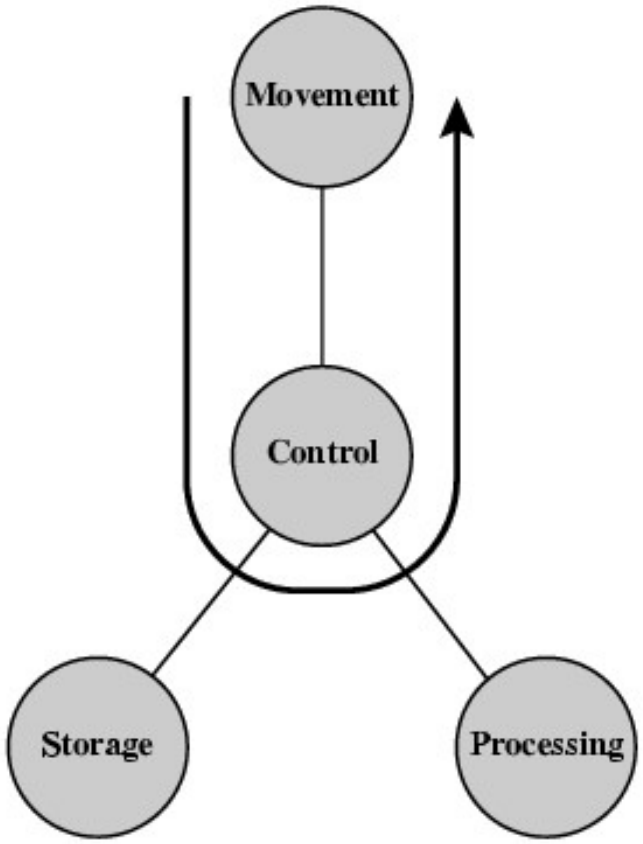
Function

All computer functions are:

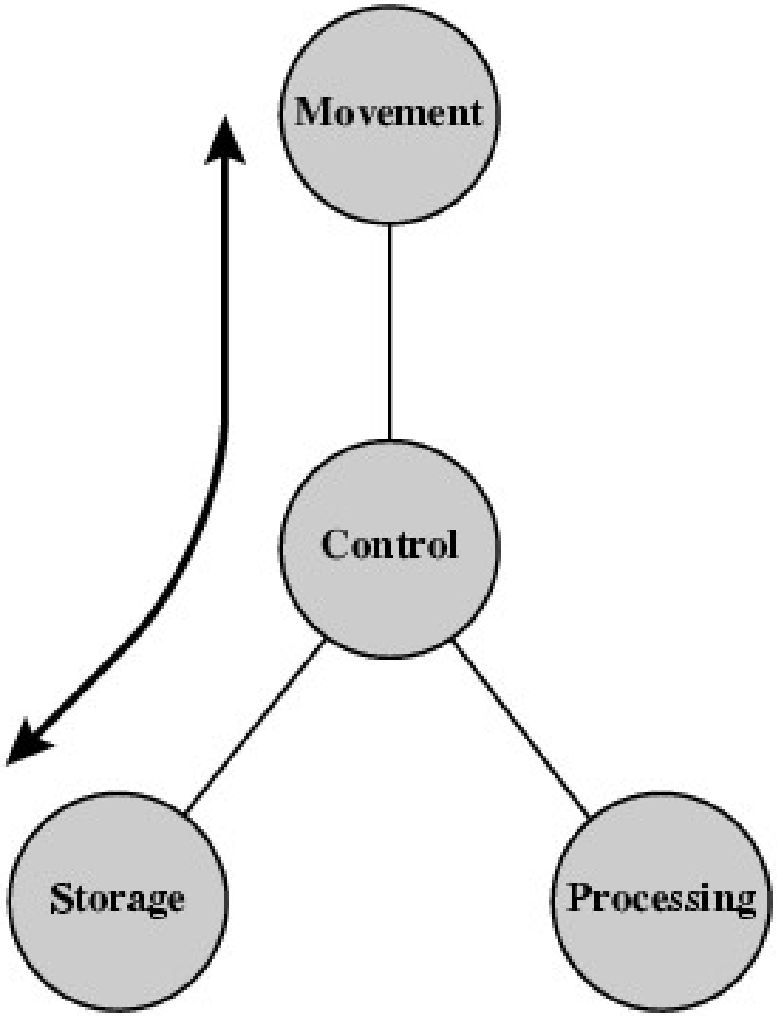
- Data processing
- Data storage
- Data movement
- Control



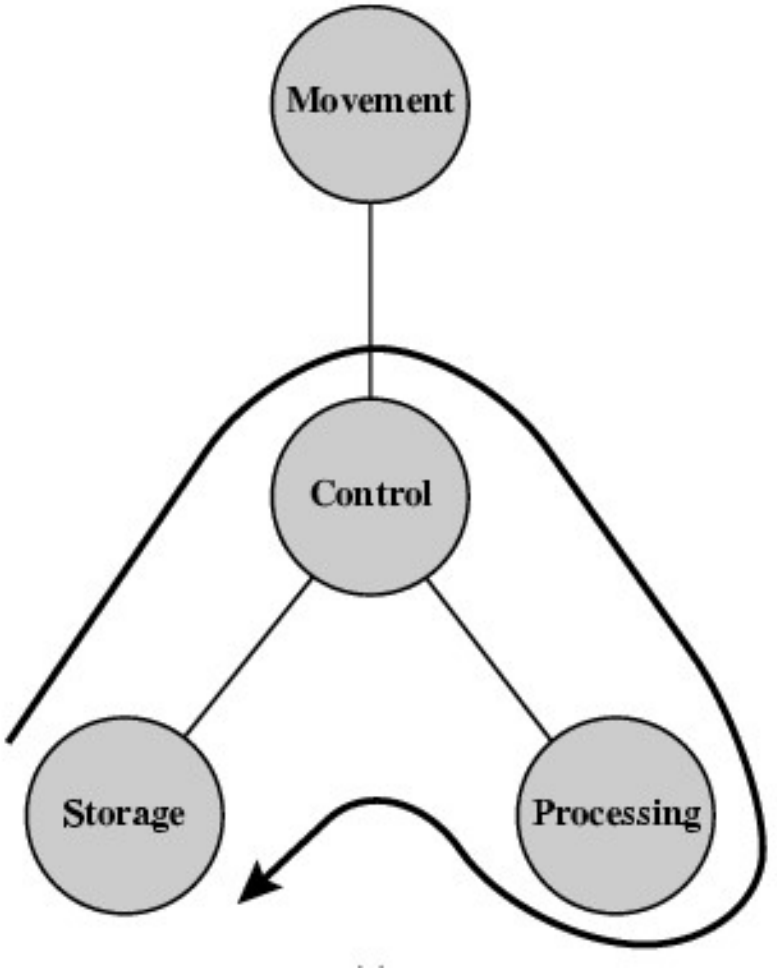
Operations (1) Data movement



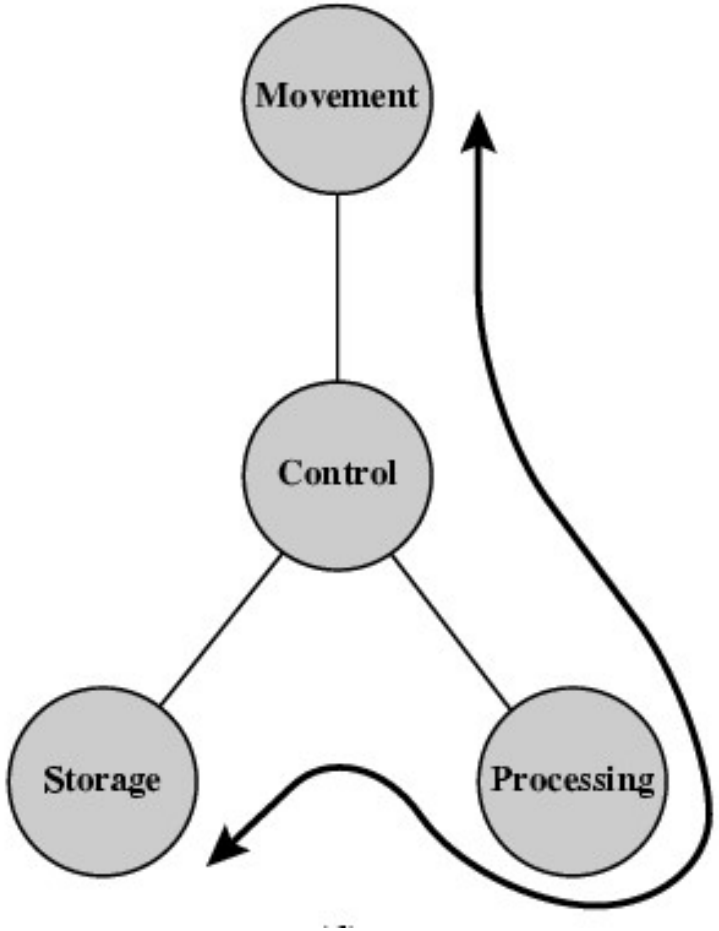
Operations (2) Storage



Operation (3) Processing from/to storage



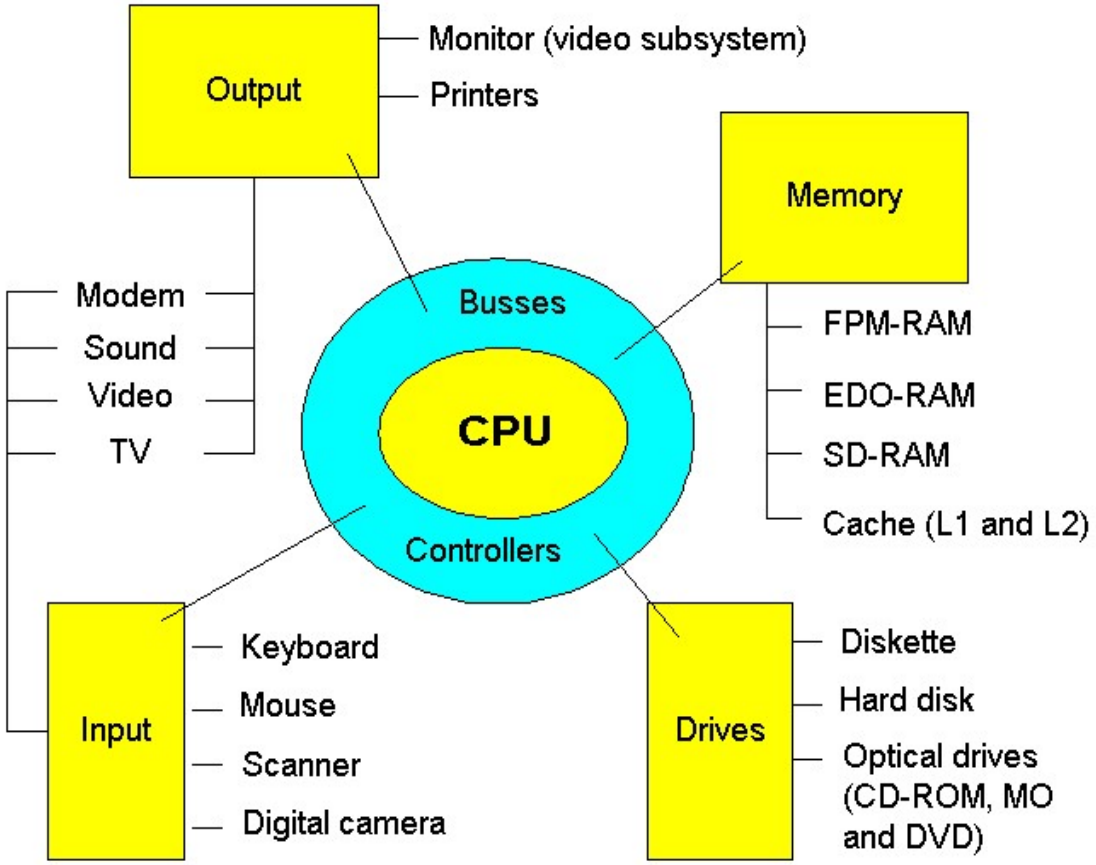
Operation (4) Processing from storage to I/O



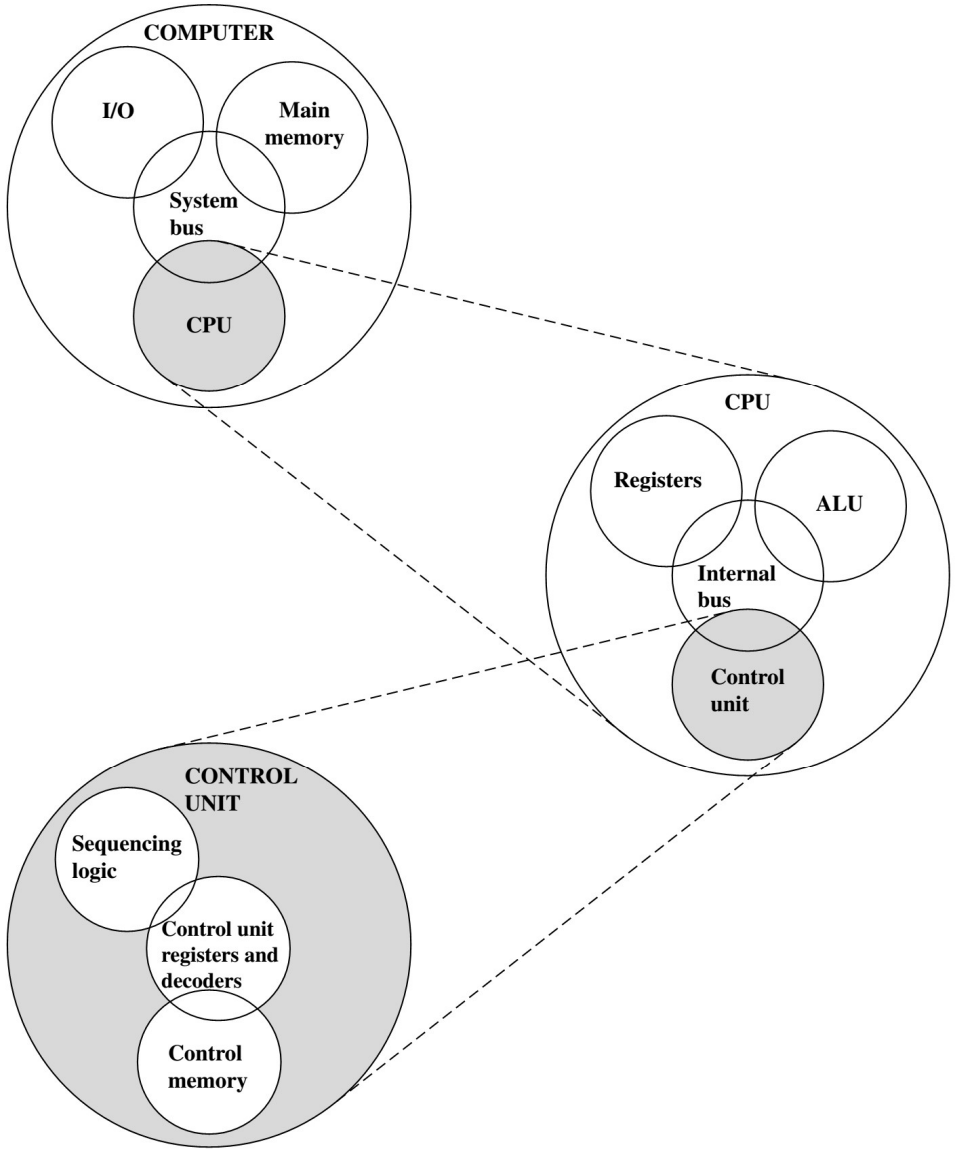
Structure

The Computer

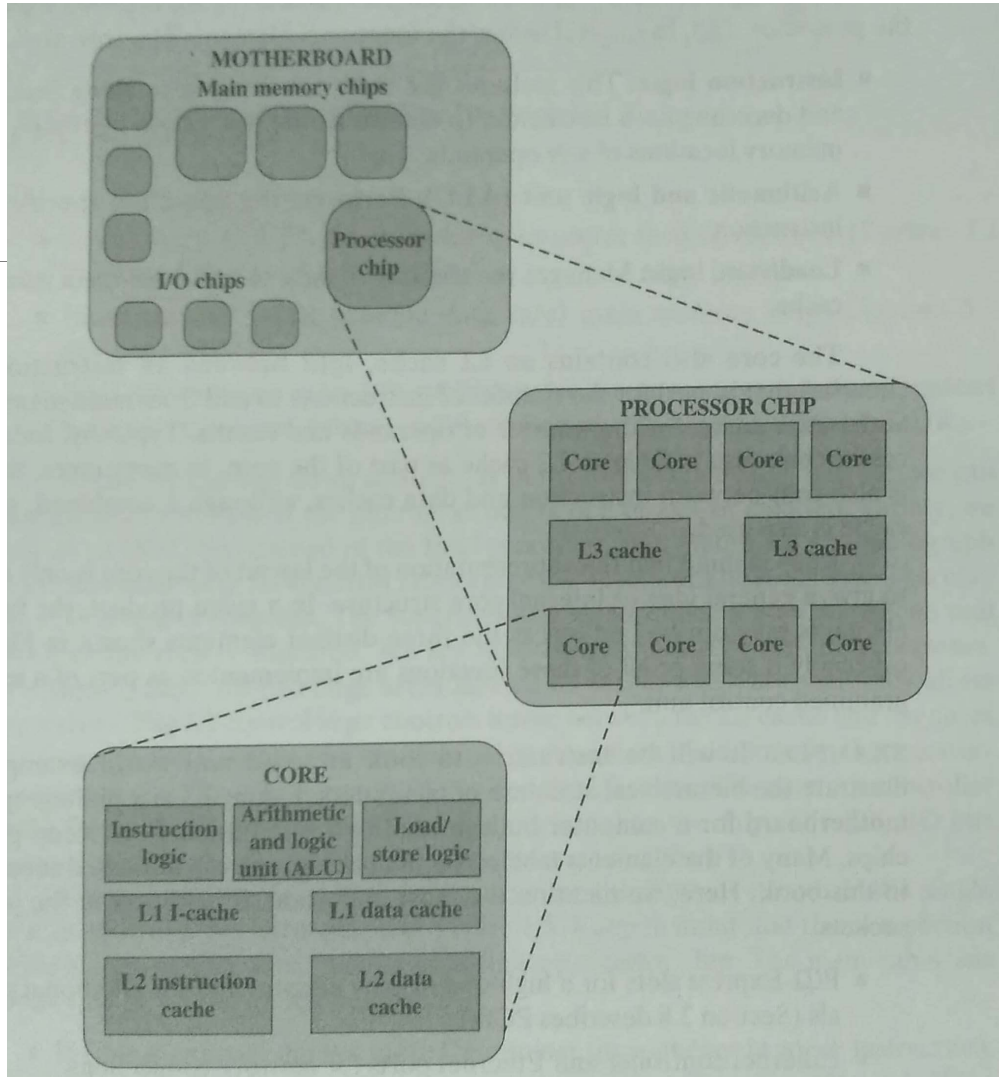
- CPU
  - Controls the operation of the computer and performs its data processing functions.
- Main memory
  - Stores data
  - Fast Page Mode RAM
  - Synchronous DRAM
  - Extended data output) **RAM**
- I/O
  - Moves data between the computer and its external environment
- System interconnection
  - Provides for communication among CPU, main memory, and I/O



Structure - Top Level



Multicore computer



Review Questions

1. What, in general terms, is the distinction between computer organization and computer architecture?
2. What, in general terms, is the distinction between computer structure and computer function?
3. What are the four main functions of a computer?
4. List and briefly define the main structural components of a computer.
5. List and briefly define the main structural components of a processor.

History of computers

First Generation: Vacuum Tubes

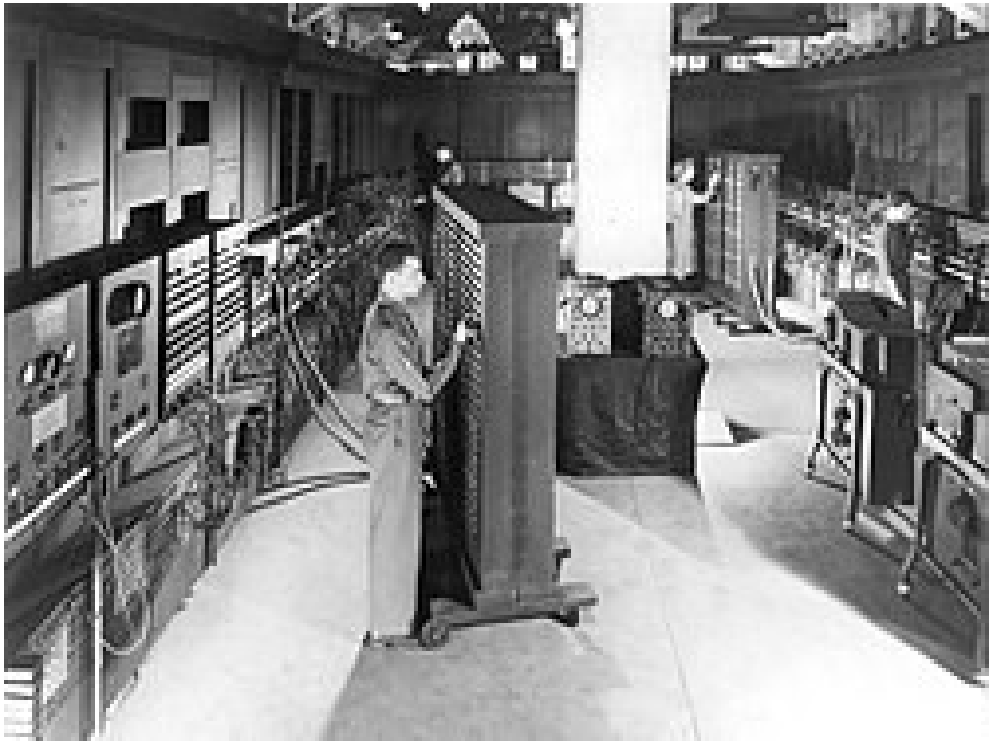
Second Generation: Transistors

Third Generation: Integrated circuits

Later generations: LSI and VLSI

First Gen: Vacuum tubes

- ENIAC (Electronic Numerical Integrator and Computer)
- Built in 1943 for WW-II
- Weighed 30 tons, occupying 1500 square feet of floor space, and containing more than 18,000 vacuum tubes
- Consumed 140 kilowatts of power
- Faster than any electromechanical computer, capable of 5000 additions per second



ENIAC

- It was a decimal computer, rather than a binary one
- Its memory consisted of 20 “accumulators,” each capable of holding a 10-digit decimal number
- A ring of 10 vacuum tubes represented each digit
- The major drawback is that it had to be programmed manually by setting switches and plugging and unplugging cables

Von Neumann machine

- Program could be represented in a form suitable for storing in memory alongside the data
- A computer could get its instructions by reading them from memory
- This program could be set or altered by setting the values of a portion of memory
- This idea, known as the *stored-program concept,* usually attributed to the mathematician John von Neumann
- Shortly, the design of a new stored program computer, referred to as the IAS computer, at the Princeton Institute for Advanced Studies
- Took 6 Years to build and is the prototype for all the subsequent general-purpose computers

Second GEN: Transistors

- Invented at Bell Labs in 1947; by 1950s → electronic revolution.
- Late 1950s → fully transistorized computers were commercially available
- Smaller, cheaper, and dissipates less heat; can be used in the same way as a vacuum tube to construct computers
- The second generation introduced more complex ALUs and CUs
- Use of high-level programming languages, and *system software* with the computer.

Table 2.3 Example members of the IBM 700/7000 Series

Model Number	First Delivery	CPU Technology	Memory Technology	Cycle Time (μs)	Memory Size (K)	Number of Opcodes	Number of Index Registers	Hardwired Floating-Point	I/O Overlap (Channels)	Instruction Fetch Overlap	Speed (relative to 701)
701	1952	Vacuum tubes	Electrostatic tubes	30	2–4	24	0	no	no	no	1
704	1955	Vacuum tubes	Core	12	4–32	80	3	yes	no	no	2.5
709	1958	Vacuum tubes	Core	12	32	140	3	yes	yes	no	4
7090	1960	Transistor	Core	2.18	32	169	3	yes	yes	no	25
7094 I	1962	Transistor	Core	2	32	185	7	yes (double precision)	yes	yes	30
7094 II	1964	Transistor	Core	1.4	32	185	7	yes (double precision)	yes	yes	50

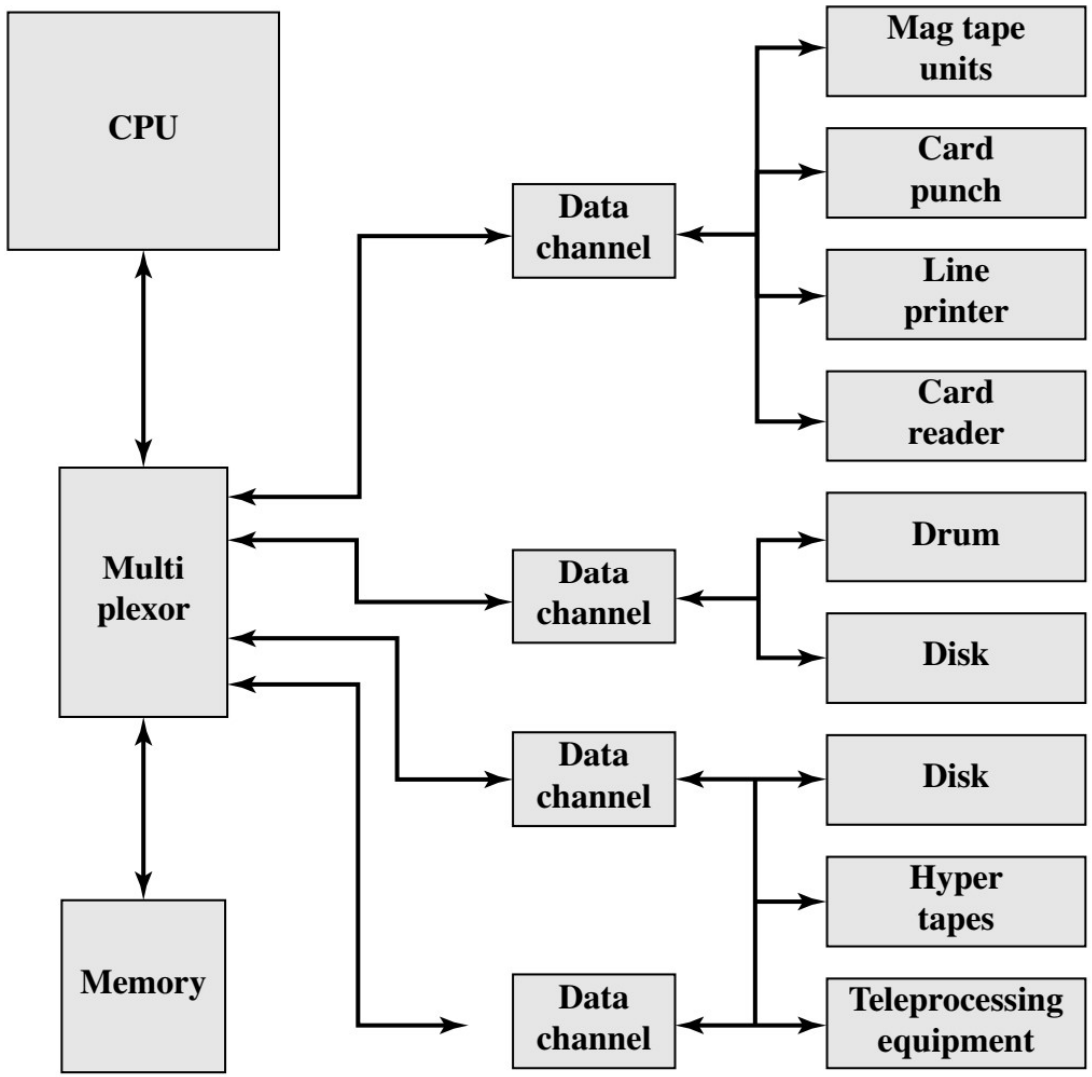


Figure 2.5 An IBM 7094 Configuration

Third gen: Integrated circuits

- A single, self-contained transistor is called a *discrete component*
- 1950s to early 1960s, electronic equipment was composed largely of discrete components—transistors, resistors, capacitors, and so on.
- Discrete components were manufactured separately → packaged in their own containers → soldered or wired together onto masonite-like circuit boards → then installed in computers, oscilloscopes, and other electronic equipment.
- Whenever an electronic device needed a transistor replacement, a small piece of silicon had to be soldered to a circuit board; made the manufacturing process expensive and cumbersome.
- The Early second-generation computers contained about 10,000 transistors. This figure grew to the hundreds of thousands; manufacturing of newer, more powerful machines became increasingly difficult.

Third gen: Integrated circuits

- 1958 → invention of the Integrated circuit
- The integrated circuit exploits the fact that → components as transistors, resistors, and conductors can be fabricated from a semiconductor such as silicon.
- Fabrication of an entire circuit in a tiny piece of silicon; rather than assemble discrete components using separate pieces of silicon
- Many transistors can be produced at the same time on a single wafer of silicon; and can be connected with a process of metallization to form circuits

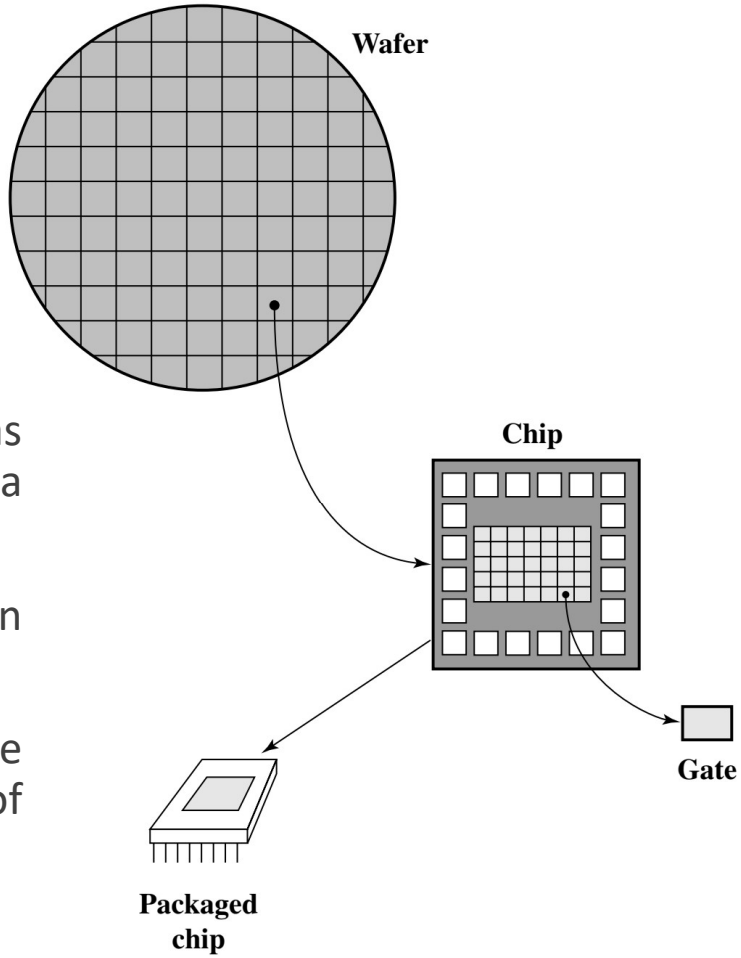


Figure 2.7 Relationship among Wafer, Chip, and Gate

Integrated circuits

- Initially, only a few gates or memory cells could be reliably manufactured and packaged together; known as SSI
- As time went on, it became possible to pack more and more components on the same chip.
- This figure reflects the famous Moore's law, which was propounded by Gordon Moore, cofounder of Intel, in 1965.

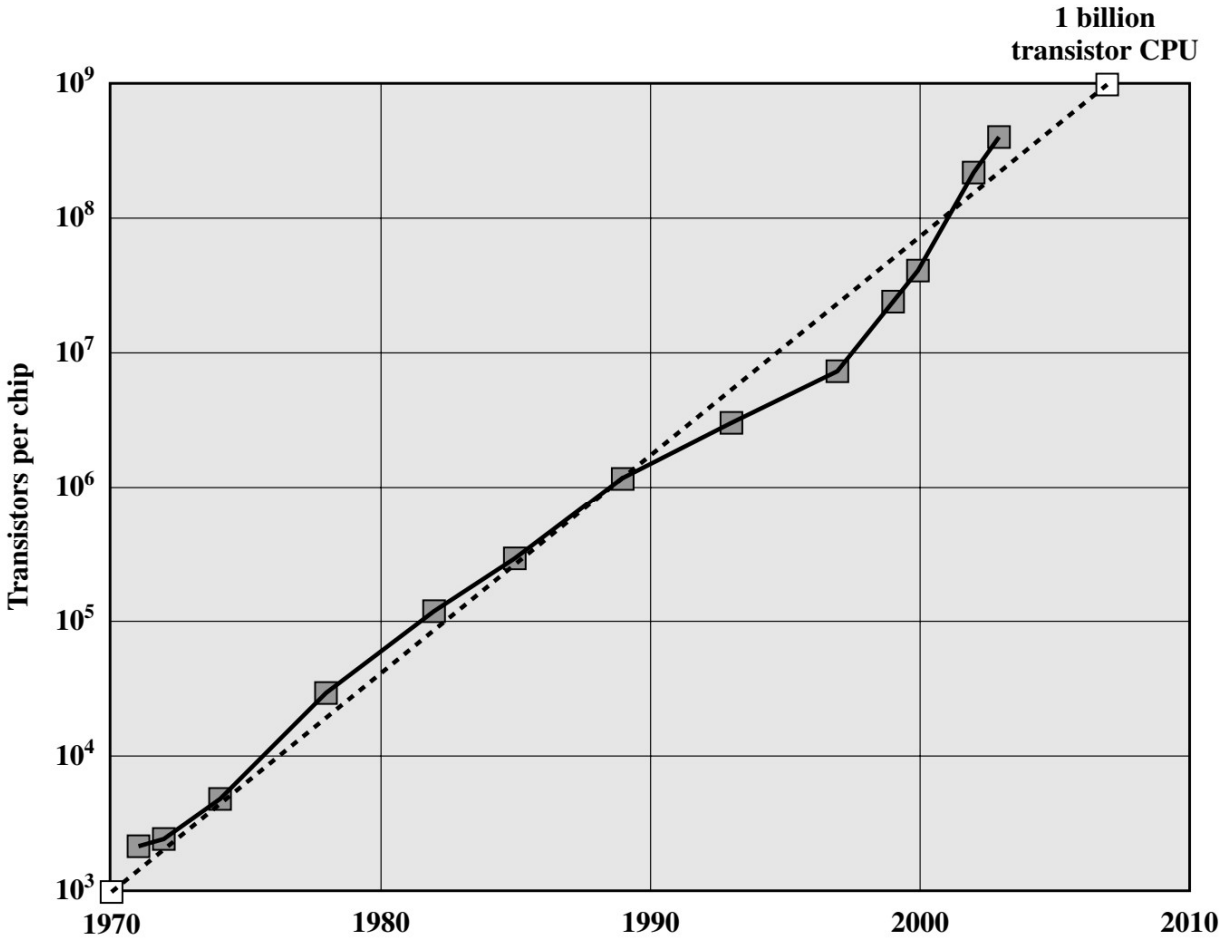


Figure 2.8 Growth in CPU Transistor Count [BOHR03]

Integrated circuits

- Moore observed that the #Transistors was doubling every year; correctly predicted that this pace would continue
- The pace continued year after year and decade after decade; it slowed to a doubling every 18 months in the 1970s but has sustained that rate ever since
- Consequences of Moore's law:
  - The cost of a chip has remained virtually unchanged during this period of rapid growth in density. This means that the cost of computer logic and memory circuitry has fallen at a dramatic rate.
  - Because logic and memory elements are placed closer together on more densely packed chips, the electrical path length is shortened, increasing operating speed.
  - The computer becomes smaller, making it more convenient to place in a variety of environments.
  - There is a reduction in power and cooling requirements.
  - The interconnections on the integrated circuit are much more reliable than solder connections. With more circuitry on each chip, there are fewer interchip connections

Later generations

Table 2.2 Computer Generations

Generation	Approximate Dates	Technology	Typical Speed (operations per second)
1	1946–1957	Vacuum tube	40,000
2	1958–1964	Transistor	200,000
3	1965–1971	Small and medium scale integration	1,000,000
4	1972–1977	Large scale integration	10,000,000
5	1978–1991	Very large scale integration	100,000,000
6	1991–	Ultra large scale integration	1,000,000,000

Later generations

- With the introduction of largescale integration (LSI), more than 1000 components can be placed on a single integrated circuit chip.
- Very-large-scale integration (VLSI) achieved more than 10,000 components per chip, while current ultra-large-scale integration (ULSI) chips can contain more than one million components.
- The first application of integrated circuit technology to computers was construction of the processor
- It was also found that this same technology could be used to construct memories

Later generations

- In the 1950s and 1960s, most computer memory was constructed from tiny rings of ferromagnetic materials
- Magnetized one way, a ring (called a *core*) represented a one; magnetized the other way, it stood for a zero.
- Magnetic-core memory was rather fast; it took as little as a millionth of a second to read a bit stored in memory.
- But it was expensive, bulky, and used destructive readout: The simple act of reading a core erased the data stored in it.
- It was therefore necessary to install circuits to restore the data as soon as it had been extracted.

Later generations

- Then, in 1970, Fairchild produced the first semiconductor memory.
- This chip, about the size of a single core, could hold 256 bits of memory.
- It was nondestructive and much faster than core. It took only 70 billionths of a second to read a bit.
- However, the cost per bit was higher than for that of core
- In 1974, the price per bit of semiconductor memory dropped below the price per bit of core memory.
- Following this, there has been a continuing and rapid decline in memory cost accompanied by a corresponding increase in physical memory density.
- This has led the way to smaller, faster machines with memory sizes of larger and more expensive machines from just a few years earlier.
- Developments in memory technology, together with developments in processor technology changed the nature of computers in less than a decade.

Microprocessors

- A breakthrough was achieved in 1971, when Intel developed its 4004.
- The 4004 was the first chip to contain *all* of the components of a CPU on a single chip: The microprocessor was born.
- The 4004 can add two 4-bit numbers and can multiply only by repeated addition.
- By today's standards, the 4004 is hopelessly primitive, but it marked the beginning of a continuing evolution of microprocessor capability and power.
- The next major step in the evolution of the microprocessor was the introduction in 1972 of the Intel 8008.
- This was the first 8-bit microprocessor and was almost twice as complex as the 4004.

Microprocessors

- The introduction, in 1974, of the Intel 8080.
- This was the first general-purpose microprocessor.
- The 4004 and the 8008 had been designed for specific applications, the 8080 was designed to be the CPU of a general-purpose microcomputer.
- Like the 8008, the 8080 is an 8-bit microprocessor.
- It was faster, has a richer instruction set, and has a large addressing capability.

Microprocessors

- About the same time, 16-bit microprocessors began to be developed.
- However, it was not until the end of the 1970s that powerful, general-purpose 16-bit microprocessors appeared. One of these was the 8086.
- The next step in this trend occurred in 1981, when both Bell Labs and Hewlett-Packard developed 32-bit, single-chip microprocessors.
- Intel introduced its own 32-bit microprocessor, the 80386, in 1985

Table 2.6 Evolution of Intel Microprocessors

(a) 1970s Processors

	4004	8008	8080	8086	8088
Introduced	1971	1972	1974	1978	1979
Clock speeds	108 kHz	108 kHz	2 MHz	5 MHz, 8 MHz, 10 MHz	5 MHz, 8 MHz
Bus width	4 bits	8 bits	8 bits	16 bits	8 bits
Number of transistors	2,300	3,500	6,000	29,000	29,000
Feature size (μm)	10		6	3	6
Addressable memory	640 Bytes	16 KB	64 KB	1 MB	1 MB

(b) 1980s Processors

	80286	386TM DX	386TM SX	486TM DX CPU
Introduced	1982	1985	1988	1989
Clock speeds	6 MHz–12.5 MHz	16 MHz–33 MHz	16 MHz–33 MHz	25 MHz–50 MHz
Bus width	16 bits	32 bits	16 bits	32 bits
Number of transistors	134,000	275,000	275,000	1.2 million
Feature size (μm)	1.5	1	1	0.8–1
Addressable memory	16 MB	4 GB	16 MB	4 GB
Virtual memory	1 GB	64 TB	64 TB	64 TB
Cache	—	—	—	8 kB

(c) 1990s Processors

	486TM SX	Pentium	Pentium Pro	Pentium II
Introduced	1991	1993	1995	1997
Clock speeds	16 MHz–33 MHz	60 MHz–166 MHz,	150 MHz–200 MHz	200 MHz–300 MHz
Bus width	32 bits	32 bits	64 bits	64 bits
Number of transistors	1.185 million	3.1 million	5.5 million	7.5 million
Feature size (μm)	1	0.8	0.6	0.35
Addressable memory	4 GB	4 GB	64 GB	64 GB
Virtual memory	64 TB	64 TB	64 TB	64 TB
Cache	8 kB	8 kB	512 kB L1 and 1 MB L2	512 kB L2

(d) Recent Processors

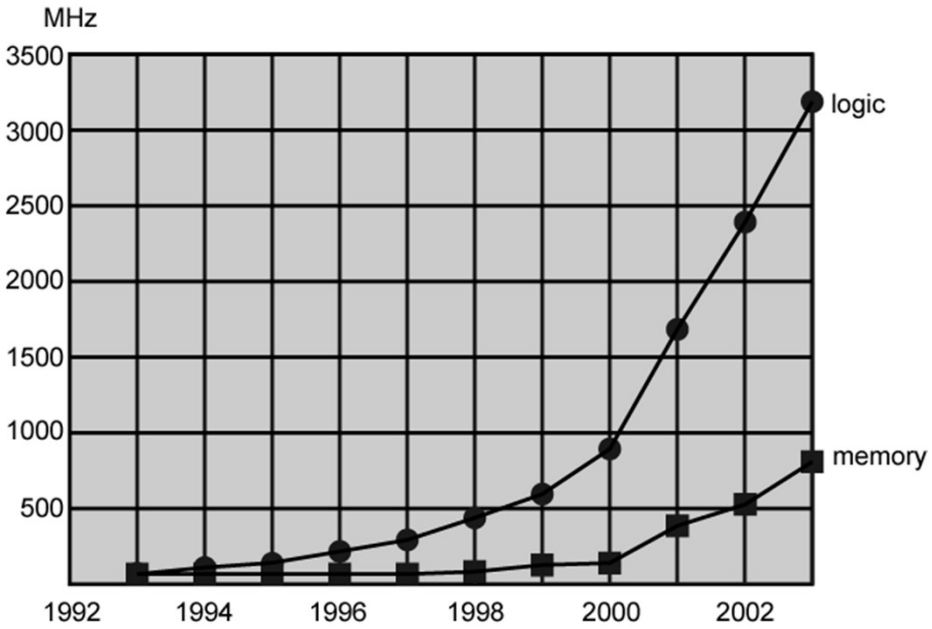
	Pentium III	Pentium 4	Core 2 Duo	Core 2 Quad
Introduced	1999	2000	2006	2008
Clock speeds	450–660 MHz	1.3–1.8 GHz	1.06–1.2 GHz	3 GHz
Bus sidth	64 bits	64 bits	64 bits	64 bits
Number of transistors	9.5 million	42 million	167 million	820 million
Feature size (nm)	250	180	65	45
Addressable memory	64 GB	64 GB	64 GB	64 GB
Virtual memory	64 TB	64 TB	64 TB	64 TB
Cache	512 kB L2	256 kB L2	2 MB L2	6 MB L2

Speeding it up

- Pipelining
- On board cache
- On board L1 & L2 cache
- Branch prediction
- Data flow analysis

Performance Balance

- Processor speed increased
- Memory capacity increased
- Memory speed lags behind processor speed



Logic and Memory Performance Gap

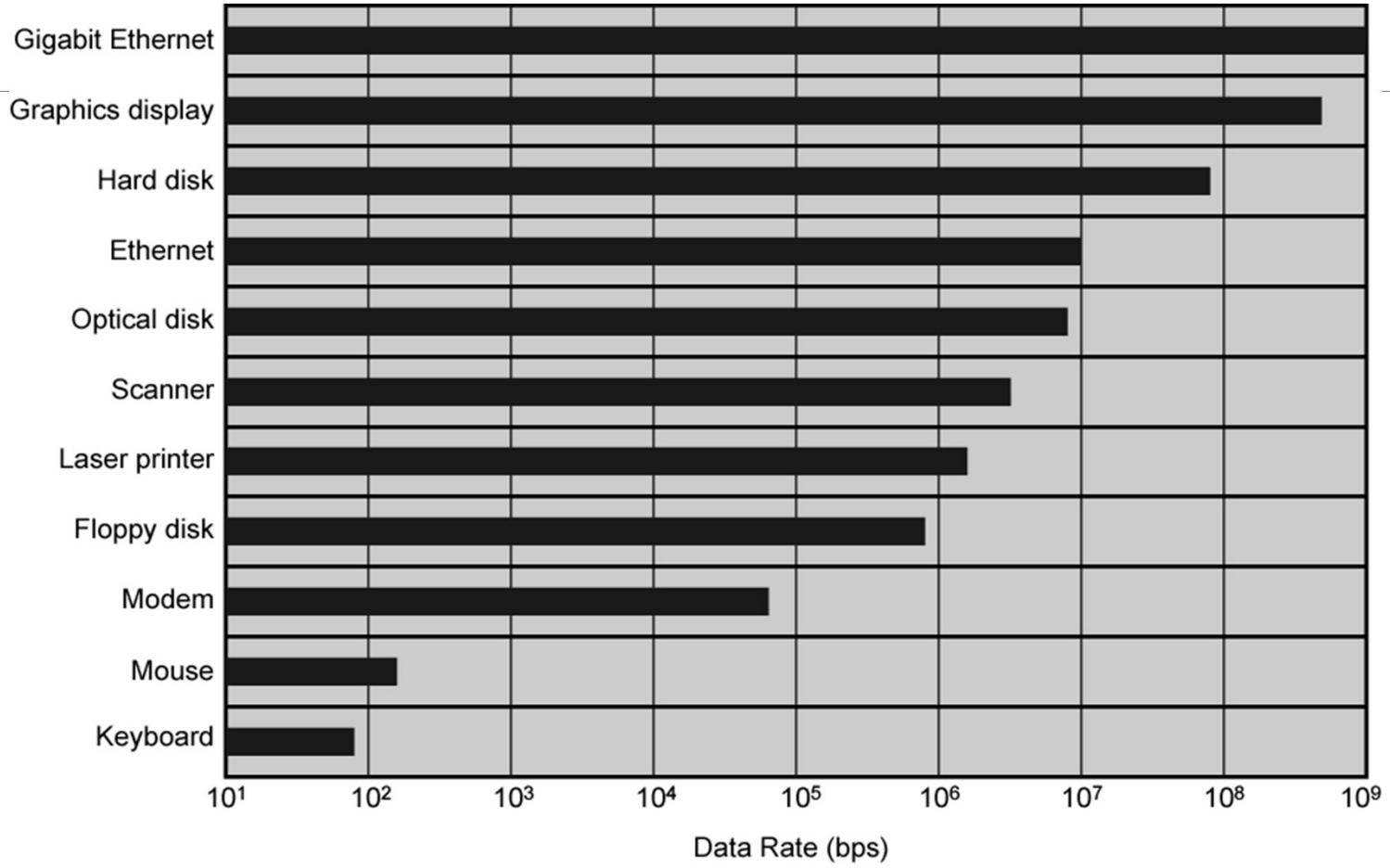
Solutions

- Increase number of bits retrieved at one time
  - Make DRAM “wider” rather than “deeper”
- Change DRAM interface
  - Cache
- Reduce frequency of memory access
  - More complex cache and cache on chip
- Increase interconnection bandwidth
  - High speed buses
  - Hierarchy of buses

I/O Devices

- Peripherals with intensive I/O demands
- Large data throughput demands
- Processors can handle this
- Solutions:
  - Caching
  - Higher-speed interconnection buses
  - More elaborate bus structures
  - Multiple-processor configurations

Typical I/O Device Data Rates



Key is Balance

- Processor components
- Main memory
- I/O devices
- Interconnection structures

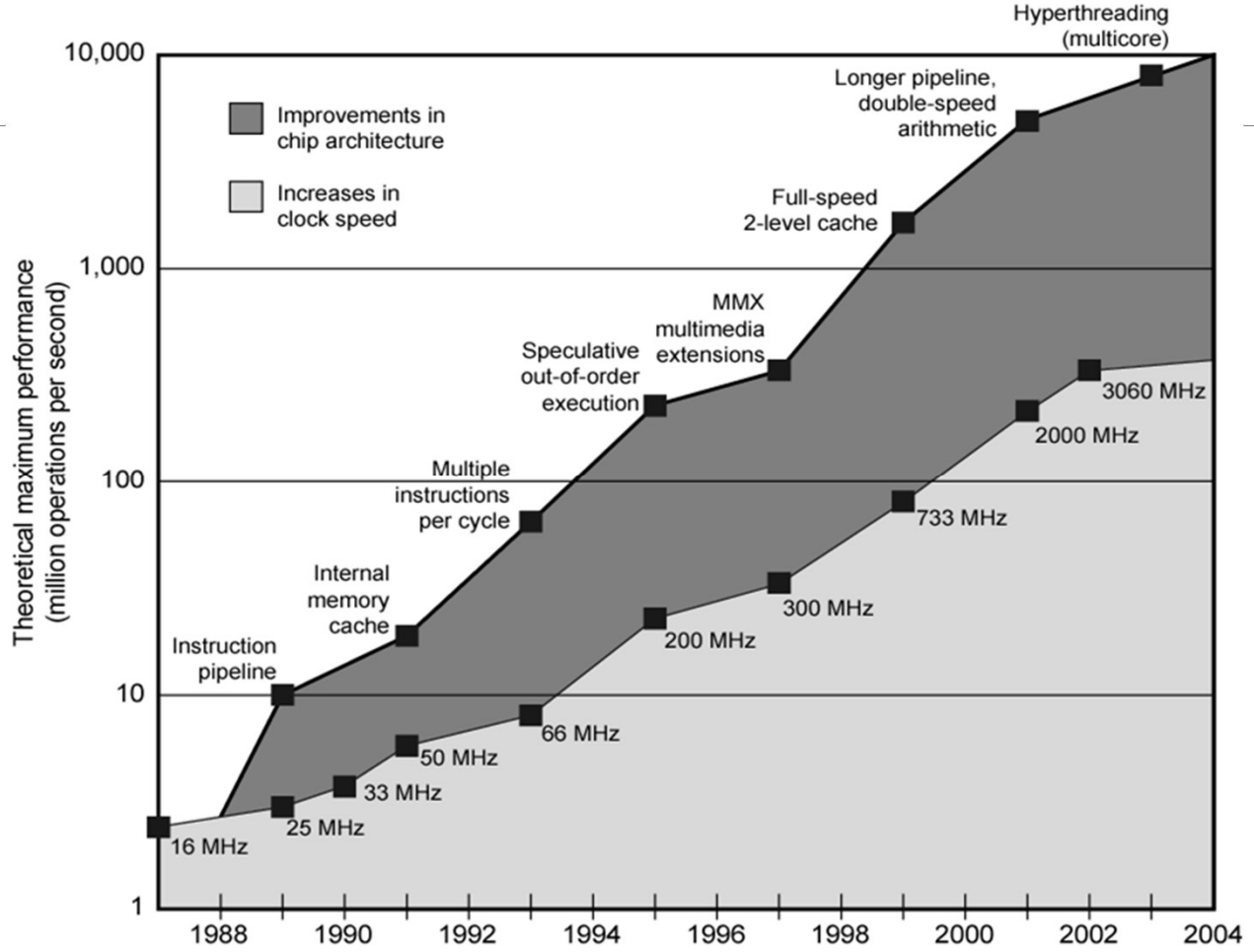
Improvements in Chip Organization and Architecture

- Increase hardware speed of processor
  - Fundamentally due to shrinking logic gate size
    - More gates, packed more tightly, increasing clock rate
    - Propagation time for signals reduced
- Increase size and speed of caches
  - Dedicating part of processor chip
    - Cache access times drop significantly
- Change processor organization and architecture
  - Increase effective speed of execution
  - Parallelism

Problems with Clock Speed and Logic Density

- Power
  - Power density increases with density of logic and clock speed
  - Dissipating heat
- RC delay
  - Speed at which electrons flow limited by resistance and capacitance of metal wires connecting them
  - Delay increases as RC product increases
  - Wire interconnects thinner, increasing resistance
  - Wires closer together, increasing capacitance
- Memory latency
  - Memory speeds lag processor speeds
- Solution:
  - More emphasis on organizational and architectural approaches

Intel Microprocessor Performance



Increased Cache Capacity

- Typically two or three levels of cache between processor and main memory
- Chip density increased
  - More cache memory on chip
    - Faster cache access
- Pentium chip devoted about 10% of chip area to cache
- Pentium 4 devotes about 50%

More Complex Execution Logic

- Enable parallel execution of instructions
- Pipeline works like assembly line
  - Different stages of execution of different instructions at same time along pipeline
- Superscalar allows multiple pipelines within single processor
  - Instructions that do not depend on one another can be executed in parallel

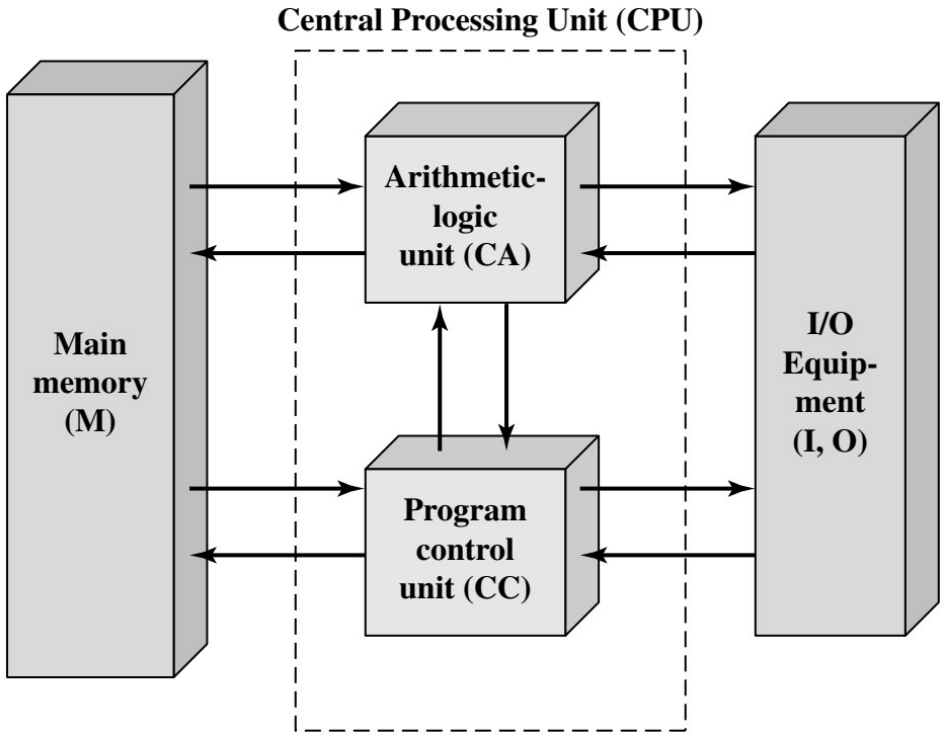
Diminishing Returns

- Internal organization of processors complex
  - Can get a great deal of parallelism
  - Further significant increases likely to be relatively modest
- Benefits from cache are reaching limit
- Increasing clock rate runs into power dissipation problem
  - Some fundamental physical limits are being reached

New Approach – Multiple Cores

- Multiple processors on single chip
  - Large shared cache
- Within a processor, increase in performance proportional to square root of increase in complexity
- If software can use multiple processors, doubling number of processors almost doubles performance
- So, use two simpler processors on the chip rather than one more complex processor
- With two processors, larger caches are justified
  - Power consumption of memory logic less than processing logic

IAS Computer

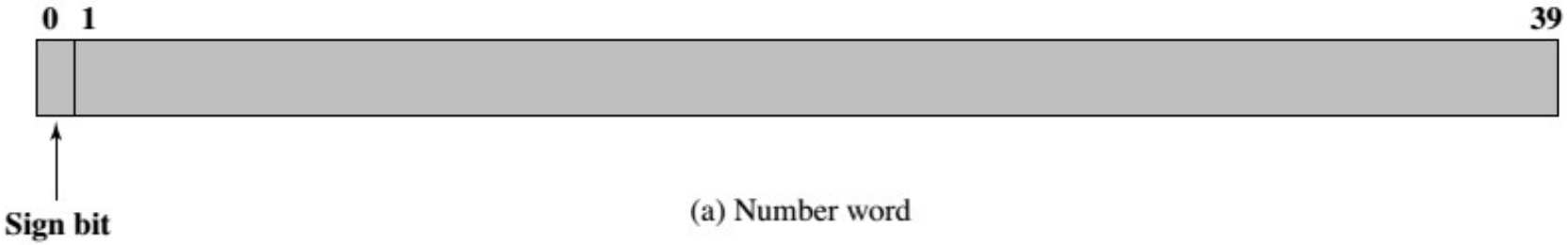


- A main memory, which stores both data and instructions
- An arithmetic and logic unit (ALU) capable of operating on binary data
- A control unit, which interprets the instructions in memory and causes them to be executed
- Input and output (I/O) equipment operated by the control unit

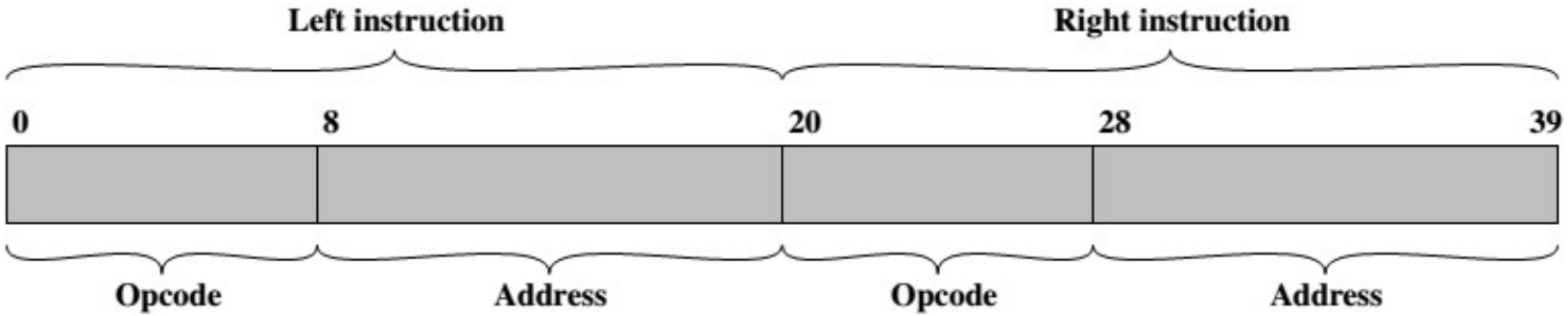
Working of IAS

- The memory of the IAS consists of 1000 storage locations, called words, of 40 binary digits (bits) each.
- Both data and instructions are stored there.
- Numbers are represented in binary form, and each instruction is a binary code.
- Each number is represented by a sign bit and a 39-bit value.
- A word may also contain two 20-bit instructions, with each instruction consisting of an 8-bit operation code (opcode) specifying the operation to be performed and a 12-bit address designating one of the words in memory (numbered from 0 to 999).

EXAMPLE



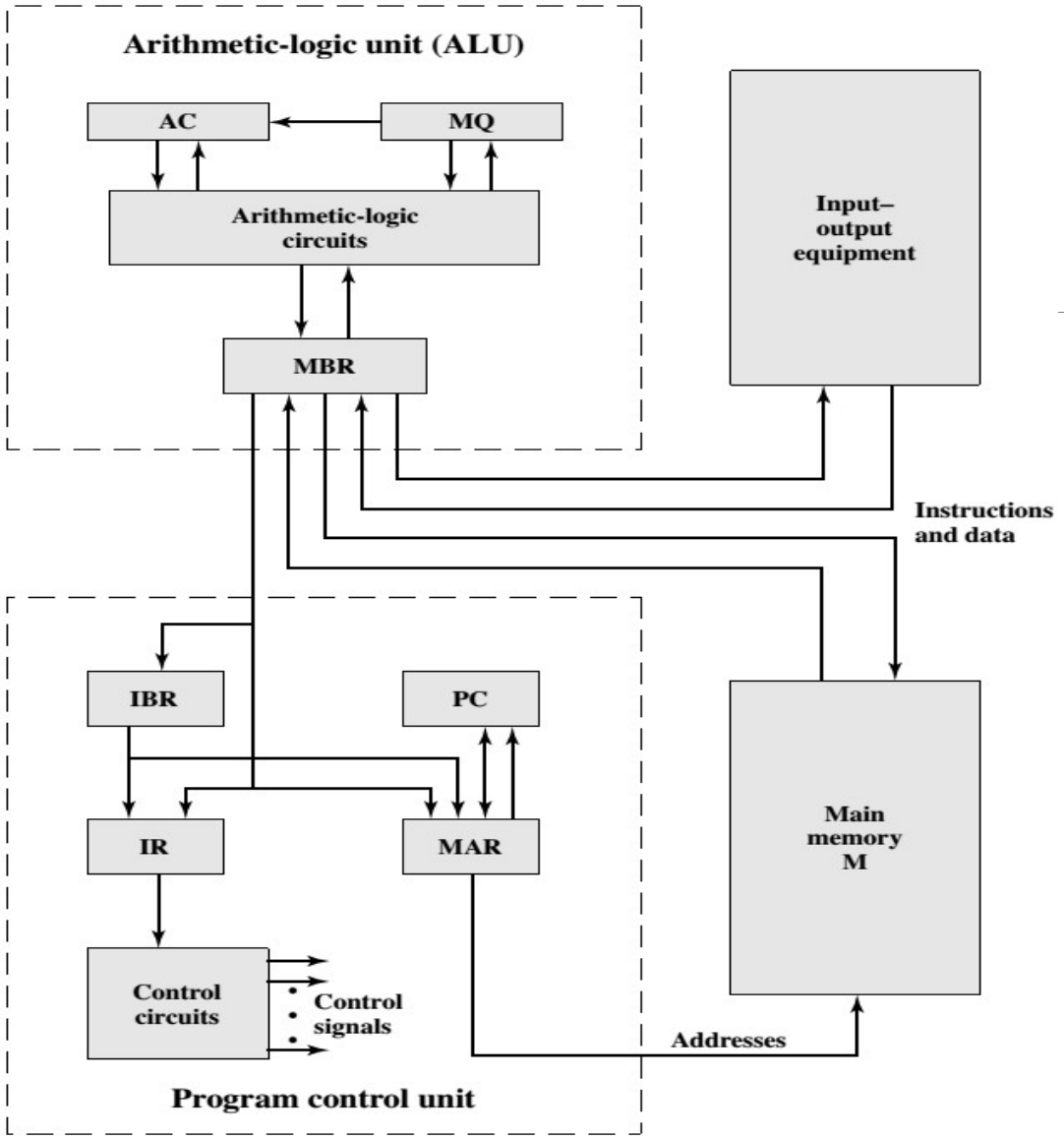
(a) Number word

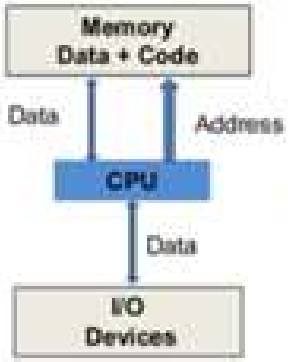


(b) Instruction word

Registers in IAS

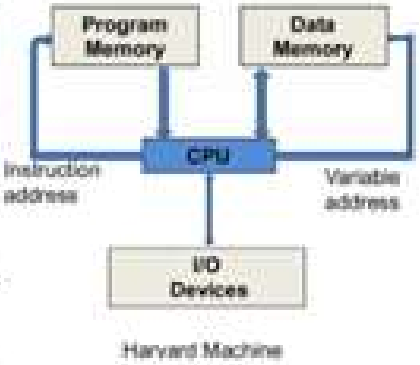
- **Memory buffer register (MBR):** Contains a word to be stored in memory or sent to the I/O unit, or is used to receive a word from memory or from the I/O unit.
- **Memory address register (MAR):** Specifies the address in memory of the word to be written from or read into the MBR.
- **Instruction register (IR):** Contains the 8-bit opcode instruction being executed.
- **Instruction buffer register (IBR):** Employed to hold temporarily the righthand instruction from a word in memory.
- **Program counter (PC):** Contains the address of the next instruction-pair to be fetched from memory.
- **Accumulator (AC) and multiplier quotient (MQ):** Employed to hold temporarily operands and results of ALU operations.


Arithmetic-logic unit (ALU)
AC
MQ
Arithmetic-logic circuits
MBR
Input–output equipment
Instructions and data
IBR
PC
IR
MAR
Main memory M
Control circuits
Control signals
Addresses
Program control unit



Von Neumann Machine

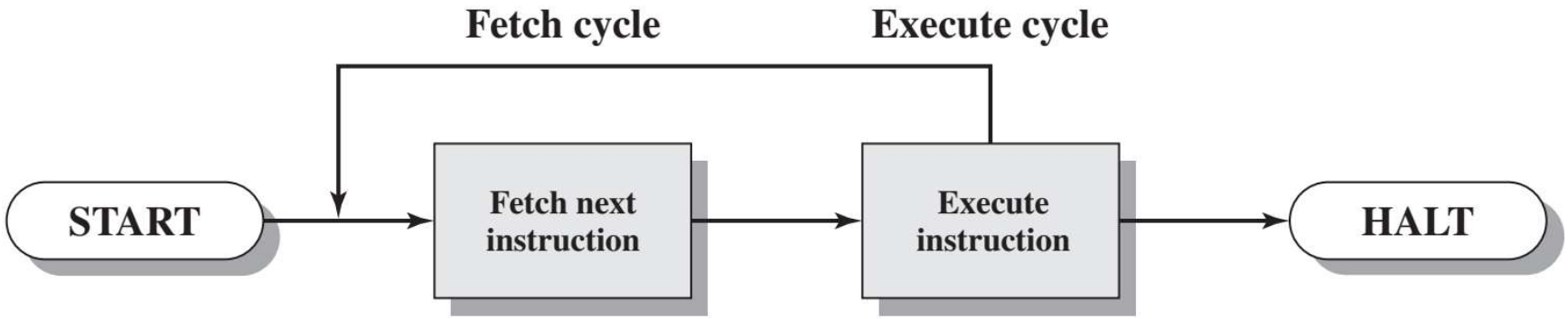
VON NEUMANN ARCHITECTURE	HARVARD ARCHITECTURE
It is ancient computer architecture based on stored program computer concept.	It is modern computer architecture based on Harvard Mark I relay based model.
Same physical memory address is used for instructions and data.	Separate physical memory address is used for instructions and data.
There is common bus for data and instruction transfer.	Separate buses are used for transferring data and instruction.
Two clock cycles are required to execute single instruction.	An instruction is executed in a single cycle.
It is cheaper in cost.	It is costly than von neumann architecture.
CPU can not access instructions and read/write at the same time.	CPU can access instructions and read/write at the same time.
It is used in personal computers and small computers because we need to change the program in RAM.	It is used in micro controllers and signal processing. In these applications the program is already dumped in to the ROM.



Harvard Machine

Computer function

- Basic function performed by the computer is execution of the program → set of instructions stored in memory
- Instruction processing consists of two steps
  - Processor reads / Fetches
  - Executing the fetched instruction
- Processing required for a single instruction is known as instruction cycle



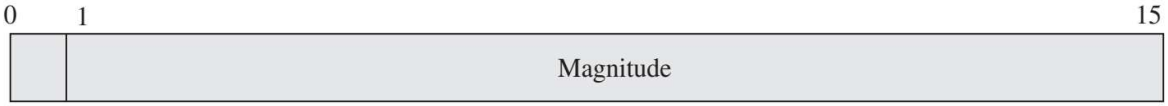
Instruction fetch and execute

- At the beginning of each instruction cycle, the processor fetches an instruction from memory
- A register called the program counter (PC) holds the address of the instruction to be fetched next
- Unless told otherwise, the processor always increments the PC after each instruction fetch so that it will fetch the next instruction in sequence i.e. the instruction located in the next higher memory address
- The fetched instruction is loaded into a register in the processor known as the instruction register (IR)
- The instruction contains bits that specify the action the processor needs to take
- The processor interprets the instruction and performs the required action

Example

- Suppose a processor contains a single data register, called an accumulator (AC)
- Both instructions and data are 16 bits long
- The instruction format provides 4 bits for the opcode → 16 opcodes and 4096 (4K) memory

(a) Instruction format



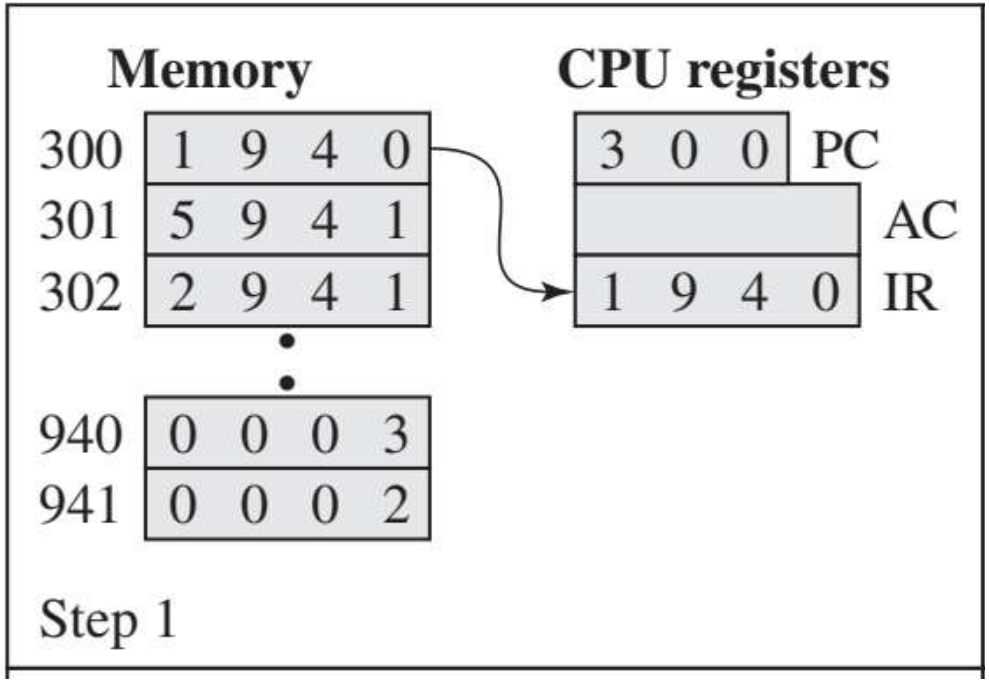
(b) Integer format

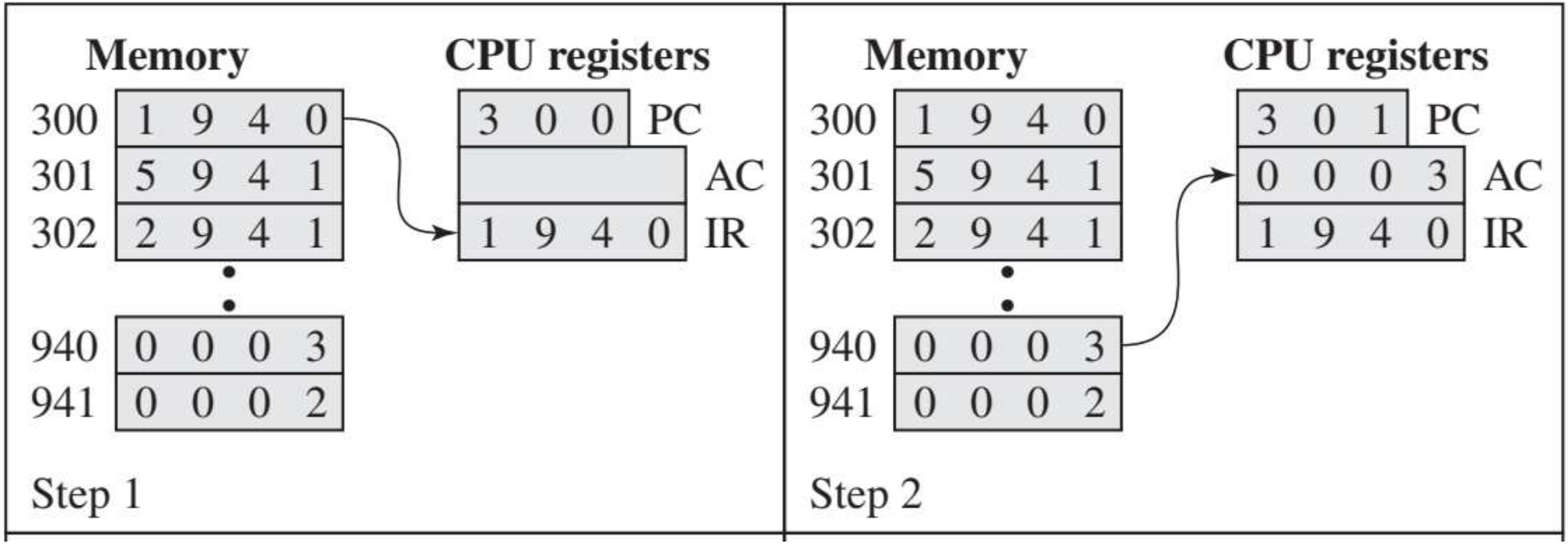
Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

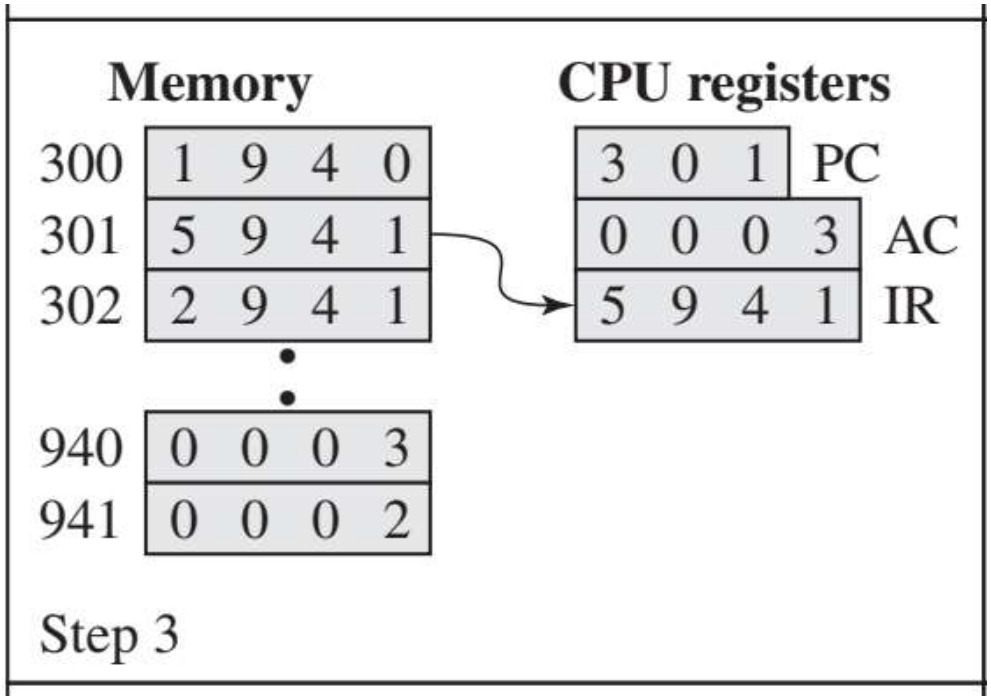
(c) Internal CPU registers

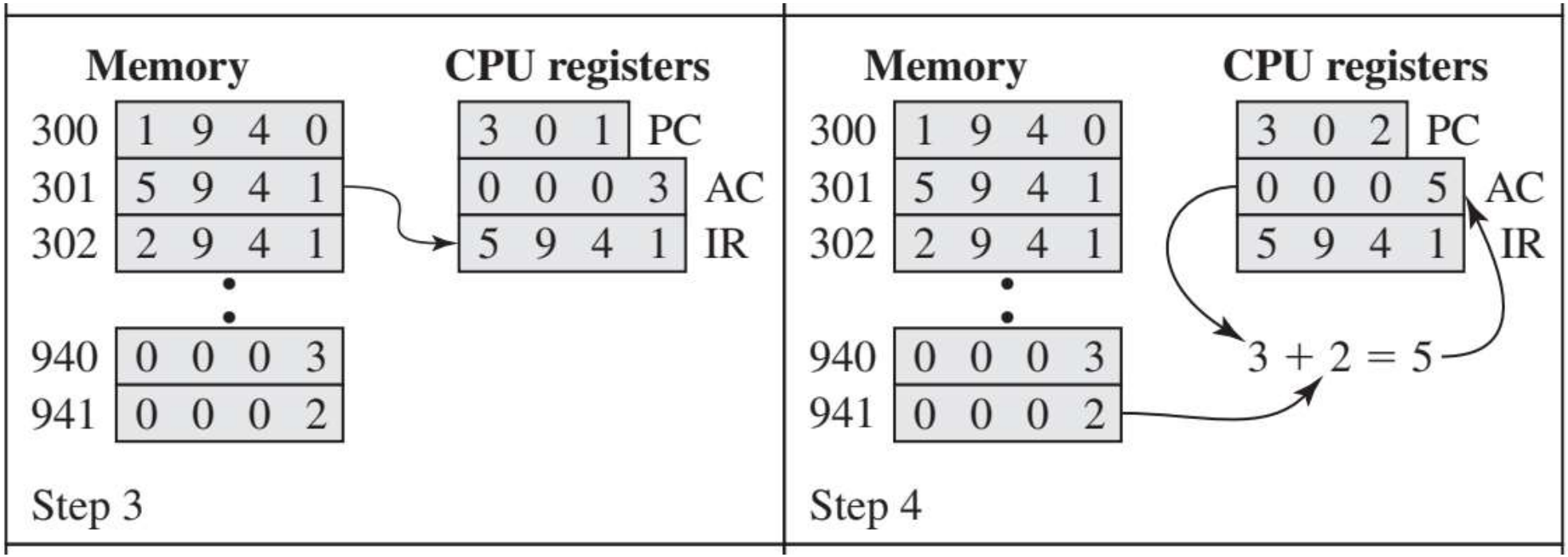
0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

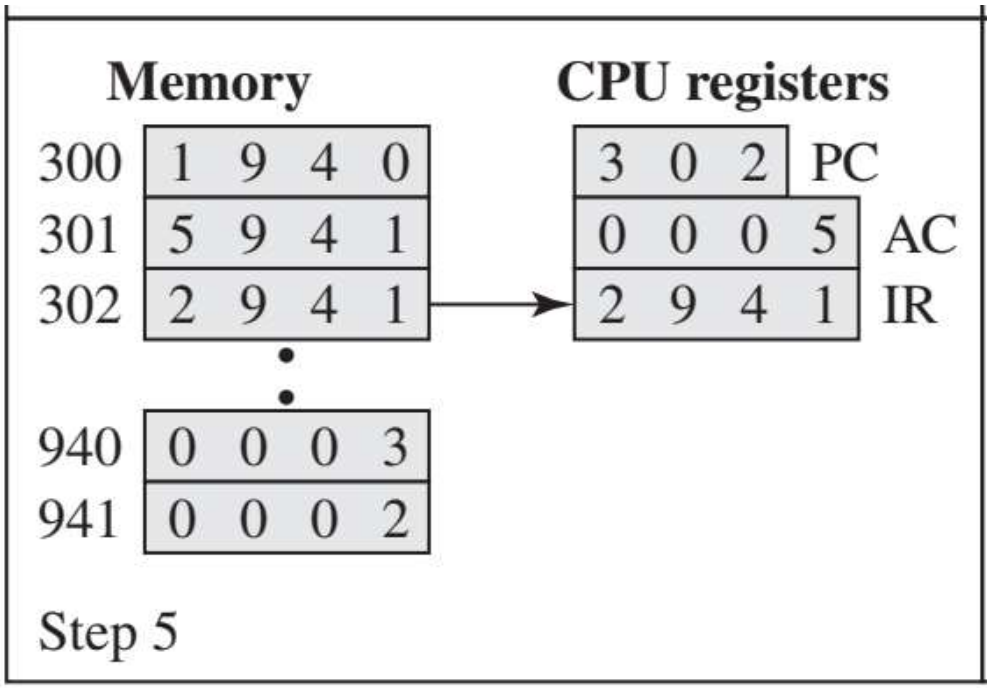
(d) Partial list of opcodes

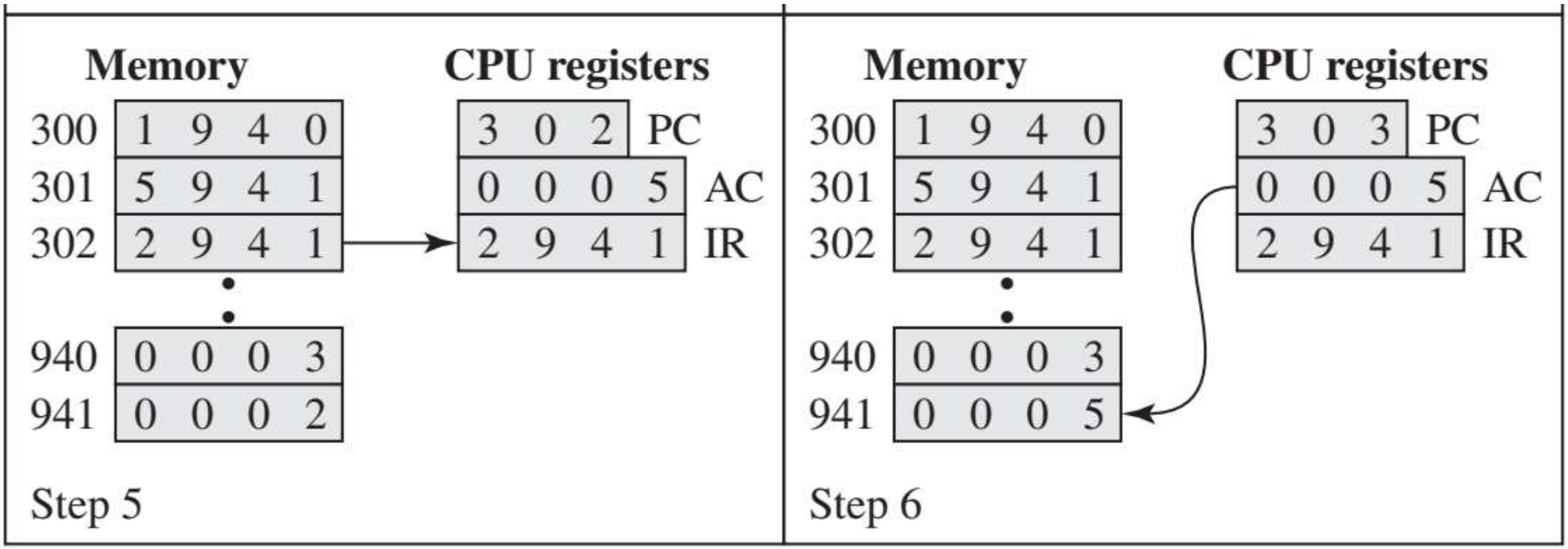

Memory
CPU registers
300 1 9 4 0
301 5 9 4 1
302 2 9 4 1
940 0 0 0 3
941 0 0 0 2
3 0 0 PC
AC
1 9 4 0 IR
Step 1


Memory
CPU registers
300 1 9 4 0
301 5 9 4 1
302 2 9 4 1
940 0 0 0 3
941 0 0 0 2
3 0 0 PC
AC
1 9 4 0 IR
Step 1
Memory
CPU registers
300 1 9 4 0
301 5 9 4 1
302 2 9 4 1
940 0 0 0 3
941 0 0 0 2
3 0 1 PC
0 0 0 3 AC
1 9 4 0 IR
Step 2


Memory
CPU registers
300 1 9 4 0
301 5 9 4 1
302 2 9 4 1
940 0 0 0 3
941 0 0 0 2
3 0 1 PC
0 0 0 3 AC
5 9 4 1 IR
Step 3

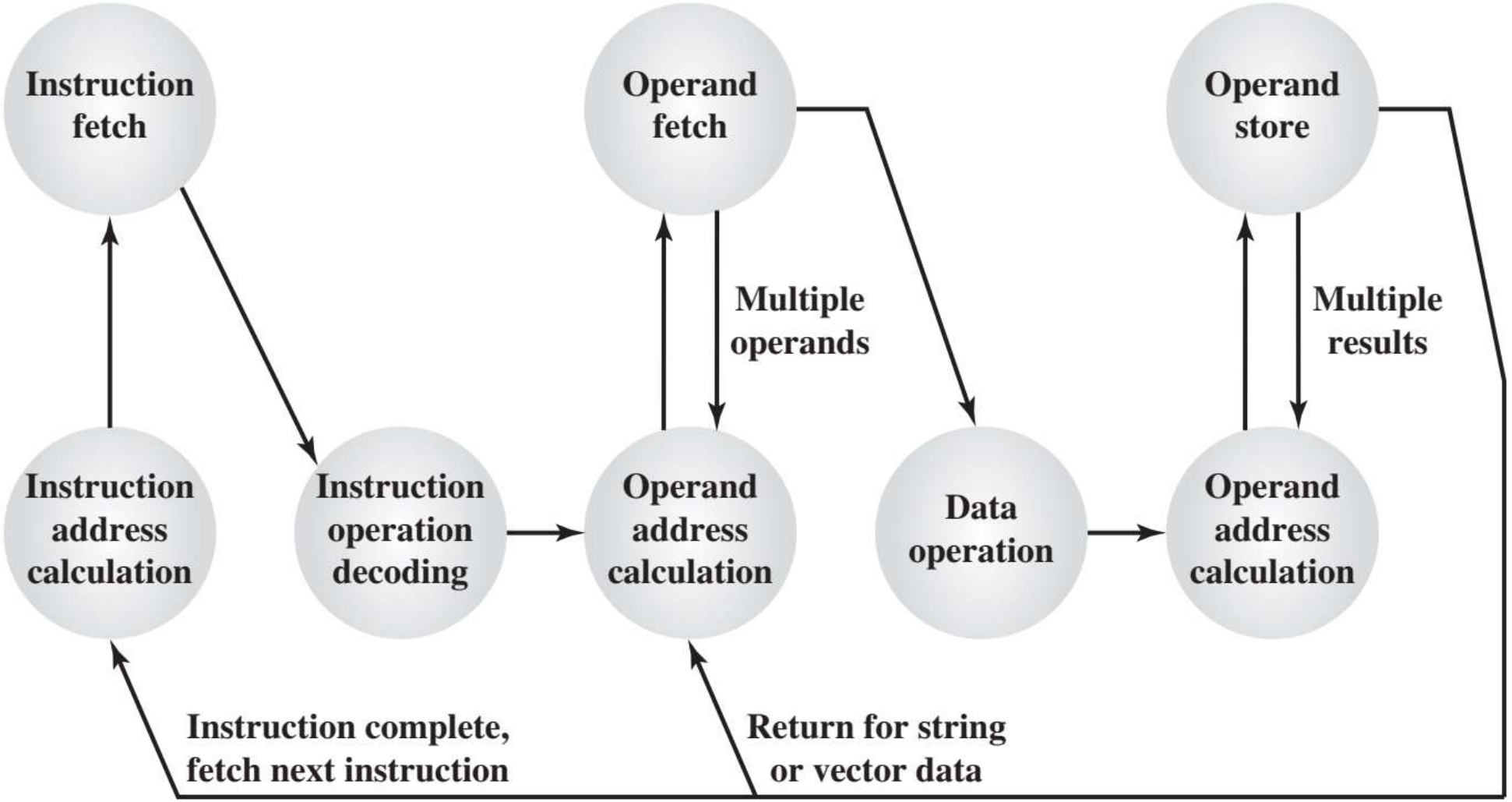

Memory
CPU registers
300 1 9 4 0
301 5 9 4 1
302 2 9 4 1
940 0 0 0 3
941 0 0 0 2
3 0 1 PC
0 0 0 3 AC
5 9 4 1 IR
Step 3
Memory
CPU registers
300 1 9 4 0
301 5 9 4 1
302 2 9 4 1
940 0 0 0 3
941 0 0 0 2
3 0 2 PC
0 0 0 5 AC
5 9 4 1 IR
3 + 2 = 5
Step 4


Memory
CPU registers
300 1 9 4 0
301 5 9 4 1
302 2 9 4 1
940 0 0 0 3
941 0 0 0 2
3 0 2 PC
0 0 0 5 AC
2 9 4 1 IR
Step 5


Memory
CPU registers
300 1 9 4 0
301 5 9 4 1
302 2 9 4 1
940 0 0 0 3
941 0 0 0 2
3 0 2 PC
0 0 0 5 AC
2 9 4 1 IR
Step 5
Memory
CPU registers
300 1 9 4 0
301 5 9 4 1
302 2 9 4 1
940 0 0 0 3
941 0 0 0 5
3 0 3 PC
0 0 0 5 AC
2 9 4 1 IR
Step 6

Example

- In this example, three instruction cycles, each consisting of a fetch cycle and an execute cycle, are needed to add the contents of location 940 to the contents of 941.
- With a more complex set of instructions, fewer cycles would be needed.
- Some older processors, for example, included instructions that contain more than one memory address.
- For example, the PDP-11 processor expressed symbolically as ADD B,A
  - Fetch the ADD instruction.
  - Read the contents of memory location A into the processor.
  - Read the contents of memory location B into the processor. In order that the contents of A are not lost, the processor must have at least two registers for storing memory values, rather than a single accumulator.
  - Add the two values.
  - Write the result from the processor to memory location A.


Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data operation
Operand address calculation
Instruction complete, fetch next instruction
Return for string or vector data

States

- **Instruction address calculation (iac):** Determine the address of the next instruction to be executed.
- **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.

States

- The **oac** state appears twice, because an instruction may involve a read, a write, or both.
- Example → ADD A,B results in the following sequence of states: iac, if, iod, oac, of, oac, of, do, oac, os.
- A single instruction can specify an operation to be performed on an array of numbers or a string of characters → involves repetitive operand fetch and store operations

IAS structure

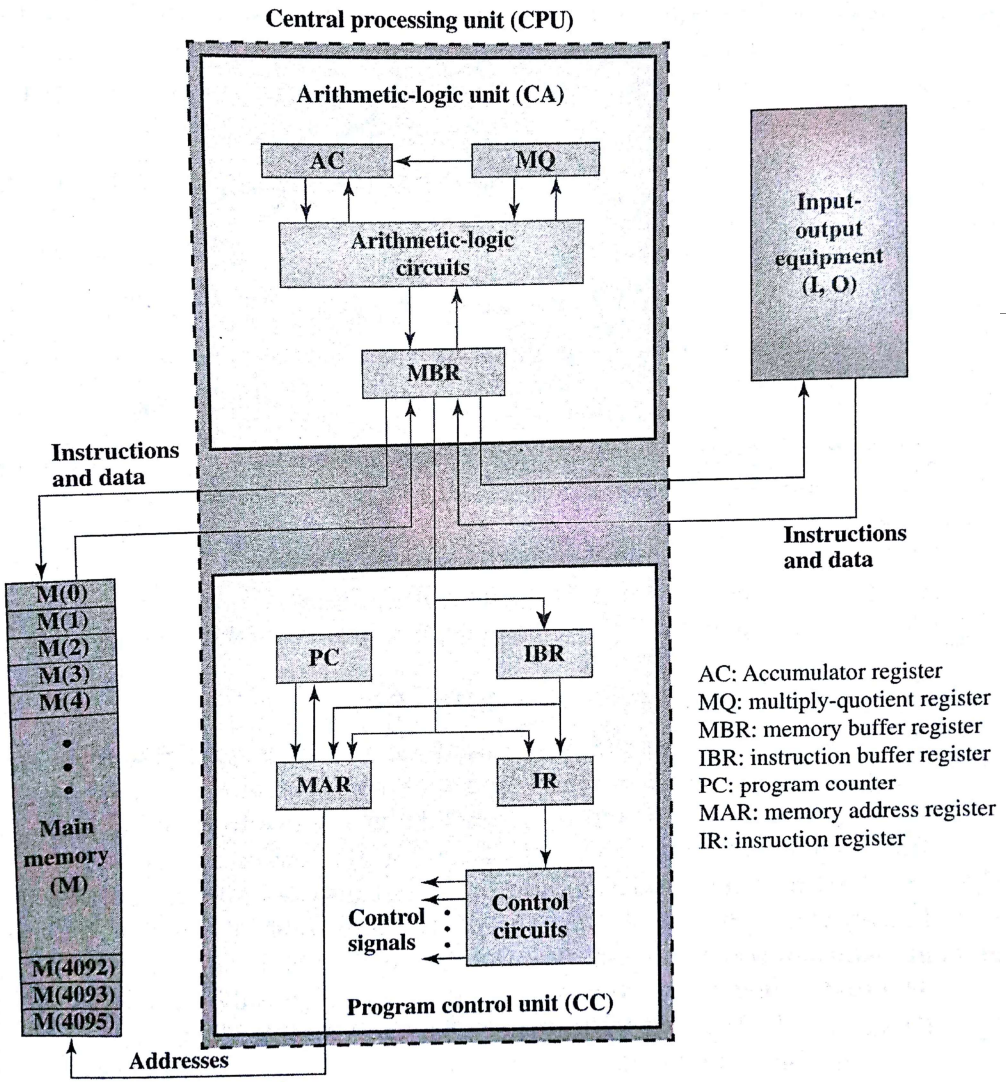
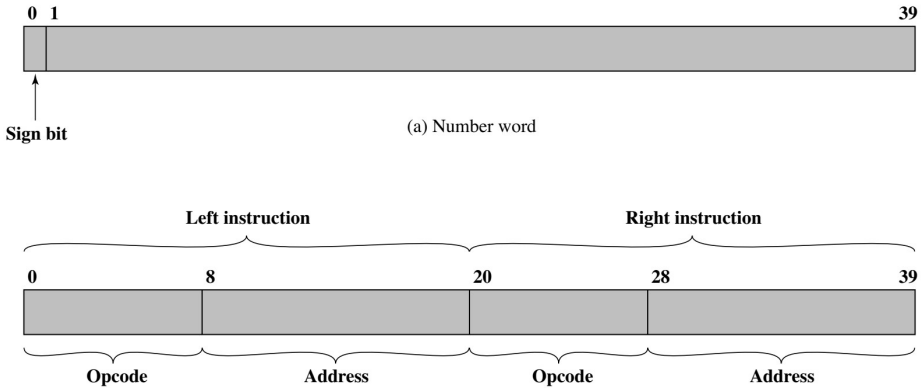
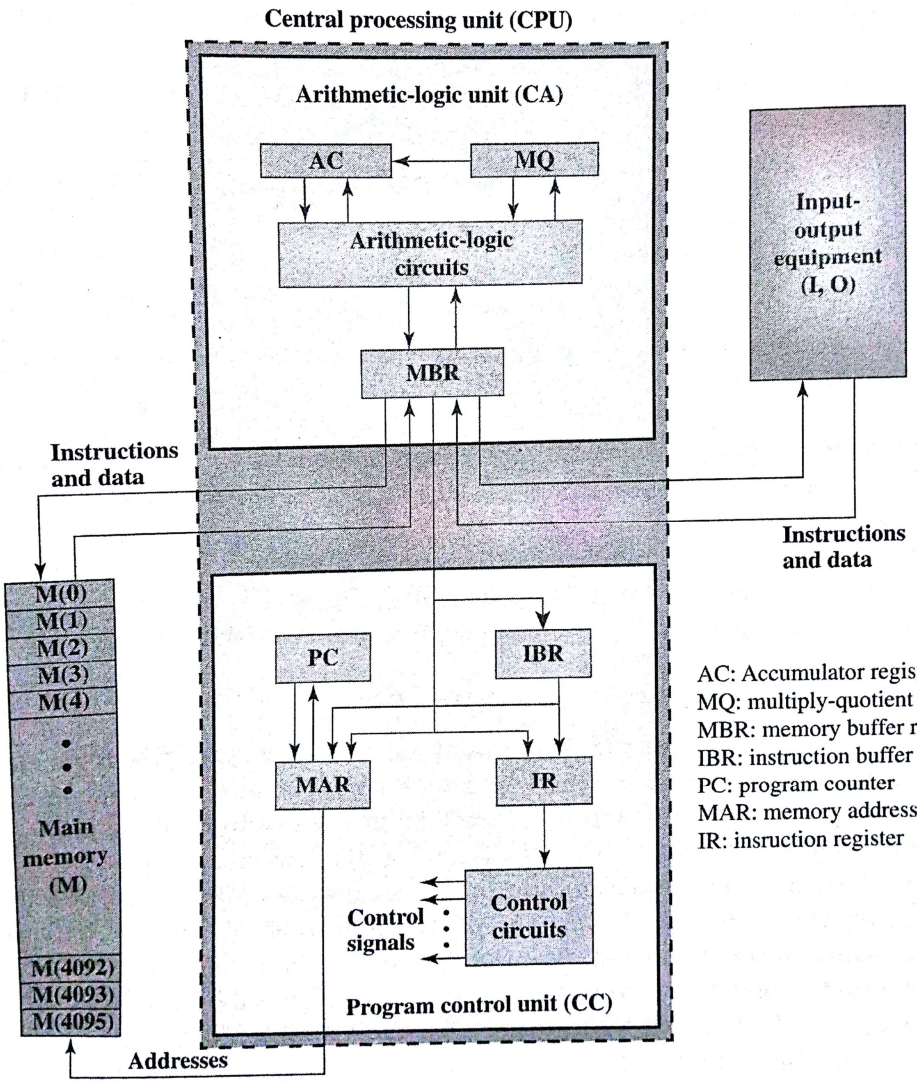


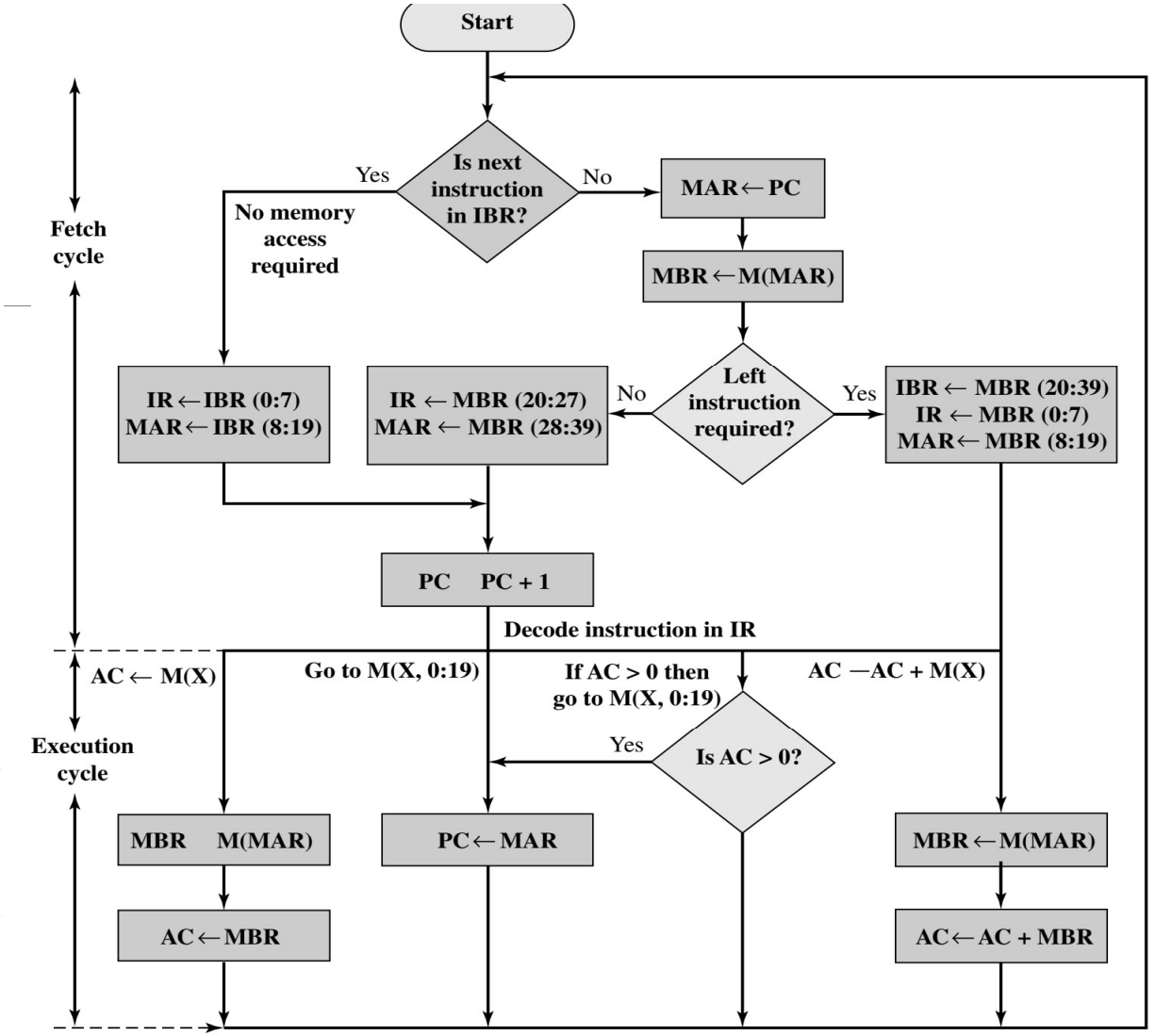
Figure 1.6 IAS Structure





AC: Accumulator register
MQ: multiply-quotient register
MBR: memory buffer register
IBR: instruction buffer register
PC: program counter
MAR: memory address register
IR: insruction register

Figure 1.6 IAS Structure



M(X) = contents of memory location whose address is X
(i:j) = bits i through j

Instruction Type	Opcode	Symbolic Representation	Description
Data transfer	00001010	LOAD MQ	Transfer contents of register MQ to the accumulator AC
	00001001	LOAD MQ,M(X)	Transfer contents of memory location X to MQ
	00100001	STOR M(X)	Transfer contents of accumulator to memory location X
	00000001	LOAD M(X)	Transfer M(X) to the accumulator
	00000010	LOAD −M(X)	Transfer −M(X) to the accumulator
	00000011	LOAD |M(X)|	Transfer absolute value of M(X) to the accumulator
	00000100	LOAD −|M(X)|	Transfer −|M(X)| to the accumulator
Unconditional branch	00001101	JUMP M(X,0:19)	Take next instruction from left half of M(X)
	00001110	JUMP M(X,20:39)	Take next instruction from right half of M(X)
Conditional branch	00001111	JUMP+ M(X,0:19)	If number in the accumulator is nonnegative, take next instruction from left half of M(X)
	00010000	JUMP+ M(X,20:39)	If number in the accumulator is nonnegative, take next instruction from right half of M(X)
Arithmetic	00000101	ADD M(X)	Add M(X) to AC; put the result in AC
	00000111	ADD |M(X)|	Add |M(X)| to AC; put the result in AC
	00000110	SUB M(X)	Subtract M(X) from AC; put the result in AC
	00001000	SUB |M(X)|	Subtract |M(X)| from AC; put the remainder in AC
	00001011	MUL M(X)	Multiply M(X) by MQ; put most significant bits of result in AC, put least significant bits in MQ
	00001100	DIV M(X)	Divide AC by M(X); put the quotient in MQ and the remainder in AC
	00010100	LSH	Multiply accumulator by 2; i.e., shift left one bit position
	00010101	RSH	Divide accumulator by 2; i.e., shift right one position
Address modify	00010010	STOR M(X,8:19)	Replace left address field at M(X) by 12 rightmost bits of AC
	00010011	STOR M(X,28:39)	Replace right address field at M(X) by 12 rightmost bits of AC

Example of addition

1. LOAD M(X) 500, ADD M(X) 501 (PC=1)

MAR<PC

MBR<M[MAR]

IBR<MBR[20:39]

IR<MBR[0:7]

MAR<MBR[8:19]

MBR<M[MAR]

AC<MBR

IR<IBR[0:7]

MAR<IBR[8:19]

PC<PC+1

MBR<M[MAR]

AC<AC+MBR

2. STOR M(X) 500, OTHER INSTRUCTION (PC=2)

MAR<PC

MBR<M[MAR]

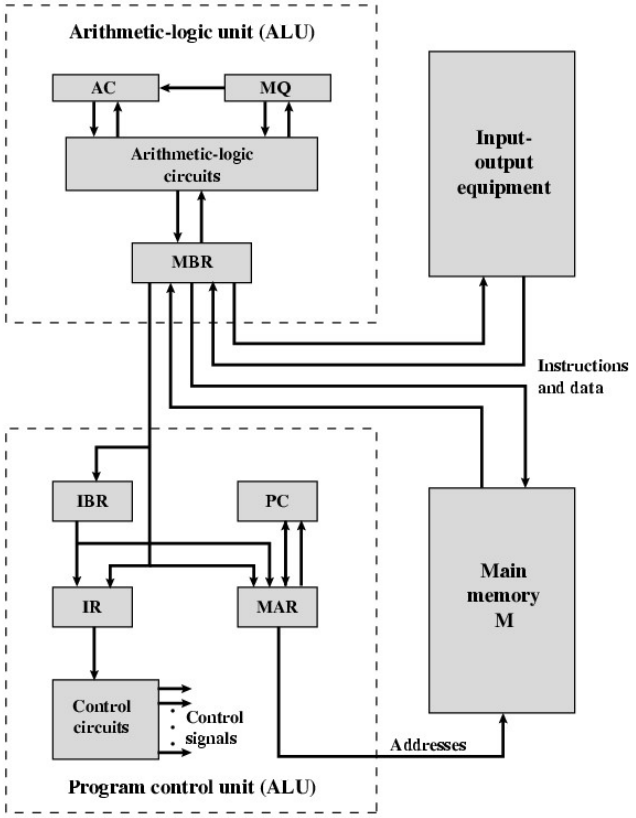
IBR<MBR[20:39]

IR<MBR[0:7]

MAR<MBR[8:19]

MBR<AC

M[MAR]<MBR



P1 Given the memory contents of the IAS computer shown below,

Address	Contents
08A	010FA210FB
08B	010FA0F08D
08C	020FA210FB

show the assembly language code for the program, starting at address 08A. Explain what this program does.

Instruction Type	Opcode	Symbolic Representation	Description
Data transfer	00001010	LOAD MQ	Transfer contents of register MQ to the accumulator AC
	00001001	LOAD MQ,M(X)	Transfer contents of memory location X to MQ
	00100001	STOR M(X)	Transfer contents of accumulator to memory location X
	00000001	LOAD M(X)	Transfer M(X) to the accumulator
	00000010	LOAD −M(X)	Transfer −M(X) to the accumulator
	00000011	LOAD |M(X)|	Transfer absolute value of M(X) to the accumulator
	00000100	LOAD −|M(X)|	Transfer −|M(X)| to the accumulator
Unconditional branch	00001101	JUMP M(X,0:19)	Take next instruction from left half of M(X)
	00001110	JUMP M(X,20:39)	Take next instruction from right half of M(X)
Conditional branch	00001111	JUMP+ M(X,0:19)	If number in the accumulator is nonnegative, take next instruction from left half of M(X)
	00010000	JUMP+ M(X,20:39)	If number in the accumulator is nonnegative, take next instruction from right half of M(X)
Arithmetic	00000101	ADD M(X)	Add M(X) to AC; put the result in AC
	00000111	ADD |M(X)|	Add |M(X)| to AC; put the result in AC
	00000110	SUB M(X)	Subtract M(X) from AC; put the result in AC
	00001000	SUB |M(X)|	Subtract |M(X)| from AC; put the remainder in AC
	00001011	MUL M(X)	Multiply M(X) by MQ; put most significant bits of result in AC, put least significant bits in MQ
	00001100	DIV M(X)	Divide AC by M(X); put the quotient in MQ and the remainder in AC
	00010100	LSH	Multiply accumulator by 2; i.e., shift left one bit position
	00010101	RSH	Divide accumulator by 2; i.e., shift right one position
Address modify	00010010	STOR M(X,8:19)	Replace left address field at M(X) by 12 rightmost bits of AC
	00010011	STOR M(X,28:39)	Replace right address field at M(X) by 12 rightmost bits of AC

P2

On the IAS, what would the machine code instruction look like to load the contents of memory address 2?

How many trips to memory does the CPU need to make to complete this instruction during the instruction cycle?

P3

Let $\mathbf{A} = A(1), A(2), \ldots, A(1000)$ and $\mathbf{B} = B(1), B(2), \ldots, B(1000)$ be two vectors (one-dimensional arrays) comprising 1000 numbers each that are to be added to form an array C such that $C(I) = A(I) + B(I)$ for $I = 1, 2, \ldots, 1000$. Using the IAS instruction set, write a program for this problem. Ignore the fact that the IAS was designed to have only 1000 words of storage.

Answer

P1

This program will store the absolute value of content at memory location 0FA into memory location 0FB.

Address	Contents
08A	LOAD M(0FA)
	STOR M(0FB)
08B	LOAD M(0FA)
	JUMP +M(08D)
08C	LOAD –M(0FA)
	STOR M(0FB)
08D	

P2

OPCODE	OPERAND
00000001	000000000010

First, the CPU must make access memory to fetch the instruction. The instruction contains the address of the data we want to load. During the execute phase accesses memory to load the data value located at that address for a total of two trips to memory.

Answer

P3

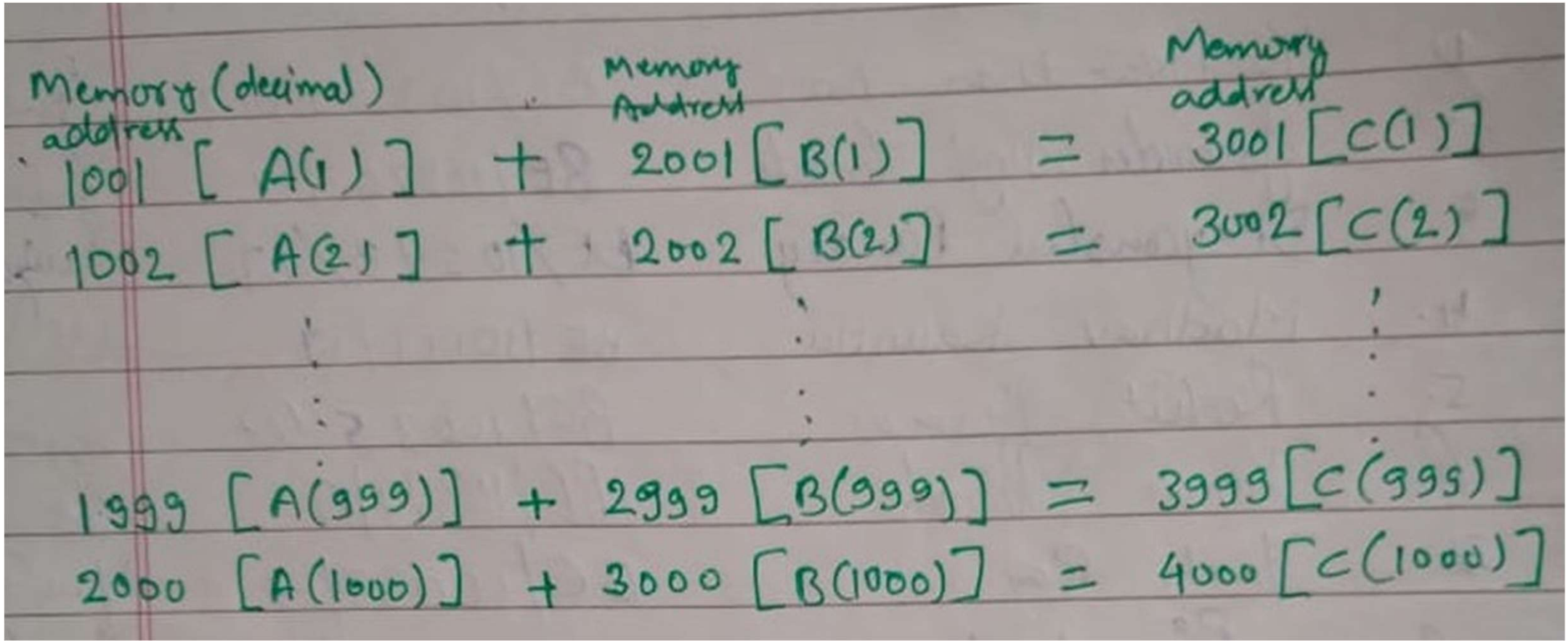
The vectors A, B, and C are each stored in 1,000 continuous locations in memory, beginning at locations 1001, 2001, and 3001, respectively.

The program begins with the left half of location 3.

A counting variable N is set to 999 and decremented after each step until it reaches -1.

Thus, the vectors are processed from high location to low location.

Location	Instruction	Comments
0	999	Constant (count N)
1	1	Constant
2	1000	Constant
3L	LOAD M(2000)	Transfer A(I) to AC
3R	ADD M(3000)	Compute A(I) + B(I)
4L	STOR M(4000)	Transfer sum to C(I)
4R	LOAD M(0)	Load count N
5L	SUB M(1)	Decrement N by 1
5R	JUMP+ M(6, 20:39)	Test N and branch to 6R if nonnegative
6L	JUMP M(6, 0:19)	Halt
6R	STOR M(0)	Update N
7L	ADD M(1)	Increment AC by 1
7R	ADD M(2)	
8L	STOR M(3, 8:19)	Modify address in 3L
8R	ADD M(2)	
9L	STOR M(3, 28:39)	Modify address in 3R
9R	ADD M(2)	
10L	STOR M(4, 8:19)	Modify address in 4L
10R	JUMP M(3, 0:19)	Branch to 3L


Memory (decimal) address
Memory Address
Memory address
1001 [A(1)] + 2001 [B(1)] = 3001 [C(1)]
1002 [A(2)] + 2002 [B(2)] = 3002 [C(2)]
1999 [A(999)] + 2999 [B(999)] = 3999 [C(999)]
2000 [A(1000)] + 3000 [B(1000)] = 4000 [C(1000)]

Example 3

```
main ()
{
int a=15, b=5, c;
if (a >= b)
   c = a – b;
else
   c = a + b;
}
```

```
0 15   a
1 5    b
2 c

4 .  If (a >=b)
4L          LOAD M(0)
4R          SUB M(1)
5L          JUMP+ M(6, 0:19)
5R          JUMP M(8, 0:19)
6.  true, c=a-b
6L          LOAD M(0)
6R          SUB M(1)
7L          STOR M(2)
7R          JUMP M(9, 20:39)
8.  false c = a+b
8L          LOAD M(0)
8R          ADD M(1)
9L          STOR M(2)
9R          HALT
```

Example 3 (continued)

- Optimized

```
0 15      a
1 5       b
2 c
3
4L        LOAD M(0)
4R        SUB M(1)
5L        JUMP+ M(6, 20:39)
5R        LOAD M(0)
6L        ADD M(1)
6R        STOR M(2)
7L        HALT
```

Example3 (with a > b)

```
main () {
int a=15, b=5, c;
if (a > b)
   c = a – b;
else
   c = a + b;
}
```

```
0 15    a
1 5     b
2 c
3 1
4 begin
5 . a > b
5 load M(0)
6 sub M(1)
7 sub M(3)
8 jump+ M(10)
9 jump M(14)
10 . True, c = a- b
10 load M(0)
11 sub M(1)
12 stor M(2)
13 jump M(17)
14 . False, c = a + b
14 load M(0)
15 add M(1)
16 stor M(2)
17 halt
```

Example 6

```
main () {
int a=2, b=2, I;
I = 1;
while (I < 10) {
   a = a +b;
   I = I +1;
}
}
```

Give it a try.

Example 6 (continued)

```
main () {
int a=2, b=2, I;
I = 1;
while (I < 10) {
   a = a +b;
   I = I +1;
}
}
```

```
0           1
1           10
2           2       a
3           2       b
4           i
7 . I =1
7L          LOAD M(0)
7R          STOR M(4)
8 . while (I < 10)
8L          LOAD M(4)
8R          SUB M(1)
9L          JUMP+ M(13,0:19)
9 . a = a +b
9R          LOAD M(2)
10L         ADD M(3)
10R         STOR M(2)
11 . I=I+1
11L         LOAD M(4)
11R         ADD M(0)
12L         STOR M(4)
12R         JUMP M(8,0:19)
13L         HALT
```

Problem

Write a program to add 5 numbers located at consecutive memory locations starting at address 500. Add another 5 numbers located at consecutive memory locations starting at address 600. Subtract the lower value from the higher value and store it at location having address 700.

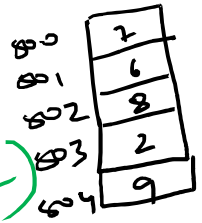
Problem

Write a program to add 5 numbers located at consecutive memory locations starting at address 500. Add another 5 numbers located at consecutive memory locations starting at address 600. Subtract the lower value from the higher value and store it at location having address 700.

Sol:

800: LOAD M(500) [Left Instr]
800: ADD M(501) [Right Instr]
801: ADD M(502) [Left Instr]
801: ADD M(503) [Right Instr]
802: ADD M(504) [Left Instr]
802: STOR M(701) [Right Instr]
803: LOAD M(600) [Left Instr]
803: ADD M(601) [Right Instr]
804: ADD M(602) [Left Instr]
804: ADD M(603) [Right Instr]
805: ADD M(604) [Left Instr]
805: SUB M(701) [Right Instr]
806: JUMP+ M(807,20:39) [Left Instr]
806: STOR M(702) [Right Instr]
807: LOAD -M(702) [Left Instr]
807: STOR M(700) [Right Instr]

IAS computer – Sample problem: There are 5 values at memory location 500. Find the greatest number among these values and store it in 700 mem location.



600 | 8 |

700 | 9 |

Ans:-

800: Load M(500) → AC = 7

800: SUB M(501) → AC = 7 − 6, AC = 1

801: Jump+ M(802, 20:39)

~~801: LOAd M(501)~~

~~802: Jump M(803, 0:19)~~

802: Load M(500) → AC = 7

803: Store M(600) → M(600) ⇐ AC 7

803: SUB M(502) → AC = 7 − 8 = −1

804: Jump+ M(805, 20:39)

804: Load M(502) → AC = 8

805: Jump M(806, 0:19)

~~805: Load M(600)~~

806: Store M(600) → M(600) = 8

806: SUB M(503) → AC = 8 − 2 = 6

807: Jump+ M(808, 20:39)

8087: Load M(503) ✗

808: Jump M(809, 0:19) ✗

808: Load M(600) AC = 8

809: Store M(600) M(600) = 8

809: SUB M(504) → AC = 8 − 9 = −1

810: Jump+ M(811, 20:39)

810: Load M(504) AC = 9

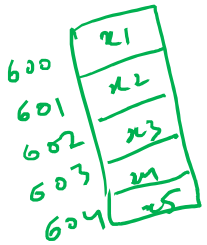
811: Jump M(812, 0:19)

811: Load M(600) ✗

812: Store M(700) → M(700) = AC = 9

IAS Computer - Sample problem 2:- Write a program to multiply 5 nos. stored consecutively at mem location 600. and store the results in mem location 700.

IAS Computer - Sample problem 2:- Write a program to multiply 5 nos. stored consecutively at mem location 600. and store the results in mem location 700.



Ans:-

```
900: Load  M(600), MQ   (LI)
900: MUL   M(601)       (RI)
901: MUL   M(602)       (LI)
901: MUL   M(603)       (RI)
902: MUL   M(604)       (LI)
902: Store M(700)       (RI)
903: Load  MQ           (LI)
903: Store M(701)       (RI)
```

Q: Write an assembly language program for the IAS computer to find the MAC operation of 5 nos located at 600h mem location.

Q: Write an assembly language program for the IAS computer to find the MAC operation of 5 nos located at 600h mem location.

MAC → Multiply & Accumulate.

Ans:-

```
900: LOAD  MQ, M(600)
     MUL   M(601)
901: LOAD  MQ
     MUL   M(602)
902: STOR  M(700)
     LOAD  MQ
903: ADD   M(700)
     STOR  M(700)
```

```
904: LOAD  MQ, M(700)
     MUL   M(603)
905: LOAD  MQ
     ADD   M(700)
906: STOR  M(700)
     LOAD  MQ, M(700)
907: MUL   M(604)
     LOAD  MQ
```

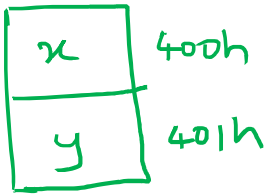
```
908: ADD   M(700)
     STOR  M(700)
```

Assumption:- MUL instruction is assumed to produce only 40-bit values.

Q:- Write an ALP for the IAS Computer to find the largest among 2 nos.

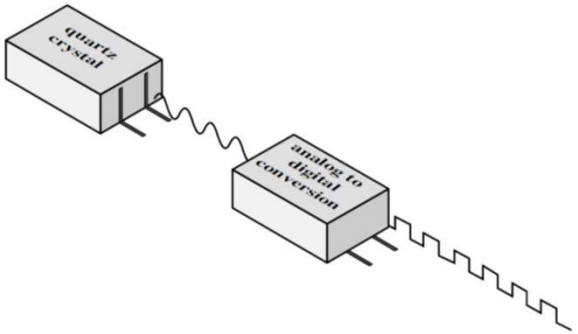
Ans:-

```
900: LOAD   M(400)
     SUB    M(401)
901: JumpT  M(903, 0:19)
     LOAD   M(401)
902: STOR   M(700)
     Jump   M(904 : 0:19)
903: LOAD   M(400)
     STOR   M(700)
904: EXIT
```



Performance Assessment

- Performance is one of the key parameters to consider, along with cost, size, security, reliability, and, in some cases power consumption.
- Application performance depends not just on the raw speed of the processor, but on the **instruction set**, choice of **implementation language**, **efficiency of the compiler**, and **skill of the programming.**
- Clock Speed
  - The System Clock: The most fundamental level, the speed of a processor is dictated by the pulse frequency produced by the clock, measured in cycles per second, or Hertz (Hz).
  - Clock signals are generated by a quartz crystal
  - The rate of pulses is known as the **clock rate**, or **clock speed**
  - One increment, or pulse, of the clock is referred to as a **clock cycle**, or a **clock tick**.
  - The time between pulses is the **cycle time**.
  - For example, a 1-GHz processor receives 1 billion pulses per second.



Performance Assessment

- Millions of instructions per second (MIPS) or MIPS rate
  - The System Clock: The most fundamental level, the speed of a processor is dictated by the pulse frequency produced by the clock, measured in cycles per second, or Hertz (Hz).

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

CPI: Cycle per instruction

f: Clock frequency (number of cycle per second)

Example: Consider a program having 500 million instructions, running on a 400 MHz processor. The program consists of three major types of instructions - ALU related, load/store, and branching. These instructions require 1, 2, and 4 CPI with a instruction mix of 60, 30, and 10% respectively in the program. Estimate the MIPS of the processor.

Solution: CPI=0.6x1+0.3x2+0.1x4=1.6

MIPS=$(400x10^6)/(1.6x10^6)$=250 MIPS

  - Similarly there are other performance parameters.

Performance Assessment :- (Sample problem 1) A program has 170 billion instructions. The processor has a clock period of 90ns. The CPI is 2, 1, 4, 3 for ALU, load, store, branching with a mix of 50%, 20%, 15%, 15% respectively. Find MIPS rate and also find the time reqd to execute the program.

Sol:-

Performance Assessment :- (Sample problem 1) A program has 270 billion instructions. The processor has a clock period of 90ns. The CPI is 2, 1, 4, 3 for ALU, load, store, branching with a mix of 50%, 20%, 15%, 15% respectively. Find MIPS rate and also find the time reqd to execute the program.

Sol:-

$$t = \frac{270 \times 10^9 \times 10^3}{49.382222 \times 10^6}$$

$$\sim \boxed{5467.5 \text{ sec}}$$

$$CPI = 2 \times 0.5 + 1 \times 0.2 + 4 \times 0.15 + 3 \times 0.15$$

$$= 2.25$$

$$\text{MIPS rate} = \frac{f}{CPI \times 10^6} = \frac{111.11 \times 10^6}{2.25 \times 10^6}$$

$$= \boxed{49.382}$$

Performance Assessment of processor : (Sample problem 2)

A program has 195 trillion instructions. The cycles required for each instruction is different. Assume that there are 60% ALU instr, 10% load/store instr, 10% branch instruct, 10% string instructs, 10% I/O instr. The cycles required to execute these instructions are 3, 2, 1, 4, 5, respectively. Each cycle is of 20 μsec ~~each~~ each. Compute the MIPS rate & total exec time.

Sol:- No of inst = 195×10^{12}

$t = 20\ \mu sec$

$f = \frac{1}{20} \times 10^6$

$= 0.5 \times 10^5$

$= \boxed{50\ KHz}$

$CPI = 3 \times 0.6 + 2 \times 0.1 + 1 \times 0.1 + 4 \times 0.1 + 5 \times 0.1$

$= 18 + 0.1(12)$

$= \boxed{3}$

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

$= \frac{50 \times 10^3}{3 \times 10^6}$ 100

$= 0.016666667 \times 10^6$

$t = \frac{195 \times 10^{12}}{16666}$

$\boxed{11.7 \times 10^9\ sec}$

16666 ~~667~~ instr per sec

Q:- Consider the MIPS rate to be '110'. Let the CPI of the instructions I, ALU, Load/store, Branching be 2, 3, 4 for 60%, 20% & 20% respectively. Calculate the maximum frequency with which the processor can run?.

Ans:-

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

$$CPI = 2 \times 0.6 + 3 \times 0.2 + 4 \times 0.2$$

$$= 2.6$$

$$f = MIPS \times CPI \times 10^6$$

$$= 110 \times 2.6 \times 10^6$$

$$\boxed{f = 286\ MHz}$$

Integer representation

- Arbitrary numbers can be represented with just the digits zero and one, the minus sign, and the period

$$-1101.0101_2 = -13.3125_{10}$$

- For computer storage and processing → minus signs and periods cannot be used
- Only binary digits (0 and 1) may be used to represent numbers
- If limited to non-negative integers, the representation is straightforward

An 8-bit word can represent the numbers from 0 to 255, including

$$\begin{aligned} 00000000 &= 0 \\ 00000001 &= 1 \\ 00101001 &= 41 \\ 10000000 &= 128 \\ 11111111 &= 255 \end{aligned}$$

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

Sign-Magnitude representation

- There are several conventions used to represent negative as well as positive integers → all of which involve treating the most significant (leftmost) bit in the word as a sign bit.
- If the sign bit is 0 → the number is positive; if the sign bit is 1 → the number is negative
- The simplest form of representation that employs a sign bit is the sign-magnitude representation
- In an n-bit word, the rightmost bits hold the magnitude of the integer

$$
\begin{aligned}
+\,18 &= 00010010 \\
-\,18 &= 10010010 \quad \text{(sign magnitude)}
\end{aligned}
$$

Sign-Magnitude representation

- General case

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases}$$

- DRAWBACKS
- Addition and subtraction requires consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation
- Another drawback is that there are two representations of 0

$$+0_{10} = 00000000$$
$$-0_{10} = 10000000$$

Sign - Magnitude representation (sample problems):-

(a) −127 (b) +63 (c) −219 (d) −66354 (e) +23655

(f) 1010111000 (g) 0101001110001 (h) 101110101010 (i) 11110011 (j) 00011010

Sign-Magnitude representation (sample problems):-

(a) −127 (b) +63 (c) −219 (d) −66354 (e) +23655

(f) 1010111000 (g) 0101001110001 (h) 1011101010 (i) 11110011 (j) 00011010

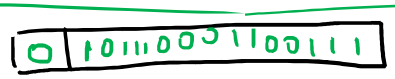
Sol:-

(a) −127 = 

(b) +63 = 

(c) −219 = 

(d) −66354 = 

(e) +23655 = 

(f) 

= −184

(g) 

= +2673

(h) 

= −938

(i) 

= −227

(j) 

= +26

2’s complement representation

- 2’s complement representation also uses the MSB as a sign bit

Range	-2^{n-1} through $2^{n-1} - 1$
Number of Representations of Zero	One
Negation	Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
Expansion of Bit Length	Add additional bit positions to the left and fill in with the value of the original sign bit.
Overflow Rule	If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.
Subtraction Rule	To subtract B from A, take the twos complement of B and add it to A.

2's complement representation

- Consider an n-bit integer, A, in 2's complement representation
- If A is positive, then the sign bit, a_{n-1} is zero. The remaining bits represent the magnitude of the number in the same fashion as for sign magnitude representation

$$A = \sum_{i=0}^{n-2} 2^i a_i \qquad \text{for } A \geq 0$$

- The number zero is identified as positive and therefore has a 0 sign bit and a magnitude of all 0s.
- The range of positive integers that may be represented is from 0 (all of the magnitude bits are 0) through $2^{n-1} - 1$ (all of the magnitude bits are 1).
- Any larger number would require more bits

2’s complement representation

- Now, for a negative number the sign bit, a_{n-1} is one
- The remaining $n - 1$ bits can take on any one of 2^{n-1} values
- Therefore, the range of negative integers that can be represented is from -1 to -2^{n-1}
- The weight of the most significant bit is -2^{n-1}
- This is the convention used in 2’s complement representation, yielding the following expression for negative numbers

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

2's complement representation

- For $a_{n-1} = 0$, the term $-2^{n-1}a_{n-1} = 0$ and the equation defines a non-negative integer
- When $a_{n-1} = 1$, the term 2^{n-1} is subtracted from the summation term, yielding a negative integer

Decimal Representation	Sign-Magnitude Representation	Twos Complement Representation
+8	–	–
+7	0111	0111
+6	0110	0110
+5	0101	0101
+4	0100	0100
+3	0011	0011
+2	0010	0010
+1	0001	0001
+0	0000	0000
−0	1000	–
−1	1001	1111
−2	1010	1110
−3	1011	1101
−4	1100	1100
−5	1101	1011
−6	1110	1010
−7	1111	1001
−8	–	1000

Working

- Consider an n-bit sequence of binary digits $a_{n-1}a_{n-2}\ldots a_1a_0$ interpreted as a 2's complement integer A, so that its value is

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

- If A is a positive number, the rule clearly works. Now, if A is negative and we want to construct an m-bit representation, with m > n. Then

$$A = -2^{m-1}a_{m-1} + \sum_{i=0}^{m-2} 2^i a_i$$

Working

$$-2^{m-1} + \sum_{i=0}^{m-2} 2^i a_i = -2^{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

$$-2^{m-1} + \sum_{i=n-1}^{m-2} 2^i a_i = -2^{n-1}$$

$$2^{n-1} + \sum_{i=n-1}^{m-2} 2^i a_i = 2^{m-1}$$

$$1 + \sum_{i=0}^{n-2} 2^i + \sum_{i=n-1}^{m-2} 2^i a_i = 1 + \sum_{i=0}^{m-2} 2^i$$

$$\sum_{i=n-1}^{m-2} 2^i a_i = \sum_{i=n-1}^{m-2} 2^i$$

$$\Rightarrow \quad a_{m-2} = \cdots = a_{n-2} = a_{n-1} = 1$$

2’s complement using value box representation

- The nature of 2’s complement representation is a value box, in which the value on the far right in the box is $1\ (2^0)$
- Each succeeding position to the left is double in value, until the leftmost position, which is negated
- The most negative 2’s complement number that can be represented is -2^{n-1} i.e. if any of the bits other than the sign bit is one, it adds a positive amount to the number

-128	64	32	16	8	4	2	1

-128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

$-128 \qquad +2 \quad +1 \quad = -125$

Sign-Magnitude representation (Sample problems):-

(a) −127 (b) +63 (c) −219 (d) −66354 (e) +23655

(f) 1010111000 (g) 0101001110001 (h) 101110101010 (i) 111100011 (j) 00011010

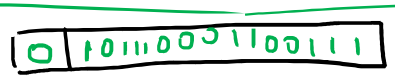
Sol:-

(a) −127 = 

(b) +63 = 

(c) −219 = 

(d) −66354 = 

(e) +23655 = 

(f) 

= −184

(g) 

= +2673

(h) 

= −938

(i) 

= −227

(j) 

= +26

Two's complement representation (Decimal to binary)

(1) A = −13

+13 ⇒ 01101

One's comp

10010

Add '1'

```
  10010
+     1
-------
  10011
```

→ two's comp of +13 ⇒ −13

(2) A = −273

+273 ⇒ 0100010001

1011101110

```
  1011101110
+          1
------------
  1011101111
```

→ −273

```
2 | 273
2 | 136 - 1
2 | 68  - 0
2 | 34  - 0
2 | 17  - 0
2 | 8   - 1
2 | 4   - 0
2 | 2   - 0
    1   - 0
```

(3) A = +186

186 ⇒ 10111010

+186 ⇒ 01011010

(4) A = −6434

6434 ⇒ 1100100010 0 10

+6434 ⇒ 0 1100100010 0 10

neg/comp → 1 0011011101 1 01

Add 1 → + 1

1 0011011101 1 10 → −6434

(5) A = −2667

2667 ⇒ 1010011010 11

+2667 ⇒ 0101001101 011

1010110010100

+ 1

1010110010101 → −2667

(6) A = +777

+777 ⇒ 0110000100 1

Binary to Decimal

① 0 1 0 1 0 0 0 1

$2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

$1\times2^6 + 1\times2^4 + 1\times2^0$

$64 + 16 + 1$

$+81$

② 0 1 1 1 0 1 1 1 0 1

$2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

$1\times2^0 + 1\times2^2 + 1\times2^3 + 1\times2^4 + 1\times2^6 + 1\times2^7 + 1\times2^8$

$1 + 4 + 8 + 16 + 64 + 128 + 256 = +477$

③ 1 0 1 1 0 0 1 0 1 1 0

$1\times2^1 + 1\times2^2 + 1\times2^4 + 1\times2^7 + 1\times2^8 + (1\times -2^{10})$

$2 + 4 + 16 + 128 + 256 - 1024$

$\Rightarrow 406 - 1024$

$\Rightarrow -618$

④ 111000110

$1\times2^1 + 1\times2^2 + 1\times2^6 + 1\times2^7 + 1\times2^8 + 1\times(-2^9)$

$2 + 4 + 64 + 128 + 256 - 512$

$406 - 512$

-106

⑤ 0101000001111111

$1\times2^0 + 1\times2^1 + 1\times2^2 + 1\times2^3 + 1\times2^4 + 1\times2^5$

$+ 1\times2^6 + 2^{12} + 1\times2^{14}$

$\Rightarrow 1+2+4+8+16+32+64+4096+16384 =$ **+20607**

⑥ 0011111011110110

$1\times2^1 + 1\times2^2 + 1\times2^4 + 1\times2^5 + 1\times2^6 + 1\times2^8 + 1\times2^9$

$+ 1\times2^{10} + 1\times2^{11} + 2^{12} +$

$2+4+16+32+64+256+512+1024+$

$2048 + 4096 =$ **+8054**

⑦ 1001111011101 1

$\Rightarrow$

-16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
1	0	0	1	1	1	1	0	1	1	1	0	1	0	1

$1024 + 512 + 256 + 0 + 64 + 32 + 16 + 0 + 4 + 0 + 1$

$+ 204 + 0 + 0 - 16384$

$\Rightarrow$ **-12427**

Converting between different bit lengths

- It is sometimes desirable to take an *n*-bit integer and store it in *m* bits, where m > n
- In sign-magnitude notation, this is easily accomplished: simply move the sign bit to the new leftmost position and fill in with zeros.

- For 2's complement negative numbers

+18	=	00010010	(twos complement, 8 bits)
+18	=	0000000000010010	(twos complement, 16 bits)
−18	=	11101110	(twos complement, 8 bits)
−32,658	=	1000000001101110	(twos complement, 16 bits)

Converting between different bit lengths

- The rule for 2's complement integers is to move the sign bit to the new leftmost position and fill in with copies of the sign bit.
- For positive numbers, fill in with zeros, and for negative numbers, fill in with ones. This is called sign extension.

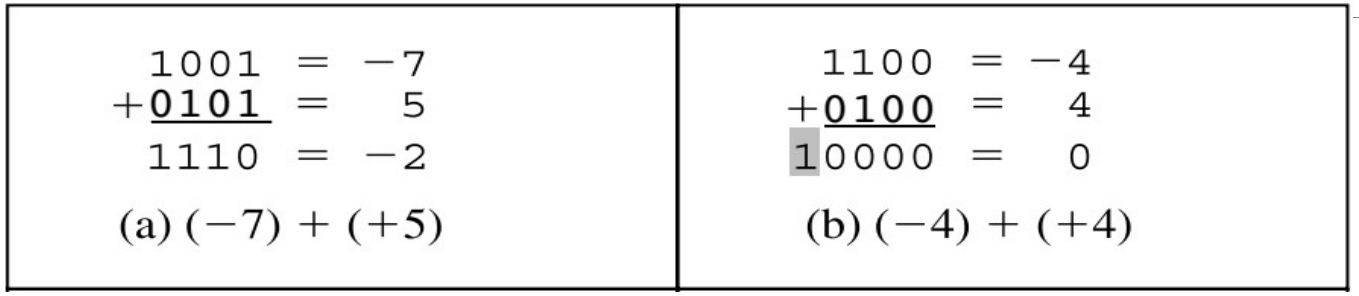
-18	$=$	11101110	(twos complement, 8 bits)
-18	$=$	1111111111101110	(twos complement, 16 bits)

Integer arithmetic: negation

- In sign-magnitude representation, the rule for forming the negation of an integer is simple: invert the sign bit
- In 2's complement notation,
  - Take the Boolean complement of each bit of the integer
  - Treating the result as an unsigned binary integer, add 1

$$
\begin{array}{rcrl}
+18 & = & 00010010 & \text{(twos complement)} \\
\text{bitwise complement} & = & 11101101 & \\
 & & + \quad 1 & \\
 & & \hline
 & & 11101110 = -18 &
\end{array}
$$

Integer arithmetic: Addition



Integer arithmetic: Addition

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)

Integer arithmetic: Addition

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)
0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Integer arithmetic: subtraction

```
        0010 =  2
       +1001 = −7
        1011 = −5

(a) M = 2 = 0010
    S = 7 = 0111
   −S =     1001
```

```
        0101 =  5
       +1110 = −2
       10011 =  3

(b) M = 5 = 0101
    S = 2 = 0010
   −S =     1110
```

Integer arithmetic: subtraction

```
        0010 =  2
       +1001 = –7
        1011 = –5

(a) M = 2 = 0010
    S = 7 = 0111
   –S =     1001
```

```
        0101 =  5
       +1110 = –2
       10011 =  3

(b) M = 5 = 0101
    S = 2 = 0010
   –S =     1110
```

```
        1011 = –5
       +1110 = –2
       11001 = –7

(c) M = –5 = 1011
    S =  2 = 0010
   –S =      1110
```

```
        0101 = 5
       +0010 = 2
        0111 = 7

(d) M =  5 = 0101
    S = –2 = 1110
   –S =      0010
```

Integer arithmetic: subtraction

```
        0010 =  2
       +1001 = –7
        1011 = –5

(a) M =  2 = 0010
    S =  7 = 0111
   –S =      1001
```

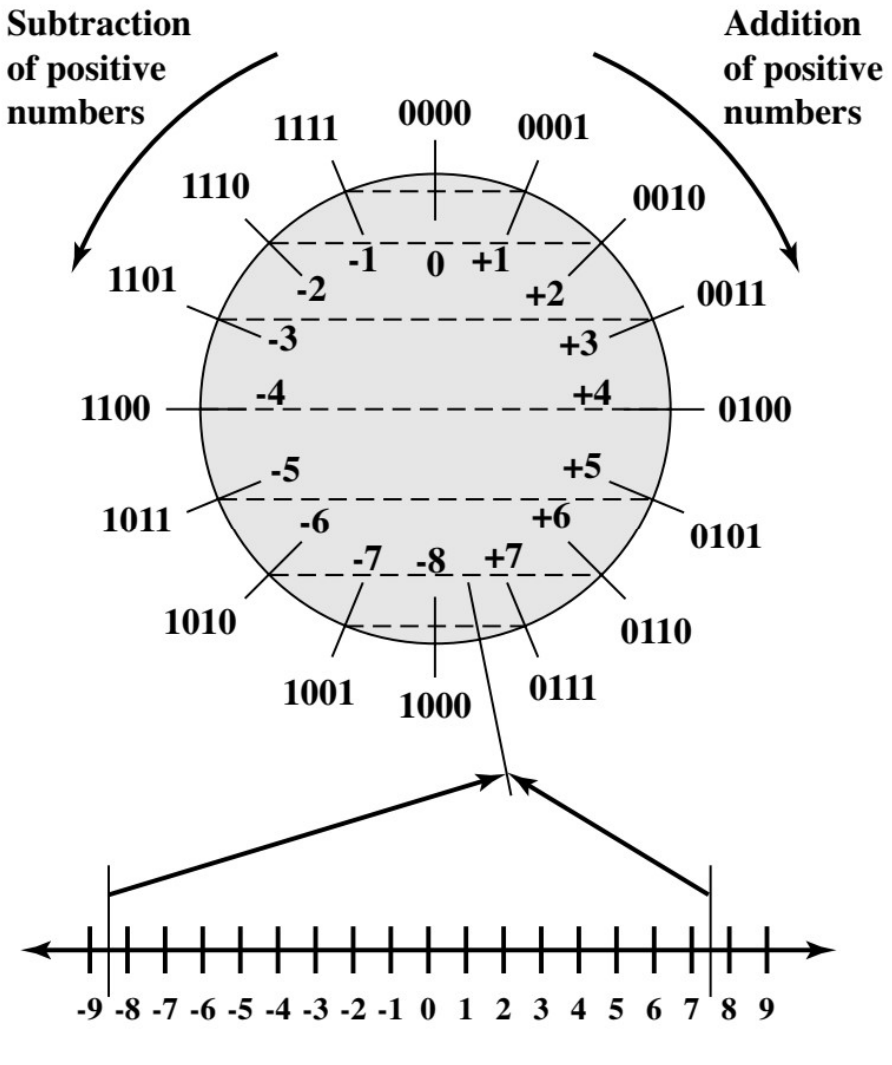
```
        0101 =  5
       +1110 = –2
       10011 =  3

(b) M =  5 = 0101
    S =  2 = 0010
   –S =      1110
```

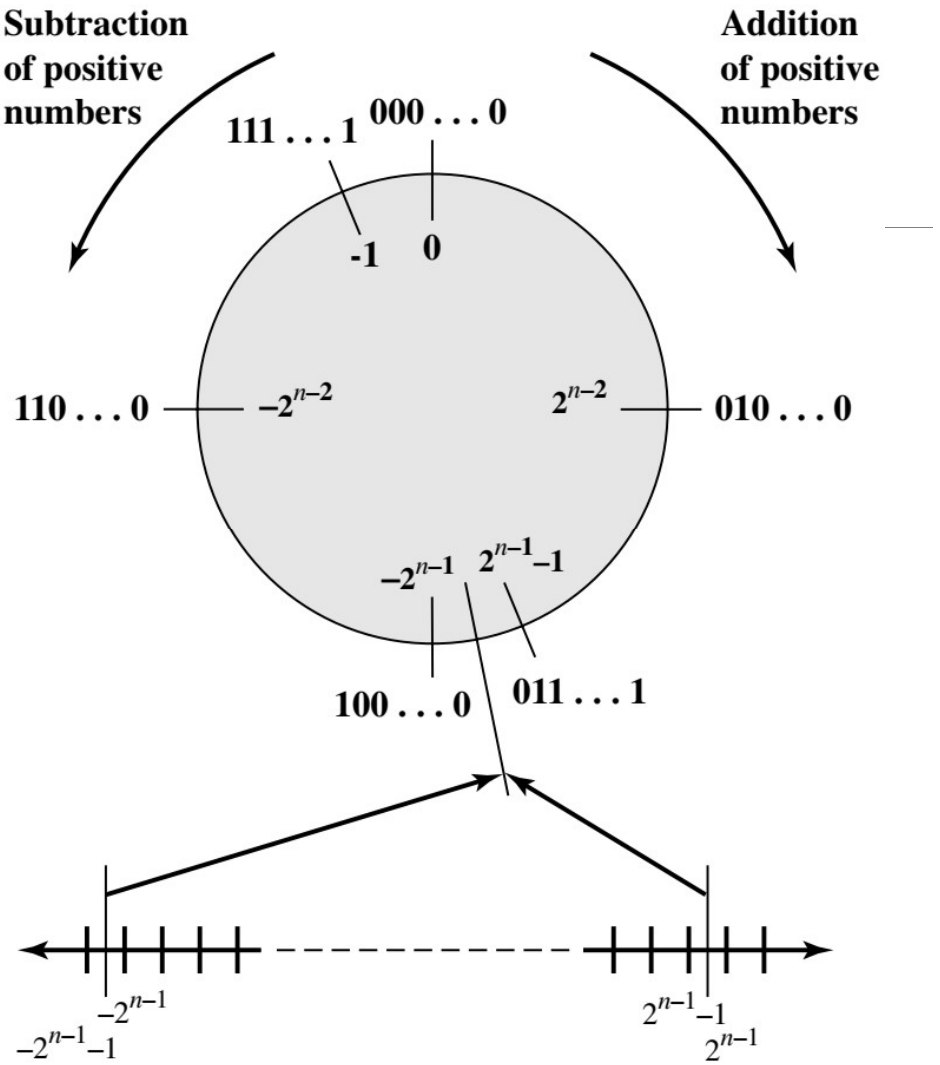
```
        1011 = –5
       +1110 = –2
       11001 = –7

(c) M = –5 = 1011
    S =  2 = 0010
   –S =      1110
```

```
        0101 = 5
       +0010 = 2
        0111 = 7

(d) M =  5 = 0101
    S = –2 = 1110
   –S =      0010
```

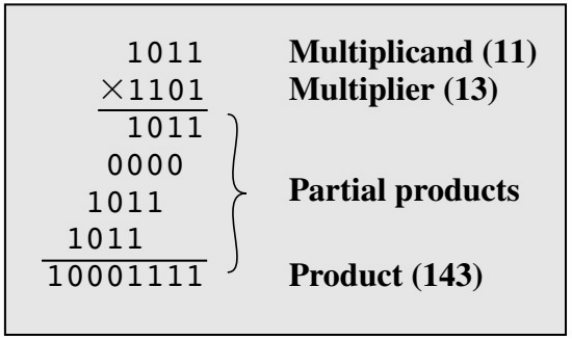
```
        0111 = 7
       +0111 = 7
        1110 = Overflow

(e) M =  7 = 0111
    S = –7 = 1001
   –S =      0111
```

```
        1010 = –6
       +1100 = –4
       10110 = Overflow

(f) M = –6 = 1010
    S =  4 = 0100
   –S =      1100
```


Subtraction of positive numbers
Addition of positive numbers
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
0
+1
+2
+3
+4
+5
+6
+7
-1
-2
-3
-4
-5
-6
-7
-8
-9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9


Subtraction of positive numbers
Addition of positive numbers
000 . . . 0
111 . . . 1
-1
0
110 . . . 0
010 . . . 0
100 . . . 0
011 . . . 1
-2^{n-2}
2^{n-2}
-2^{n-1}
$2^{n-1}-1$
-2^{n-1}
$-2^{n-1}-1$
$2^{n-1}-1$
2^{n-1}

Integer arithmetic: multiplication



- Compared with addition and subtraction, multiplication is a complex operation, whether performed in hardware or software
- A wide variety of algorithms have been used in various computers
- Multiplication of unsigned integers
  - Multiplication involves the generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product
  - The partial products are easily defined. When the multiplier bit is 0, the partial product is 0. When the multiplier bit is 1, the partial product is the multiplicand
  - The total product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product
  - The multiplication of two n-bit binary integers results in a product of up to 2n bits in length (e.g., 11 x 11 = 1001)

Integer arithmetic: multiplication

- Compared with the pencil-and-paper approach, there are several things we can do to make computerized multiplication more efficient
- First, we can perform a running addition on the partial products rather than waiting until the end.
- For each 1 on the multiplier, an add and a shift operation are required; but for each 0, only a shift is required.

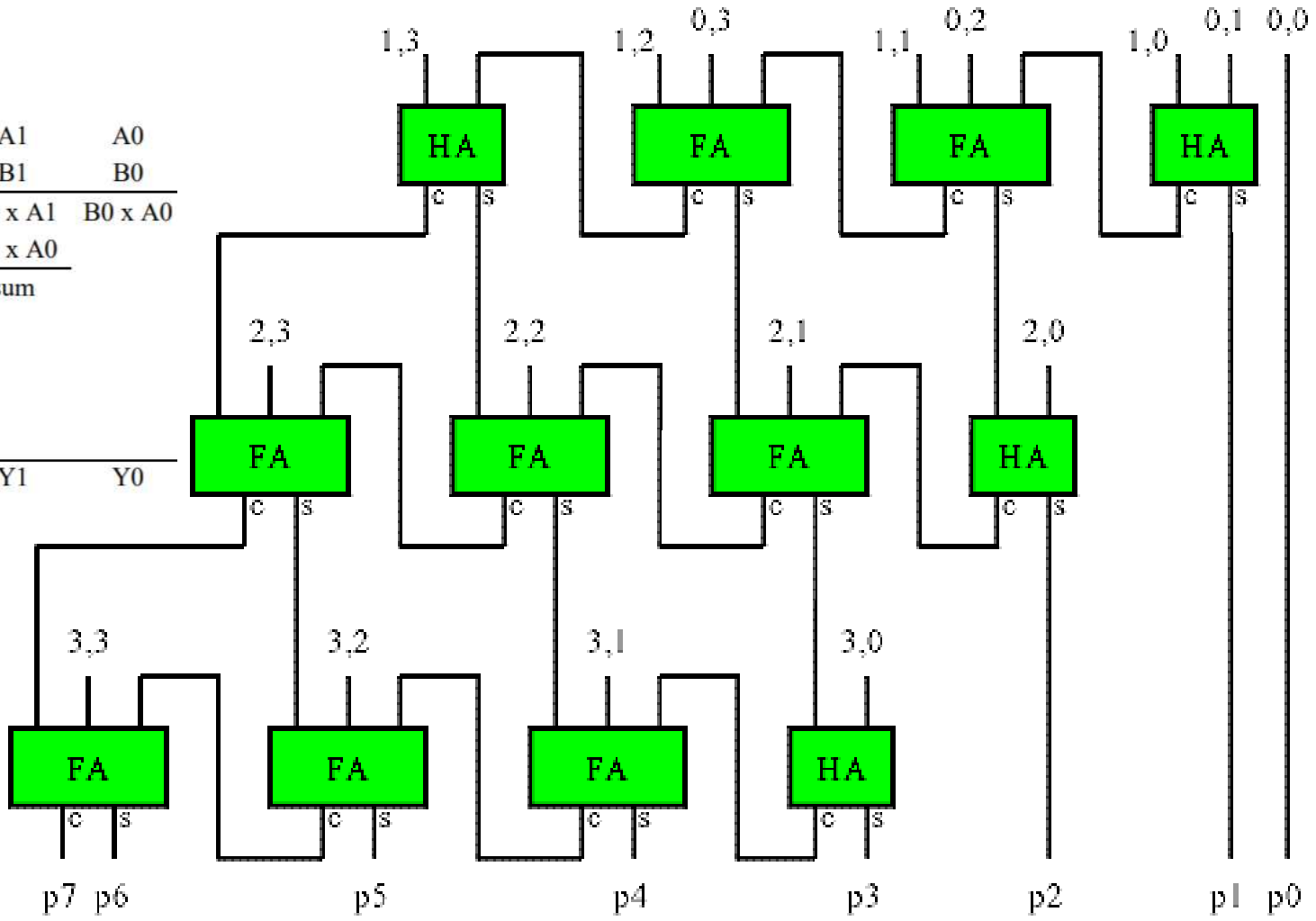
Array multiplication

- Let us consider the multiplicand to be M = (A_3, A_2, A_1, A_0) and the multiplier to be Q = (B_3, B_2, B_1, B_0)

			x	A3 B3	A2 B2	A1 B1	A0 B0	Inputs
			C	B0 x A3	B0 x A2	B0 x A1	B0 x A0	
		+	B1 x A3	B1 x A2	B1 x A1	B1 x A0		
		C	sum	sum	sum	sum		
	+	B2 x A3	B2 x A2	B2 x A1	B2 x A0			Internal Signals
	C	sum	sum	sum	sum			
+	B3 x A3	B3 x A2	B3 x A1	B3 x A0				
C	sum	sum	sum	sum				
Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	Outputs

Hardware configuration of 4*4 Array multiplier

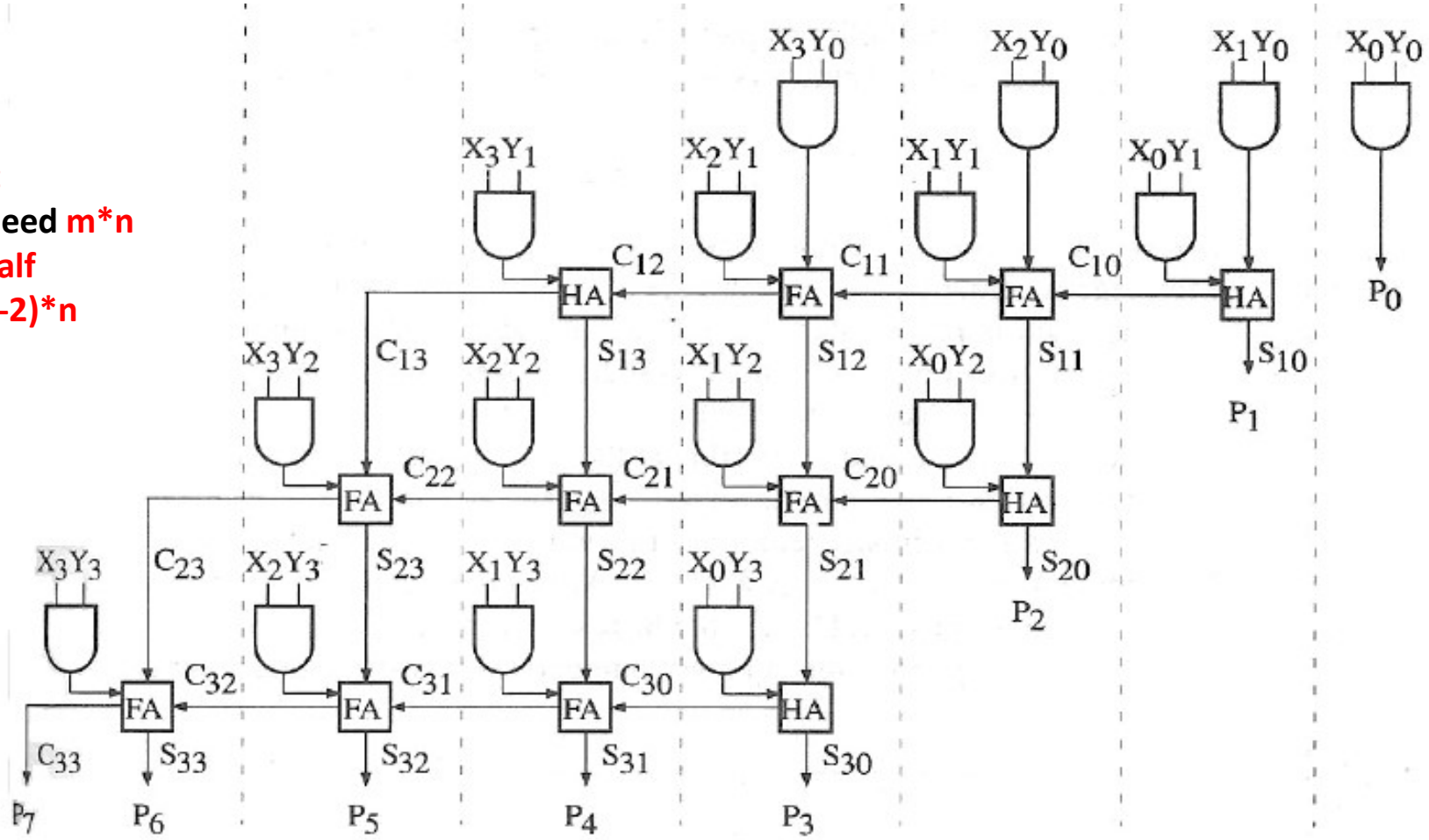
				A3	A2	A1	A0
			x	B3	B2	B1	B0
			C	B0 x A3	B0 x A2	B0 x A1	B0 x A0
		+	B1 x A3	B1 x A2	B1 x A1	B1 x A0	
		C	sum	sum	sum	sum	
	+	B2 x A3	B2 x A2	B2 x A1	B2 x A0		
	C	sum	sum	sum	sum		
+	B3 x A3	B3 x A2	B3 x A1	B3 x A0			
C	sum	sum	sum	sum			
Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0



For m-bit*n-bit multiplier we need m*n AND gates, n Half adders, and (m-2)*n Full adders

Hardware configuration of 4*4 Array multiplier

For m-bit*n-bit multiplier we need m*n AND gates, n Half adders, and (m-2)*n Full adders

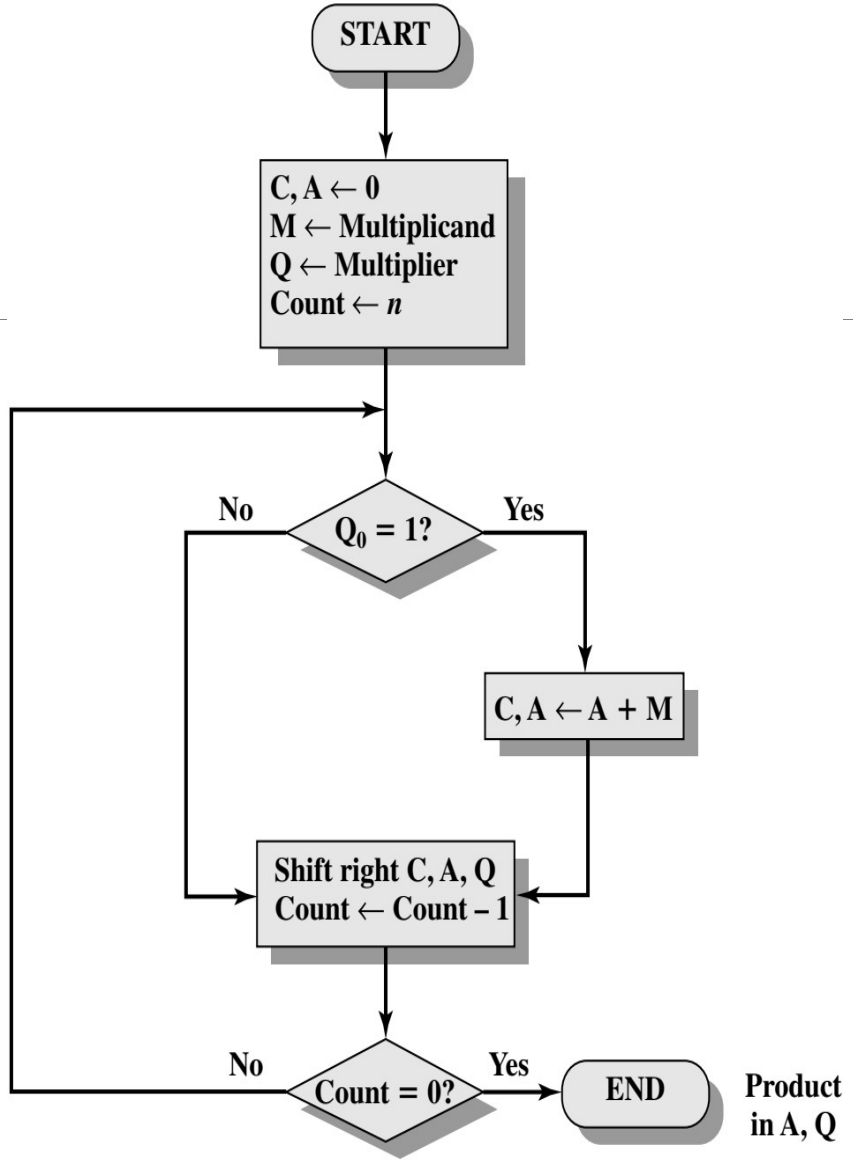


Signed multiplication using array multiplier

```
                                                p0[7] p0[6] p0[5] p0[4] p0[3] p0[2] p0[1] p0[0]
                                      + p1[7] p1[6] p1[5] p1[4] p1[3] p1[2] p1[1] p1[0] 0
                                + p2[7] p2[6] p2[5] p2[4] p2[3] p2[2] p2[1] p2[0] 0     0
                          + p3[7] p3[6] p3[5] p3[4] p3[3] p3[2] p3[1] p3[0] 0     0     0
                    + p4[7] p4[6] p4[5] p4[4] p4[3] p4[2] p4[1] p4[0] 0     0     0     0
              + p5[7] p5[6] p5[5] p5[4] p5[3] p5[2] p5[1] p5[0] 0     0     0     0     0
        + p6[7] p6[6] p6[5] p6[4] p6[3] p6[2] p6[1] p6[0] 0     0     0     0     0     0
  + p7[7] p7[6] p7[5] p7[4] p7[3] p7[2] p7[1] p7[0] 0     0     0     0     0     0     0
-------------------------------------------------------------------------------------------
P[15] P[14] P[13] P[12] P[11] P[10]  P[9]  P[8]  P[7]  P[6]  P[5]  P[4]  P[3]  P[2]  P[1]  P[0]
```

```
                                                   1  ~p0[7]  p0[6]  p0[5]  p0[4]  p0[3]  p0[2]  p0[1]  p0[0]
                                              ~p1[7] +p1[6] +p1[5] +p1[4] +p1[3] +p1[2] +p1[1] +p1[0]   0
                                        ~p2[7] +p2[6] +p2[5] +p2[4] +p2[3] +p2[2] +p2[1] +p2[0]   0      0
                                  ~p3[7] +p3[6] +p3[5] +p3[4] +p3[3] +p3[2] +p3[1] +p3[0]   0      0      0
                            ~p4[7] +p4[6] +p4[5] +p4[4] +p4[3] +p4[2] +p4[1] +p4[0]   0      0      0      0
                      ~p5[7] +p5[6] +p5[5] +p5[4] +p5[3] +p5[2] +p5[1] +p5[0]   0      0      0      0      0
                ~p6[7] +p6[6] +p6[5] +p6[4] +p6[3] +p6[2] +p6[1] +p6[0]   0      0      0      0      0      0
    1  +p7[7] ~p7[6] ~p7[5] ~p7[4] ~p7[3] ~p7[2] ~p7[1] ~p7[0]   0      0      0      0      0      0      0
 ---------------------------------------------------------------------------------------------------------------
P[15]  P[14]  P[13]  P[12]  P[11]  P[10]   P[9]   P[8]   P[7]   P[6]   P[5]   P[4]   P[3]   P[2]   P[1]  P[0]
```

Multiplication



Multiplication

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

Multiplying Negative Numbers

This does not work! Eg. If we multiply 11 (1011) by 13 (1101) we should get 143 (10001111).

If we interpret these as two's complement numbers, we have -5 (1011) times

-3(1101) which equals -113(10001111) which is incorrect.

It will also not work if either the multiplicand or the multiplier is negative.

If 9 and 3 are treated as unsigned integers, the multiplication proceeds simply fig a.

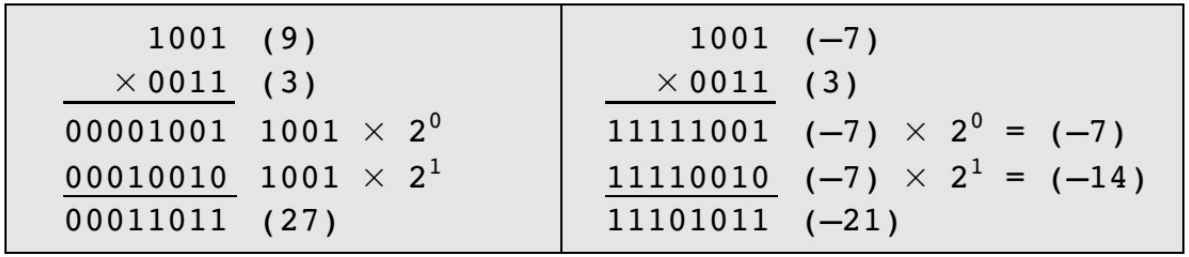
But if 1001 is interpreted as the

twos complement value -7, then

each partial product must be a

negative twos complement number of 2n(8) bits

Note: this is accomplished by padding out each pa



1001 (9)	1001 (−7)
× 0011 (3)	× 0011 (3)
00001001 1001 × 2^0	11111001 (−7) × 2^0 = (−7)
00010010 1001 × 2^1	11110010 (−7) × 2^1 = (−14)
00011011 (27)	11101011 (−21)

(a) Unsigned integers (b) Twos complement integers

If the multiplier is negative

If the multiplier is negative, straightforward multiplication also will not work.

The reason is that the bits of the multiplier no longer correspond to the shifts or multiplications that must take place.

For example, the 4-bit decimal number -3 is written 1101 in twos complement. If we simply took partial products based on each bit position, we have

instead of $1101 \longleftrightarrow -(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) = -(2^3 + 2^2 + 2^0)$

$$-(2^1 + 2^0)$$

Solutions to the above issues

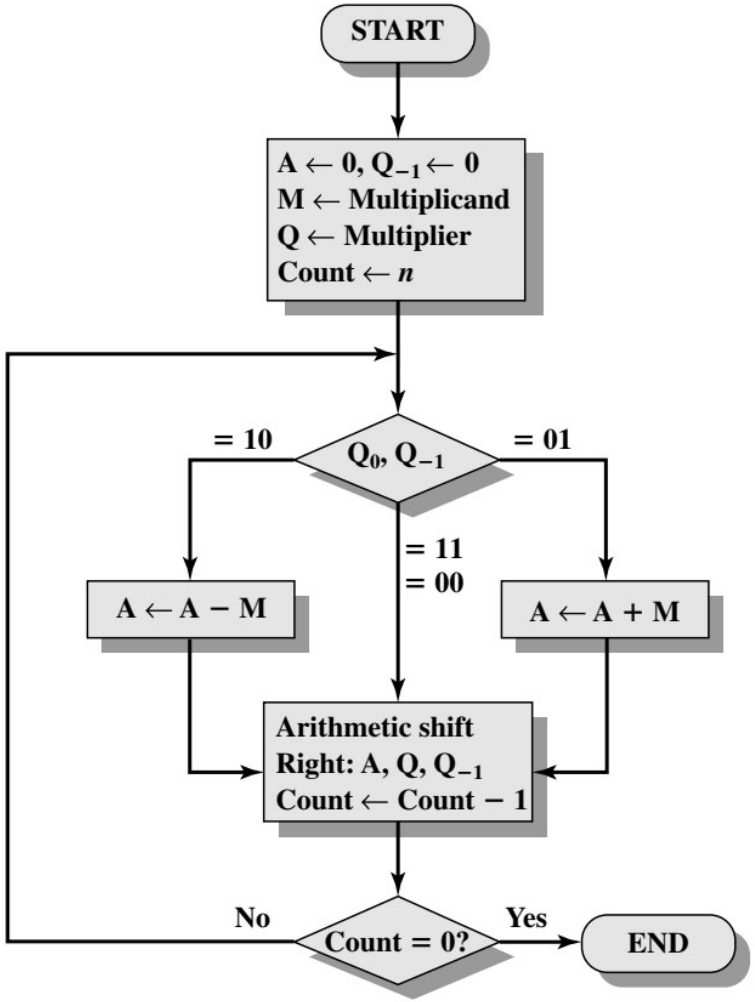
Solution 1

- Convert to positive if required
- Multiply as above
- If signs were different, negate answer

Solution 2

- Booth's algorithm

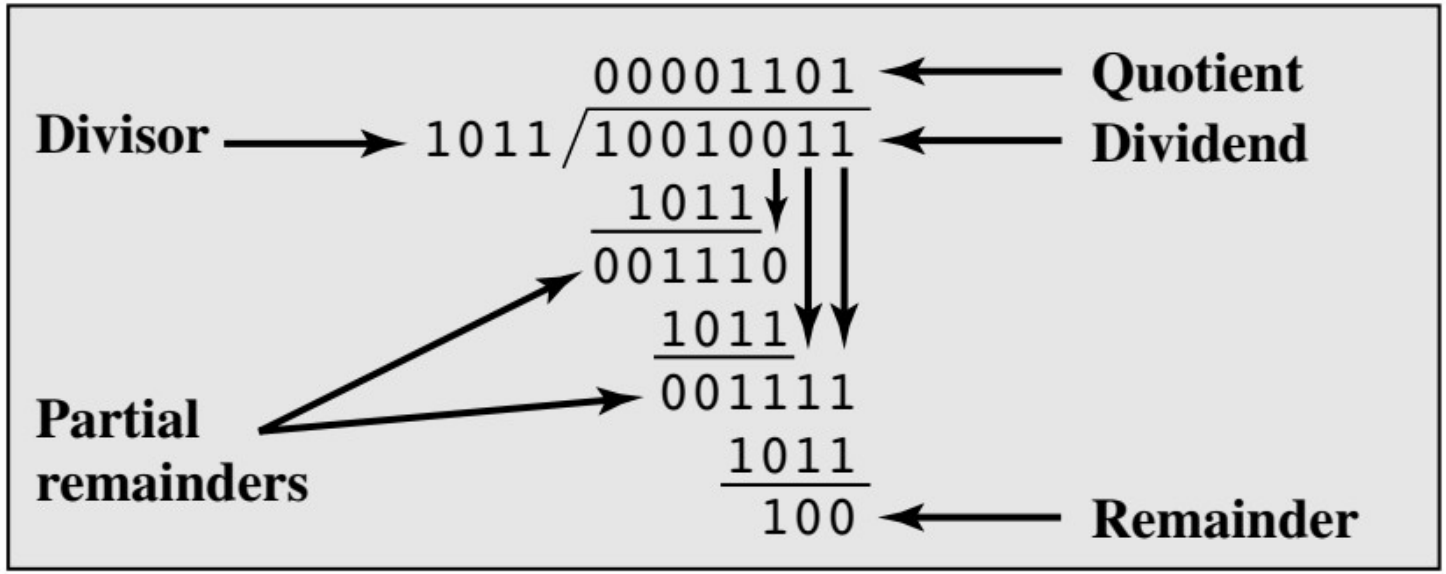
Booth Multiplication



Booth Multiplication: Example

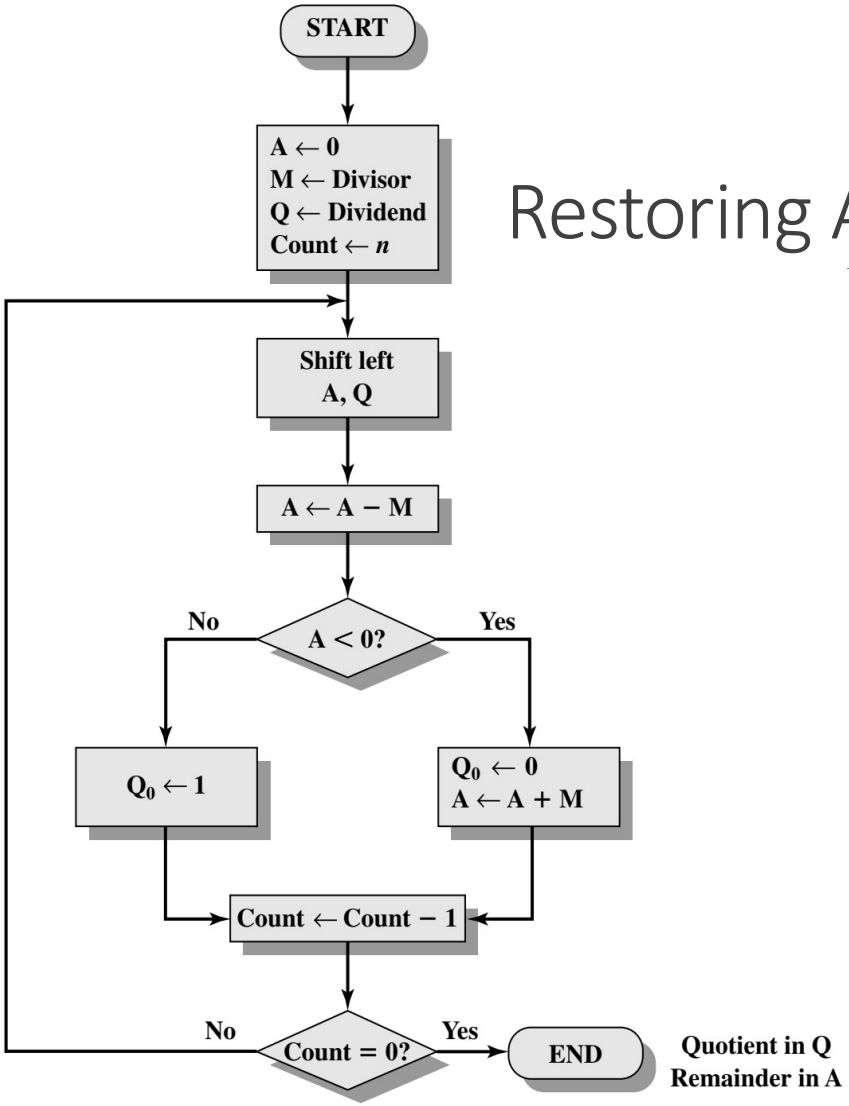
A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	$A \leftarrow A - M$	First cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	$A \leftarrow A + M$	Third cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth cycle

Division

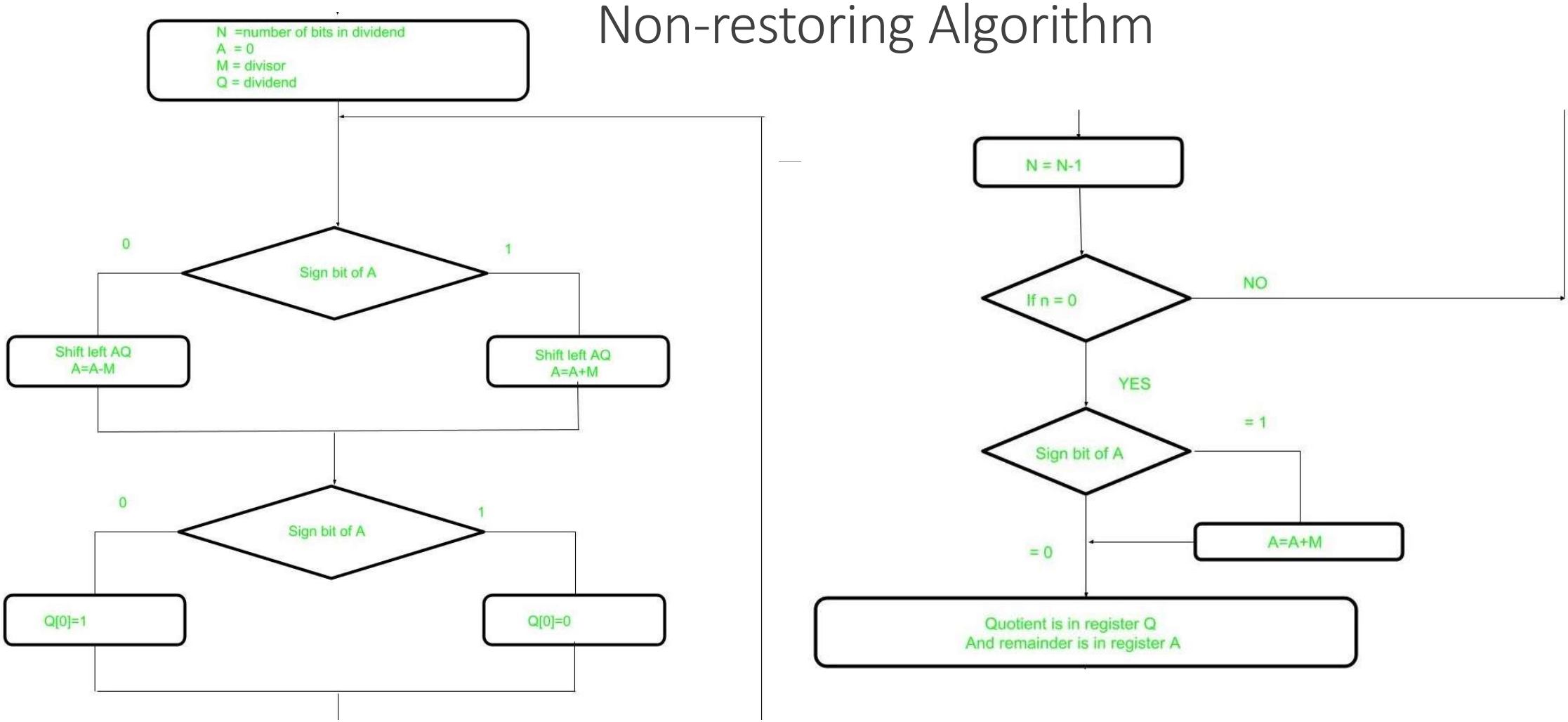

00001101
Quotient
Divisor
1011/10010011
Dividend
1011
001110
1011
001111
Partial
remainders
1011
100
Remainder

Division

Restoring Algorithm



Non-restoring Algorithm



Signed Division process

- To deal with negative numbers, the remainder is defined by

$$D = Q \times V + R$$

- Consider the following examples of integer division with all possible combinations of signs of D and V

$$\begin{array}{llll} D = 7 & V = 3 & \Rightarrow Q = 2 & R = 1 \\ D = 7 & V = -3 & \Rightarrow Q = -2 & R = 1 \\ D = -7 & V = 3 & \Rightarrow Q = -2 & R = -1 \\ D = -7 & V = -3 & \Rightarrow Q = 2 & R = -1 \end{array}$$

$$\text{sign}(R) = \text{sign}(D)$$

$$\text{sign}(Q) = \text{sign}(D) \times \text{sign}(V)$$

Floating Point Numbers

Now we have seen **unsigned** and **signed** integers. In real life we also need to be able represent numbers with fractional parts (like: -12.5 & 45.39).

- Called **Floating Point** numbers.
- We will learn the IEEE 32-bit floating point representation.

Floating Point Numbers

In the decimal system, a decimal point (**radix point**) separates the whole numbers from the fractional part

Examples:

37.25 (whole = 37, fraction = 25/100)

123.567

10.12345678

Floating Point Numbers

For example, 37.25 can be analyzed as:

10^1	10^0	10^{-1}	10^{-2}
Tens	Units	Tenths	Hundredths
3	**7**	**2**	**5**

37.25 = (3 x 10) + (7 x 1) + (2 x 1/10) + (5 x 1/100)

Binary Equivalence

The binary equivalent of a floating point number can be determined by computing the binary representation for each part separately.

1) For the **whole** part:

   Use subtraction or division method previously learned.

2) For the **fractional** part:

   Use the subtraction or multiplication method (to be shown next)

Fractional Part – Multiplication Method

In the binary representation of a floating point number the column values will be as follows:

…	2^5	2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4} …
…	32	16	8	4	2	1	.	1/2	1/4	1/8	1/16…
…	32	16	8	4	2	1	.	.5	.25	.125	.0625…

Fractional Part – Multiplication Method

Ex 1. Find the binary equivalent of **0.25**

Step 1: Multiply the fraction by 2 until the fractional part becomes $_0$

.25

x 2

0.5

x 2

1.0

Step 2: Collect the whole parts in forward order. Put them after the radix point

. .5 .25 .125 .0625

. 0 1

Fractional Part – Multiplication Method

Ex 2. Find the binary equivalent of **0.625**

Step 1: Multiply the fraction by 2 until the fractional part becomes 0

.625

x 2

1.25

x 2

0.50

x 2

1.0

Step 2: Collect the whole parts in forward order. Put them after the radix point

. .5 .25 .125 .0625

. 1 0 1

Fractional Part – Subtraction Method

Start with the column values again, as follows:

… 2^0 .	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}…
… 1 .	1/2	1/4	1/8	1/16	1/32	1/64…
… 1 .	.5	.25	.125	.0625	.03125	.015625…

Fractional Part – Subtraction Method

Starting with 0.5, subtract the column values from left to right.
Insert a 0 in the column if the value cannot be subtracted or 1 if it can be. Continue until the fraction becomes .0

Ex 1.

.25	.5	.25	.125	.0625
– .25	.0	1		
.0				

Binary Equivalent of FP number

Ex 2. Convert 37.25, using subtraction method.

64 32 16 8 4 2 1 . .5 .25 .125 .0625

2^6 2^5 2^4 2^3 2^2 2^1 2^0 . 2^{-1} 2^{-2} 2^{-3} 2^{-4}

1 0 0 1 . 0 0 1 1

37 .25

- 32 - .25 5

.0

- 4

1

-1

0

$37.25_{10} = 100101.01_2$

Binary Equivalent of FP number

Ex 3. Convert 18.625, using subtraction method.

64	32	16	8	4	2	1	.	.5	.25	.125	.0625
2^6	2^5	2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}
		1	0	0	1	0		1	0	1	

18	.625
- 16	- .5
2	.125
- 2	- .125
0	

$$18.625_{10} = 10010.101_2^0$$

Problem storing binary form

We have no way to store the radix point!

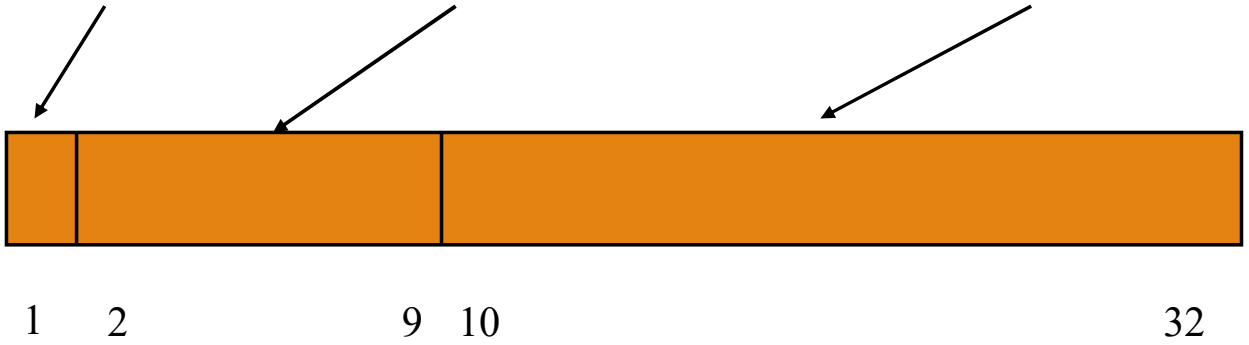
Large numbers will take so much space

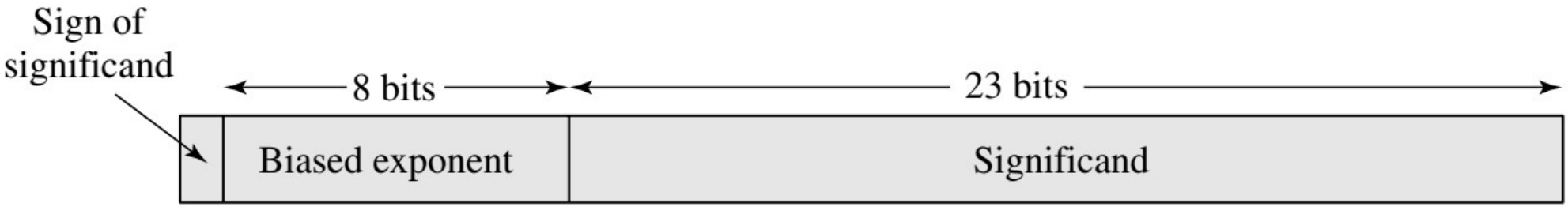
Standards committee came up with a way to store floating point numbers (that have a decimal point)

IEEE Floating Point Representation

Floating point numbers can be stored into 32-bits, by dividing the bits into three parts:

the **sign**, the **exponent**, and the **mantissa.**





- Sign
- Base = 2
- Exponent = Value of the 8bit exponent – Bias
  (where Bias = $2^{k-1}-1$, k = no. of bits in the exponent)
- Significand

Floating point representation

0.110×2^5

110×2^2

0.0110×2^6

- Any floating-point number can be expressed in many ways
- To simplify operations on floating-point numbers, it is typically required that they be normalized
- A **normalized number** is one in which the most significant digit of the significand is nonzero
- For base 2 representation, a normalized number is therefore one in which the most significant bit of the significand is one
- The typical convention is that there is one bit to the left of the radix point

$$\pm 1.\text{bbb} \ldots \text{b} \times 2^{\pm E}$$

- where b is a binary digit (either 0 or 1)
- Thus, the 23-bit field is used to store a 24-bit significand with a value in the half open interval [1, 2)

Sample problems

Convert the following decimal numbers into binary using IEEE 754 floating point representation.

(i) -3347.7991×2^{21}

(ii) 157.4773×2^{-11}

(iii) -1234.5997×2^{17}

(iv) -488.6791×2^{-31}

Floating point representation

- In the IBM base-16 format, the exponent is stored to represent 5 rather than 20

$$0.11010001 \times 2^{10100} = 0.11010001 \times 16^{101}$$

- The advantage of using a larger exponent is that a greater range can be achieved for the same number of exponent bits.
- However, a larger exponent base gives a greater range at the expense of less precision.

Floating point Arithmetic

While performing arithmetic operations, a Floating point number is treated as two fixed point numbers: Exponent and Mantissa.

Consider X and Y as two Floating point numbers.

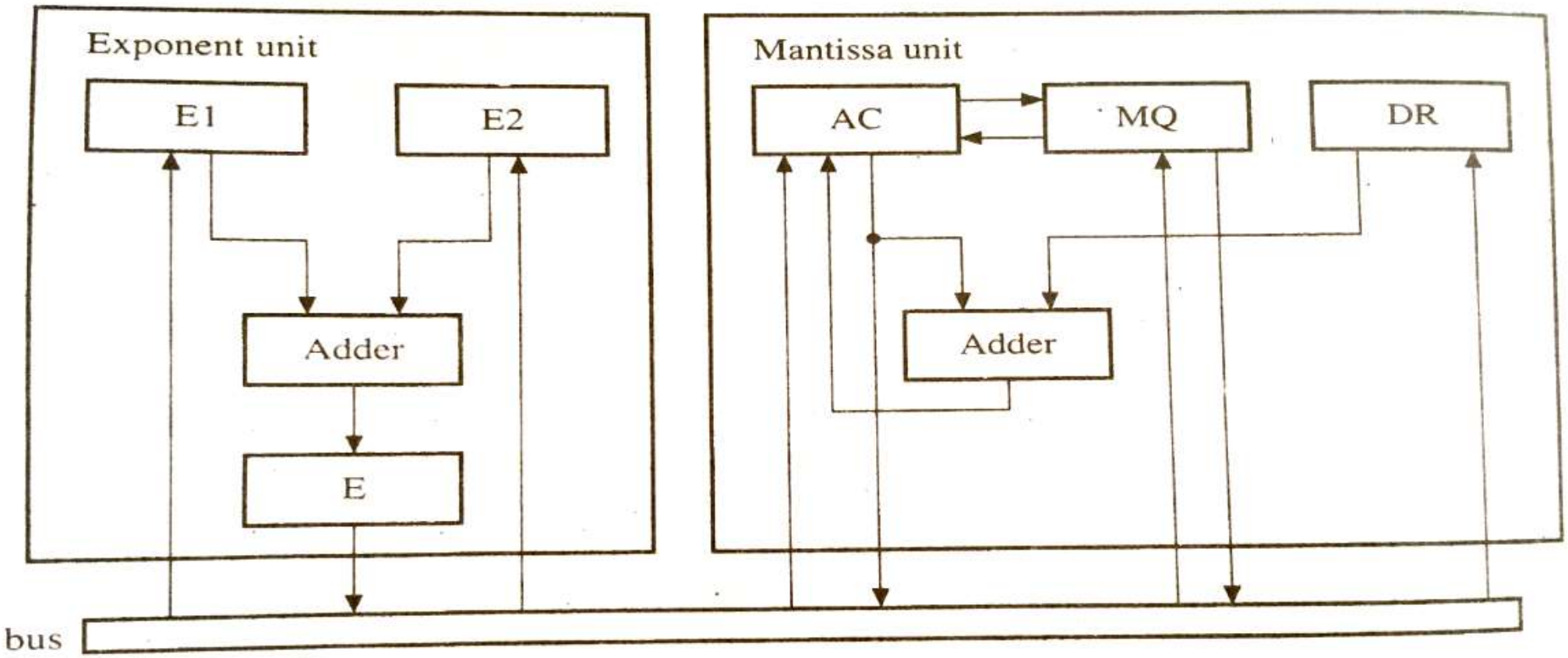
$X = X_M.2^{X_E}$

$Y = Y_M.2^{Y_E}$

Then the arithmetic operations on these two numbers will be performed as:

OPERATION	MANTISSA	EXPONENT
Addition	Add	Equalize
Subtraction	Sub	Equalize
Multiplication	Mul	Add
Division	Div	Sub

From the above table it is evident that Mantissa and Exponent are dealt with, in different ways. Hence a Floating Point ALU has two units internally: Exponent Unit and Mantissa Unit.


Exponent unit
E1
E2
Adder
E
Mantissa unit
AC
MQ
DR
Adder
bus

ADDITION ALGORITHM

```
LOAD:       E1 ← XE,  E2 ← YE;  ...          {Exponents}
            AC ← XM,  DR ← YM; ...           {Mantissas}
            Error ← 0,  AC_Overflow ← 0; ...  {Error Variables}

{Compare and Equalize}
COMPARE:    E ← E1 – E2;

EQUALIZE:   If (E < 0) then
                AC ← right-shift (AC);
                E ← E + 1;
                go to Equalize;
            Else
            If (E > 0) then
                DR ← right-shift (DR);
                E ← E – 1;
                go to Equalize;

{Add the mantissas}
ADD:        AC ← AC + DR;
            E ← Max (E1, E2);
```

```
{Add the mantissas}
ADD:        AC ← AC + DR;
            E ← Max (E1, E2);

{Adjust for overflow}
OVERFLOW:   If (AC_Overflow = 1) then
                If (E = Emax) then go to ERROR;
                AC ← right-shift (AC);
                E ← E + 1;
                go to END;

{Adjust for Zero result}
ZERO:       If (AC = 0) then
                E ← 0;

{Normalize the result}
NORMALIZE:  If AC is normalized then
                go to END;

UNDERFLOW:      If E > EMIN then
                AC ← left-shift (AC);
                E ← E – 1;
                go to Normalize;

{Set error flag}
ERROR:          Error ← 1;

{End the program}
END:            End of process.
```

Floating point arithmetic

- For addition and subtraction, it is necessary to ensure that both operands have the same exponent value.
- This may require shifting the radix point on one of the operands to achieve alignment.
- Multiplication and division are more straightforward.

Floating point arithmetic

- **Exponent overflow:** A positive exponent exceeds the maximum possible exponent value. In some systems, this may be designated as or +∞ to - ∞
- **Exponent underflow:** A negative exponent is less than the minimum possible exponent value (e.g., is less than). This means that the number is too small to be represented, and it may be reported as 0.
- **Significand underflow:** In the process of aligning significands, digits may flow off the right end of the significand. As we shall discuss, some form of rounding is required.
- **Significand overflow:** The addition of two significands of the same sign may result in a carry out of the most significant bit. This can be fixed by realignment, as we shall explain.

Floating point arithmetic: Addition and subtraction

Floating Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E}$ $X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E}$ } $X_E \leq Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \times B^{X_E - Y_E}$

Examples:

$X = 0.3 \times 10^2 = 30$

$Y = 0.2 \times 10^3 = 200$

$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$

$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$

$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$

$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$

Floating point arithmetic: Addition and subtraction

- **Phase 1: Zero check.** Because addition and subtraction are identical except for a sign change, the process begins by changing the sign of the subtrahend if it is a subtract operation. Next, if either operand is 0, the other is reported as the result.
- **Phase 2: Significand alignment.** The next phase is to manipulate the numbers so that the two exponents are equal.

To see the need for aligning exponents, consider the following decimal addition:

$$(123 \times 10^0) + (456 \times 10^{-2})$$

Clearly, we cannot just add the significands. The digits must first be set into equivalent positions, that is, the 4 of the second number must be aligned with the 3 of the first. Under these conditions, the two exponents will be equal, which is the mathematical condition under which two numbers in this form can be added. Thus,

$$(123 \times 10^0) + (456 \times 10^{-2}) = (123 \times 10^0) + (4.56 \times 10^0) = 127.56 \times 10^0$$

Floating point arithmetic: Addition and subtraction

- **Phase 3: Addition.** Next, the two significands are added together, taking into account their signs. Because the signs may differ, the result may be 0. There is also the possibility of significand overflow by 1 digit. If so, the significand of the result is shifted right and the exponent is incremented. An exponent overflow could occur as a result; this would be reported and the operation halted.
- **Phase 4: Normalization.** The final phase normalizes the result. Normalization consists of shifting significand digits left until the most significant digit (bit, or 4 bits for base-16 exponent) is nonzero. Each shift causes a decrement of the exponent and thus could cause an exponent underflow. Finally, the result must be rounded off and then reported. We defer a discussion of rounding until after a discussion of multiplication and division.

Floating point arithmetic: Addition and subtraction

Floating Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E}$ $X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E}$ } $X_E \leq Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \times B^{X_E - Y_E}$

Examples:

$X = 0.3 \times 10^2 = 30$

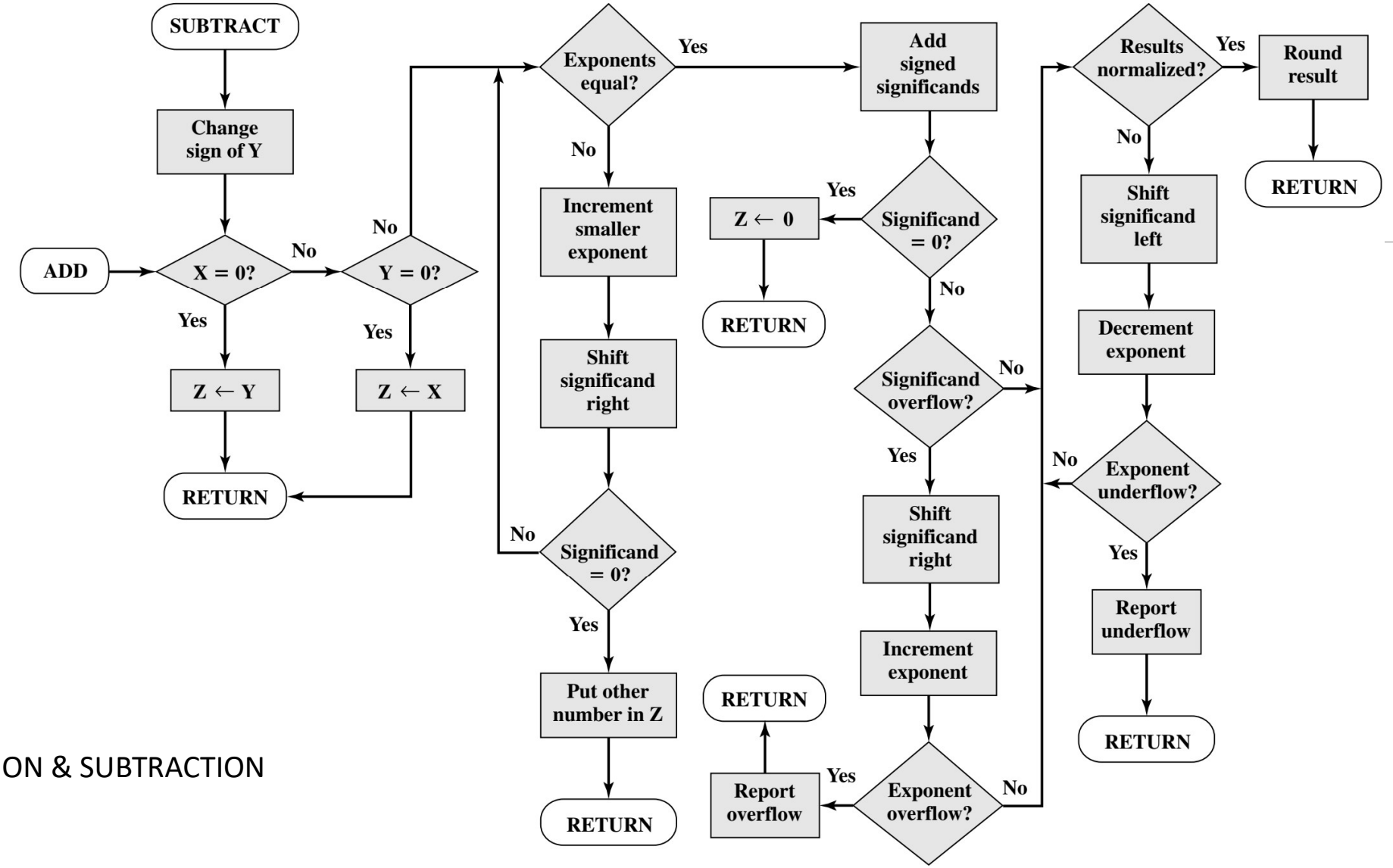
$Y = 0.2 \times 10^3 = 200$

$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$

$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$

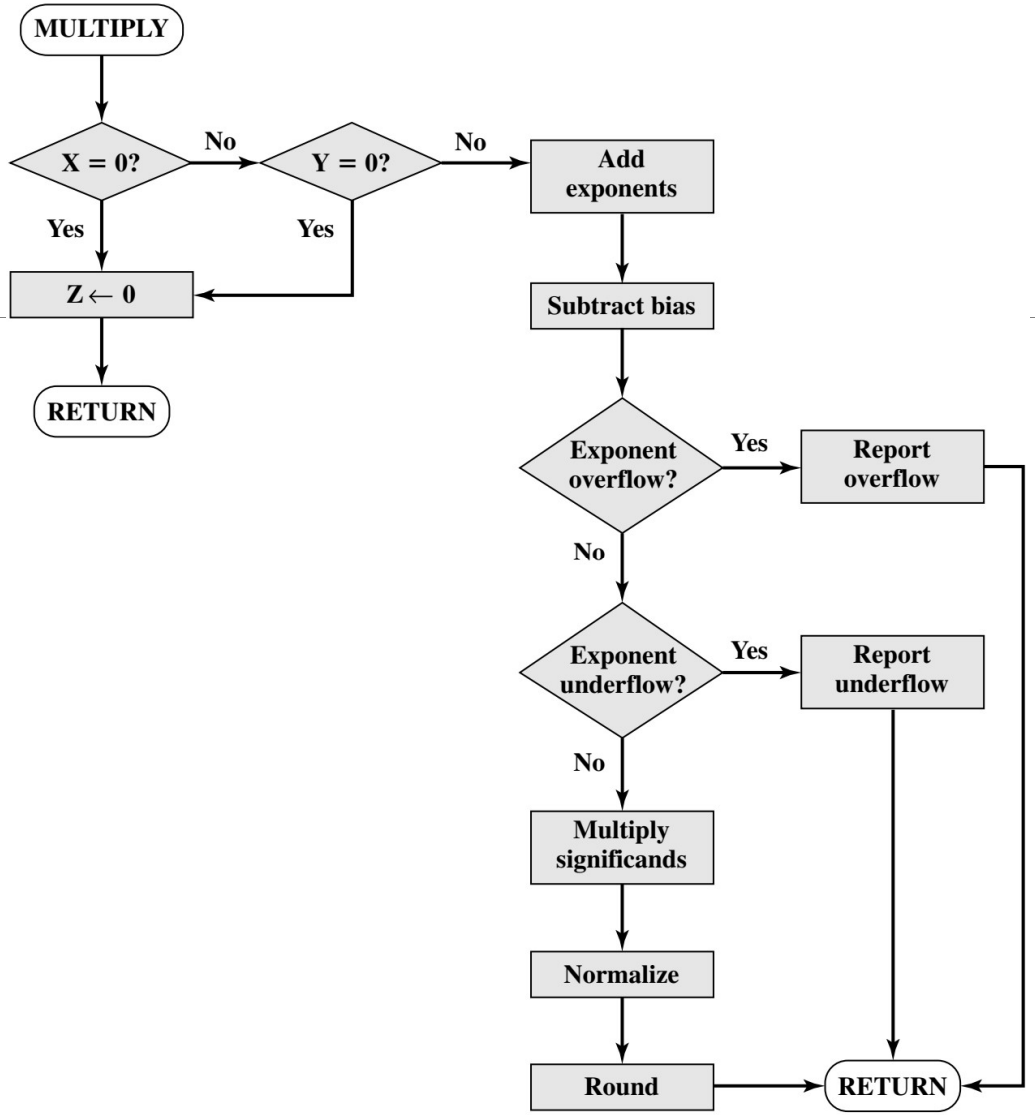
$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$

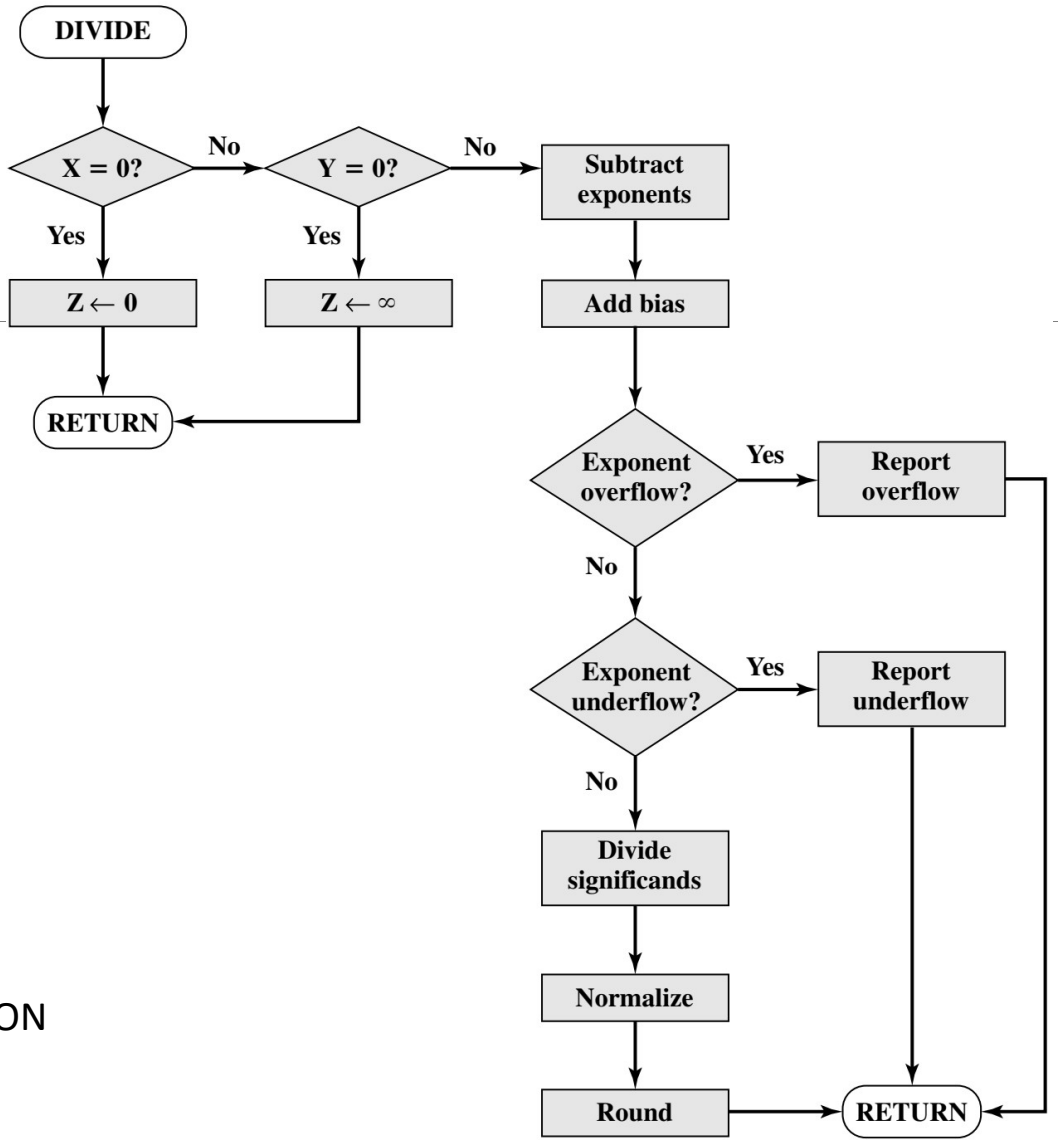
$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$



ADDITION & SUBTRACTION

MULTIPLICATION




DIVIDE
X = 0?
No
Y = 0?
No
Subtract exponents
Yes
Yes
Z ← 0
Z ← ∞
Add bias
RETURN
Exponent overflow?
Yes
Report overflow
No
Exponent underflow?
Yes
Report underflow
No
Divide significands
Normalize
Round
RETURN

DIVISION

Precision considerations

- The exponent and significand are stored in ALU registers.
- The length of the register is almost always greater than the length of the significand plus an implied bit.
- The register contains additional bits, called guard bits, which are used to pad out the right end of the significand with 0s.

$$
\begin{aligned}
x &= 1.000.....00 \times 2^{1} \\
\underline{-y} &= \underline{0.111.....11} \times 2^{1} \\
z &= 0.000.....01 \times 2^{1} \\
&= 1.000.....00 \times 2^{-22}
\end{aligned}
$$

$$
\begin{aligned}
x &= 1.000.....00\ 0000 \times 2^{1} \\
\underline{-y} &= \underline{0.111.....11\ 1000} \times 2^{1} \\
z &= 0.000.....00\ 1000 \times 2^{1} \\
&= 1.000.....00\ 0000 \times 2^{-23}
\end{aligned}
$$

$$
\begin{aligned}
x &= .100000 \times 16^{1} \\
\underline{-y} &= \underline{.0FFFFF \times 16^{1}} \\
z &= .000001 \times 16^{1} \\
&= .100000 \times 16^{-4}
\end{aligned}
$$

$$
\begin{aligned}
x &= .100000\ 00 \times 16^{1} \\
\underline{-y} &= \underline{.0FFFFF\ F0 \times 16^{1}} \\
z &= .000000\ 10 \times 16^{1} \\
&= .100000\ 00 \times 16^{-5}
\end{aligned}
$$

Rounding

- The result of any operation on the significands is generally stored in a longer register.
- When the result is put back into the floating-point format, the extra bits must be disposed of.
- **Round to nearest:** The result is rounded to the nearest representable number.
- **Round toward +∞:** The result is rounded up toward plus infinity.
- **Round toward -∞:** The result is rounded down toward negative infinity.
- **Round toward 0:** The result is rounded toward zero.

Rounding

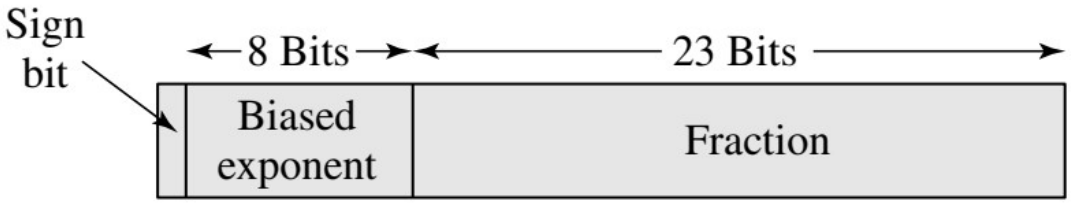
- **Round to nearest** is the default rounding mode listed in the standard and is defined as follows: The representable value nearest to the infinitely precise result shall be delivered.
- If the extra bits, beyond the 23 bits that can be stored, are 10010, then the extra bits amount to more than one-half of the last representable bit position.
- In this case, the correct answer is to add binary 1 to the last representable bit, rounding up to the next representable number.
- If the extra bits are 01111, they are less than one-half of the last representable bit position.
- The correct way is simply to drop the extra bits (truncate).
- If the result of a computation is exactly midway between two representable numbers, the value is rounded up if the last representable bit is currently 1 and not rounded up if it is currently 0.

Rounding

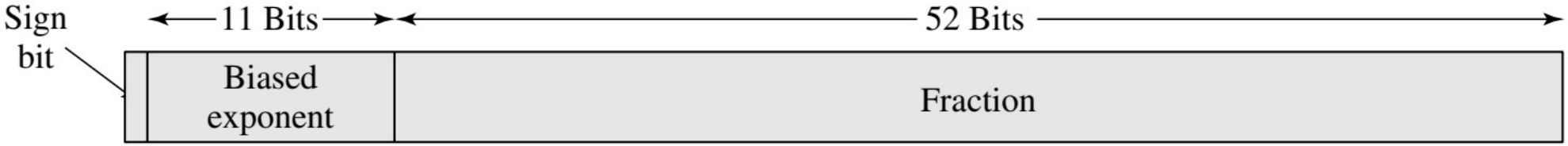
- **Rounding to plus** and **minus infinity** provides an efficient method for monitoring and controlling errors in floating-point computations by producing two values for each result.
- The two values correspond to the lower and upper endpoints of an interval that contains the true result.
- If the range between the upper and lower bounds is sufficiently narrow, then a sufficiently accurate result has been obtained.

Rounding

- **Rounding toward zero** is, in fact, simple truncation: The extra bits are ignored.
- However, the result is that the magnitude of the truncated value is always less than or equal to the more precise original value, and it affects every operation for which there are non-zero extra bits.



(a) Single format



(b) Double format

Parameter	Format			
	Single	**Single Extended**	**Double**	**Double Extended**
Word width (bits)	32	≥43	64	≥79
Exponent width (bits)	8	≥11	11	≥15
Exponent bias	127	unspecified	1023	unspecified
Maximum exponent	127	≥1023	1023	≥16383
Minimum exponent	−126	≤−1022	−1022	≤−16382
Number range (base 10)	$10^{-38}, 10^{+38}$	unspecified	$10^{-308}, 10^{+308}$	unspecified
Significand width (bits)*	23	≥31	52	≥63
Number of exponents	254	unspecified	2046	unspecified
Number of fractions	2^{23}	unspecified	2^{52}	unspecified
Number of values	1.98×2^{31}	unspecified	1.99×2^{63}	unspecified

Special BIT patterns

- An exponent of zero together with a fraction of zero represents positive or negative zero, depending on the sign bit. As was mentioned, it is useful to have an exact value of 0 represented.
- An exponent of all ones together with a fraction of zero represents positive or negative infinity, depending on the sign bit. It is also useful to have a representation of infinity. This leaves it up to the user to decide whether to treat overflow as an error condition or to carry the value and proceed with whatever program is being executed.
- An exponent of zero together with a nonzero fraction represents a denormalized number. In this case, the bit to the left of the binary point is zero and the true exponent is -126 or -1022. The number is positive or negative depending on the sign bit.

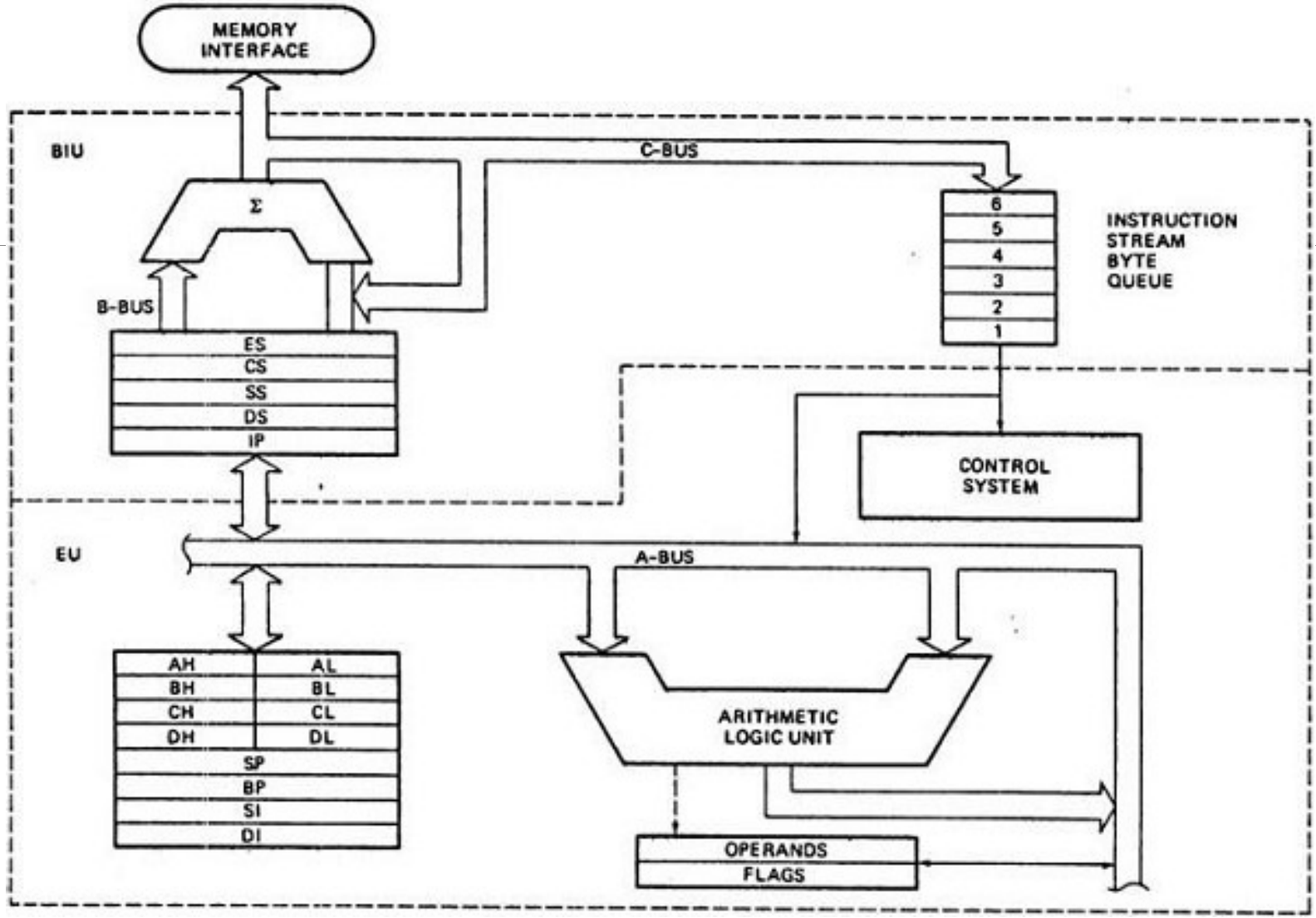
Special BIT patterns

- An exponent of all ones together with a nonzero fraction is given the value NaN, which means *Not a Number,* and is used to signal various exception conditions.

8086 microprocessor

- The 8086 Microprocessor has two units

1. Bus Interface Unit
2. Execution Unit


MEMORY INTERFACE
BIU
C-BUS
Σ
B-BUS
6
5
4
3
2
1
INSTRUCTION STREAM BYTE QUEUE
ES
CS
SS
DS
IP
CONTROL SYSTEM
EU
A-BUS
AH
AL
BH
BL
CH
CL
DH
DL
SP
BP
SI
DI
ARITHMETIC LOGIC UNIT
OPERANDS
FLAGS

Bus interface Unit

- The Bus Interface Unit consists of different units such as the Instruction Queue, Segment Registers, Instruction Pointer, Address adder.
- It interfaces the processor to the outside → performing all the all external bus operations like fetch, read, write, input and output of data.
- The BIU uses instruction queue for pipelined instructions → 6-byte First-in-First-out register.
- It provides a full 16-bit bidirectional data bus and 20-bit address bus.
- Specifically, it performs Instruction fetch, Instruction queuing, Operand fetch and storage, Address relocation and Bus control.
- The BIU uses a mechanism known as an instruction stream queue to implement a pipeline architecture.

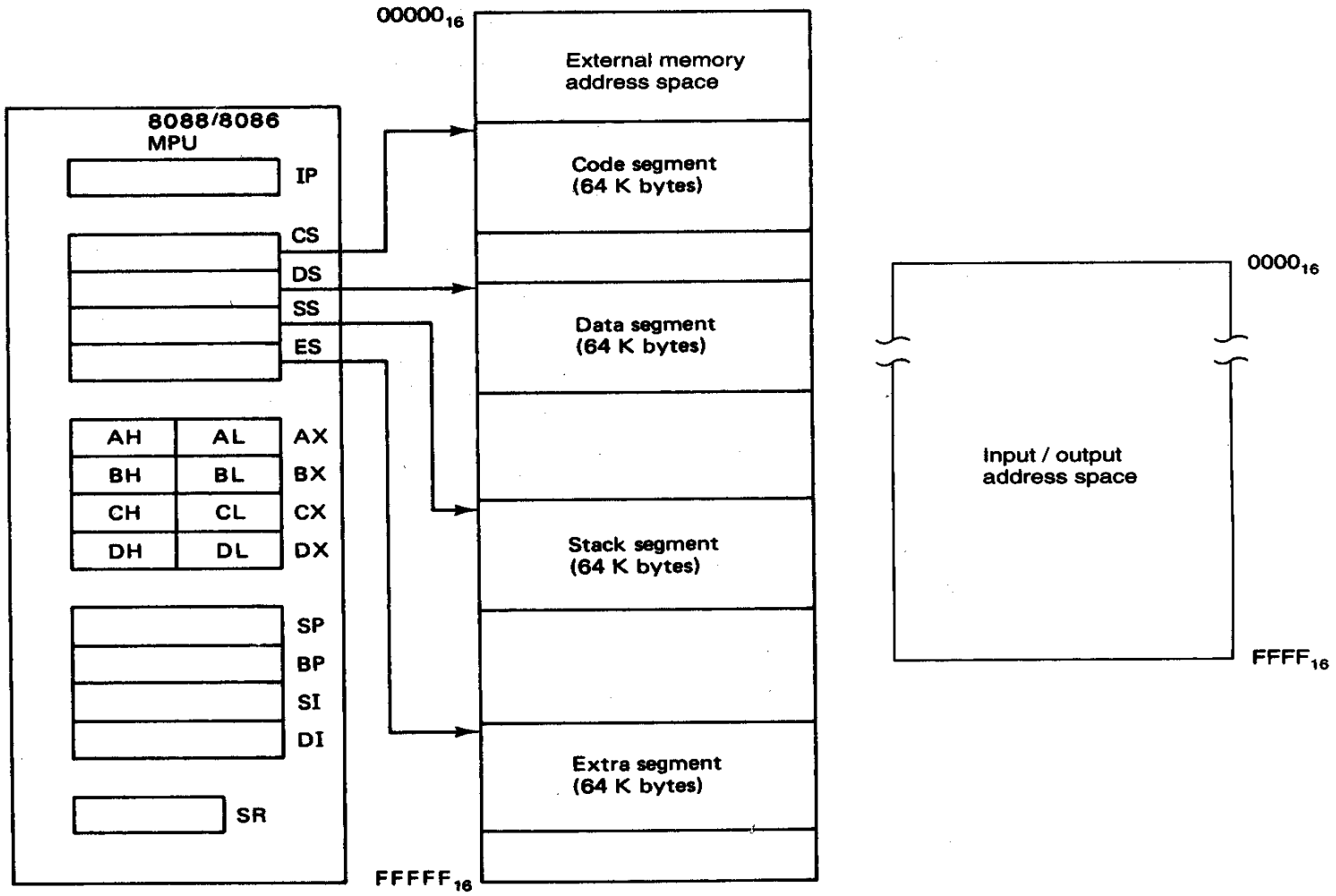
Bus interface Unit

- This queue permits prefetch of up to six bytes of instruction code.
- Whenever the queue of the BIU is not full, it has room for at least two more bytes and at the same time the EU is not requesting it to read or write operands from memory, the BIU is free to look ahead in the program by prefetching the next sequential instruction.
- These prefetching instructions are held in its FIFO queue.
- With its 16-bit data bus, the BIU fetches two instruction bytes in a single memory cycle.
- After a byte is loaded at the input end of the queue, it automatically shifts up through the FIFO to the empty location nearest the output.

Execution Unit

- The Execution Unit consists of the following units such as Control circuitry, Instruction decoder, ALU, Pointer and Index register, Flag register.
- The EU decodes and executes the instructions fetched by the BIU.
- It extracts instructions from the top of the queue in the BIU, decodes them, generates operands if necessary, passes them to the BIU and requests it to perform the read or write bus cycles to memory or I/O and perform the operation specified by the instruction on the operands.
- If the BIU is already in the process of fetching an instruction when the EU requests it to read or write operands from memory or I/O, the BIU first completes the instruction fetch bus cycle before initiating the operand read / write cycle.
- It also tests the status and control flags and updates these flags based on the results of the instruction.

Software Model of the 8086 Microprocessors



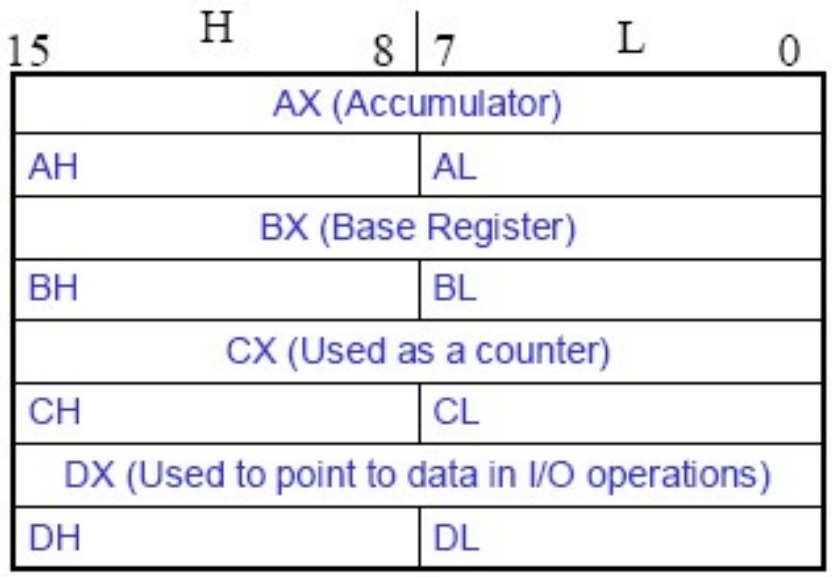
Register organization

- 8086 has a powerful set of registers of 16-bit each
- The registers are categorized into 4 groups

1. General data registers
2. Segment registers
3. Pointer and index registers
4. Flag register

General Registers

- 8086 contains 4 general data registers AX, BX, CX, and DX.
- They are used to hold data, variables, results etc., temporarily for faster operation.



AX - the Accumulator
BX - the Base Register
CX - the Count Register
DX - the Data Register

General Registers

- AX is used as a 16-bit accumulator, with the lower 8-bits designated as AL and higher 8-bits as AH for 8-bit operations.
- It performs all the arithmetic and logic operations and it is also used to store the result of any operation.
- BX register is used as a general purpose register as well as to store the offset for forming physical address in certain addressing modes.
- CX register is used as a default counter in case of string and loop instructions.
- It is also used for the count of the number of bits by which the contents of an operand must be shifted or rotated during the execution of the multibit shift or rotate instructions.

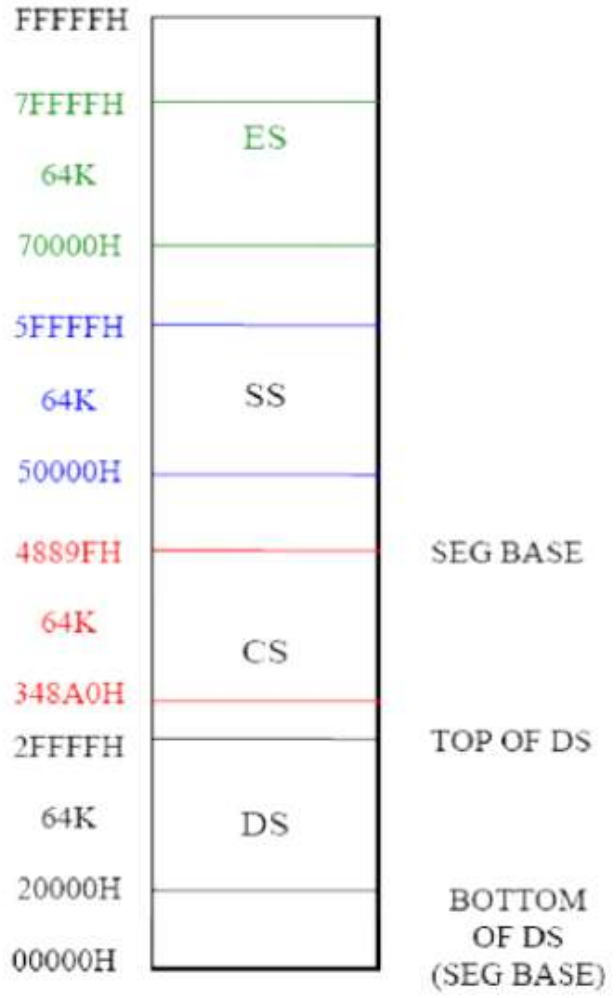
General Registers

- DX register is used in I/O operations to hold the address of the I/O port.
- DX register also holds the remainder after a word division and holds the high-order bits (MSB) of the result after a word multiplication (32-bit).

Segment Registers

- There are 4 segment registers

1. Code segment
2. Data segment
3. Stack segment
4. Extra segment



Segment Registers

- Code segment (CS) register is used to address a memory location in the code segment of memory where the executable program or instructions are stored
- Stack segment (SS) register is used for addressing stack segment of memory which is used to store stack data.
- The data segment (DS) register points to the data segment of the memory where the data is stored
- The extra segment (ES) register points to the extra segment of the memory. This is used as another data segment for extra data storage.

Pointers and index Registers

SP	Stack Pointer
BP	Base Pointer
SI	Source Index
DI	Destination Index
IP	Instruction Pointer

Segment	Offset Registers	Function
CS	IP	Address of the next instruction
DS	BX, DI, SI	Address of data
SS	SP, BP	Address in the stack
ES	BX, DI, SI	Address of destination data (for string operations)

- All 16 bits wide, L/H bytes separately are not accessible
- Used as memory pointers
  - Example: MOV AH, [SI]
    - *Move the byte stored in memory location whose address is contained in register SI to register AH*
- IP is not under direct control of the programmer

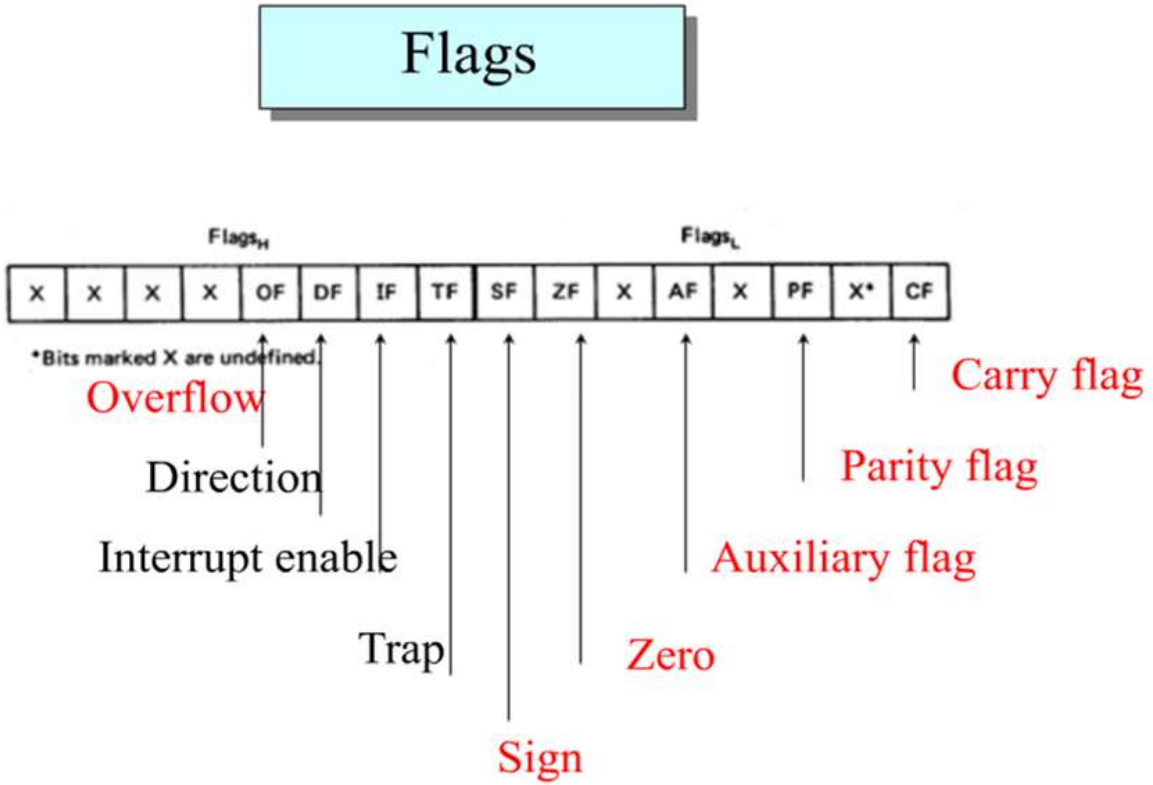
Pointers and index Registers

- The function of IP is similar to a program counter, but it contains the offset address instead of the actual address of the next instruction.
- It contains the offset address within the code segment and the IP is combined with the CS register to generate the actual address of the next instruction to be executed.
- Stack pointer also contains the offset value which is added to the SS register to obtain the actual address of the stack segment.
- Base pointer is similar to the SP since it also contains the offset value pointing to the stack segment.

Pointers and index Registers

- The index registers are used as general purpose registers as well as for offset storage purpose.
- The source index register is used to store the offset of the source data in the data segment.
- And the destination index register is used to store the offset of the destination data in the data or extra segment.

FLAG Register



FLAG Registers

- Overflow flag is based on the (n-1)th bit carry of the ALU result.
- Overflow occurs when signed numbers are added or subtracted.
- For 8-bit operation, if there is carry from the D6 bit to the D7 bit, then the overflow flag is set
- Similarly, for 16-bit operation, if there is carry from the D14 bit to the D15 bit, then the overflow flag is set
- Trap flag is when the processor enters into single step mode or else it is reset.
- In single step mode, the processor executes one instruction at a time and it is useful for debugging programs.

FLAG Registers

- Interrupt flag is set when an maskable interrupt or INTR is received by the processor.
- Direction flag is used for string manipulation instructions i.e. the direction flag selects the increment or decrement mode for DI and SI registers in string instructions
- If DF = 1; then the registers are automatically decremented, or else DF = 0 then the registers are incremented.
- Carry Flag is set when there is a carry generated from the MSB bit addition. Otherwise, CF=0.
- Auxillary flag is set when there is a carry from D3 bit to D4 bit in 8-bit operations / D7 bit to D8 bit in 16-bit operations. If there is no carry generated for these bits, then AF=0.
- Zero flag is set when the result of ALU operation is 0. Otherwise, for any non-zero value, ZF = 0.
- Sign flag is set when the result of ALU operation has 1 as its MSB. Otherwise, SF = 0.
- Parity flag is set when the result of ALU operation has even number of 1's. Otherwise, PF = 0.

Addressing modes

- Addressing mode is the way of locating the data or operands, the types of operands used and the way they are accessed for executing an instruction.
- Based on the flow of instructions, the instructions in 8086 can be categorized as

1. Sequential control flow instructions
2. Control transfer instructions

- Sequential control flow are the instructions in which after execution, the control is transferred to the next instruction appearing immediately after it in the program. Eg. Arithmetic instructions, logical, data transfer, and processor control instructions.
- Control transfer instructions transfer their control to some predefined address after their execution. Eg. INT, CALL, RET, and JUMP instructions.

Addressing modes

1. Register addressing mode
2. Immediate addressing mode
3. Direct addressing mode
4. Register indirect addressing mode
5. Register relative addressing mode
6. Indexed addressing mode
7. Based indexed addressing mode
8. Relative based indexed addressing mode

1. Intrasegment direct addressing mode
2. Intrasegment indirect addressing mode
3. Intersegment direct addressing mode
4. Intersegment indirect addressing mode

Sequential Control flow modes

1. Register addressing mode
2. Immediate addressing mode
3. Direct addressing mode
4. Register indirect addressing mode
5. Register relative addressing mode
6. Indexed addressing mode
7. Based indexed addressing mode
8. Relative based indexed addressing mode

Register addressing mode

- In this mode, both the operands are specified by registers.
- Eg. MOV AX, BX
- All the registers can be used in this mode.
- Both the source and destination registers should be of the same size
- A segment to segment movement of data is not allowed

Immediate addressing mode

- In this mode, the source operand is specified as immediate data byte or word and the destination is either a register or a memory location.
- Eg. MOV AL, 22H; MOV BX, 3456H
- All the registers can be used in this mode.
- Both the source and destination should be of the same size

Direct addressing mode

- In this mode, the source is a memory location and the destination is a register.
- Eg. MOV AL, [1234H]; MOV BX, [5000H]
- Here, a 16-bit memory address i.e. the offset address is directly specified in the instruction as a part of it.
- The content of the physical address which is formed from the offset address is the source data.

Register indirect addressing mode

- Register indirect addressing mode allows data to be addressed at any memory location using the offset registers: BP, BX, DI or SI
- DS is the default segment when the registers BX, DI or SI are used.
- SS is the default segment when the register BP is used.
- Eg. MOV AX, [BX]

Register relative addressing mode

- In this mode, the data in a segment of memory are addressed by adding an 8-bit or 16-bit displacement to the contents of a base register (BX or BP) or an index register (SI or DI).
- ES and DS are the default segments when the registers BX, DI or SI are used.
- SS is the default segment when the register BP is used.
- Eg. MOV AX, 1000H [BX]

Indexed addressing mode

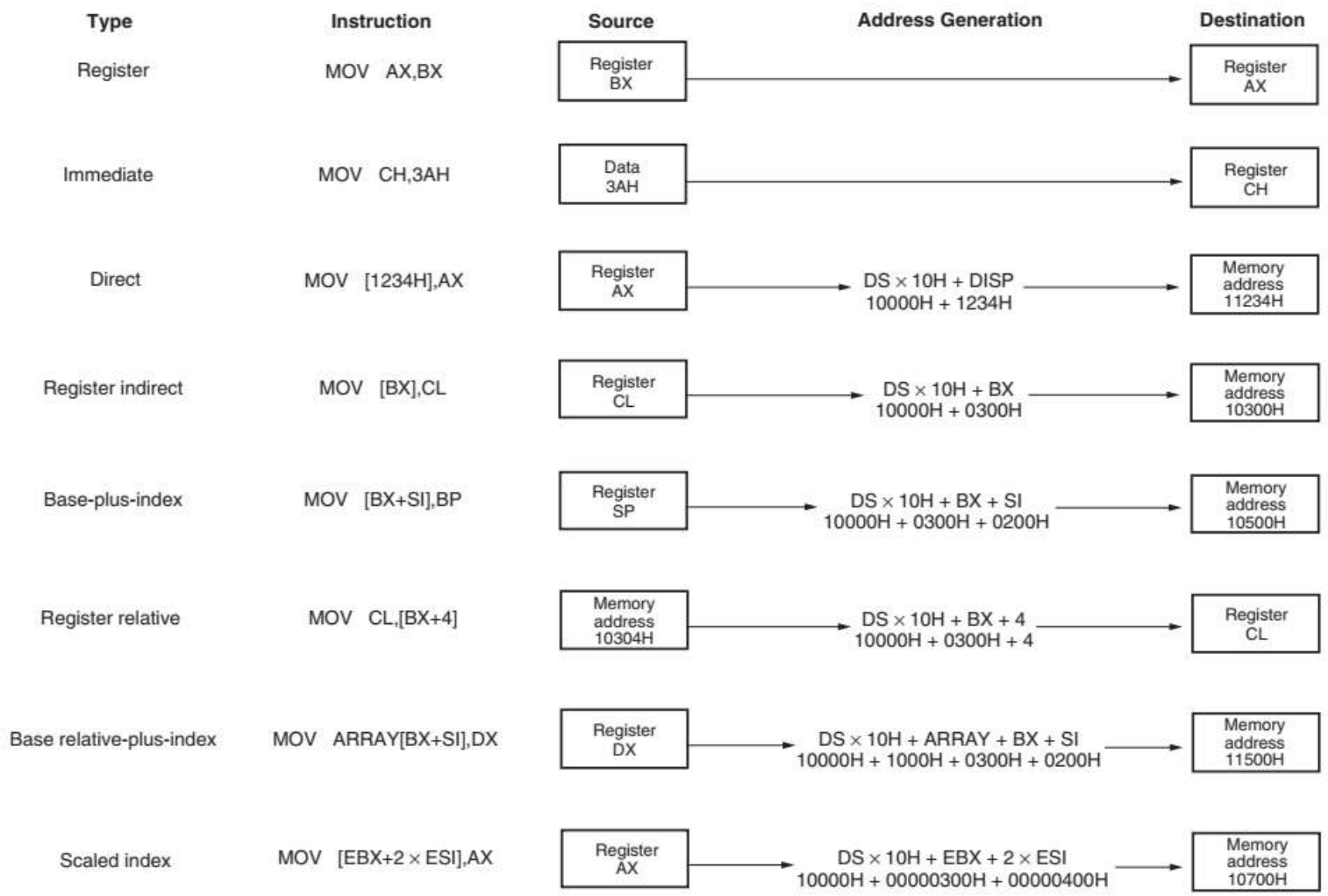
- In this mode, the offset address of the operand is stored in one of the index registers like SI or DI.
- ES and DS are the default segments for DI or SI
- Eg. MOV AX, [SI]

Based-Indexed addressing mode

- In this mode, one base register (BX or BP) and one index register (SI or DI) are used to indirectly address memory.
- The effective address is formed by adding contents of a base register to the contents of the index register.
- Eg. MOV AX, [BX] [DI]

Relative Based-Indexed addressing mode

- It is similar to the based indexed mode, but it adds a displacement along with the base register and index register to form the memory address
- The effective address is formed by adding the 8-bit or 16-bit displacement with the addition result of the base register and the index register.
- Eg. MOV AX, 2000H [BX] [DI]



Notes: EBX = 00000300H, ESI = 00000200H, ARRAY = 1000H, and DS = 1000H

Control transfer instruction modes

1. Intrasegment direct addressing mode
2. Intrasegment indirect addressing mode
3. Intersegment direct addressing mode
4. Intersegment indirect addressing mode

- If the address location to which the control is to be transferred lies in a different segment other than the current one, the mode is called intersegment mode.
- If the destination lies in the same segment, the mode is called intrasegment mode

Intrasegment direct mode

- In this mode, the effective branch address to which the control is to be transferred lies in the same segment in which the control transfer instruction lies and it appears directly in the instruction as an immediate displacement value.
- The effective branch address is the sum of an 8-bit or 16-bit displacement in the current contents of IP.
- Eg. JMP [02](Eff. offset Addr = [IP] + [02])

Intrasegment indirect mode

- In this mode, the effective branch address is the contents of the register or memory location that is accessed using any one of the data addressing modes.
- The contents of the IP will be replaced by the effective branch address
- Eg. JMP BX (Eff. offset Addr = [IP] + [BX])
- In this instruction, the control is jumped to an address specified by the 16-bit register.
- The value of IP+BX is copied into IP with CS value unchanged.
- Then the physical address of the next instruction is obtained using the current content of CS and new value of IP

Intersegment direct mode

- This addressing mode is used to provide means of branching from one segment to another segment.
- Eg. JMP 2000H: 3000H
- i.e. the JMP instruction loads CS with 2000H and loads IP with 3000H

Intersegment indirect mode

- This addressing mode replaces the contents of the IP and CS with the contents of two consecutive words in memory that are addressed using indirect addressing.
- Eg. JMP [5000H] or JMP [BX or SI or DI]
- i.e. the contents of [5000H] & [5000H+1] in DS is loaded into IP and loads the contents of [5000H +2] & [5000H +3] in DS into CS.

What is an Instruction Set?

The complete collection of instructions that are understood by a CPU

Machine Code

Binary

Usually represented by programmer as assembly codes

Elements of an Instruction

Operation code (Op code)
- Do this

Source Operand reference
- To this

Result Operand reference
- Put the answer here

Next Instruction Reference
- When you have done that, do this...

Instruction Representation

In machine code each instruction has a unique bit pattern

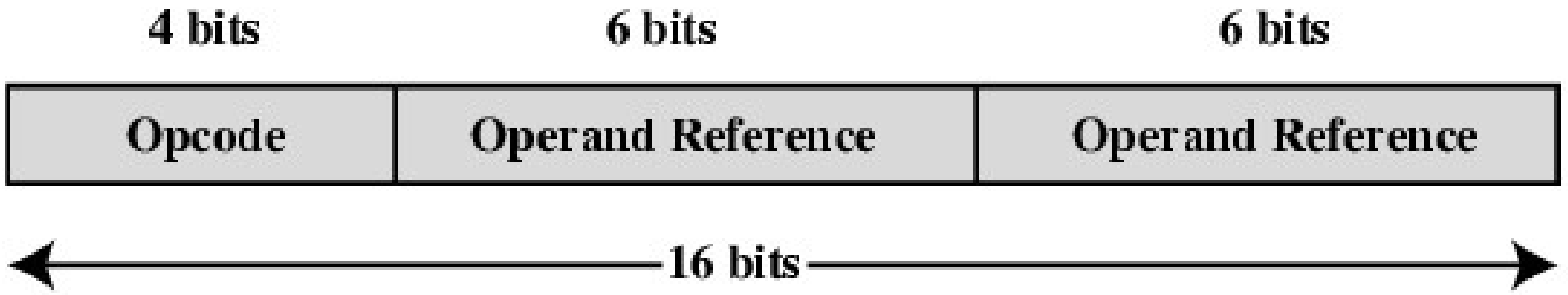
For human consumption (well, programmers anyway) a symbolic representation is used

- e.g. ADD, SUB, LOAD

Operands can also be represented in this way

- ADD A,B

Simple Instruction Format



Instruction Types

Data processing

Data storage (main memory)

Data movement (I/O)

Program flow control

Number of Addresses (a)

3 addresses

- Operand 1, Operand 2, Result
- a = b + c;
- May be a forth - next instruction (usually implicit)
- Not common
- Needs very long words to hold everything

Number of Addresses (b)

2 addresses

- One address doubles as operand and result
- a = a + b
- Reduces length of instruction
- Requires some extra work
  - Temporary storage to hold some results

Number of Addresses (c)

1 address

- Implicit second address
- Usually a register (accumulator)
- Common on early machines

Number of Addresses (d)

0 (zero) addresses

- All addresses implicit
- Uses a stack
- e.g. push a
- push b
- add
- pop c

- c = a + b

Programs to Execute $Y = \dfrac{A - B}{C + (D \times E)}$

Instruction		Comment
SUB	Y, A, B	$Y \leftarrow A - B$
MPY	T, D, E	$T \leftarrow D \times E$
ADD	T, T, C	$T \leftarrow T + C$
DIV	Y, Y, T	$Y \leftarrow Y \div T$

(a) Three-address instructions

Instruction		Comment
MOVE	Y, A	$Y \leftarrow A$
SUB	Y, B	$Y \leftarrow Y - B$
MOVE	T, D	$T \leftarrow D$
MPY	T, E	$T \leftarrow T \times E$
ADD	T, C	$T \leftarrow T + C$
DIV	Y, T	$Y \leftarrow Y \div T$

(b) Two-address instructions

Instruction		Comment
LOAD	D	$AC \leftarrow D$
MPY	E	$AC \leftarrow AC \times E$
ADD	C	$AC \leftarrow AC + C$
STOR	Y	$Y \leftarrow AC$
LOAD	A	$AC \leftarrow A$
SUB	B	$AC \leftarrow AC - B$
DIV	Y	$AC \leftarrow AC \div Y$
STOR	Y	$Y \leftarrow AC$

(c) One-address instructions

Instruction Set

8086 supports 6 types of instructions.

1. **Data Transfer Instructions**
2. **Arithmetic Instructions**
3. **Logical Instructions**
4. **String manipulation Instructions**
5. **Process Control Instructions**
6. **Control Transfer Instructions**

Instruction Set

1. Data Transfer Instructions

Instructions that are used to transfer data/ address in to registers, memory locations and I/O ports.

Generally involve two operands: Source operand and Destination operand of the same size.

Source: Register or a memory location or an immediate data
Destination : Register or a memory location.

The size should be a either a byte or a word.

A 8-bit data can only be moved to 8-bit register/ memory and a 16-bit data can be moved to 16-bit register/ memory.

Instruction Set

1. Data Transfer Instructions

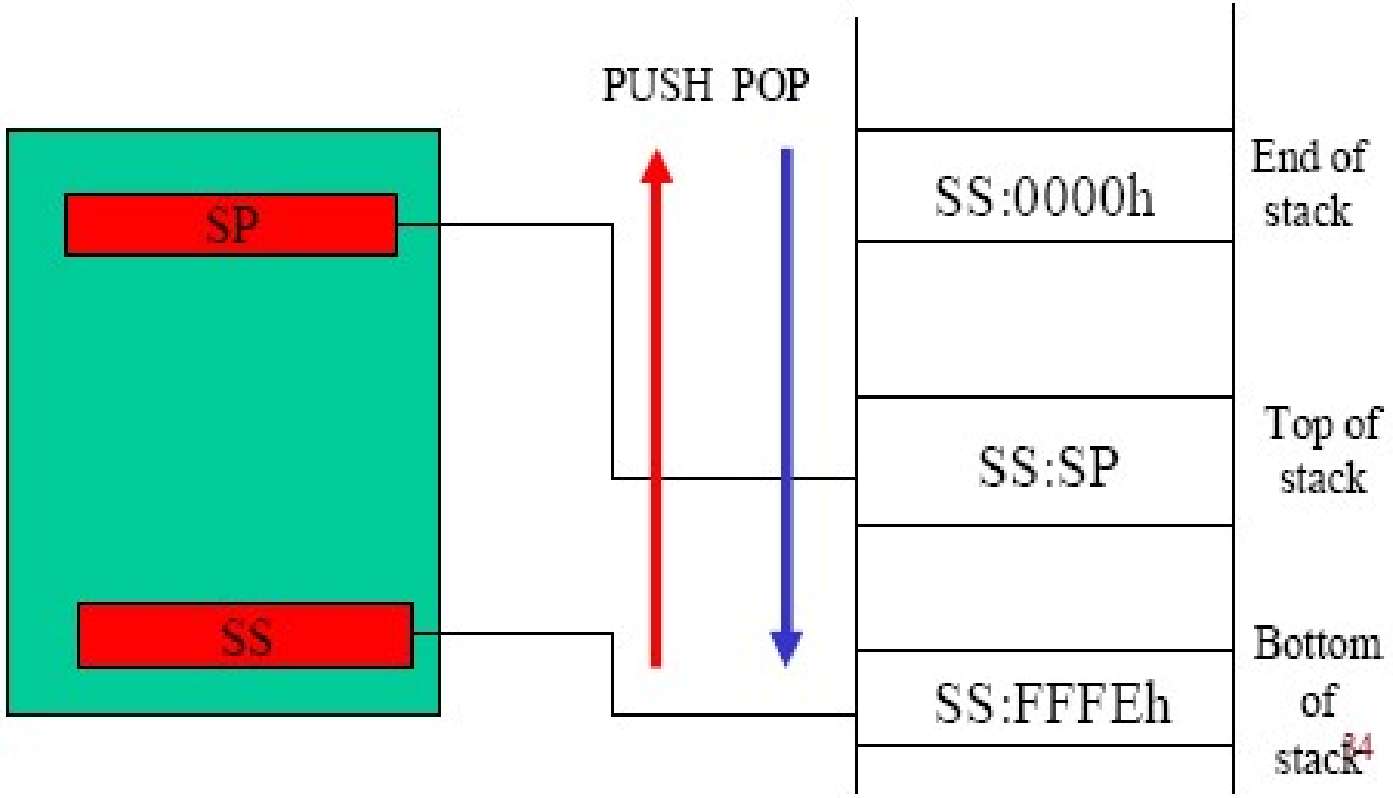
Mnemonics: MOV, XCHG, PUSH, POP, IN, OUT …

MOV reg2/ mem, reg1/ mem	
MOV reg2, reg1	(reg2) ← (reg1)
MOV mem, reg1	(mem) ← (reg1)
MOV reg2, mem	(reg2) ← (mem)
MOV reg/ mem, data	
MOV reg, data	(reg) ← data
MOV mem, data	(mem) ← data
XCHG reg2/ mem, reg1	
XCHG reg2, reg1	(reg2) ↔ (reg1)
XCHG mem, reg1	(mem) ↔ (reg1)

Stack operations

- Data is placed on the stack using the PUSH instruction and removed from the stack using the POP instruction.
- When an item is pushed onto the stack, the processor decrements the SP register, then writes the item at the new top of stack.
- When an item is popped off the stack, the processor reads the item from the top of stack, then increments the SP register.
- Therefore, the stack grows **down** in memory (towards lesser addresses) when items are pushed on the stack and shrinks **up** (towards greater addresses) when the items are popped from the stack.

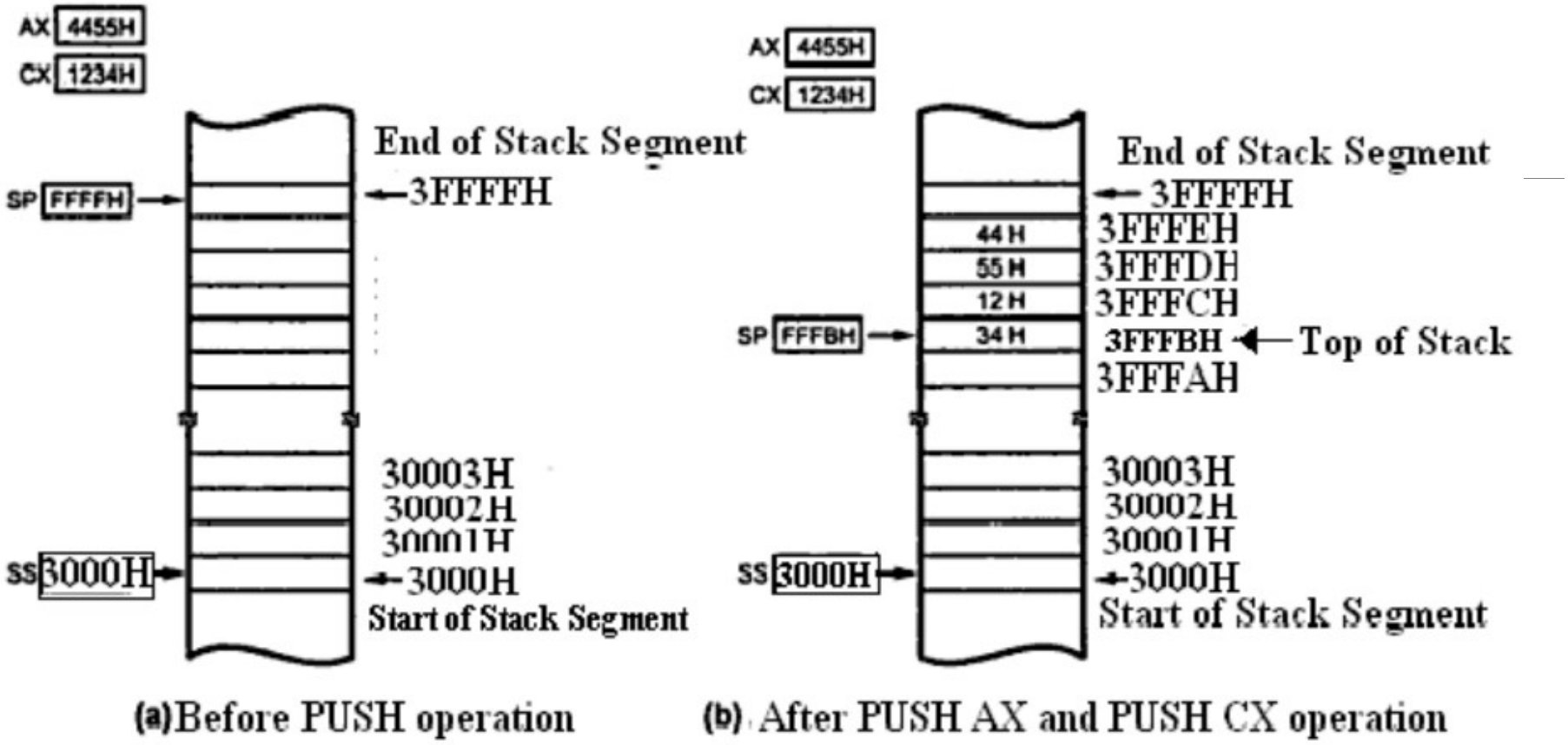
The Stack


PUSH POP
SP
SS
SS:0000h
End of stack
SS:SP
Top of stack
SS:FFFEh
Bottom of stack

PUSH/POP

- These instructions are used to copy a word on top of the stack or remove the word from top of the stack in the register specified.
- The operand in both (PUSH and POP) instructions can be a general purpose register, segment register(except CS) or a memory location.

Instruction	Format	Examples	Comments
PUSH	PUSH operand	PUSH BX	Copies the BH at SP-1 and BL at SP-2. Thus after the complete execution of PUSH instruction SP is decremented by 2, this new value (SP-2) is the new top of stack.
POP	POP operand	POP CX	Copies byte from the top of stack in CL and sets SP to SP+1, copies the byte from this location to CH and sets the SP to SP+1. Thus after the complete execution of POP instruction SP is increments by 2, this new value (SP+2) is the new top of stack.


AX 4455H
CX 1234H
End of Stack Segment
SP FFFFH
3FFFFH
30003H
30002H
30001H
SS 3000H
3000H
Start of Stack Segment
(a) Before PUSH operation
AX 4455H
CX 1234H
End of Stack Segment
3FFFFH
3FFFEH
44H
3FFFDH
55H
3FFFCH
12H
SP FFFBH
34H
3FFFBH
Top of Stack
3FFFAH
30003H
30002H
30001H
SS 3000H
3000H
Start of Stack Segment
(b) After PUSH AX and PUSH CX operation

Instruction Set

1. Data Transfer Instructions

Mnemonics: MOV, XCHG, PUSH, POP, IN, OUT …

PUSH reg16/ mem	
PUSH reg16	**(SP) ← (SP) – 2** **MA_S = (SS) x 16_{10} + SP** **(MA_S ; MA_S + 1) ← (reg16)**
PUSH mem	**(SP) ← (SP) – 2** **MA_S = (SS) x 16_{10} + SP** **(MA_S ; MA_S + 1) ← (mem)**
POP reg16/ mem	
POP reg16	**MA_S = (SS) x 16_{10} + SP** **(reg16) ← (MA_S ; MA_S + 1)** **(SP) ← (SP) + 2**
POP mem	**MA_S = (SS) x 16_{10} + SP** **(mem) ← (MA_S ; MA_S + 1)** **(SP) ← (SP) + 2**

Instruction Set

1. Data Transfer Instructions

Mnemonics: MOV, XCHG, PUSH, POP, IN, OUT ...

IN A, [DX]	
IN AL, [DX]	**$PORT_{addr}$ = (DX)** **(AL) ← (PORT)**
IN AX, [DX]	**$PORT_{addr}$ = (DX)** **(AX) ← (PORT)**
IN A, addr8	
IN AL, addr8	**(AL) ← (addr8)**
IN AX, addr8	**(AX) ← (addr8)**

OUT [DX], A	
OUT [DX], AL	**$PORT_{addr}$ = (DX)** **(PORT) ← (AL)**
OUT [DX], AX	**$PORT_{addr}$ = (DX)** **(PORT) ← (AX)**
OUT addr8, A	
OUT addr8, AL	**(addr8) ← (AL)**
OUT addr8, AX	**(addr8) ← (AX)**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

ADD reg2/ mem, reg1/mem	
ADD reg2, reg1	(reg2) ← (reg1) + (reg2)
ADD reg2, mem	(reg2) ← (reg2) + (mem)
ADD mem, reg1	(mem) ← (mem)+(reg1)
ADD reg/mem, data	
ADD reg, data	(reg) ← (reg)+ data
ADD mem, data	(mem) ← (mem)+data
ADD A, data	
ADD AL, data8	(AL) ← (AL) + data8
ADD AX, data16	(AX) ← (AX) +data16

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

ADC reg2/ mem, reg1/mem	
ADC reg2, reg1 ADC reg2, mem ADC mem, reg1	(reg2) ← (reg1) + (reg2)+CF (reg2) ← (reg2) + (mem)+CF (mem) ← (mem)+(reg1)+CF
ADC reg/mem, data	
ADC reg, data ADC mem, data	(reg) ← (reg)+ data+CF (mem) ← (mem)+data+CF
ADC A, data	
ADC AL, data8 ADC AX, data16	(AL) ← (AL) + data8+CF (AX) ← (AX) +data16+CF

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

SUB reg2/ mem, reg1/mem	
SUB reg2, reg1 **SUB reg2, mem** **SUB mem, reg1**	**(reg2) ← (reg1) - (reg2)** **(reg2) ← (reg2) - (mem)** **(mem) ← (mem) - (reg1)**
SUB reg/mem, data	
SUB reg, data **SUB mem, data**	**(reg) ← (reg) - data** **(mem) ← (mem) - data**
SUB A, data	
SUB AL, data8 **SUB AX, data16**	**(AL) ← (AL) - data8** **(AX) ← (AX) - data16**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

SBB reg2/ mem, reg1/mem	
SBB reg2, reg1	(reg2) ← (reg1) - (reg2) - CF
SBB reg2, mem	(reg2) ← (reg2) - (mem)- CF
SBB mem, reg1	(mem) ← (mem) - (reg1) –CF
SBB reg/mem, data	
SBB reg, data	(reg) ← (reg) – data - CF
SBB mem, data	(mem) ← (mem) - data - CF
SBB A, data	
SBB AL, data8	(AL) ← (AL) - data8 - CF
SBB AX, data16	(AX) ← (AX) - data16 - CF

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

INC reg/ mem	
INC reg8	**(reg8) ← (reg8) + 1**
INC reg16	**(reg16) ← (reg16) + 1**
INC mem	**(mem) ← (mem) + 1**
DEC reg/ mem	
DEC reg8	**(reg8) ← (reg8) - 1**
DEC reg16	**(reg16) ← (reg16) - 1**
DEC mem	**(mem) ← (mem) - 1**

After DEC instruction we can use any JMP (Cond. Or Non-conditional) incase of loop

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

MUL reg/ mem	
MUL reg	**For byte : (AX) ← (AL) x (reg8)** **For word : (DX)(AX) ← (AX) x (reg16)**
MUL mem	**For byte : (AX) ← (AL) x (mem8)**
IMUL reg/ mem	
IMUL reg	**For byte : (AX) ← (AL) x (reg8)** **For word : (DX)(AX) ← (AX) x (reg16)**
IMUL mem	**For byte : (AX) ← (AX) x (mem8)**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

DIV reg/ mem	
DIV reg	For 16-bit :- 8-bit : (AL) ← (AX) :- (reg8) Quotient (AH) ← Remainder
DIV mem	For 16-bit :- 8-bit : (AL) ← (AX) :- (mem8) Quotient (AH) ← Remainder

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

IDIV reg/ mem	
IDIV reg	For 16-bit :- 8-bit : (AL) ← (AX) :- (reg8) Quotient (AH) ← Remainder
IDIV mem	For 16-bit :- 8-bit : (AL) ← (AX) :- (mem8) Quotient (AH) ← Remainder

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

CMP reg2/mem, reg1/ mem	
CMP reg2, reg1	**Modify flags ← (reg2) – (reg1)** **If (reg2) > (reg1) then CF=0, ZF=0, SF=0** **If (reg2) < (reg1) then CF=1, ZF=0, SF=1** **If (reg2) = (reg1) then CF=0, ZF=1, SF=0**
CMP reg2, mem	**Modify flags ← (reg2) – (mem)** **If (reg2) > (mem) then CF=0, ZF=0, SF=0** **If (reg2) < (mem) then CF=1, ZF=0, SF=1** **If (reg2) = (mem) then CF=0, ZF=1, SF=0**
CMP mem, reg1	**Modify flags ← (mem) – (reg1)** **If (mem) > (reg1) then CF=0, ZF=0, SF=0** **If (mem) < (reg1) then CF=1, ZF=0, SF=1** **If (mem) = (reg1) then CF=0, ZF=1, SF=0**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

CMP reg/mem, data	

CMP reg, data	Modify flags ← (reg) – (data)
	
	If (reg) > data then CF=0, ZF=0, SF=0
	If (reg) < data then CF=1, ZF=0, SF=1
	If (reg) = data then CF=0, ZF=1, SF=0

CMP mem, data	Modify flags ← (mem) – (data)
	
	If (mem) > data then CF=0, ZF=0, SF=0
	If (mem) < data then CF=1, ZF=0, SF=1
	If (mem) = data then CF=0, ZF=1, SF=0

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

CMP A, data	
CMP AL, data8	**Modify flags ← (AL) – data8** **If (AL) > data8 then CF=0, ZF=0, SF=0** **If (AL) < data8 then CF=1, ZF=0, SF=1** **If (AL) = data8 then CF=0, ZF=1, SF=0**
CMP AX, data16	**Modify flags ← (AX) – data16** **If (AX) > data16 then CF=0, ZF=0, SF=0** **If (mem) < data16 then CF=1, ZF=0, SF=1** **If (mem) = data16 then CF=0, ZF=1, SF=0**

Eg: SUM OF ‘N’ CONSECUTIVE NUMBERS

```
        MOV SI, 2000
        MOV CL, [SI]
        ----------------
        MOV AL, 00
        MOV BL, 01
LOOP    ADD AL, BL
        INC BL
        DEC CL
        JNZ LOOP
        MOV DI, 2002
        MOV [DI], AX
        HLT
```

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …**

AND A, data AND AL, data8	(AL) ← (AL) & data8
AND AX, data16	(AX) ← (AX) & data16

AND reg/mem, data AND reg, data	(reg) ← (reg) & data
AND mem, data	(mem) ← (mem) & data

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …**

OR reg2/mem, reg1/mem	
OR reg2, reg1	(reg2) ← (reg2) | (reg1)
OR reg2, mem	(reg2) ← (reg2) | (mem)
OR mem, reg1	(mem) ← (mem) | (reg1)

OR reg/mem, data	
OR reg, data	(reg) ← (reg) | data
OR mem, data	(mem) ← (mem) | data

OR A, data	
OR AL, data8	(AL) ← (AL) | data8
OR AX, data16	(AX) ← (AX) | data16

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …**

XOR reg2/mem, reg1/mem	
XOR reg2, reg1	(reg2) ← (reg2) ^ (reg1)
XOR reg2, mem	(reg2) ← (reg2) ^ (mem)
XOR mem, reg1	(mem) ← (mem) ^ (reg1)

XOR reg/mem, data	
XOR reg, data	(reg) ← (reg) ^ data
XOR mem, data	(mem) ← (mem) ^ data

XOR A, data	
XOR AL, data8	(AL) ← (AL) ^ data8
XOR AX, data16	(AX) ← (AX) ^ data16

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …**

TEST reg2/mem, reg1/mem	
TEST reg2, reg1	Modify flags ← (reg2) & (reg1)
TEST reg2, mem	Modify flags ← (reg2) & (mem)
TEST mem, reg1	Modify flags ← (mem) & (reg1)

TEST reg/mem, data	
TEST reg, data	Modify flags ← (reg) & data
TEST mem, data	Modify flags ← (mem) & data

TEST A, data	
TEST AL, data8	Modify flags ← (AL) & data8
TEST AX, data16	Modify flags ← (AX) & data16

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …**

SHR reg/mem

SHR reg

i) SHR reg, 1

ii) SHR reg, CL

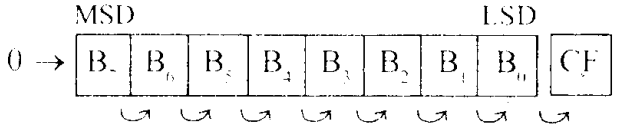
SHR mem

i) SHR mem, 1

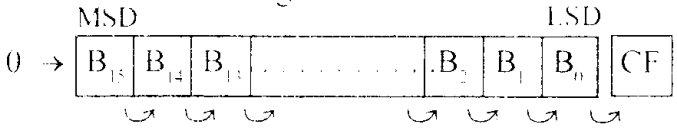
ii) SHR mem, CL

$CF \leftarrow B_{LSD} ; B_n \leftarrow B_{n+1} ; B_{MSD} \leftarrow 0$

reg 8 / mem 8



reg 16 / mem 16



Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …**

SHL reg/mem or SAL reg/mem

SHL reg or SAL reg

i) SHL reg, 1 or SAL reg, 1

ii) SHL reg, CL or SAL reg, CL

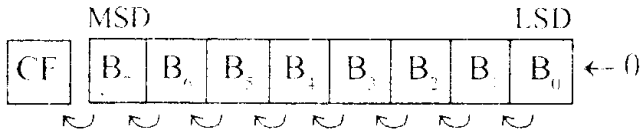
SHL mem or SAL mem

i) SHL mem, 1 or SAL mem, 1

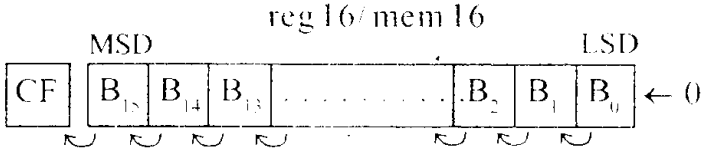
ii) SHL mem, CL or SAL mem, CL

$CF \leftarrow B_{MSD} ; B_{n-1} \leftarrow B_n ; B_{LSD} \leftarrow 0$

reg 8 / mem 8



reg 16 / mem 16



Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …**

RCR reg/mem

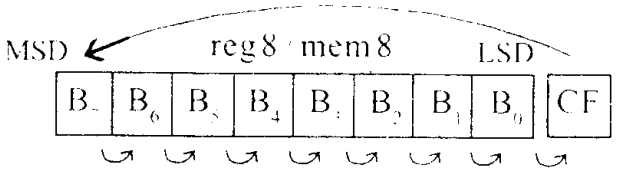
RCR reg

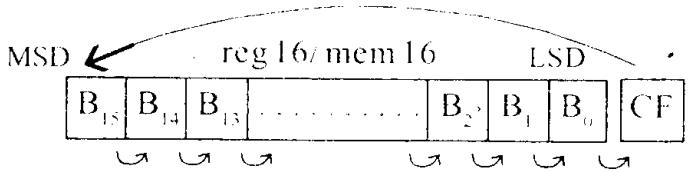
i) RCR reg, 1
ii) RCR reg, CL

RCR mem

i) RCR mem, 1
ii) RCR mem, CL

$B_n \leftarrow B_{n-1}$; $B_{MSD} \leftarrow CF$; $CF \leftarrow B_{LSD}$





Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, ROL ...**

ROL reg/mem

ROL reg

i) ROL reg, 1

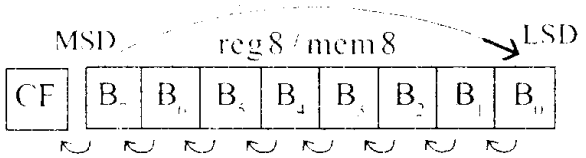
ii) ROL reg, CL

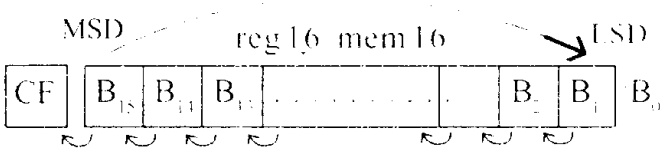
ROL mem

i) ROL mem, 1

ii) ROL mem, CL

$B_{n+1} \leftarrow B_n ; CF \leftarrow B_{MSD} ; B_{LSD} \leftarrow B_{MSD}$





Instruction Set

4. String Manipulation Instructions

- **String : Sequence of bytes or words**
- **8086 instruction set includes instruction for string movement, comparison, scan, load and store.**
- **REP instruction prefix : used to repeat execution of string instructions**
- **String instructions end with S or SB or SW.**
  S represents string, SB string byte and SW string word.
- **Offset or effective address of the source operand is stored in SI register and that of the destination operand is stored in DI register.**
- **Depending on the status of DF, SI and DI registers are automatically updated.**
- **DF = 0 ⇒ SI and DI are incremented by 1 for byte and 2 for word.**
- **DF = 1 ⇒ SI and DI are decremented by 1 for byte and 2 for word.**

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

MOVS	
MOVSB	**MA = (DS) x 16_{10} + (SI)** **MA_E = (ES) x 16_{10} + (DI)** **(MA_E) ← (MA)** **If DF = 0, then (DI) ← (DI) + 1; (SI) ← (SI) + 1** **If DF = 1, then (DI) ← (DI) - 1; (SI) ← (SI) - 1**
MOVSW	**MA = (DS) x 16_{10} + (SI)** **MA_E = (ES) x 16_{10} + (DI)** **(MA_E ; MA_E + 1) ← (MA; MA + 1)** **If DF = 0, then (DI) ← (DI) + 2; (SI) ← (SI) + 2** **If DF = 1, then (DI) ← (DI) - 2; (SI) ← (SI) - 2**

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

REP	
REPZ/ REPE **(Repeat string instruction until ZF = 0)**	**While CX ≠ 0 and ZF = 1, repeat execution of string instruction and** **(CX) ← (CX) – 1**
REPNZ/ REPNE **(Repeat string instruction until ZF = 1)**	**While CX ≠ 0 and ZF = 0, repeat execution of string instruction and** **(CX) ← (CX) - 1**

Example:
MOVSB
REP MOVSB

MOVSW
REP MOVSW

In the example, the first form copies a single byte from the source string, at address DS:SI, to the destination string, at address ES:DI, then increments (or decrements, if the Direction flag is set) both SI and DI.

The second form performs this operation and then decrements CX; if CX is not zero, the operation is repeated.

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, LODS, STOS

Compare two string byte or string word

CMPS	
CMPSB	**MA = (DS) x 16_{10} + (SI)** **MA_E = (ES) x 16_{10} + (DI)** **Modify flags ← (MA) - (MA_E)** **If (MA) > (MA_E), then CF = 0; ZF = 0; SF = 0** **If (MA) < (MA_E), then CF = 1; ZF = 0; SF = 1**
CMPSW	**If (MA) = (MA_E), then CF = 0; ZF = 1; SF = 0** **<u>For byte operation</u>** **If DF = 0, then (DI) ← (DI) + 1; (SI) ← (SI) + 1** **If DF = 1, then (DI) ← (DI) - 1; (SI) ← (SI) - 1** **<u>For word operation</u>** **If DF = 0, then (DI) ← (DI) + 2; (SI) ← (SI) + 2** **If DF = 1, then (DI) ← (DI) - 2; (SI) ← (SI) - 2**

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

Load string byte in to AL or string word in to AX

LODS	
LODSB	**MA = (DS) x 16_{10} + (SI)** **(AL) ← (MA)** **If DF = 0, then (SI) ← (SI) + 1** **If DF = 1, then (SI) ← (SI) – 1**
LODSW	**MA = (DS) x 16_{10} + (SI)** **(AX) ← (MA ; MA + 1)** **If DF = 0, then (SI) ← (SI) + 2** **If DF = 1, then (SI) ← (SI) – 2**

Address	Program
	MOV CL, 08
	MOV SI, 1400
	MOV DI, 1450
Loop	LODSB
	MOV [DI], AL
	INC DI
	DEC CL
	JNC Loop
	INT 3

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

Store byte from AL or word from AX in to string

STOS	
STOSB	$MA_E = (ES) \times 16_{10} + (DI)$ $(MA_E) \leftarrow (AL)$ If DF = 0, then (DI) ← (DI) + 1 If DF = 1, then (DI) ← (DI) – 1
STOSW	$MA_E = (ES) \times 16_{10} + (DI)$ $(MA_E ; MA_E + 1) \leftarrow (AX)$ If DF = 0, then (DI) ← (DI) + 2 If DF = 1, then (DI) ← (DI) – 2

Instruction Set

5. Processor Control Instructions

Mnemonics	Explanation
STC	Set CF ← 1
CLC	Clear CF ← 0
CMC	Complement carry CF ← CF/
STD	Set direction flag DF ← 1
CLD	Clear direction flag DF ← 0
STI	Set interrupt enable flag IF ← 1
CLI	Clear interrupt enable flag IF ← 0
NOP	No operation
HLT	Halt after interrupt is set

Instruction Set

6. Control Transfer Instructions

- **Transfer the control to a specific destination or target instruction**
- **Do not affect flags**

❑ **8086 Unconditional transfers**

Mnemonics	Explanation
CALL reg/ mem/ disp16	Call subroutine
RET	Return from subroutine
JMP reg/ mem/ disp8/ disp16	Unconditional jump

Instruction Set

6. Control Transfer Instructions

- **Checks flags**
- **If conditions are true, the program control is transferred to the new memory location in the same segment by modifying the content of IP**

Instruction Set

6. Control Transfer Instructions

Name	Alternate name
JE disp8 Jump if equal	JZ disp8 Jump if result is 0
JNE disp8 Jump if not equal	JNZ disp8 Jump if not zero
JG disp8 Jump if greater	JNLE disp8 Jump if not less or equal
JGE disp8 Jump if greater than or equal	JNL disp8 Jump if not less
JL disp8 Jump if less than	JNGE disp8 Jump if not greater than or equal
JLE disp8 Jump if less than or equal	JNG disp8 Jump if not greater

Name	Alternate name
JE disp8 Jump if equal	JZ disp8 Jump if result is 0
JNE disp8 Jump if not equal	JNZ disp8 Jump if not zero
JA disp8 Jump if above	JNBE disp8 Jump if not below or equal
JAE disp8 Jump if above or equal	JNB disp8 Jump if not below
JB disp8 Jump if below	JNAE disp8 Jump if not above or equal
JBE disp8 Jump if below or equal	JNA disp8 Jump if not above

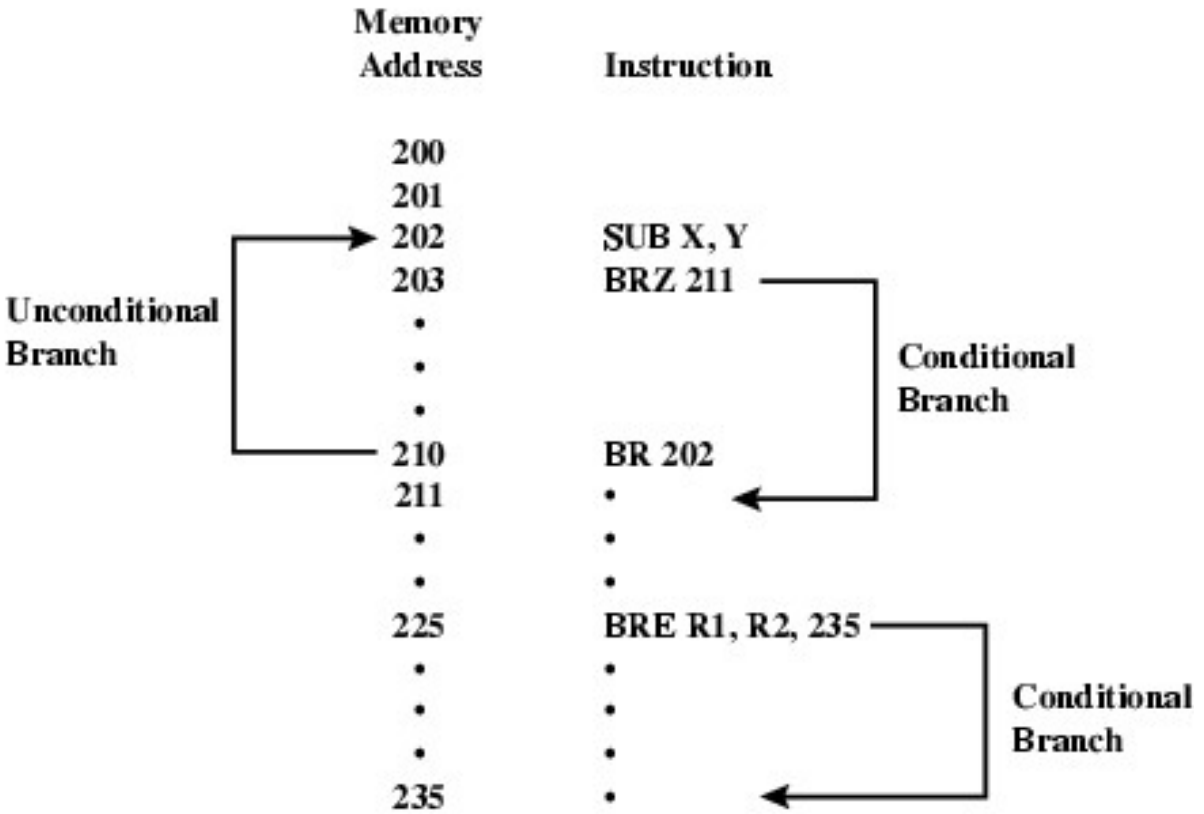
Instruction Set

6. Control Transfer Instructions

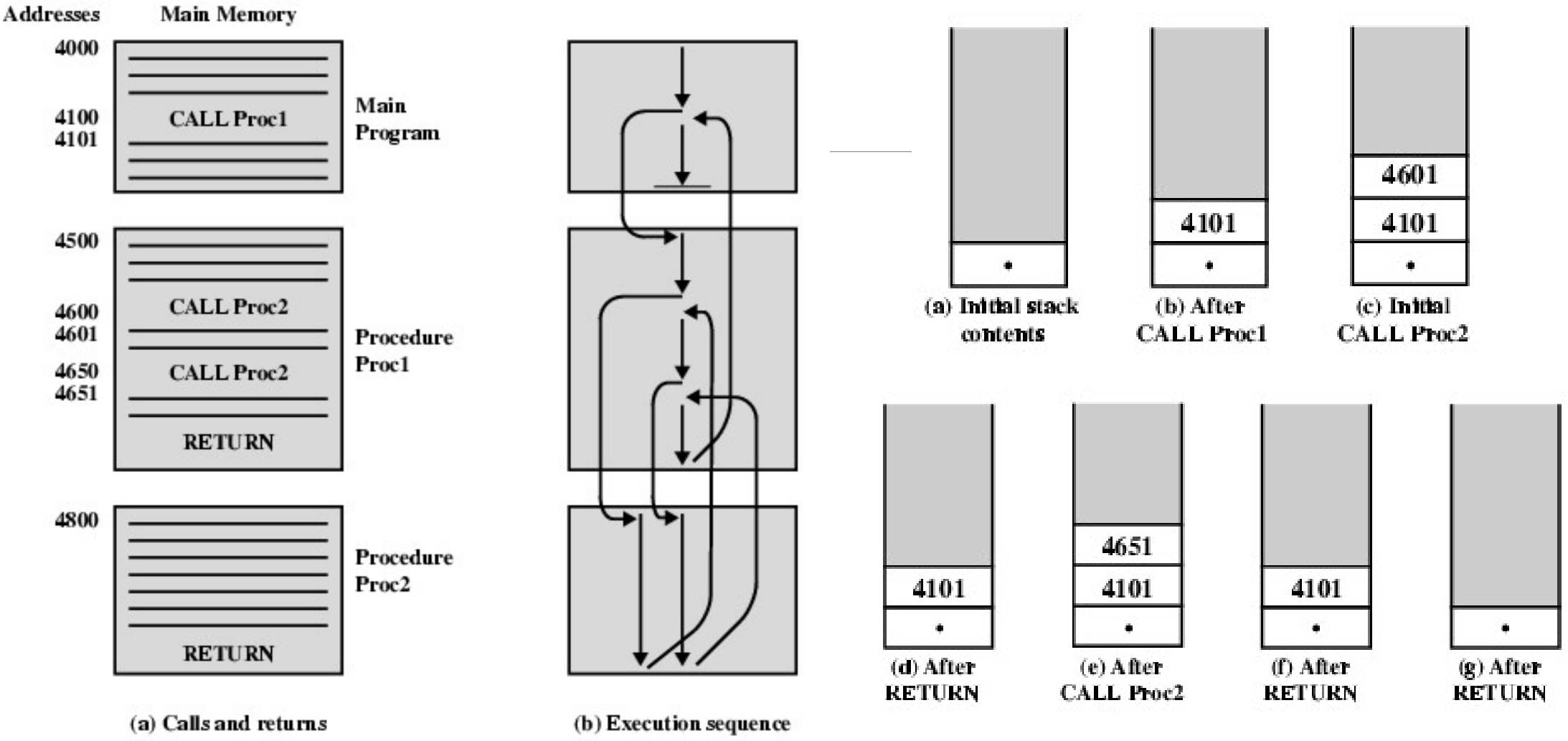
- **8086 conditional branch instructions affecting individual flags**

Mnemonics	Explanation
JC disp8	Jump if CF = 1
JNC disp8	Jump if CF = 0
JP disp8	Jump if PF = 1
JNP disp8	Jump if PF = 0
JO disp8	Jump if OF = 1
JNO disp8	Jump if OF = 0
JS disp8	Jump if SF = 1
JNS disp8	Jump if SF = 0
JZ disp8	Jump if result is zero, i.e, Z = 1
JNZ disp8	Jump if result is not zero, i.e, Z = 1

Branch Instruction



Nested Procedure Calls



String Manipulation Instructions

- A series of data bytes or words available in the memory at consecutive locations are called byte strings or word strings
- Length of the string is stored in CX reg
- For referring to a string, starting/ending address of the string and length of the string is required

LODS/STOS

- Load string byte or word

Eg: LODSB → Loads a byte into AL

AL = DS : [SI]

LODSW → Loads a word into AX

AX = DS : [SI]

Eg: STOSB → Stores a byte in AL

ES : [DI] = AL

STOSW → Stores a word in AX

ES : [DI] = AX

MOVS

- Moves a string byte or word in mem to another mem location

Eg: MOVSB → Moves string byte

ES : [DI] = DS : [SI]

MOVSW → Moves string word

ES : [DI] = DS : [SI]

CMPS

- Compares two strings

Eg: CMPSB → Compares string bytes

ES : [DI] ↔ DS : [SI]

CMPSW → Compares string words

ES : [DI] ↔ DS : [SI]

SCAS

- It scans the string of bytes (SCASB) or word (SCASW) for an operand byte or word specified in the register AL or AX

Eg: SCASB → Compares AL with byte in memory

AL ↔ ES : [DI]

SCASW → Compares string words

AX ↔ ES : [DI]

SCAS

- It scans the string of bytes (SCASB) or word (SCASW) for an operand byte or word specified in the register AL or AX

Eg: SCASB → Compares AL with byte in memory

AL ↔ ES : [DI]

SCASW → Compares string words

AX ↔ ES : [DI]

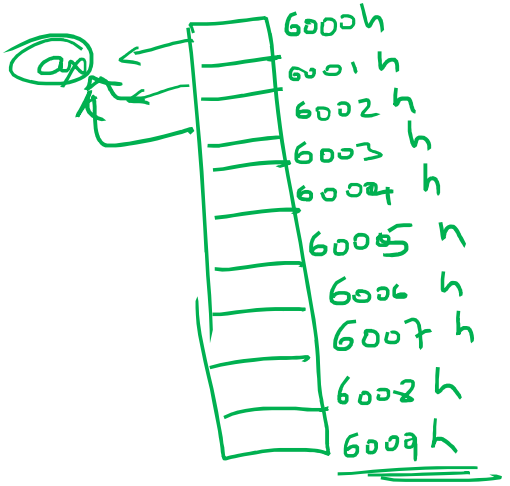
REP

Prefix	Used with:	Meaning
REP	MOVS STOS	Repeat while not end of string CX ≠ 0
REPE/REPZ	CMPS SCAS	Repeat while not end of string and strings are equal CX ≠ 0 and ZF = 1
REPNE/REPNZ	CMPS SCAS	Repeat while not end of string and strings are not equal CX ≠ 0 and ZF = 0

Q: Write a program to find the addition of 10 nos at offset location 6000h.

Sol:-

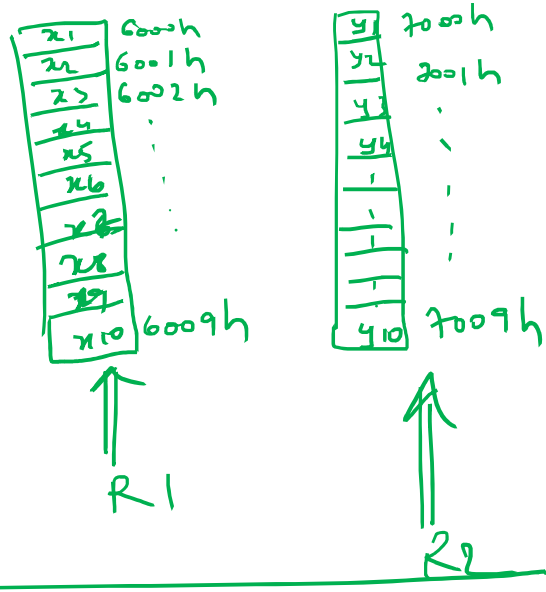
```
        org 100h
        mov SI, 6000h
        mov ax, [SI]
        inc SI
        add ax, [SI]
next:   inc SI
        adc ax, [SI]
        cmp SI, 6009h
        jnz next
        ret
```



Q. WAP to add 10 nos. at 6000h & 7000h location seperately and subtract the lighter result from the lower one. Store the result in 8000h.

Sol:-

```
      org 100h
      mov SI, 6000h
      mov al, [SI]
next: inc SI
      add al, [SI]
      cmp SI, 6009h
      jnz next
      mov [9000h], al
      mov SI, 7000h
      mov al, [SI]
next1: inc SI
      add al, [SI]
      cmp SI, 7009h
      JNZ next
      mov bL, [9000h]
      cmp al, bl
      JG code1
      sub bL, aL
      mov [8000h], bl
      jmp exit
code1: sub al, bl
      mov [8000h], al
exit: ret
```



Q:- WAP to multiply two nos at two labels 'operand1' & 'operand2'. Send the result to a port of 8086 having address 6791h.

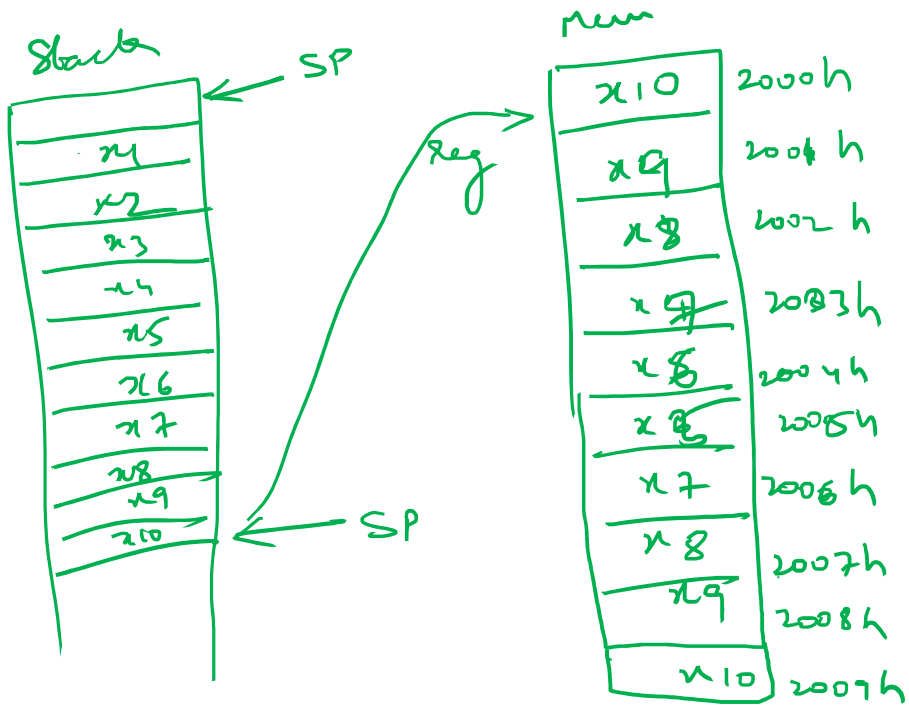
Sol:-

```
lea  SI, operand1
mov  al, [SI]
lea  SI, operand2
mov  bl, [SI]
mul  bl
mov  dx, 6791h
out  dx, ax
```

Q:- WAP to reverse an array ~~and~~ having 10 values using stack operations.

Sol:-

```
        org 100h
        mov SI, 2000h
next:   mov al, [SI]
        push al
        inc SI
        cmp SI, 200Ah
        JNZ next
        mov SI, 2000h
next1:  pop al
        mov [SI], al
        inc SI
        cmp SI, 200Ah
        JNZ next1
        ret
```

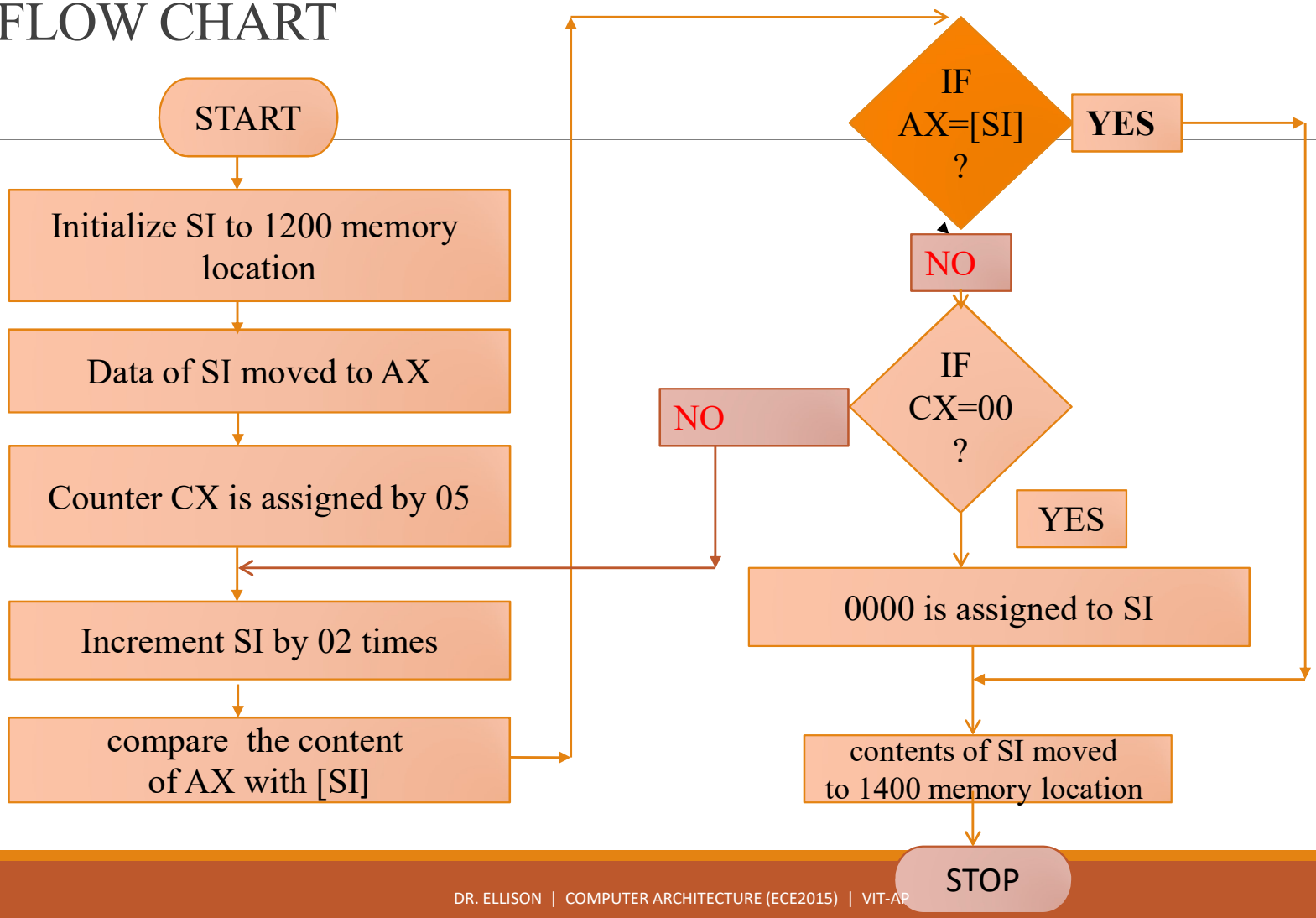


WAP to add a series of 8 bit numbers stored in memory locations starting from 1200 to 1209. Store the result in 120A

```
        MOV SI,1200H
        MOV AL,00H
        MOV CL,0AH
L1:     ADD AL,[SI]
        INC SI
        DEC CL
        JNZ L1
        MOV [SI],AL
        HLT
```

Searching the existence of a certain data in a given data array

FLOW CHART


START
Initialize SI to 1200 memory location
Data of SI moved to AX
Counter CX is assigned by 05
Increment SI by 02 times
compare the content of AX with [SI]
IF AX=[SI] ?
YES
NO
IF CX=00 ?
NO
YES
0000 is assigned to SI
contents of SI moved to 1400 memory location
STOP

PROGRAM

MEMORY ADDRESS	DATA IN
1200	AB96
1202	89CD
1204	AB96
1206	4EDF
1208	9197
120A	9600

Memory address	label	mnemonics
1000		MOV SI,1200H
		MOV AX,[SI]
		MOV CX,0005H
	GG	INC SI
		INC SI
		CMP AX,[SI]
		JE SS
		DEC CX
		JNZ GG
		MOV SI,0000H
	SS	MOV [1400], SI
		HLT

MEMORY ADDRESS	DATA IN
1200	AB96
1202	89CD
1204	AB96
1206	4EDF
1208	9197
120A	9600

MEMORY ADDRESS	DATA OUT
1400	1204

To separate odd and even numbers in a given data array

Address	Program	Explanation
	MOV CL, 06	Set counter in CL register
	MOV SI, 1600	Set source index as 1600
	MOV DI, 1500	Set Destination index as memory address 1500
Loop1:	MOV AL,[SI]	Load data from source memory
	ROR AL, 01	Rotate AL once to right
	JNC Loop1	If bit is one Jump to Loop1
	ROL AL,01	Rotate AL once to left
	MOV [DI], AL	Move result to Destination
	INC SI	Increment Destination index
	INC DI	Increment Destination index
	DEC CL	Decrement the count
	JNZ Loop1	Jump if CL not 0 to Loop1
	HLT	Stop the program

INPUT & OUTPUT:

INPUT	1600	2
	1601	5
	1602	7
	1603	6
	1604	12
	1605	15
OUTPUT	1500	5
	1501	7
	1503	15

Similarly write a program to separate –ve numbers from +ve numbers in a given set of data

WAP to count the number of odd and even numbers in a given set of data array

WAP to find the greatest number in a given set of data array

Memory address(offset	Data (Input)
1500	05
1501	25
1502	35
1503	20
1504	30
1505	15

Memory address(offset	Data (Output)
1600	35

	MNEMONICS	COMMENT
	MOV SI, 1500H	SI<-1500
	MOV CL, [SI]	CL<-[SI]
	INC SI	SI<-SI+1
	MOV AL, [SI]	AL<-[SI]
	DEC CL	CL<-CL-1
	INC SI	SI<-SI+1
L1:	CMP AL, [SI]	AL-[SI]
	JNC L2	JUMP TO L2 IF CY=0
	MOV AL, [SI]	AL<-[SI]
L2:	INC SI	SI<-SI+1
	LOOP L1	**CX<-CX-1 & JUMP TO L1 IF CX NOT 0**
	MOV [1600], AL	AL->[1600]
	HLT	END

How the program will be modify to write for finding the smallest number?

Write a program to sort a 8 bit data array in ascending order. The array consists of 5 numbers starting from location 3000H:4000H.

```
        MOV AX,3000H
        MOV DS,AX
        MOV CH, 04H
L3:     MOV CL,04H
        MOV SI, 4000H
L2:     MOV AL, [SI]
        MOV AH, [SI+01H]
        CMP AL, AH
        JB L1
        JZ L1
        MOV [SI+1], AL
        MOV [SI], AH
L1:     INC SI
        DEC CL
        JNZ L2
        DEC CH
        JNZ L3
        HLT
```

Address offset	Initial data	Ext loop 1	Ext loop 2	Ext loop 3	Ext loop 4
4000	55	45	35	25	15
4001	45	35	25	15	25
4002	35	25	15	35	35
4003	25	15	45	45	45
4004	15	55	55	55	55

WAP to reverse a string of 10 bytes using stack. Check whether the string is palindrome or not. If it is palindrome display FFH on a display unit whose address is 52H else display 00H.

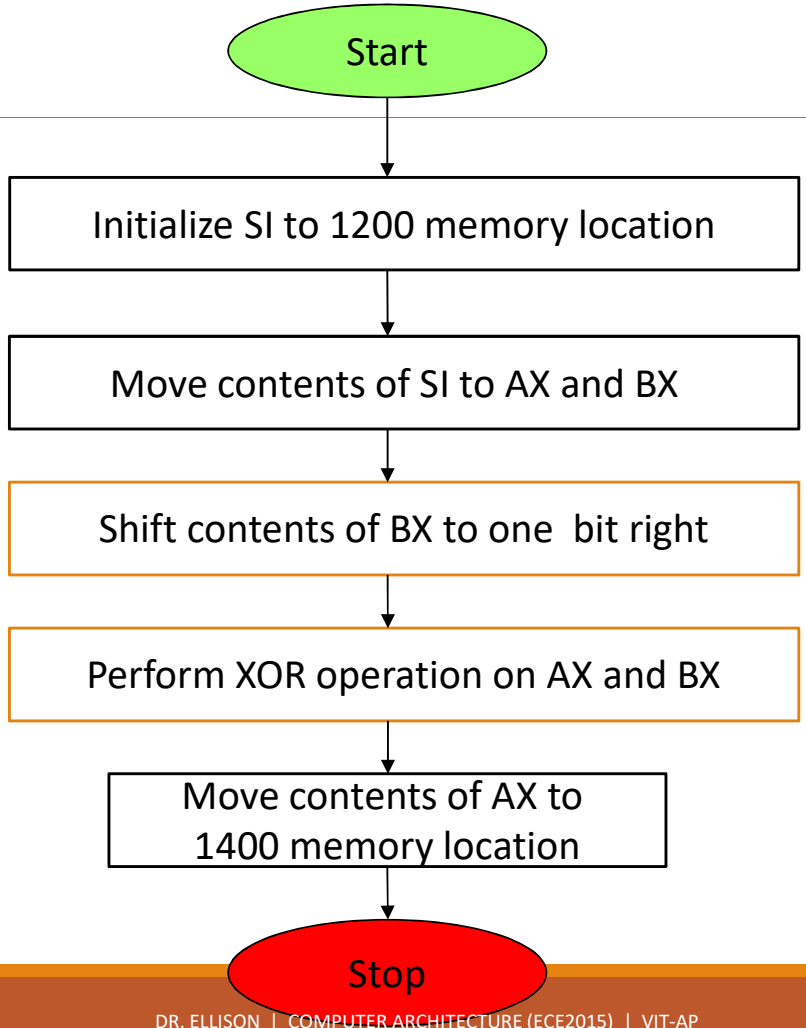
```
        MOV SI, 2300H
        MOV DI, 2500H
        MOV CL, 0AH
L1:     MOV AL, [SI]
        PUSH AL
        INC SI
        DEC CL
        JNZ L1
        MOV CL, 0AH
L2:     POP AL
        MOV [DI], AL
        INC DI
        DEC CL
        JNZ L2
```

SI		STACK		DI	
2300	11	FFF1	AA	2500	AA
2301	22	FFF2	99	2501	99
2302	33	FFF3	88	2502	88
2303	44	FFF4	77	2503	77
2304	55	FFF5	66	2504	66
2305	66	FFF6	55	2505	55
2306	77	FFF7	44	2506	44
2307	88	FFF8	33	2507	33
2308	99	FFF9	22	2508	22
2309	AA	FFFA	11	2509	11

ALP to find the Sum of cubes of an array of size 10 by using 8086.

```
        MOV SI,0200 H
        MOV DI,0220H
        MOV CL,0AH
Up:     MOV AL,[SI]
        MOV BL,AL
        MUL BL
        MUL BL
        MOV [DI],AX
        INC SI
        INC DI
        INC DI
        DEC CL
        JNZ Up
        HLT
```

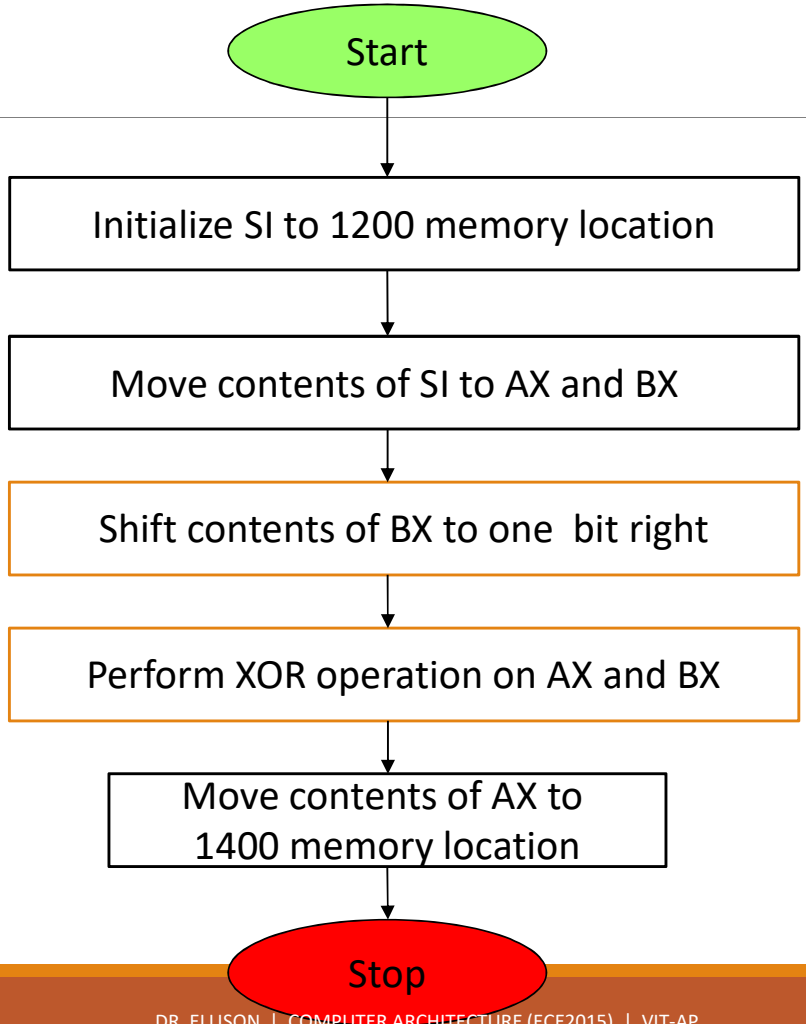
Write an ALP to convert binary number to gray code



Program

```
MOV SI,  1200
MOV AX,  [SI]
MOV BX,  [SI]
SHR BX,  01
XOR AX,  BX
MOV  [1400], AX
HLT
```

Write an ALP to convert binary number to gray code



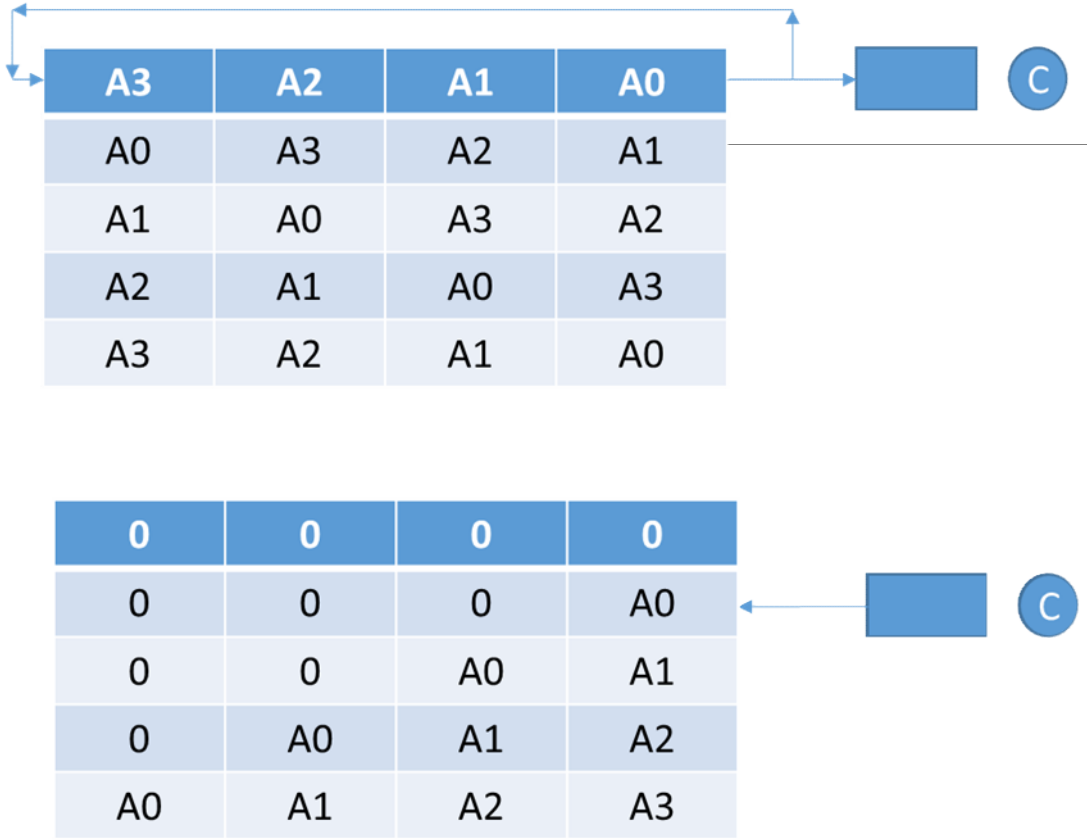
A3	A2	A1	A0
0	A3	A2	A1

A3XOR0	A2XORA3	A1XORA2	A0XORA1

Program

```
MOV SI,  1200
MOV AX,  [SI]
MOV BX,  [SI]
SHR BX,  01
XOR AX,  BX
MOV  [1400], AX
HLT
```

Program to check a number for bit wise palindrome. If palindrome place FFH at 2500H or place 00H at 2500H

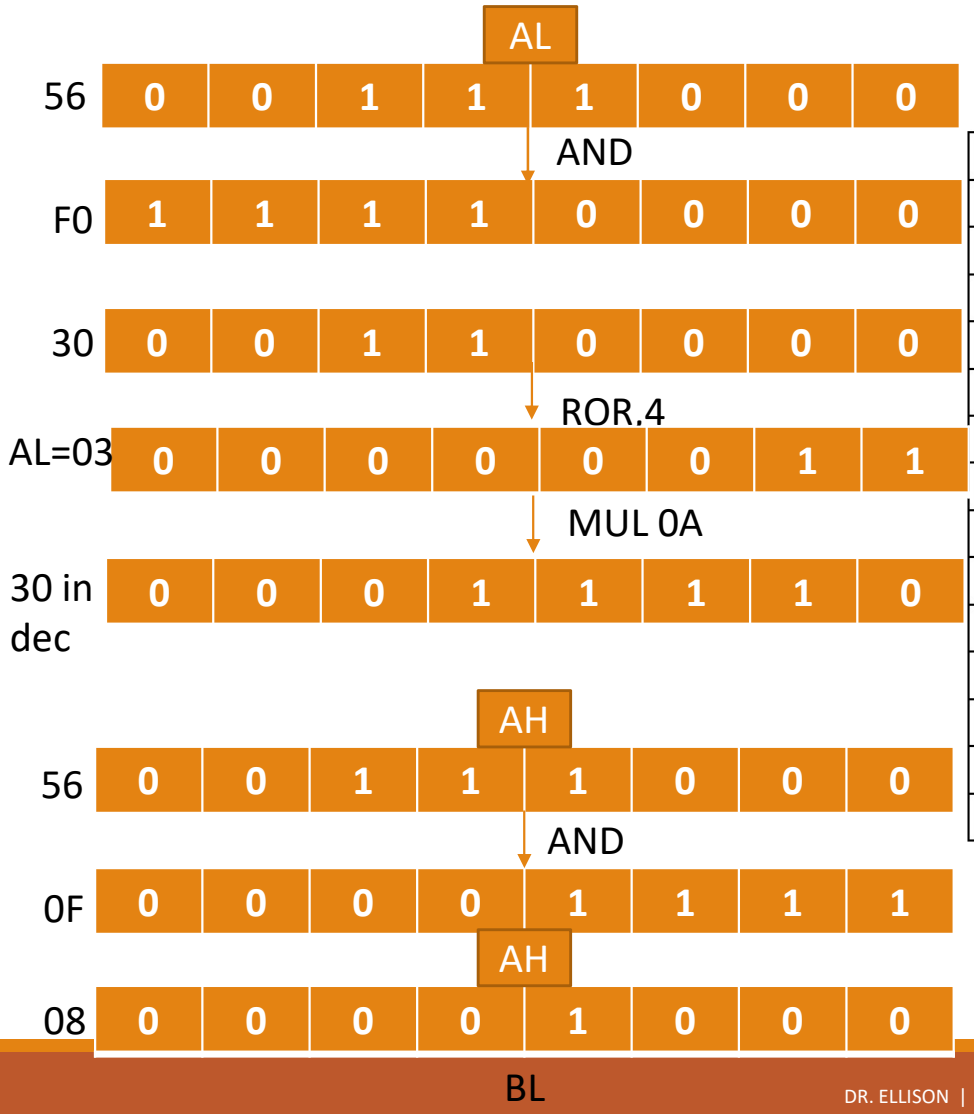



```
        MOV AX, [2300H]
        MOV CL,10H ;Initialize the counter 10.
UP:     ROR AX,1 ;Rotate right one time.
        RCL DX,1 ;Rotate left with carry one time.
        DEC CL
        JNZ UP ;Loop the process.
        CMP AX,DX ;Compare AX and DX.
        JNZ DOWN ;If no zero go to DOWN label.
        MOV [2500H], FFH ;Declare as a PALINDROME.
        JMP EXIT ;Jump to EXIT label.
DOWN:   MOV [2500H], 00H ; Declare as not a PALINDROME
EXIT:   HLT
```

Program to get the square root of a number

```
      M0V AX, [0500H]        move the data from offset 500 to register AX
      MOV CX, 0000H          move 0000 to register CX
      MOV BX, FFFFH          move FFFF to register BX
L1:   ADD BX, 0002H          add BX and 02
      INC CX                 increment the content of CX by 1
      SUB AX, BX             subtract contents of BX from AX
      JNZ L1                 jump to address 040A if zero flag(ZF) is 0
      MOV [0600], CX         store the contents of CX to offset 600
      HLT                    end the program
```

CONVERSION OF BCD TO HEXADECIMAL



Program	Explanation
MOV SI, 1600	Set source index as 1600
MOV DI, 1500	Set Destination index as memory address 1500
MOV AL, [SI]	Load BCD number into AL register
MOV AH, AL	Move content of AL to AH
AND AH, 0F	Mask MSD of BCD number
MOV BL, AH	Save LSD in BL register
AND AL, F0	Mask LSD of BCD number
MOV CL, 04	Load 04 to counter
ROR AL, CL	rotate AL by counter times
MOV BH, 0A	move 0A to BH register
MUL BH	Multiply BH register
ADD AL, BL	Add AL, BL
MOV [DI], AL	Move result to Destination
HLT	Stop

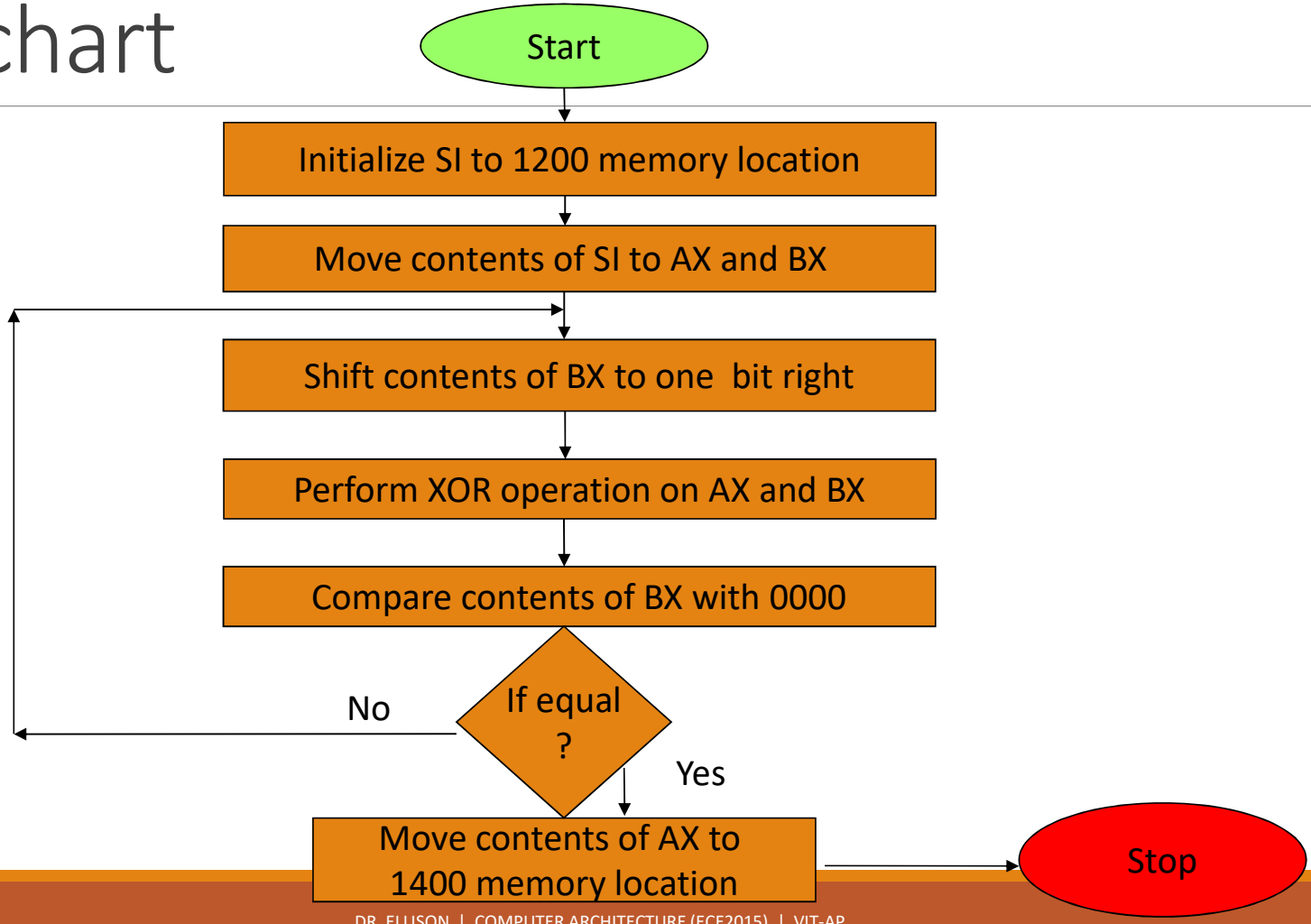
INPUT & OUTPUT:

INPUT	1600	56
OUTPUT	1500	38

Write a program to get Factorial of 10 numbers stored from the starting location 4000H:1000H. The results should be stored in 4000H:2000H

```
      MOV 4000H, AX
      MOV DS, AX
      MOV SI,1000H
      MOV DI, 2000H
      Mov CL,0AH
      MOV AL, 01H
Next: MOV BL,[SI]
LOOK: MUL BL
      DEC BL
      JNZ LOOK
      MOV [DI], AL
      INC SI
      INC DI
      LOOP NEXT
      HLT
```

Flowchart


Start
Initialize SI to 1200 memory location
Move contents of SI to AX and BX
Shift contents of BX to one bit right
Perform XOR operation on AX and BX
Compare contents of BX with 0000
If equal ?
No
Yes
Move contents of AX to 1400 memory location
Stop

Program

	Mnemonics
	MOV SI, 1200
	MOV AX, [SI]
	MOV BX, [SI]
LOOP 1	SHR BX, 01
	XOR AX, BX
	CMP BX, 0000
	JE LOOP 2
	JMP LOOP 1
LOOP 2	MOV [1400], AX
	HLT

Program to find Greatest common divisor (GCD) of given numbers

```
        MOV SI, 2300H           ;Store Offset address 2300h in SI
        MOV DI, 2400H           ;Store offset address 2400h in DI
        MOV AX, [SI]    ;Move the first number to AX.
        MOV BX, [SI+1]          ;Move the second number to BX.
UP:     CMP AX, BX      ;Compare the two numbers.
        JE EXIT                 ;If equal, go to EXIT label.
        JB EXCG                 ;If first number is below than second, go to EXCG label.
UP1:    MOV DX,0000H            ;Initialize the DX.
        DIV BX                  ;Divide the first number by second number.
        CMP DX,0000H            ;Compare remainder is zero or not.
        JE EXIT                 ;If zero, jump to EXIT label.
        MOV AX,DX       ;If non-zero, move remainder to AX.
        JMP UP                  ;Jump to UP label.
EXCG:   XCHG AX,BX      ;Exchange the remainder and quotient.
        JMP UP1                 ;Jump to UP1.
EXIT:   MOV [DI], BX    ;Store the result in DI.
        HLT                     ; Stop
```

Program to find least common multiple (LCM) of a given numbers

```
        MOV SI, 2300H       ;Store Offset address 2300h in SI
        MOV DI, 2400H       ;Store offset address 2400h in DI
        MOV DX,0000H        ;Initialize the DX
        MOV AX,[SI]         ;Move the first number to AX
        MOV BX,[SI+2]       ;Move the second number to AX
UP:     PUSH AX             ;Store the quotient/first number STACK
        PUSH DX             ;Store the remainder STACK
        DIV BX              ;Divide the first number by second number
        CMP DX,0000H        ;Compare the remainder.
        JE EXIT             ;If remainder is zero, go to EXIT label
        POP DX              ;If remainder is non-zero, ;Retrieve the remainder from stack
        POP AX              ;Retrieve the quotient from stack
        ADD AX,[SI]         ;Add first number with AX
        JNC DOWN            ;If no carry jump to DOWN label
        INC DX              ;Increment DX
DOWN:   JMP UP              ;Jump to Up label
EXIT:   MOV [DI], AX        ;If remainder is zero, store the value of LCM at destination
```

Some important programs

Program to count logical 1’s and 0’s in a given data

Program for getting square of array of numbers

Program to find LCM of a given numbers

Program to find GCD of a given numbers

Program to check a number is Bit wise palindrome or not

Program to check a 16 bit number is Nibble wise palindrome or not

Program to reverse a string

Program to search for a character in a string

Write a program to swap the nibbles of 10 data stored in the memory which starts from 2000

```
        MOV CL, 0A
        MOV SI, 2000
        MOV DI, 3000
L1:     MOV AL, [SI]
        ROR AL, 04
        MOV [DI], AL
        INC SI
        INC DI
        DEC CL
        JNZ L1
        HLT
```

Consider all the 8 bits of an output port having address 54H are connected to 8 LEDs. Write a program to blink all the LEDs ON and OFF with a time gap of 5 seconds. Consider the time taken by any instruction to be executed is 1 Second.

```
L1:     MOV AL, FFH
        OUT 54H
        CALL DELAY
        MOV AL, 00H
        OUT 54H
        CALL DELAY
        JMP L1
```


```
DELAY:  MOV CL, 02H
L2:     DEC CL
        JNZ L2
        RET
```

Write a program to add the contents of the memory location 0500H to register BX and CX. Add immediate byte 05H to the data residing in memory location, whose address is computed using offset=0600H. Store the result of the addition in 0700H. Assume data segment's starting physical address is 20000H.

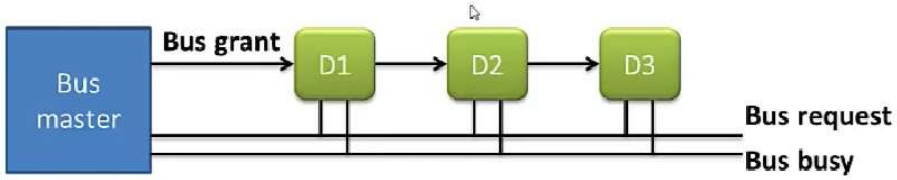
```
MOV AX, 2000H

MOV DS, AX

ADD BX, [0500H]

ADD CX, BX

MOV DL, 05H

ADD DL, [0600]

MOV [0700], DL

HLT
```

Write a program to get Factorial of 10 numbers stored from the starting location 4000H:1000H. The results should be stored in 4000H:2000H

```
      MOV 4000H, AX
      MOV DS, AX
      MOV SI,1000H
      MOV DI, 2000H
      MOV CL,0AH
      MOV AL, 01H
Next: MOV BL,[SI]
LOOK: MUL BL
      DEC BL
      JNZ LOOK
      MOV [DI], AL
      INC SI
      INC DI
      LOOP NEXT
      HLT
```

Bus arbitration for I/O devices
Data transfer techniques

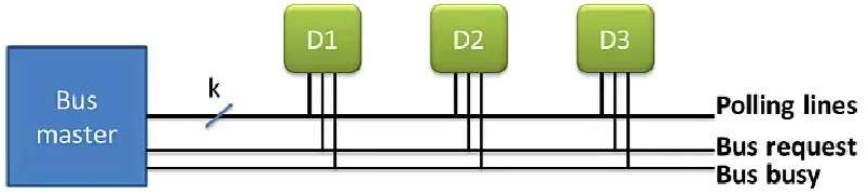
Daisy Chaining



Steps:
1. If bus not busy, make bus request
2. Master activates bus grant
3. If device gets bus grant, mark bus busy

(+) Simple
(+) Only three extra bus lines
(–) Hardwired priority
(–) Poor fault tolerance

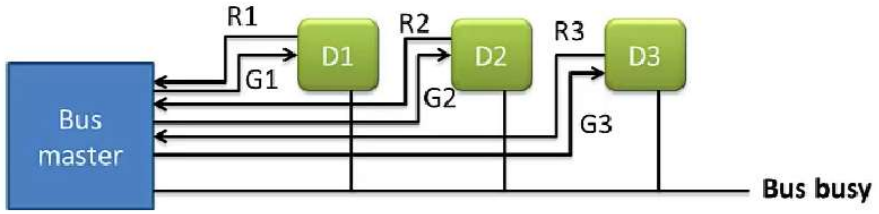
Polling



Steps:
1. If bus not busy, make bus request
2. Master polls by placing device ID on polling lines (master decides priority)
3. If device gets bus grant, mark bus busy

(+) No disadvantage of daisy chain
(+) Dynamic priority possible
(–) Extra poll lines
(–) Polling delay

Independent Request



Steps:
1. If bus not busy, make bus request
2. Master decides who to grant access, and indicates through grant line
3. If device gets bus grant, mark bus busy

(+) Fast
(+) Dynamic priority possible
(–) 2n lines for n devices!

The Purpose of Interrupts

I/O Organizations can be of 3 different types

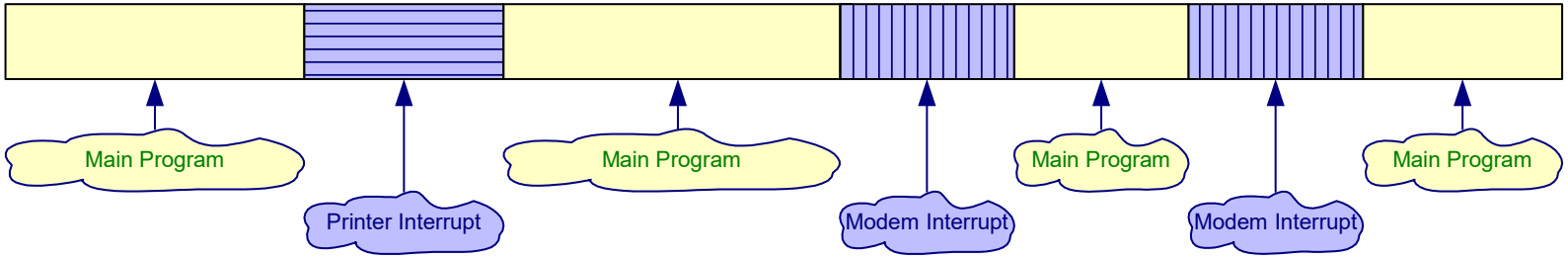
1. Programmed I/O (Polling/Daisy chain)
2. Interrupt driven I/O
3. DMA

Interrupts are useful when interfacing I/O devices with low data-transfer rates, like a keyboard or a mouse, in which case polling the device wastes valuable processing time

The peripheral interrupts the normal application execution, requesting to send or receive data.

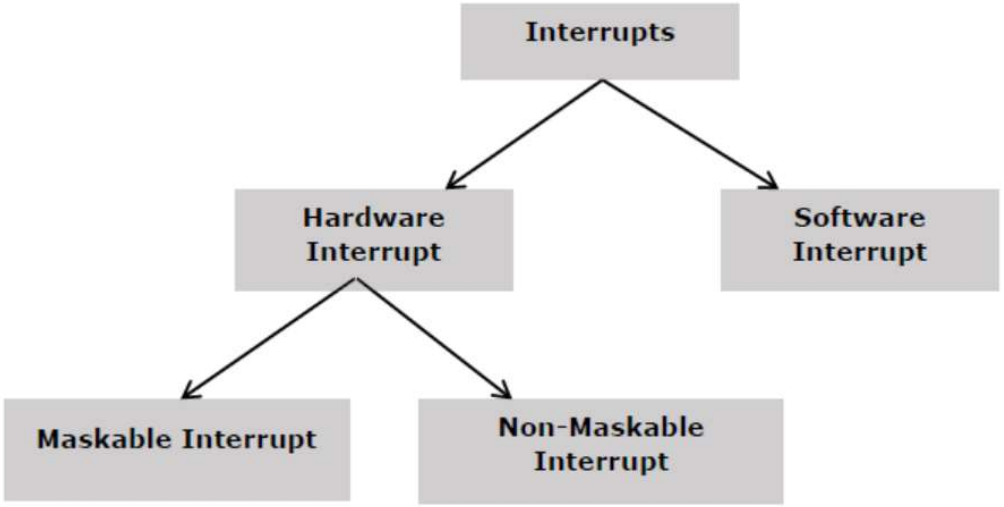
The processor jumps to a special program called *Interrupt Service Routine* to service the peripheral

After the processor services the peripheral, the execution of the interrupted program continues.



Interrupts

- An interrupt is used to cause a temporary halt in the execution of program.
- Microprocessor responds to the interrupt with an interrupt service routine → short program or subroutine that instructs the microprocessor on how to handle the interrupt.



BASIC INTERRUPT TERMINOLOGY

***Interrupt pins*:** Set of pins used in hardware interrupts

***Interrupt Service Routine (ISR) or Interrupt handler*:** code used for handling a specific interrupt

***Interrupt priority*:** In systems with more than one interrupt inputs, some interrupts have a higher priority than other

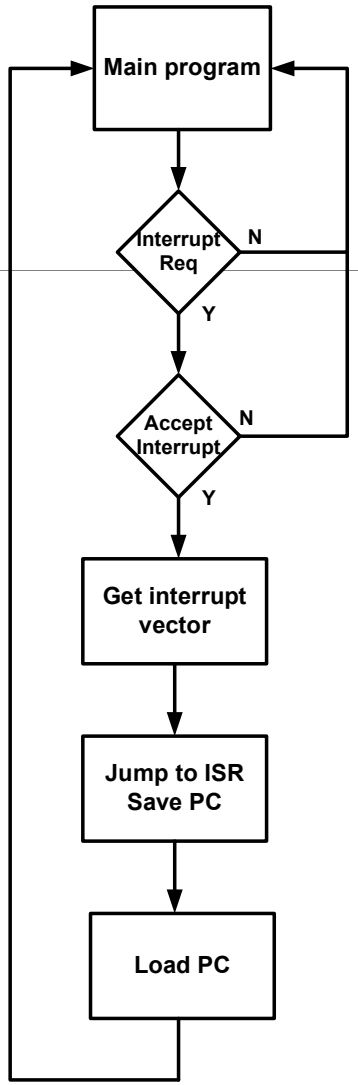
- They are serviced first if multiple interrupts are triggered simultaneously

***Interrupt vector*:** Code loaded on the bus by the interrupting device that contains the Address (segment and offset) of specific interrupt service routine

***Interrupt Masking*:** Ignoring (disabling) an interrupt

***Non-Maskable Interrupt*:** Interrupt that cannot be ignored (power-down)

Interrupt processing flow

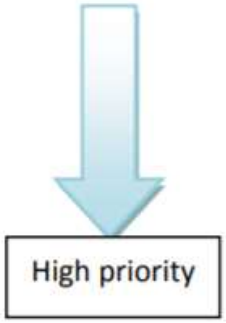


Interrupts

- A non-maskable interrupt requires an immediate response by microprocessor → usually used for serious circumstances like power failure
- A maskable interrupt is an interrupt that the microprocessor can ignore depending upon some predetermined condition defined by status register

Interrupt can divide to five groupe:

1. hardware interrupt
2. Non-maskable interrupt
3. Software interrupt
4. Internal interrupt
5. Reset



Interrupts

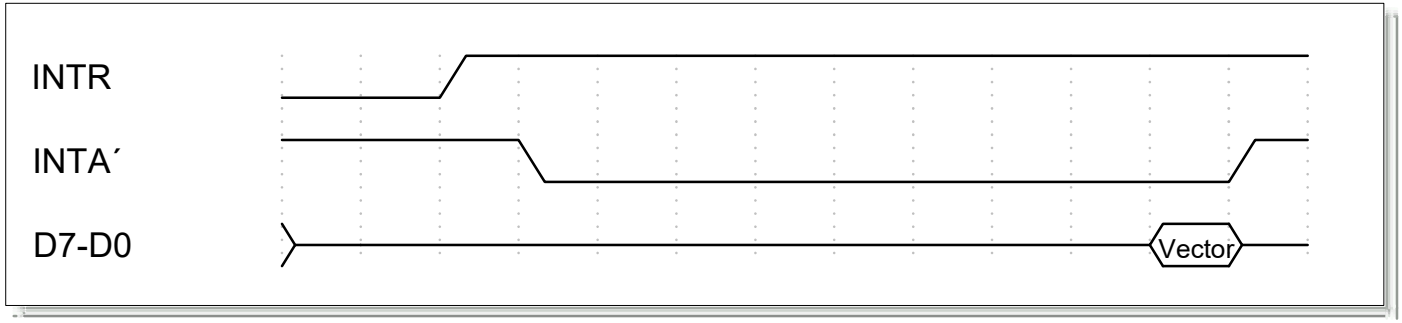
- Hardware, software and internal interrupt are serviced on priority basis.
- Each interrupt is given a different priority level by assigning it to a type number.
- Type 0 identifies the highest-priority and type 255 identifies the lowest-priority interrupt.
- The 8086 chips allow up to 256 vectored interrupts → it can have up to 256 different sources for an interrupt and the 8086 will directly call the service routine for that interrupt without any software processing.
- This is in contrast to non-vectored interrupts that transfer control directly to a single interrupt service routine, regardless of the interrupt source.

Hardware Interrupts – Interrupt pins and timing

x86 Interrupt Pins

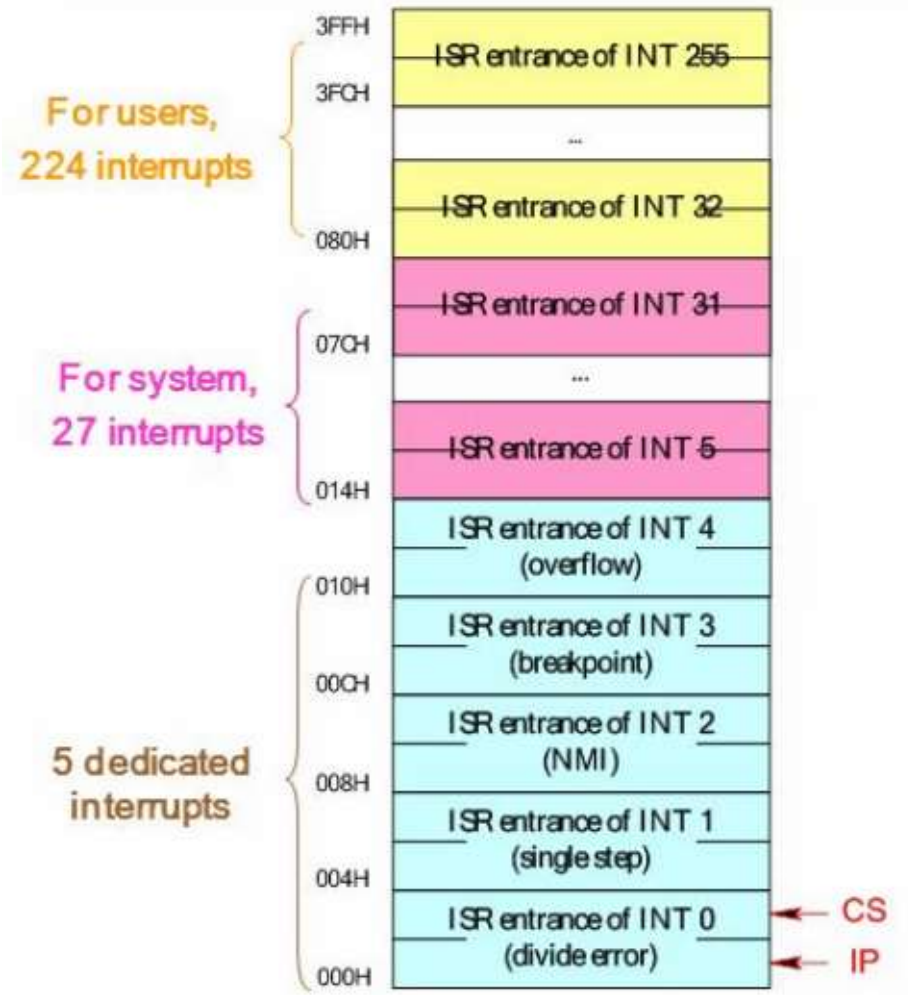
- INTR: Interrupt Request. Activated by a peripheral device to interrupt the processor.
  - Level triggered. Activated with a logic 1.

- /INTA: Interrupt Acknowledge. Activated by the processor to inform the interrupting device the the interrupt request (INTR) is accepted.
  - Level triggered. Activated with a logic 0.
- NMI: Non-Maskable Interrupt. Used for major system faults such as power failures.
  - Edge triggered. Activated with a positive edge (0 to 1) transition.
  - Must remain at logic 1, until it is accepted by the processor.
  - Before the 0 to 1 transition, NMI must be at logic 0 for at least 2 clock cycles.
  - No need for interrupt acknowledgement.



Interrupts

- The 8086 provides a 256 entry interrupt vector table beginning at address 0:0 in memory.
- The Interrupt Vector Table occupies the address range from 00000H to 003FFH (the first 1024 bytes in the memory map).
- This is a 1K table containing 256 4-byte entries.
- Each entry in this table contains a segmented address that points at the interrupt service routine in memory.
- The lowest five types are dedicated to specific interrupts such as the divide by zero interrupt and the non maskable interrupt.
- The next 27 interrupt types, from 5 to 31 are reserved by Intel for use in future microprocessors.
- The upper 224 interrupt types, from 32 to 255, are available to use for hardware and software interrupts.



- 256 interrupts
  - 0 ~ 4 dedicated
  - 5 ~ 31reserved for system use
    - 08H ~ 0FH : 8259A
    - 10H ~ 1FH : BIOS
  - 32 ~ 255 reserved for users
    - 20H ~ 3FH : DOS
    - 40H ~ FFH : open

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.

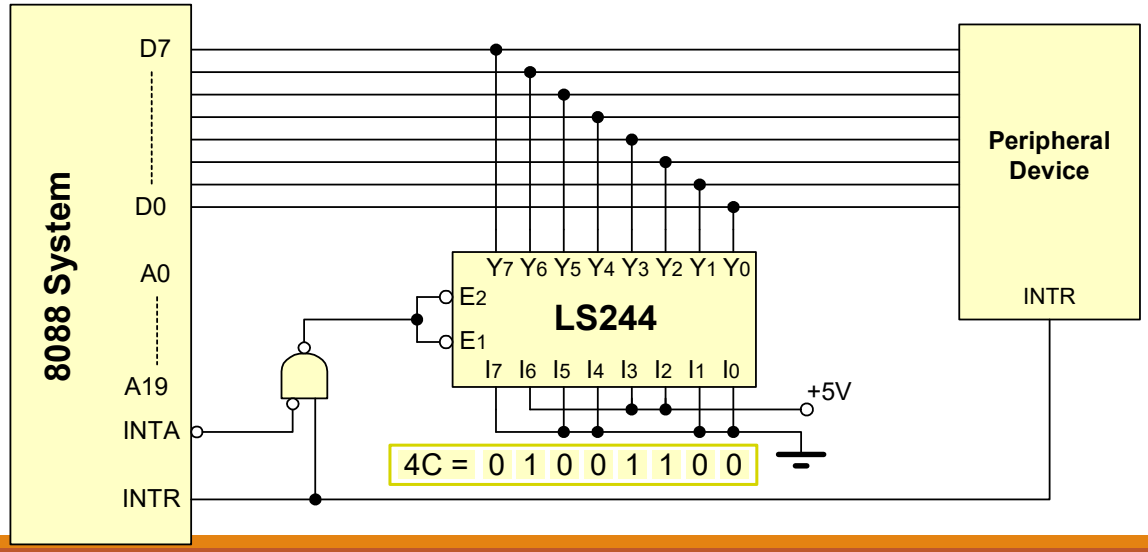
Vector No.	Mnemonic	Description	Source
9		CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		(Intel reserved. Do not use.)	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19-31		(Intel reserved. Do not use.)	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

Interrupt Vector - Example

Draw a circuit diagram to show how a device with interrupt vector 4CH can be connected on an 8088 microprocessor system.

Answer:

- **The peripheral device activates the INTR line**
- **The processor responds by activating the INTA signal**
- **The NAND gate enables the 74LS244 octal buffer**
  - **the number 4CH appears on the data bus**
- **The processor reads the data bus to get the interrupt vector**



Interrupt Vector Table – Real Mode (16-bit) Example

Using the Interrupt Vector Table shown below, determine the address of the ISR of a device with interrupt vector 42H.

Answer: Address in table = 4 X 42H = 108H

(Multiply by 4 since each entry is 4 bytes)

Offset Low = [108] = 2A, Offset High = [109] = 33

Segment Low = [10A] = 3C, Segment High = [10B] = 4A

Address = 4A3C:332A = 4A3C0 + 332A = 4D6EAH

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000	3C	22	10	38	6F	13	2C	2A	33	22	21	67	EE	F1	32	25
00010	11	3C	32	88	90	16	44	32	14	30	42	58	30	36	34	66
.....	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
00100	4A	33	3C	4A	AA	1A	1B	A2	2A	33	3C	4A	AA	1A	3E	77
00110	C1	58	4E	C1	4F	11	66	F4	C5	58	4E	20	4F	11	F0	F4
.....	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
00250	00	10	10	20	3F	26	33	3C	20	26	20	C1	3F	10	28	32
00260	20	4E	00	10	50	88	22	38	10	5A	38	10	4C	55	14	54
.....	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
003E0	3A	10	45	2F	4E	33	6F	90	3A	44	37	43	3A	54	54	7F
003F0	22	3C	80	01	3C	4F	4E	88	22	3C	50	21	49	3F	F4	65

Example

Write a sequence of instructions that initialize vector 40H to point to the ISR “isr40”.

Answer: Address in table = 4 X 40 = 100H

Set ds to 0 since the Interrupt Vector Table begins at 00000H

Get the offset address of the ISR using the Offset directive

and store it in the addresses 100H and 101H

Get the segment address of the ISR using the Segment directive

and store it in the addresses 102H and 103H

```
push    ax
push    ds
mov     ax,0
mov     ds,ax
mov     ax,offset isr40
mov     [0100h],ax
mov     ax,segment isr40
mov     [0102h],ax
pop     ds
pop     ax
```

Save registers in the stack

Set ds to 0 to point to the interrupt vector table

Get the offset address of the ISR and store it in the address 0100h (4X40 = 100h)

Get the segment address of the ISR and store it in the address 0102h

Restore registers from the stack

Interrupt Masking

The processor can inhibit certain types of interrupts by use of a special interrupt mask bit.

This mask bit is part of the flags/condition code register, or a special interrupt register.

If this bit is clear, and an interrupt request occurs on the Interrupt Request input, it is ignored.

NMI cannot be masked

S/W Interrupt Processing

Save state

- **Disable interrupts for the duration of the ISR or allow it to be interrupted too?**
- **Save program counter/Instruction Pointer**
- **Save flags**
- **Save register values?**

Jump to interrupt service routine

- **Location obtained by interrupt vector**

Process interrupt

Restore state

- **Load PC/IP, flags, registers etc.**

H/W Interrupt Processing

External interface sends an interrupt signal, to the Interrupt Request (INTR) pin, (or an internal interrupt occurs.)

The CPU finishes the present instruction (for a hardware interrupt) and checks the INTR pin.

If IF=0 the processor ignores the interrupt, else sends Interrupt Acknowledge (INTA) to hardware interface.

The interrupt type N is sent to the Central Processor Unit (CPU) via the Data bus from the hardware interface.

The contents of the flag registers are pushed onto the stack.

Both the interrupt (IF – FR bit 9) and (TF – FR bit 8) flags are cleared. This disables the INTR pin and the trap or single-step feature.

The contents of the code segment register (CS) are pushed onto the Stack.

The contents of the instruction pointer (IP) are pushed onto the Stack.

The interrupt vector contents are fetched, from (4 x N) and then placed into the IP and from (4 x N +2) into the CS so that the next instruction executes at the interrupt service procedure addressed by the interrupt vector.

While returning from the interrupt-service routine by the Interrupt Return (IRET) instruction, the IP, CS and Flag registers are popped from the Stack and return to their state prior to the interrupt.

The Intel x86 Interrupt Software Instructions

All x86 processors provide the following instructions related to interrupts:

- **INT nn: Interrupt. Run the ISR pointed by vector nn.**
  - **INT 0 is reserved for the Divide Error**
  - **INT 1 is reserved for Single Step operation**
  - **INT 2 is reserved for the NMI pin**
  - **INT 3 is reserved for setting a Breakpoint**
  - **INT 4 is reserved for Overflow (Same as the INTO (Interrupt on overflow) instruction.**
- **CLI: Clear Interrupt Flag. IF is set to 0, thus interrupts are disabled.**
- **STI: Set Interrupt Flag. IF is set to 1, thus interrupts are enabled.**
- **IRET: Return from interrupt. This is the last instruction in the ISR (Real Mode only). It pops from the stack the Flag register, the IP and the CS.**
  - **After returning from an ISR the interrupts are enabled, since the initial value of the flag register is poped from the stack.**

Reset

- Processor initialization or start up is accomplished with activation (HIGH) of the RESET pin.
- The 8086 RESET is required to be HIGH for greater than 4 CLK cycles.
- The 8086 will terminate operations on the high-going edge of RESET and will remain dormant as long as RESET is HIGH.
- The low-going transition of RESET triggers an internal reset sequence for approximately 10 CLK cycles.
- After this interval the 8086 operates normally beginning with the instruction in absolute location FFFF0H.

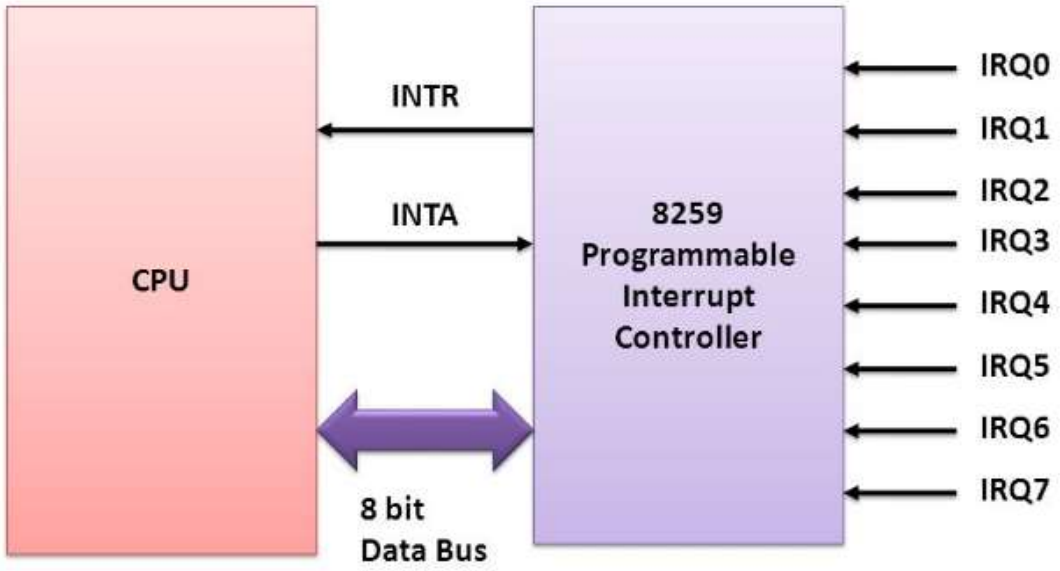
Non-Maskable interrupt

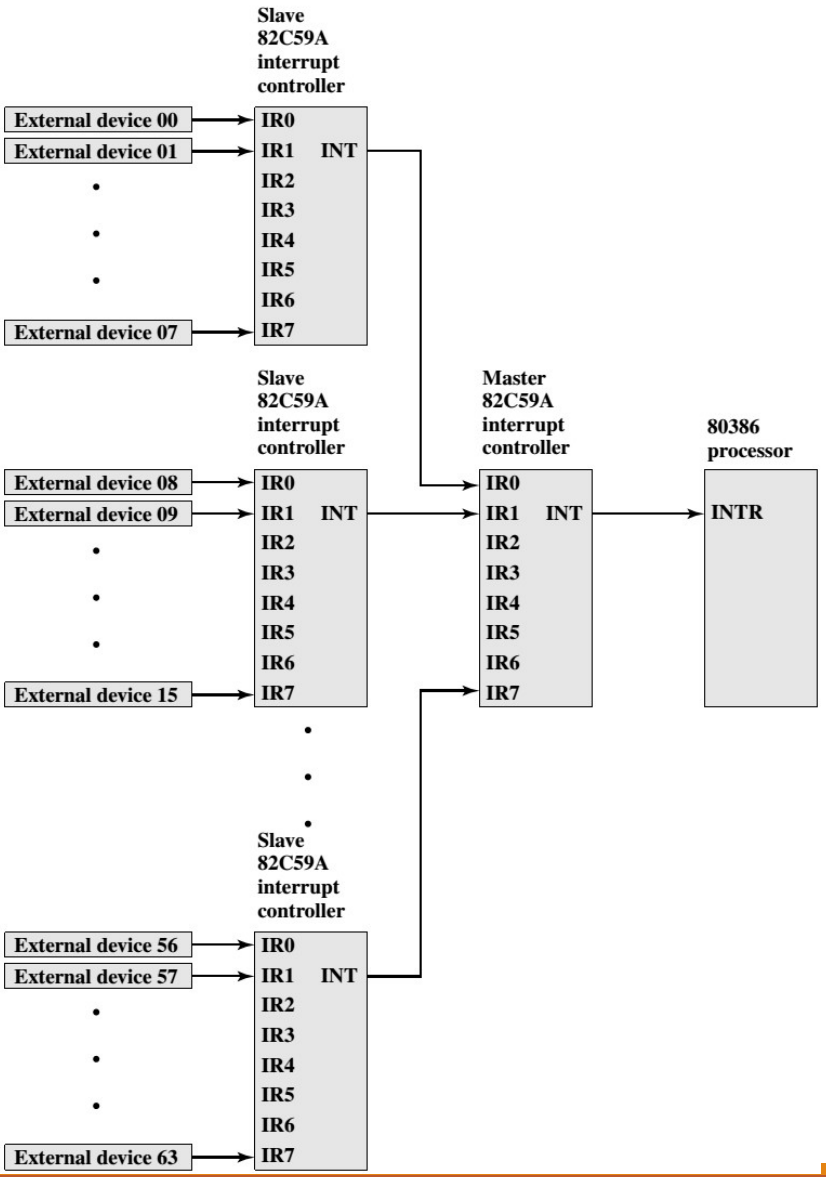
- The processor provides a single non-maskable interrupt pin (NMI) which has higher priority than the maskable interrupt request pin (INTR).
- A typical use would be to activate a power failure routine.
- The NMI is edge-triggered on a LOW-to-HIGH transition.
- The activation of this pin causes a type 2 interrupt.
- NMI is required to have a duration (in the HIGH state) of greater than two CLK cycles, but is not required to be synchronized to the clock.

Intel 8259A Interrupt Controller

- The primary sources of interrupts are the PCs timer chip, keyboard, serial ports, parallel ports, disk drives, CMOS real-time clock, mouse, sound cards, and other peripheral devices.
- These devices connect to an Intel 8259A programmable interrupt controller (PIC) that prioritizes the interrupts and interfaces with the 8086 CPU.
- The 8259A chip adds considerable complexity to the software that processes interrupts.

8259: Programmable Interrupt Controller




Slave
82C59A
interrupt
controller
External device 00
External device 01
External device 07
IR0
IR1
INT
IR2
IR3
IR4
IR5
IR6
IR7
Slave
82C59A
interrupt
controller
Master
82C59A
interrupt
controller
80386
processor
External device 08
External device 09
External device 15
INTR
Slave
82C59A
interrupt
controller
External device 56
External device 57
External device 63

Programmable 82C59A

The 82C59A is programmable. The 8386 determines the priority scheme to be used by setting a control word in the 82C59A.

The following interrupt modes are possible:

- **Fully nested:** The interrupt requests are ordered in priority from 0 (IR0) through 7 (IR7)
- **Rotating:** In some applications a number of interrupting devices are of equal priority. In this mode a device, after being serviced, receives the lowest priority in the group.
- **Special mask:** This allows the processor to inhibit interrupts from certain devices

Necessary materials to read for Interrupts

Stallings book chapter: Input / Output (Chapter 7)

Other materials will be provided in VTOP

Maskable interrupt

- Whenever an external signal activates the INTR pin, the microprocessor will be interrupted only if interrupts are enabled using set interrupt Flag instruction (STI).
- If the interrupts are disabled using clear interrupt Flag instruction (CLI), the microprocessor will not get interrupted even if INTR is activated.
- That is, INTR can be masked.
- INTR is a non vectored interrupt, which means, the 8086 does not know where to branch to service the interrupt.
- The 8086 has to be told by an external device like a Programmable Interrupt controller regarding the branch.
- Whenever the INTR pin is activated by an I/O port, if Interrupts are enabled and NMI is not active at that time, the microprocessor finishes the current instruction that is being executed and gives out a '0' on INTA pin twice.

Maskable interrupt

- When INTA pin goes low for the first time, it asks the external device to get ready.
- In response to the second INTA the microprocessor receives the 8 bit, say N, from a programmable Interrupt controller.
- The action taken is as follows:

1. Complete the current instruction.
2. Activates INTA output, and receives type Number, say N
3. Flag register value, CS value of the return address & IP value of the return address are pushed on to the stack.
4. IP value is loaded from contents of word location N x 4.
5. CS is loaded from contents of the next word location.
6. Interrupt Flag and trap Flag are reset to 0.

Maskable interrupt

- At the end of the ISR, there will be an IRET instruction.
- This performs popping off from the stack top to IP, CS and Flag registers.
- Finally, the register values which are also saved on the stack at the start of ISR, are restored from the stack and a return to the interrupted program takes place using the IRET instruction.

INT 21h

- **INT 21h** / **AH=1** - read character from standard input, with echo, result is stored in **AL**. if there is no character in the keyboard buffer, the function waits until any key is pressed.

```
mov ah, 1
int 21h
```

- **INT 21h** / **AH=2** - write character to standard output.
  entry: **DL** = character to write, after execution **AL = DL**.

```
mov ah, 2
mov dl, 'a'
int 21h
```

INT 21h

- **INT 21h** / **AH=5** - output character to printer.
  entry: **DL** = character to print, after execution **AL = DL**.

```
mov ah, 5
mov dl, 'a'
int 21h
```

- **INT 21h** / **AH=6** - direct console input or output.
  parameters for output: **DL** = 0..254 (ascii code)
  parameters for input: **DL** = 255
  for output returns: AL = DL
  for input returns: **ZF** set if no character available and **AL = 00h**, **ZF** clear if character available.
  AL = character read; buffer is cleared.

```
mov ah, 6
mov dl, 'a'
int 21h

mov ah, 6
mov dl, 255
int 21h
```

INT 21h

- **INT 21h** / **AH=1** - read character from standard input, with echo, result is stored in **AL**. if there is no character in the keyboard buffer, the function waits until any key is pressed.

```
mov ah, 1
int 21h
```

- **INT 21h** / **AH=2** - write character to standard output.
  entry: **DL** = character to write, after execution **AL = DL**.

```
mov ah, 2
mov dl, 'a'
int 21h
```

INT 21h

- **INT 21h** / **AH=5** - output character to printer.
  entry: **DL** = character to print, after execution **AL = DL**.

```
mov ah, 5
mov dl, 'a'
int 21h
```

- **INT 21h** / **AH=6** - direct console input or output.
  parameters for output: **DL** = 0..254 (ascii code)
  parameters for input: **DL** = 255
  for output returns: AL = DL
  for input returns: **ZF** set if no character available and **AL = 00h**, **ZF** clear if character available.
  AL = character read; buffer is cleared.

```
mov ah, 6
mov dl, 'a'
int 21h

mov ah, 6
mov dl, 255
int 21h
```

INT 21h

- **INT 21h** / **AH=7** - character input without echo to AL.
  if there is no character in the keyboard buffer, the function waits until any key is pressed.

```
mov ah, 7
int 21h
```

- **INT 21h** / **AH=9** - output of a string at **DS:DX**. String must be terminated by '**$**'.

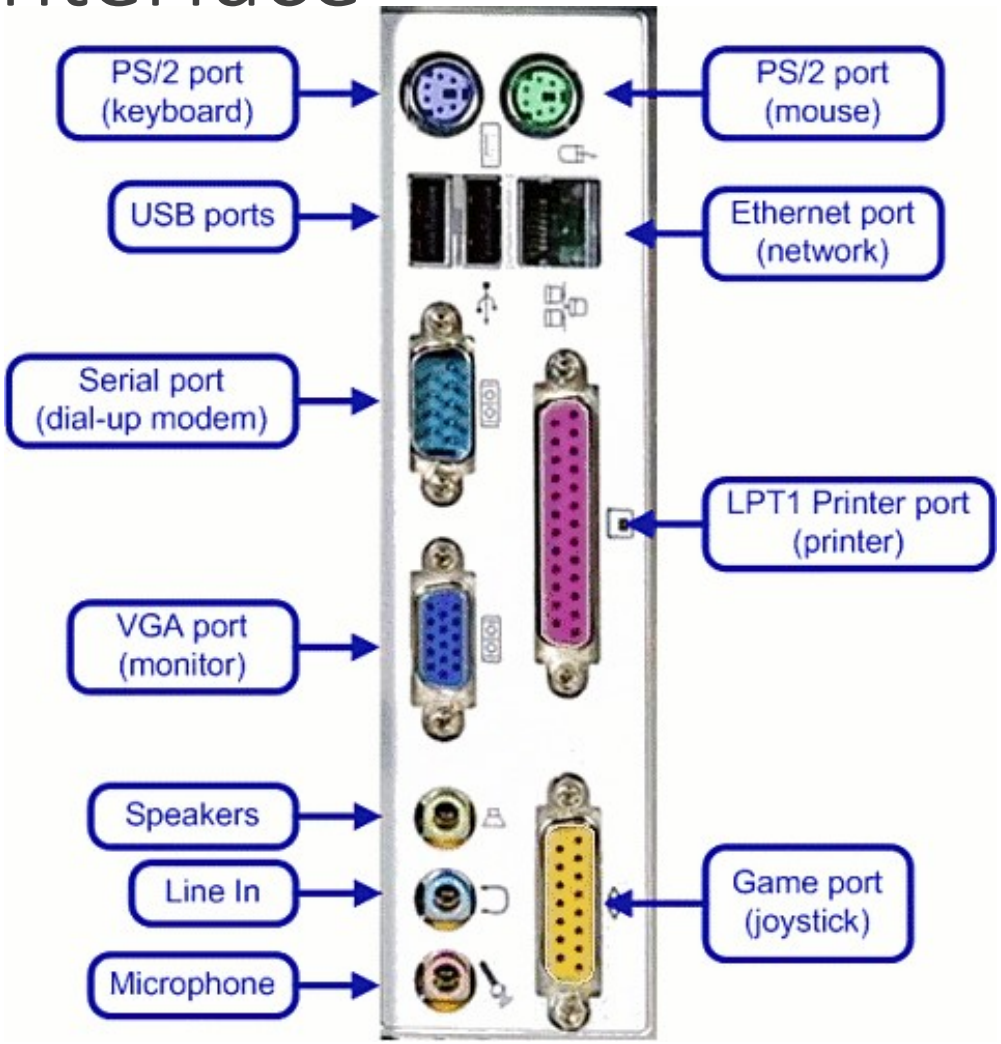
```
org 100h
mov dx, offset msg
mov ah, 9
int 21h
ret
msg db "hello world $"
```

INT 21h

- **INT 21h** / **AH=0Ah** - input of a string to **DS:DX**, first byte is buffer size, second byte is number of chars actually read. this function does **not** add '$' in the end of string. to print using **INT 21h** / **AH=9** you must set dollar character at the end of it and start printing from address **DS:DX+2**.

```
org 100h
mov dx, offset buffer
mov ah, 0ah
int 21h
jmp print
buffer db 10,?, 10 dup(' ')
print:
xor bx, bx
mov bl, buffer[1]
mov buffer[bx+2], '$'
mov dx, offset buffer + 2
mov ah, 9
int 21h
ret
```

Serial I/O Interface



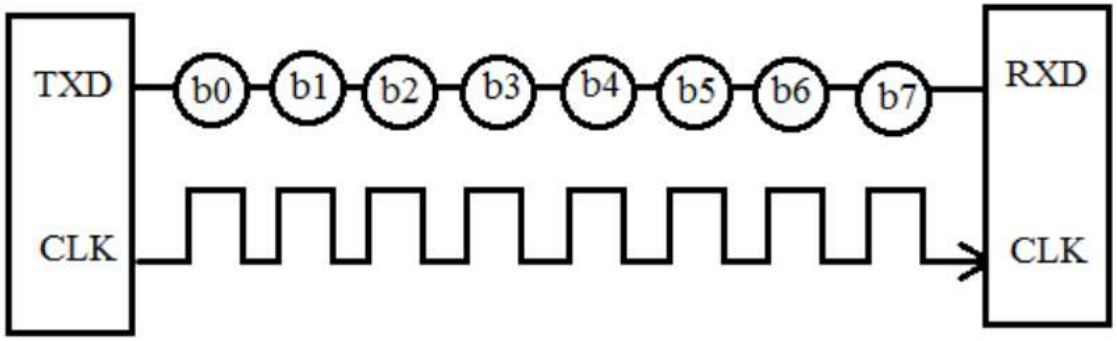
VGA cable

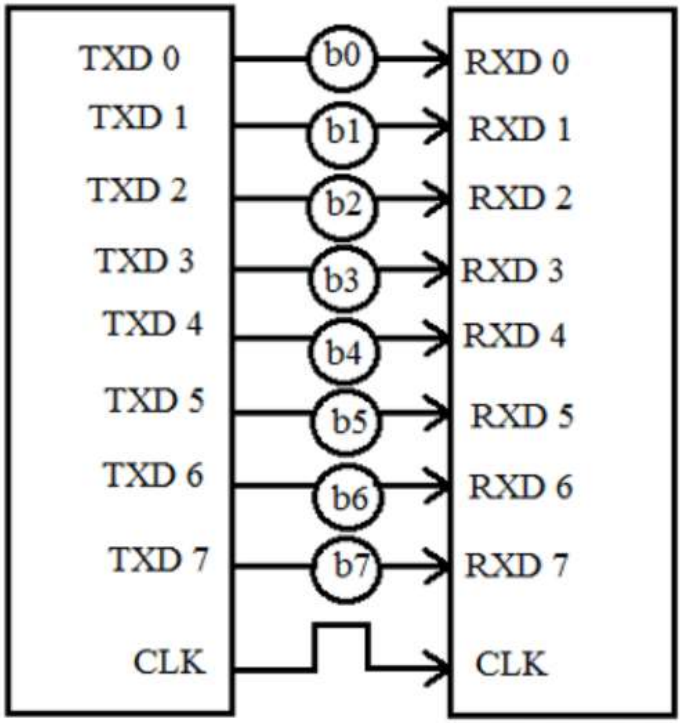
VGA connector

Serial I/O Interface

- Serial data transmission is used for digital communication between
  - Sensors and computers
  - Computers and computers
  - Computers and peripheral devices (printer, stylus, mouse, ..)
- It is one of the most widely used communication techniques to interface external equipment.
- The process of sending data sequentially over a computer bus is called as **serial communication**, which means the data will be transmitted bit by bit.
- While in parallel communication the data is transmitted in a byte (8-bit) or character on several data lines or buses at a time.
- Serial communication is slower than parallel communication but used for long data transmission due to lower cost and practical reasons.

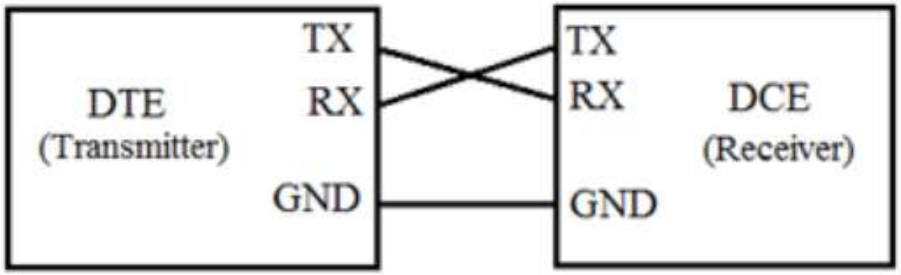
Serial I/O Interface





RS232

- RS232 is a standard protocol used for serial communication
- It is used in serial communication up to 50 feet with the rate of 1.492kbps.
- RS232 is used for connecting **Data Terminal Equipment (DTE)** and **Data Communication Equipment (DCE)**.



Handshaking

- Handshaking is a process of dynamically setting the parameters of a communication between the transmitter and receiver before the communication begins.
- The need for handshaking is dictated by the speed at with the transmitter (DTE) transmits the data, the speed at which the receiver (DCE) receives the data.

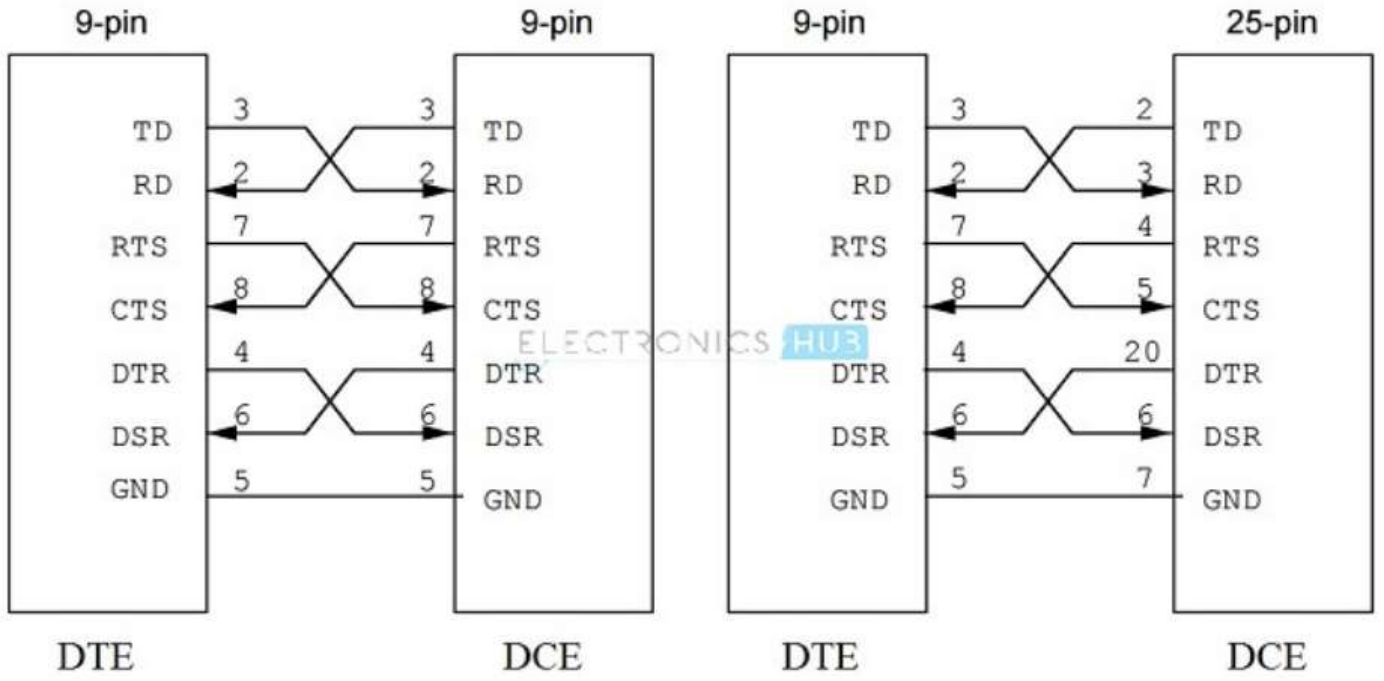
Hardware Handshaking

- In Hardware Handshaking, the transmitter first asks the receiver whether it is ready to receive data.
- The receiver then checks its buffer and if the buffer is empty, it will then tell the transmitter that it is ready to receive.
- The transmitter will transmit the data and it is loaded into the receiver buffer.
- During this time, the receiver tells the transmitter not to send any further data until the data in the buffer has been read by the receiver.

Hardware Handshaking

➢The RS232 Protocol defines four signals for the purpose of Handshaking:

1. Ready to Send (RTS)
2. Clear to Send (CTS)
3. Data Terminal Ready (DTR)
4. Data Set Ready (DSR)

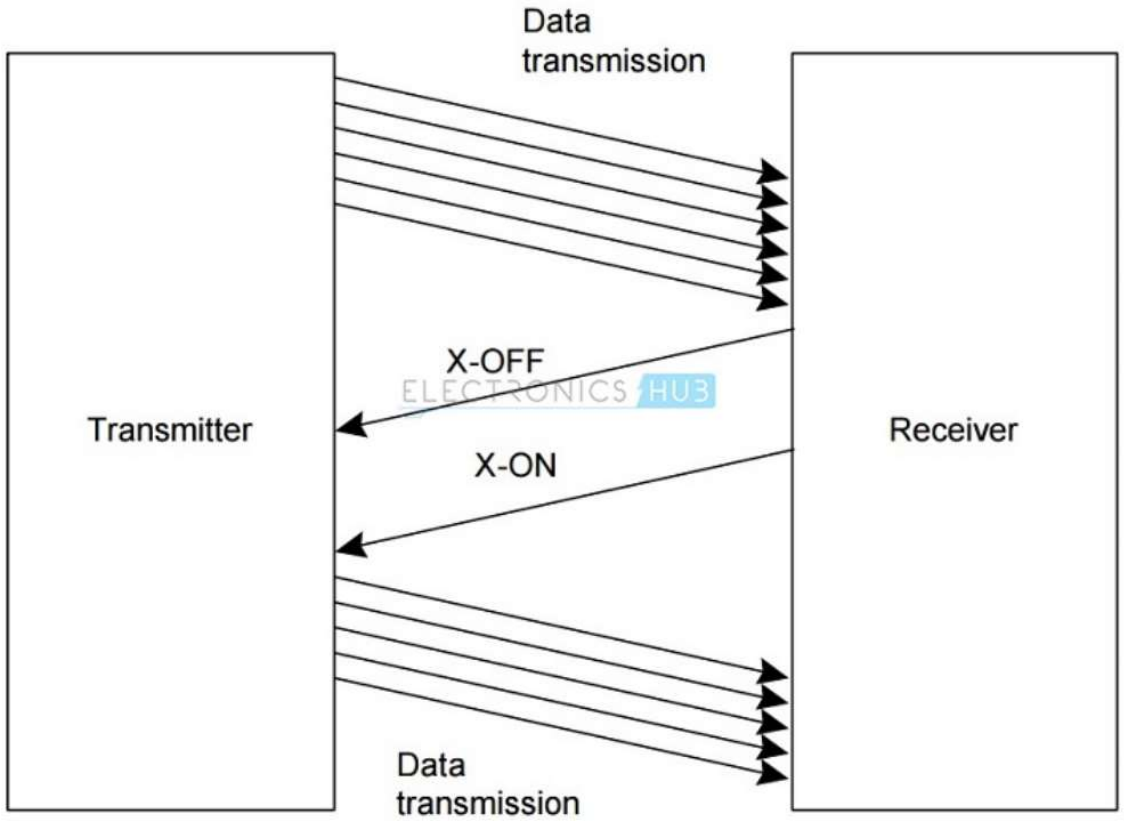


Hardware Handshaking

- With the help of Hardware Handshaking, the data from the transmitter is never lost or overwritten in the receiver buffer.
- When the transmitter (DTE) wants to send data, it pulls the RTS (Ready to Send) line to high.
- Then the transmitter waits for CTS (Clear to Send) to go high and hence it keeps on monitoring it.
- If the CTS line is low, it means that the receiver (DCE) is busy and not yet ready to receive data.
- When the receiver is ready, it pulls the CTS line to high.
- The transmitter then transmits the data. This method is also called as RTS/CTS Handshaking.

Software Handshaking

- Software Handshaking in RS232 involves two special characters for starting and stopping the communication.
- These characters are X-ON and X-OFF (Transmitter On and Transmitter OFF).
- When the receiver sends an X-OFF signal, the transmitter stops sending the data.
- The transmitter starts sending data only after it receives the X-ON signal.



RS232

- Electrical characteristics

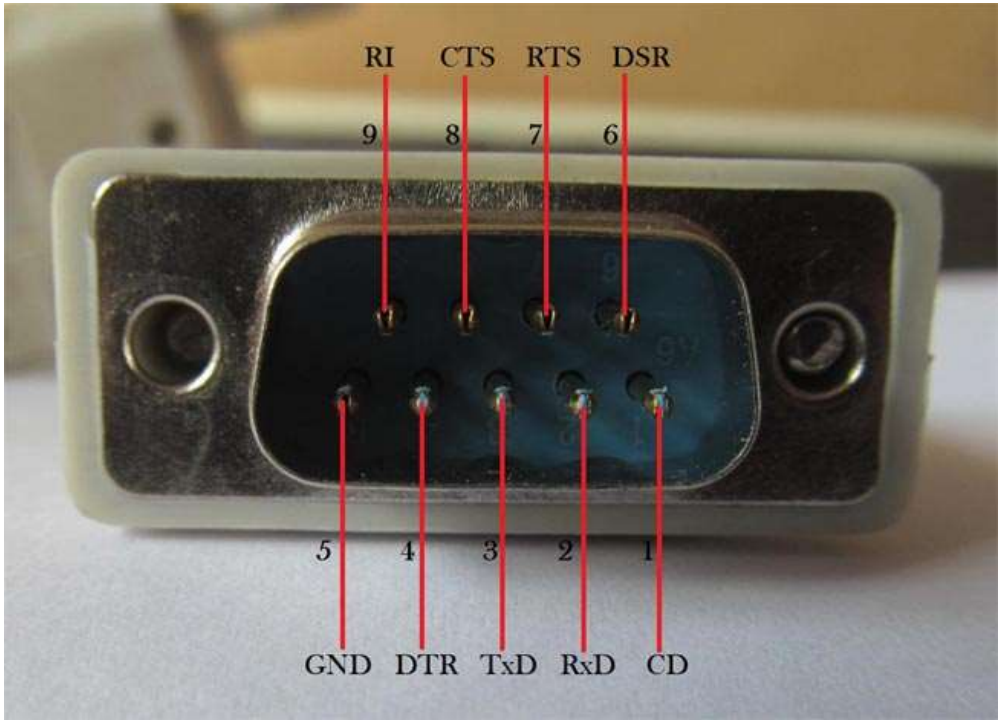
1. Logic 1: -3V to -25V; typically -12V
2. Logic 0: +3v to +25V; typically +12V
3. Any signal in the range -3V to +3V has an indeterminate logical state
4. Quiescent or inactive state is -12V (i.e. logic 1)

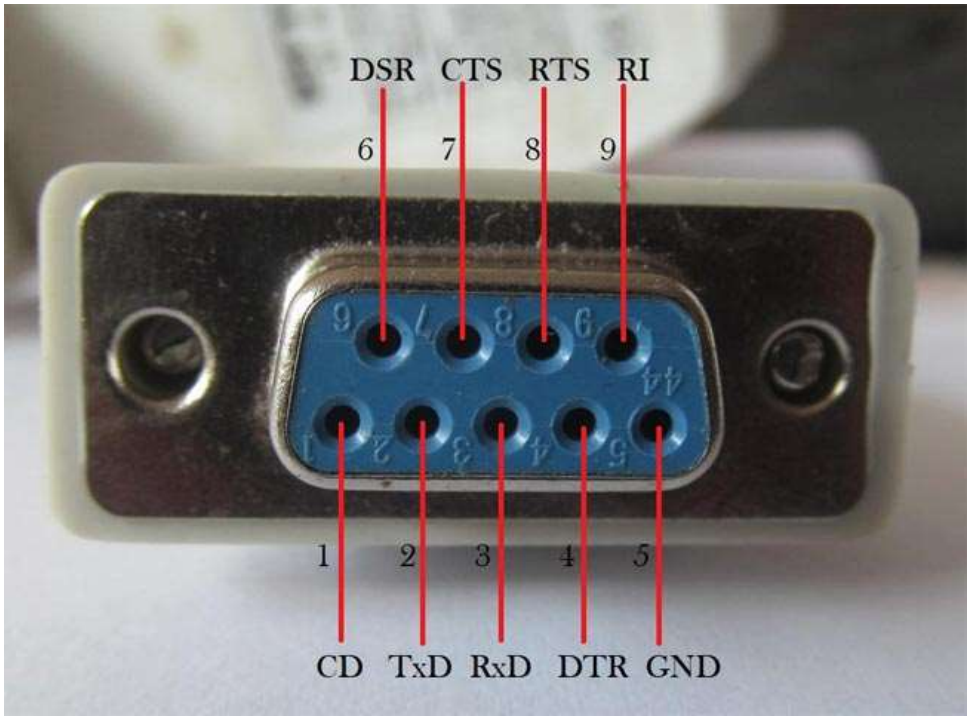
RS232 (Serial i/o Interface)

- Connectors

1. DB25S is a 25 pin connector with full RS-232 functionality
2. The computer socket has a female outer casing with male connecting pins
3. The terminating cable connector has a male outer casing with female connecting pins

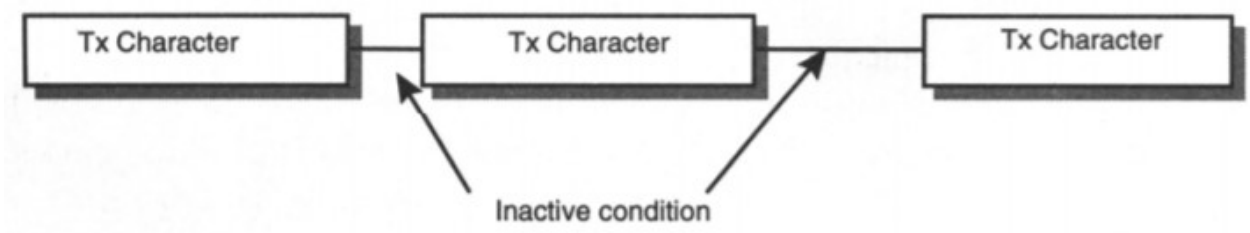
RS232 (DB9 connector)





RS232

- RS232 takes bytes and transmits the individual bits in a sequential fashion in a frame.
- A frame is a defined structure, carrying meaningful sequence of bit or bytes of data.
- It has a start bit followed by 8 data bits, a parity bit and a stop bit.
- **Frame character** is as shown below

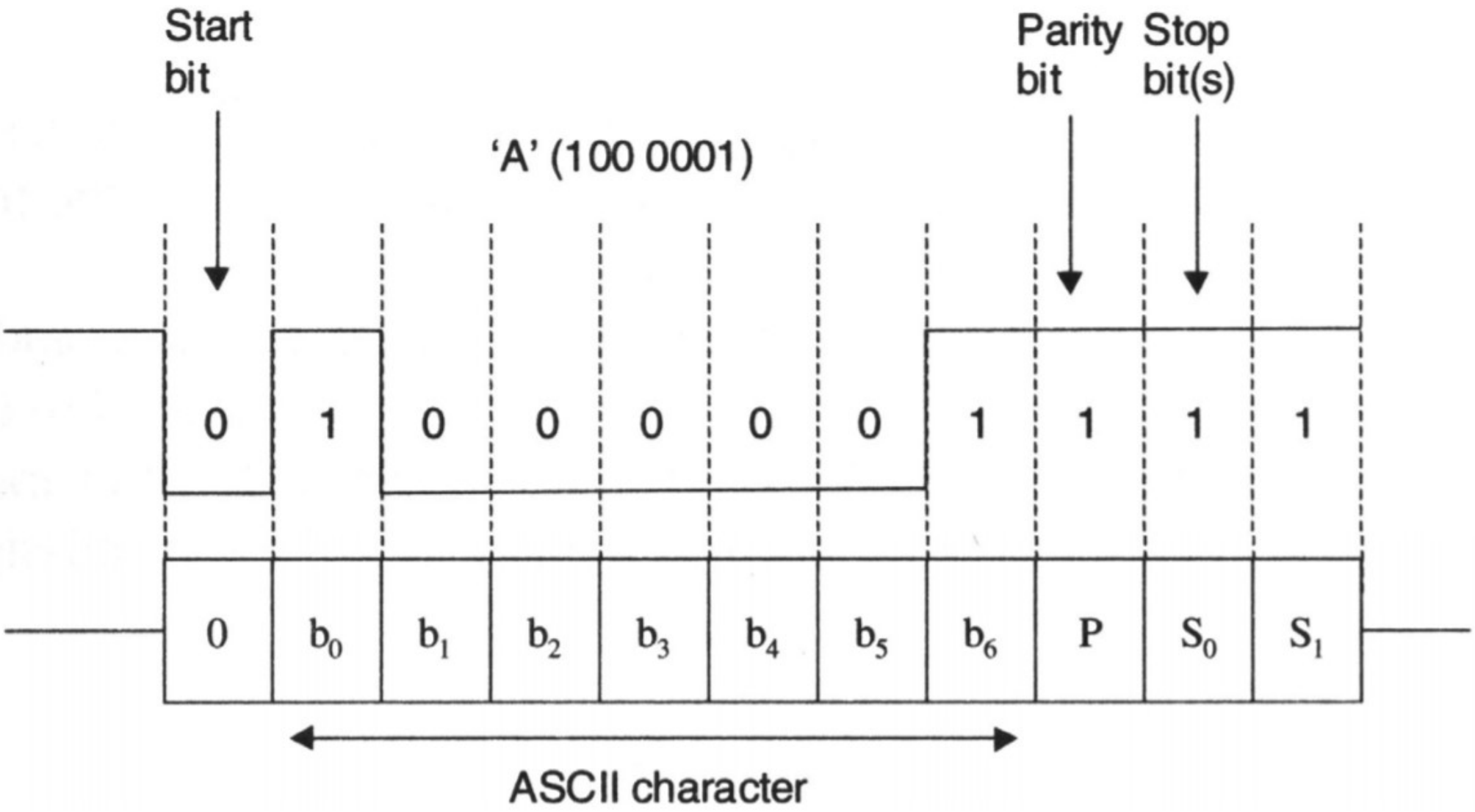


RS232

- Each bit is sent one after the other.
- This mode of transmission requires that receiver is aware when the actual data bits are arriving to synchronize itself with coming data.
- So logic 0 is sent as a start bit. The start bit in the frame signals the receiver that a new character is coming.
- Once the receiver acknowledges, the next five to eight bits are sent which represents the character.
- This is followed by parity bit used for error detection.
- Parity bit is used to specify even or odd number of one's in the set of bits.
- The stop bit helps the receiver to identify the end of message.

RS232

- If a receiver detects a value other than mark when stop bit should be present, it knows that's there is synchronization error.
- This causes a framing error condition during reception
- The device then tries to resynchronize on new incoming bits.


Start bit
'A' (100 0001)
Parity bit
Stop bit(s)
0 1 0 0 0 0 0 1 1 1 1
0 b0 b1 b2 b3 b4 b5 b6 P S0 S1
ASCII character

RS232

- Example
- ASCII coding, even parity, 2 stop bits:
- 1111101000001011000001111111111111100000111111110001100111101010011111111111
  1
- {inactive}11111 {start bit} 0 {‘A’}1000001 {parity bit} 0 {stop bits} 11 {start bit}0 {‘p’}0000111 {parity bit} 1 {stop bits}11 {inactive}11111111 {start bit}0 {‘p’}0000111 {parity bit} 1 {stop bits}11 {inactive}11 {start bit}0 {‘L’}0011001 {parity bit} 1 {stop bits}11
- Message is ‘AppL’

Universal serial bus

- Industry standard that defines the cables, connectors and communications protocols used in a bus for connection, communication, and power supply between computers and electronic devices
- Developed in1996
- By Joint efforts of Intel, Microsoft, Digital Equipment Corporation, IBM, NEC, Nortel, Compaq
- USB has superceded and effectively replaced Serial, Parallel & PS/2 ports 4

WHY USB?

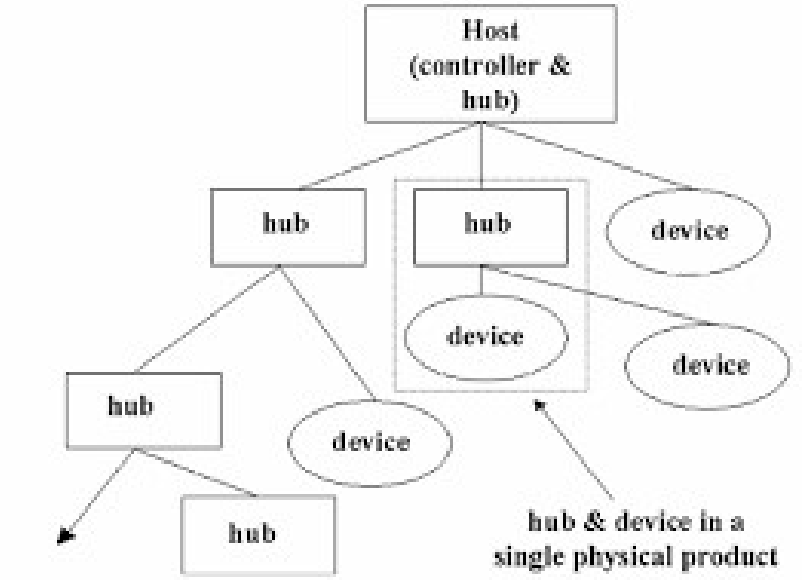
- To standardize connections of electrical devices, while maintaining and optimizing
  - High speed
  - Reliability
  - Cost of manufacturing
- To make it fundamentally easier to connect external devices to PCs
  - by replacing the multitude of connectors at the back of PCs
  - addressing the usability issues of existing interfaces
  - simplifying software configuration of all devices connected to USB

Comparison of Speeds of USB

USB Version	Speed	Type
USB version 1.1/1.0	1.5Mbits/s	Low Speed
USB version 1.1/1.0	12Mbits/s	Full Speed
USB version 2.0	480Mbits/s	High Speed
USB version 3.0	4.8 Gbit/s	High Speed
USB version 3.1	10 Gbit/s	Super Speed

Architectural Overview

- Host-controlled
  - i.e. there can only be one host per bus
- Tiered Star Topology
  - Many hubs or Devices can be connected to the root host



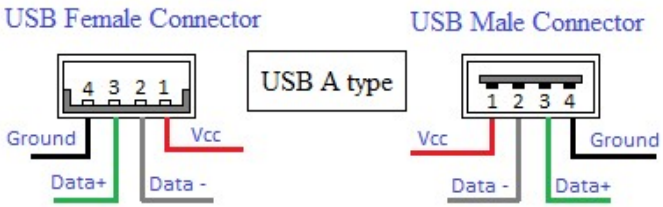
Tiered Star Topology

- Uses 7-bit addressing of devices
  - Allowing connection of up to 127 devices on a single USB bus
  - Allows expandability and ease of use
  - Each device can be handled and removed individually without interrupting others
- Allows plug'n'play connectivity which allows for dynamically loadable and unloadable devices and drivers.
  - Plug the device in and the host loads the drivers without needing a reboot or initiation/termination of connection, etc
  - Unplug the device the absence is automatically detected by the host and the drivers are unloaded

Electrical and Mechanical Characteristics

- Mechanical Characteristics
- There are commonly two kinds of USB connectors

1. Type A
2. Type B



- Many cables are made with Type A connector on one end and Type B connector on the other

- This is an attempt to prevent improper physical connections between devices, as the connectors are not physically interchangeable

Protocol layer

- USB Packet Types - Data is sent in packets Least Significant Bit (LSB) first
- There are 4 main USB packet types:

1. Token
2. Data
3. Handshake
4. Start of Frame

- The packets are then bundled into frames to create a USB message

USB Packet Fields

- Each packet is constructed from different field types

1. SYNC (synchronize)
2. PID (packet ID)
3. Address
4. Data
5. Endpoint
6. CRC (cyclic redundancy check)
7. EOP (end of packet)

USB Packet Fields

- Synchronize (Sync):
  - All packets must start with a sync field.
  - Used to synchronize clock rate between device and host.
  - It is 8 bits for low speed and 32 bits for full/high speed connection.
- Packet ID (PID):
  - Packet ID identifies the type of packet being sent.
  - It has 4 bits for the value, and another 4 bits of the inverted value to prevent errors

USB Packet Fields

- Address:
  - 7 bit field
  - Specifies the device the packet is intended for out of 127 devices that can be connected to a single bus.
- Endpoint:
  - 4-bit long
  - Allows for further flexibility in addressing
  - Can also be split for IN or OUT data

USB Packet Fields

- Cyclic Redundancy Checks (CRC):
  - Performed on the data with the packet payload.
  - All token packets have 5 bit CRC while data packets have 16 bit CRC.
- End of Packet (EOP):
  - Signaled by a ‘0’ for approximately 2 bit times followed by a J for 1 bit time.

Token Packet Format

- In
  - Informs the USB device that host wishes to read information
- Out
  - Informs the USB device that the host wishes to send information
- Setup
  - Used to begin control transfers

Data Packet Format

- Maximum data payload size for low-speed devices is 8 bytes.
- Maximum data payload size for full-speed devices is 1024 bytes
- Payloads must be sent in multiples of bytes

Handshake Packet Format

- There are three types of packets which consist simply of PID
- ACK: Acknowledgement that the packet has been successfully received
- NACK: Reports that the device can not send or receive data. Also used to interrupt when no data is available to send
- STALL: The device is in a state that requires intervention from the host

Start of Frame Packet Format

- Data is split into frames before being transmitted.
- The start of frame packet is used to signify the frame number to the host.
- The frame number field is 11 bits long

Data Transfer Types

- Data Transfer Types

1. Control
2. Isochronous
3. Bulk
4. Interrupt

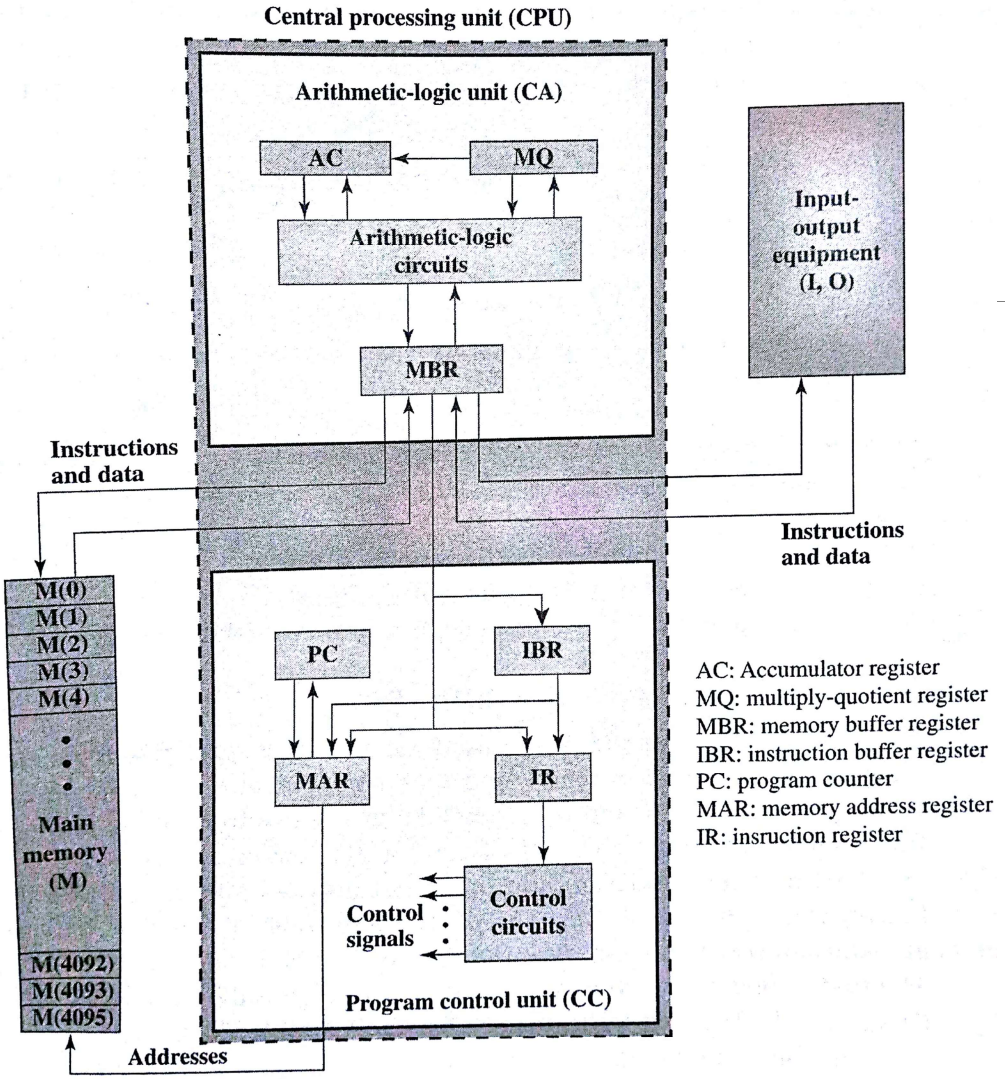
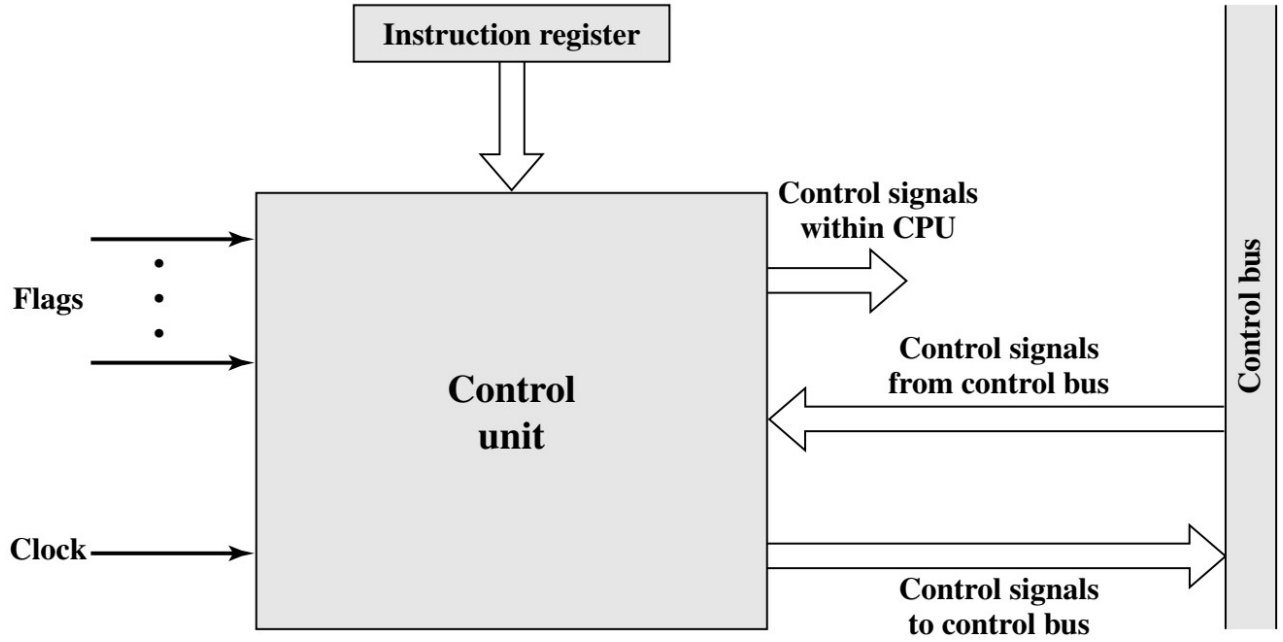


Figure 1.6 IAS Structure

Control unit



Control unit

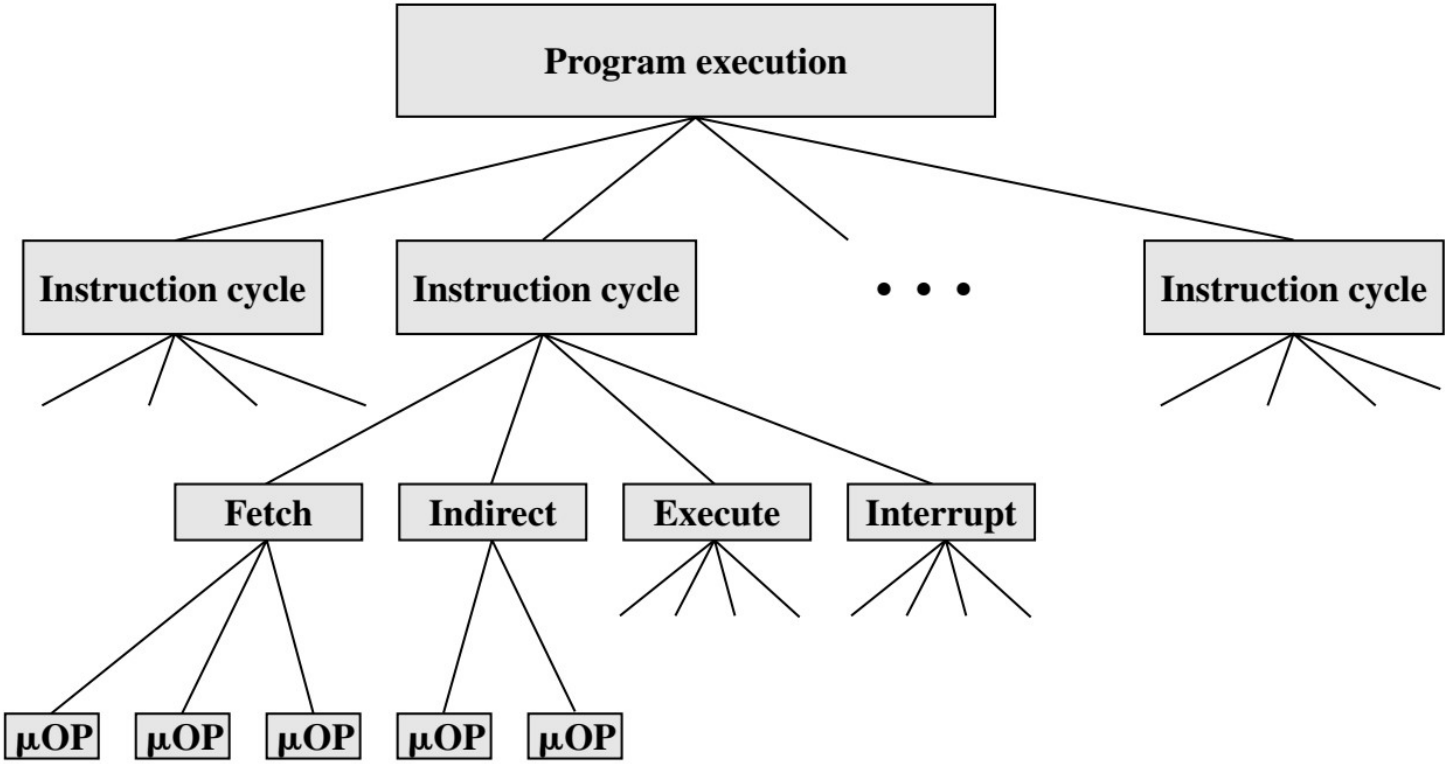
- Each instruction is executed in one instruction cycle.
- But each instruction cycle is made up of a number of smaller units
- These smaller units, in general, are

1. Fetch
2. Indirect
3. Execute
4. Interrupt

Control unit

- It can be seen that further decomposition is possible
- Each of these smaller cycles involves a series of steps, each of which involves the processor registers
- Such steps are known as **micro-operations**, since they are very simple and accomplish very little
- Micro-operations are the functional, or atomic, operations of a processor

Micro-operations



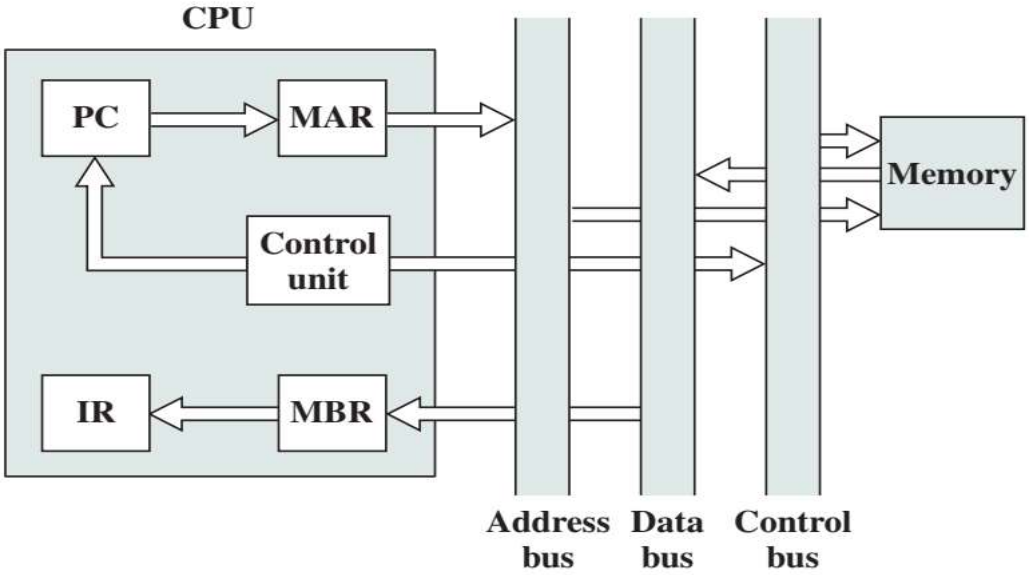
Fetch - 4 Registers

Memory Address Register (MAR)
- Connected to address bus
- Specifies address for read or write op

Memory Buffer Register (MBR)
- Connected to data bus
- Holds data to write or last data read

Program Counter (PC)
- Holds address of next instruction to be fetched

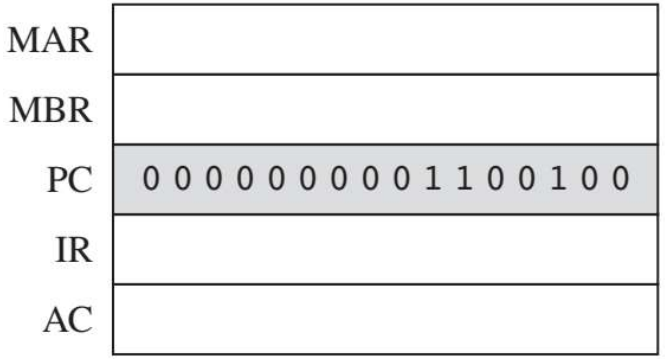
Instruction Register (IR)
- Holds last instruction fetched



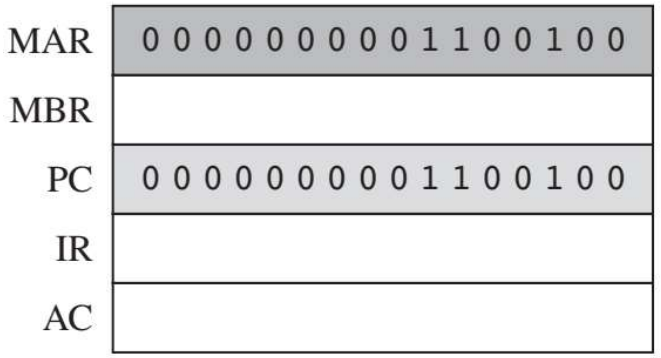
MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

Fetch cycle

- At the beginning of the fetch cycle, the address of the next instruction to be executed is in the program counter (PC); in this case, the address is 1100100

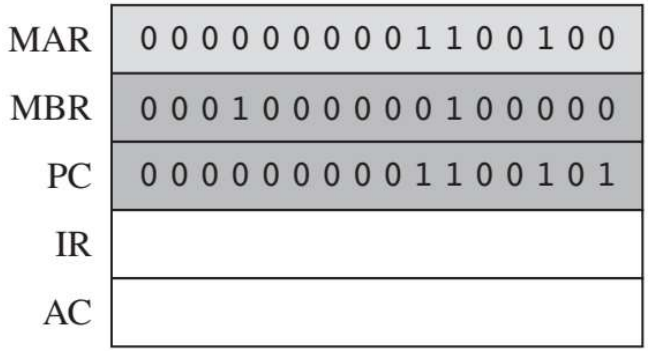


- The first step is to move that address to the memory address register (MAR) because this is the only register connected to the address lines of the system bus.



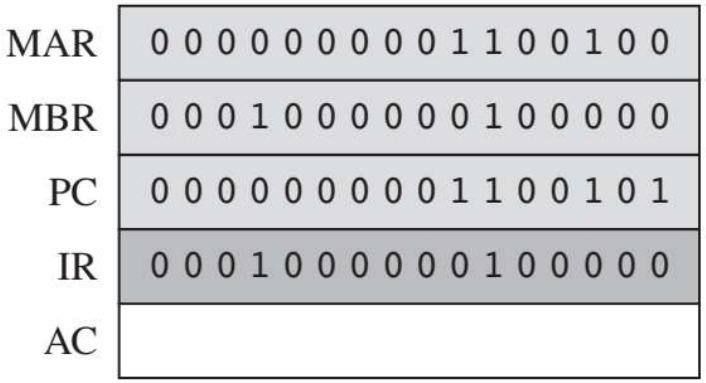
Fetch cycle

- The second step is to bring in the instruction.
- The desired address (in the MAR) is placed on the address bus → the control unit issues a READ command on the control bus → the result appears on the data bus and is copied into the memory buffer register (MBR).
- We also need to increment the PC by the instruction length to get ready for the next instruction.
- Because these two actions (read word from memory, increment PC) do not interfere with each other, we can do them simultaneously to save time.



Fetch cycle

- The third step is to move the contents of the MBR to the instruction register (IR). This frees up the MBR for use during a possible indirect cycle.
- Thus, the simple fetch cycle actually consists of three steps and four microoperations.
- Each micro-operation involves the movement of data into or out of a register.
- So long as these movements do not interfere with one another, several of them can take place during one step, saving time.



Fetch cycle

- Symbolically, we can write this sequence of events as follows:

$$t_1: \text{MAR} \leftarrow (\text{PC})$$
$$t_2: \text{MBR} \leftarrow \text{Memory}$$
$$\text{PC} \leftarrow (\text{PC}) + I$$
$$t_3: \text{IR} \leftarrow (\text{MBR})$$

- where I is the instruction length.
- We assume that a clock is available for timing purposes and that it emits regularly spaced clock pulses. Each clock pulse defines a time unit.
- Thus, all time units are of equal duration. Each micro-operation can be performed within the time of a single time unit.

Fetch cycle

- **First time unit:** Move contents of PC to MAR.
- **Second time unit:** Move contents of memory location specified by MAR to MBR. Increment by I the contents of the PC.
- **Third time unit:** Move contents of MBR to IR
- The third micro-operation could have been grouped with the fourth without affecting the fetch operation:

$$
\begin{aligned}
t_1&: \text{MAR} \leftarrow (\text{PC}) \\
t_2&: \text{MBR} \leftarrow \text{Memory} \\
t_3&: \text{PC} \leftarrow (\text{PC}) + I \\
&\quad\ \text{IR} \leftarrow (\text{MBR})
\end{aligned}
$$

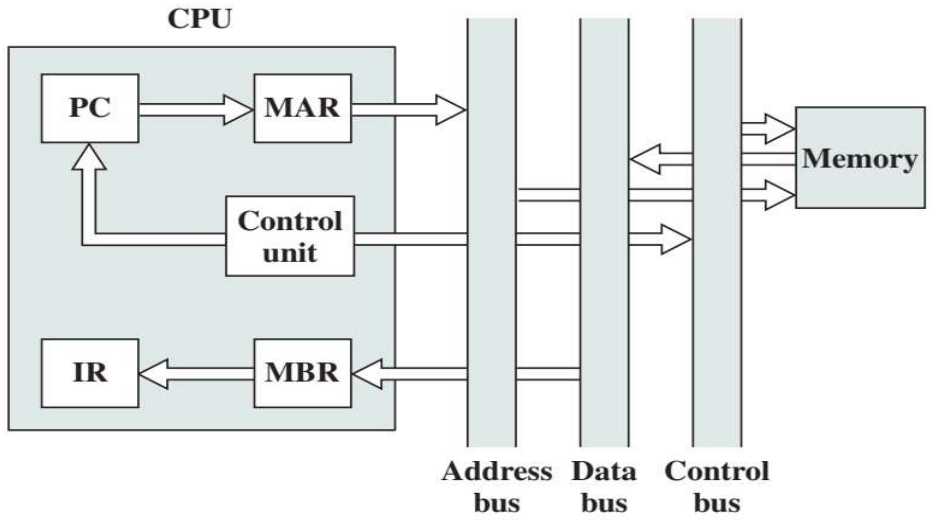
Rules for Grouping

1. The proper sequence of events must be followed. Thus (MAR←PC) must precede (MBR ← Memory) because the memory read operation makes use of the address in the MAR.
2. Conflicts must be avoided. One should not attempt to read to and write from the same register in one time unit, because the results would be unpredictable. For example, the micro-operations (MBR ← Memory) and (IR ← MBR) should not occur during the same time unit.

Indirect cycle

- Once an instruction is fetched, the next step is to fetch source operands.
- If the instruction specifies an indirect address, then an indirect cycle must precede the execute cycle.
- The data flow includes the following micro-operations:

t_1: MAR ← (IR(Address))
t_2: MBR ← Memory
t_3: IR(Address) ← (MBR(Address))



MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

Indirect cycle

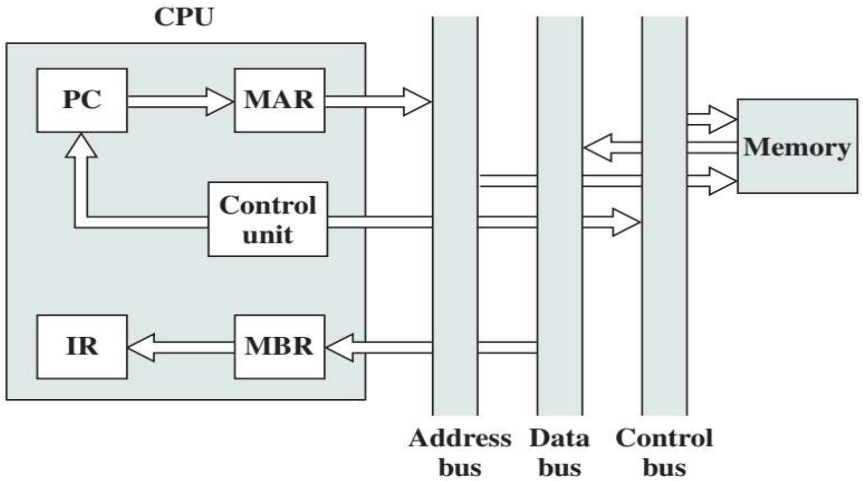
t_1: MAR ← (IR(Address))
t_2: MBR ← Memory
t_3: IR(Address) ← (MBR(Address))

- The address field of the instruction is transferred to the MAR.
- This is then used to fetch the address of the operand.
- Finally, the address field of the IR is updated from the MBR, so that it now contains a direct rather than an indirect address.
- The IR is now in the same state as if indirect addressing had not been used, and it is ready for the execute cycle.

Interrupt cycle

- At the completion of the execute cycle, a test is made to determine whether any enabled interrupts have occurred. If so, the interrupt cycle occurs.
- The nature of this cycle varies greatly from one machine to another.

```
t1: MBR ← (PC)
t2: MAR ← Save_Address
    PC ← Routine_Address
t3: Memory ← (MBR)
```



Interrupt cycle

t_1: MBR ← (PC)
t_2: MAR ← Save_Address
PC ← Routine_Address
t_3: Memory ← (MBR)

- In the first step, the contents of the PC are transferred to the MBR, so that they can be saved for return from the interrupt.
- Then the MAR is loaded with the address at which the contents of the PC are to be saved, and the PC is loaded with the address of the start of the interrupt-processing routine.
- These two actions may each be a single micro-operation.
- Once this is done, the final step is to store the MBR, which contains the old value of the PC, into memory.
- The processor is now ready to begin the next instruction cycle which is the ISR.

Execute cycle

- The fetch, indirect, and interrupt cycles are simple and predictable involving a small, fixed sequence of micro-operations and these micro-operations are repeated.
- Because of the variety opcodes, there are a number of different sequences of micro-operations that can occur.
- Consider an add instruction: **ADD R1, X** which adds the contents of the location X to register R1.
- The following sequence of micro-operations might occur:

t_1: MAR ← (IR(address))
t_2: MBR ← Memory
t_3: R1 ← (R1) + (MBR)

Execute cycle

t_1: MAR ← (IR(address))
t_2: MBR ← Memory
t_3: R1 ← (R1) + (MBR)

- We begin with the IR containing the ADD instruction.
- In the first step, the address portion of the IR is loaded into the MAR.
- Then the referenced memory location is read. Finally, the contents of R1 and MBR are added by the ALU.
- However, this is a simplified example.

Execute cycle

- Let us look at two more complex examples.
- A common instruction is increment and skip if zero: **ISZ X**
- The content of location X is incremented by 1. If the result is 0, the next instruction is skipped.
- A possible sequence of micro-operations is

t_1: MAR ← (IR(address))
t_2: MBR ← Memory
t_3: MBR ← (MBR) + 1
t_4: Memory ← (MBR)
If ((MBR) = 0) then (PC ← (PC) + I)

Execute cycle

t_1: MAR ← (IR(address))
t_2: MBR ← Memory
t_3: MBR ← (MBR) + 1
t_4: Memory ← (MBR)
If ((MBR) = 0) then (PC ← (PC) + I)

- The new feature introduced here is the conditional action.
- The PC is incremented if (MBR) = 0.
- This test and action can be implemented as one micro-operation.
- Note also that this micro-operation can be performed during the same time unit during which the updated value in MBR is stored back to memory.

Execute cycle

- Consider a subroutine call instruction.
- As an example, consider a branch-and-save-address instruction: **BSA X**
- The address of the instruction that follows the BSA instruction is saved in location X, and execution continues at location X+I
- The saved address will later be used for return.
- This is a straightforward technique for providing subroutine calls.
- The following micro-operations suffice:

t_1: MAR ← (IR(address))
MBR ← (PC)
t_2: PC ← (IR(address))
Memory ← (MBR)
t_3: PC ← (PC) + I

Execute cycle

t_1: MAR ← (IR(address))
MBR ← (PC)
t_2: PC ← (IR(address))
Memory ← (MBR)
t_3: PC ← (PC) + I

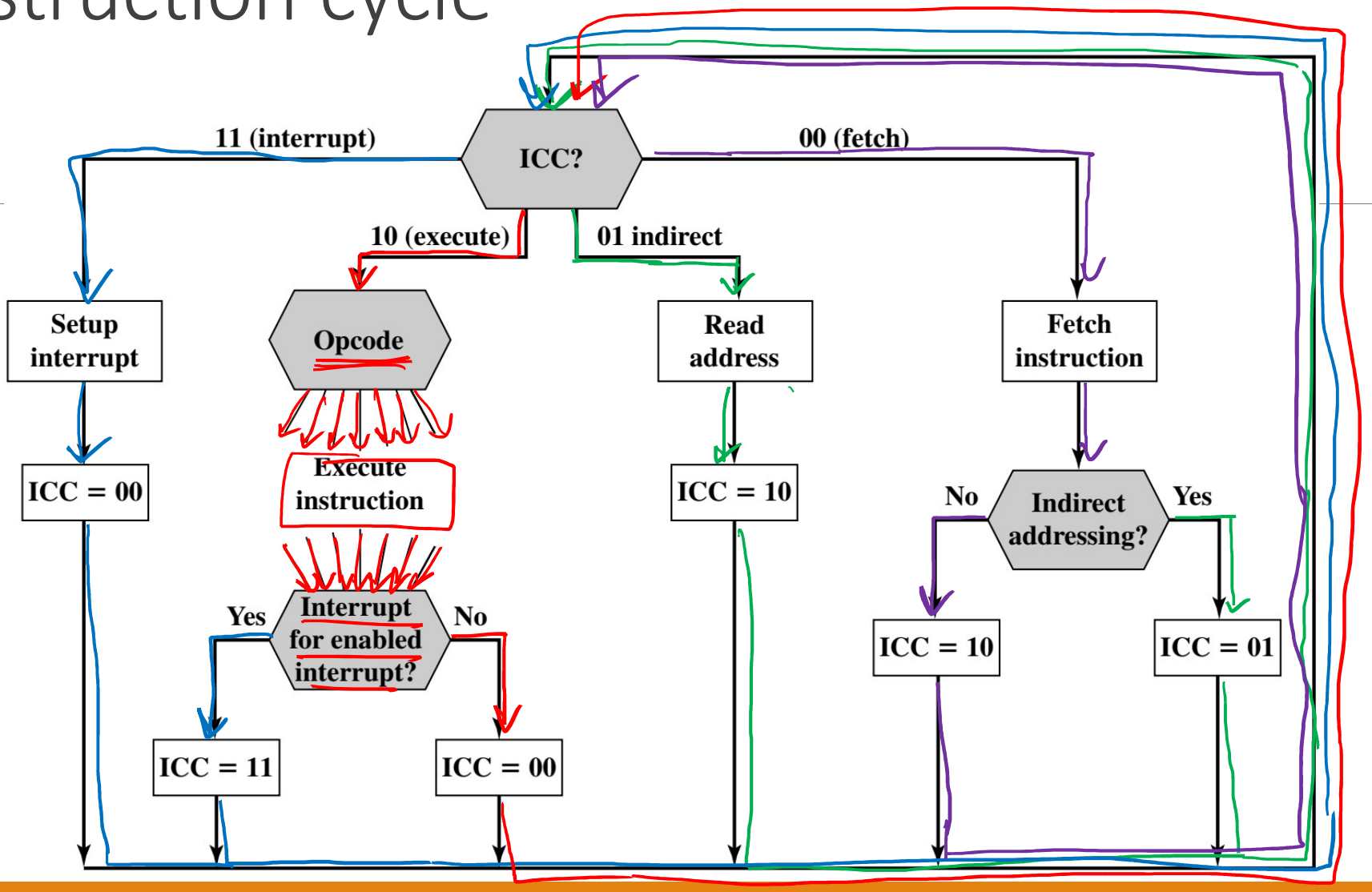
- The address in the PC at the start of the instruction is the address of the next instruction in sequence.
- This is saved at the address designated in the IR.
- The latter address is also incremented to provide the address of the instruction for the next instruction cycle.

Instruction cycle

- Each phase of the instruction cycle can be decomposed into a sequence of elementary micro-operations.
- To complete the picture, we need to tie sequences of micro-operations together.
- We assume a new 2-bit register called the *instruction cycle code* (ICC). The ICC designates the state of the processor in terms of which portion of the cycle it is in:

1. 00: Fetch
2. 01: Indirect
3. 10: Execute
4. 11: Interrupt

Instruction cycle



Control of the processor

- By reducing the operation of the processor to its most fundamental level, we are able to define exactly what it is that the control unit must cause to happen.
- Thus, we can define the ***functional requirements*** for the control unit: those functions that the control unit must perform.

1. Define the basic elements of the processor.
2. Describe the micro-operations that the processor performs.
3. Determine the functions that the control unit must perform to cause the micro-operations to be performed.

Control of the processor

1. **Sequencing:** The control unit causes the processor to step through a series of micro-operations in the proper sequence, based on the program being executed.
2. **Execution:** The control unit causes each micro-operation to be performed.

- The preceding is a functional description of what the control unit does.
- The key to how the control unit operates is the use of **control signals**.

Control unit implementation

Implementation of Control unit is broadly of two types

1. Hardwired implementation (RISC)
2. Microprogrammed implementation (CISC)

Hardwired Implementation

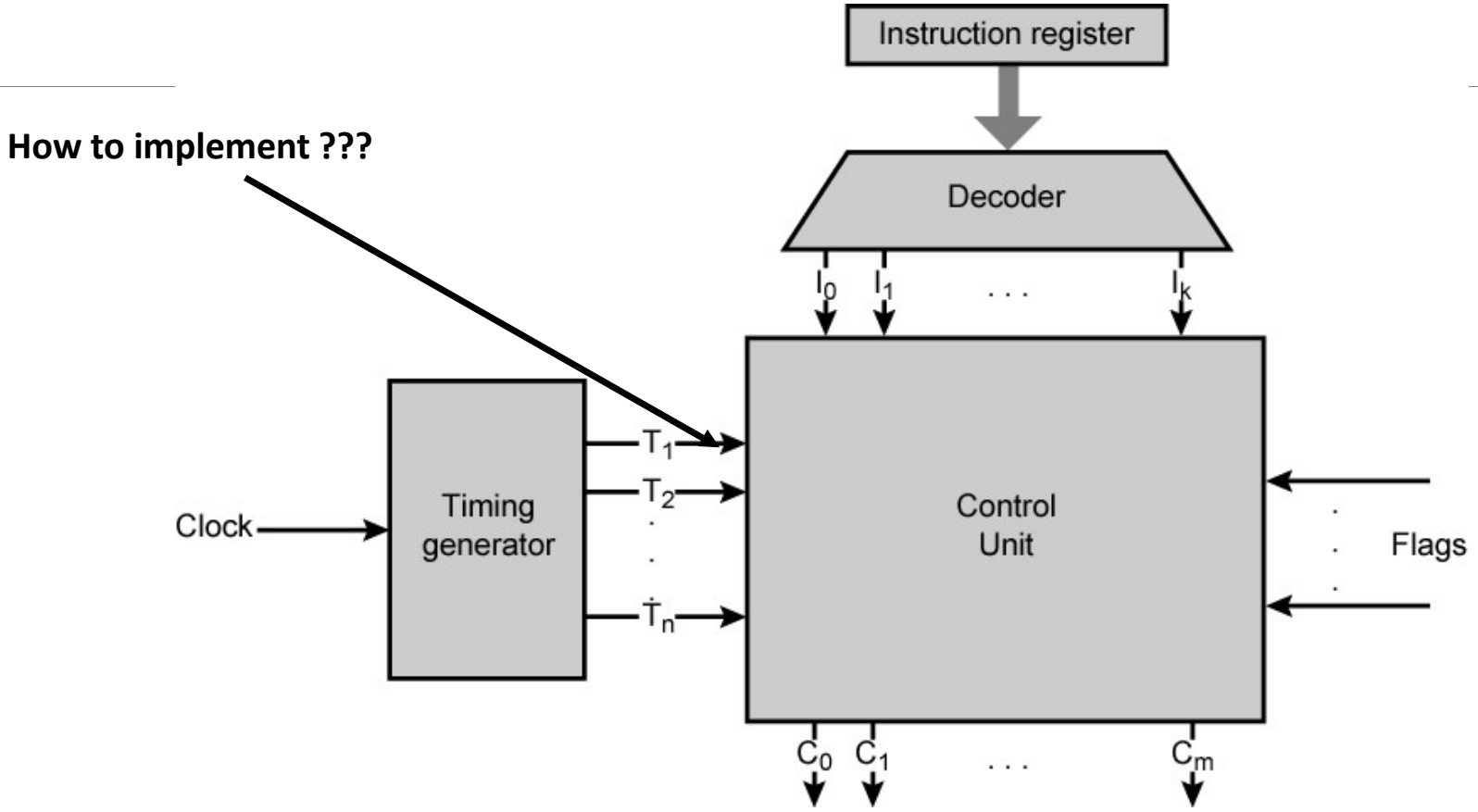
Control unit inputs

Flags and control bus
- Each bit means something

Instruction register
- Op-code causes different control signals for each different instruction
- Unique logic for each op-code
- Decoder takes encoded input and produces single output
- *n* binary inputs and 2^n outputs

Clock
- Repetitive sequence of pulses
- Useful for measuring duration of micro-ops
- Must be long enough to allow signal propagation
- Different control signals at different times within instruction cycle
- Need a counter with different control signals for t1, t2 etc.

Control Unit with Decoded Inputs



4:16 DECODER

I1	I2	I3	I4	O1	O2	O3	O4	O5	O6	O7	O8	O9	O10	O11	O12	O13	O14	O15	O16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Hardwired control unit methods

State table method:

T-States	I1	I2	I3	----	In
T1	C11	C12	C13	---	C1n
T2	C21	C22	C23	---	C2n
---	---	---	---	---	---
Tm	Cm1	Cm2	Cm3	---	Cmn

1) It is the most basic type of hardwired control unit.
2) Here the behavior of the control unit is represented in the form of a table called the state table.
3) The rows represent the T-states and the columns indicate the instructions.
4) Each intersection indicates the control signal to be produced, in the corresponding T-state of every instruction.
5) A circuit is then constructed based on every column of this table, for each instruction.

ADVANTAGE:
It is the simplest method and is ideally suited for very small instruction sets.

DRAWBACK:
As the number of instructions increase, the circuit becomes bigger and hence more complicated.
As a tabular approach is used, instead of a logical approach (flowchart), there are duplications of many circuit elements in various instructions.

Hardwired Control Unit Logic

For each control signal, to derive a Boolean expression of that signal as a function of the inputs

Let us consider a **single control signal, C5**, which causes data to be read from the external data bus into the MBR

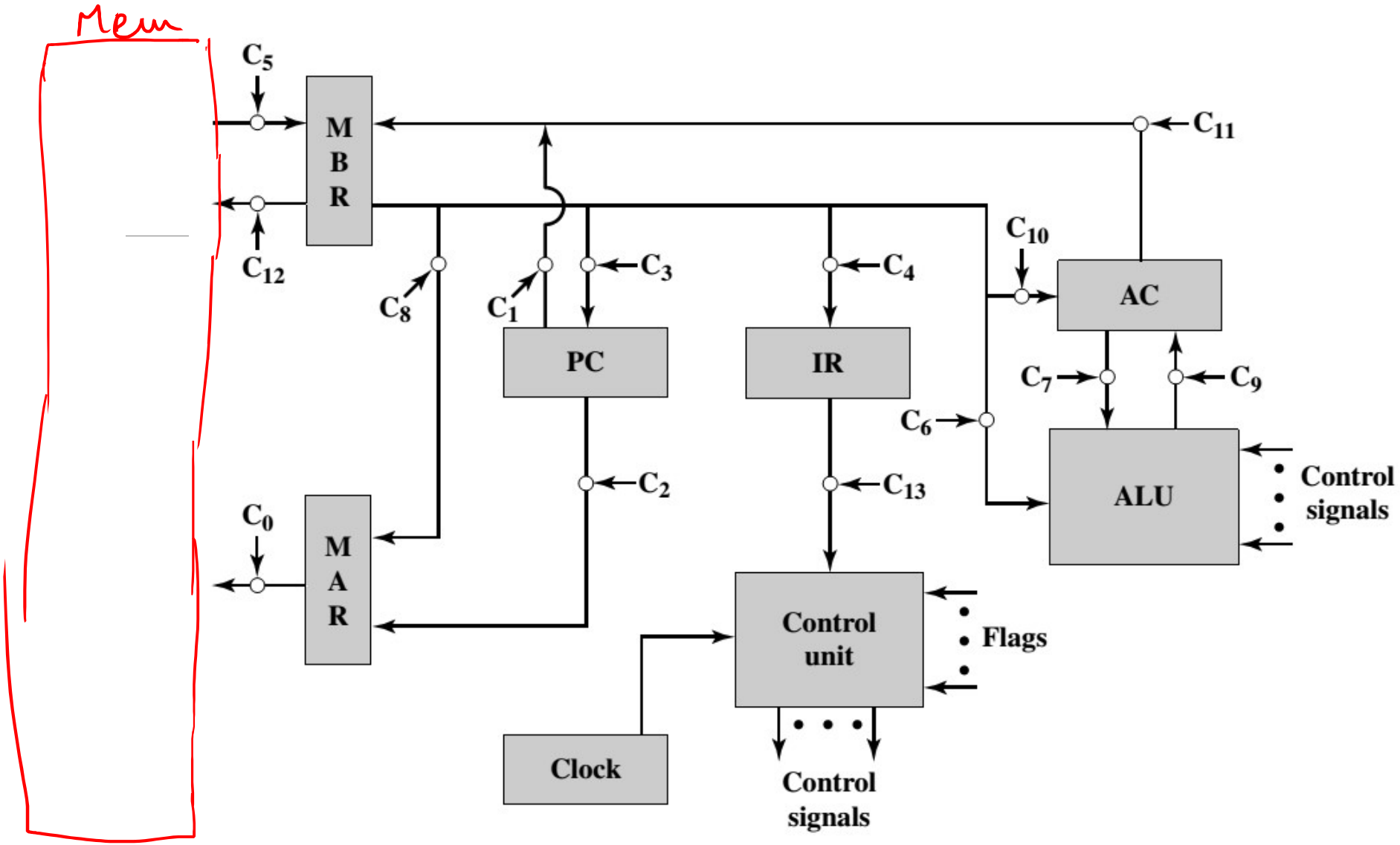
Let us define two new **control signals, P and Q**, that have the following interpretation:

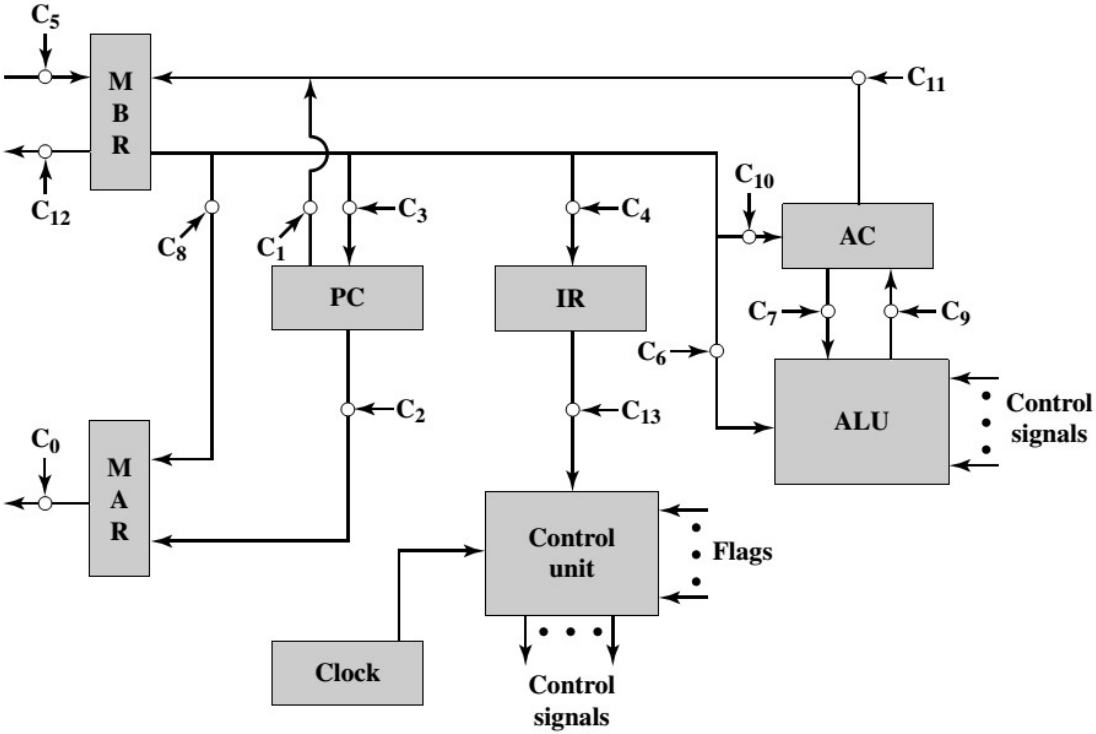
PQ = 00 Fetch Cycle

PQ = 11 Interrupt Cycle

PQ = 10 Execute Cycle

PQ = 01 Indirect Cycle


Mem
C5
MBR
C11
C12
C10
C8
C1
C3
C4
AC
PC
IR
C7
C9
C6
C2
C13
ALU
Control signals
C0
MAR
Control unit
Flags
Clock
Control signals



	Micro-operations	Active Control Signals
Fetch:	t_1: MAR ← (PC)	C_2
	t_2: MBR ← Memory PC ← (PC) + 1	C_5, C_R
	t_3: IR ← (MBR)	C_4
Indirect:	t_1: MAR ← (IR(Address))	C_8
	t_2: MBR ← Memory	C_5, C_R
	t_3: IR(Address) ← (MBR(Address))	C_4
Interrupt:	t_1: MBR ← (PC)	C_1
	t_2: MAR ← Save-address PC ← Routine-address	
	t_3: Memory ← (MBR)	C_{12}, C_W

State table Logic Example

Then C5 can be defined as:

$$C_5 = \overline{P} \cdot \overline{Q} \cdot T_2 + \overline{P} \cdot Q \cdot T_2$$

That is, the control signal C5 will be asserted during the second time unit of both the fetch and indirect cycles.

This is not complete

C5 is also needed during the execute cycle. For our simple example, let us assume that there are only three instructions that read from memory: LDA, ADD and AND. Now we can define C5 as

$$C_5 = \overline{P} \cdot \overline{Q} \cdot T_2 + \overline{P} \cdot Q \cdot T_2 + P \cdot \overline{Q} \cdot (LDA + ADD + AND) \cdot T_2$$

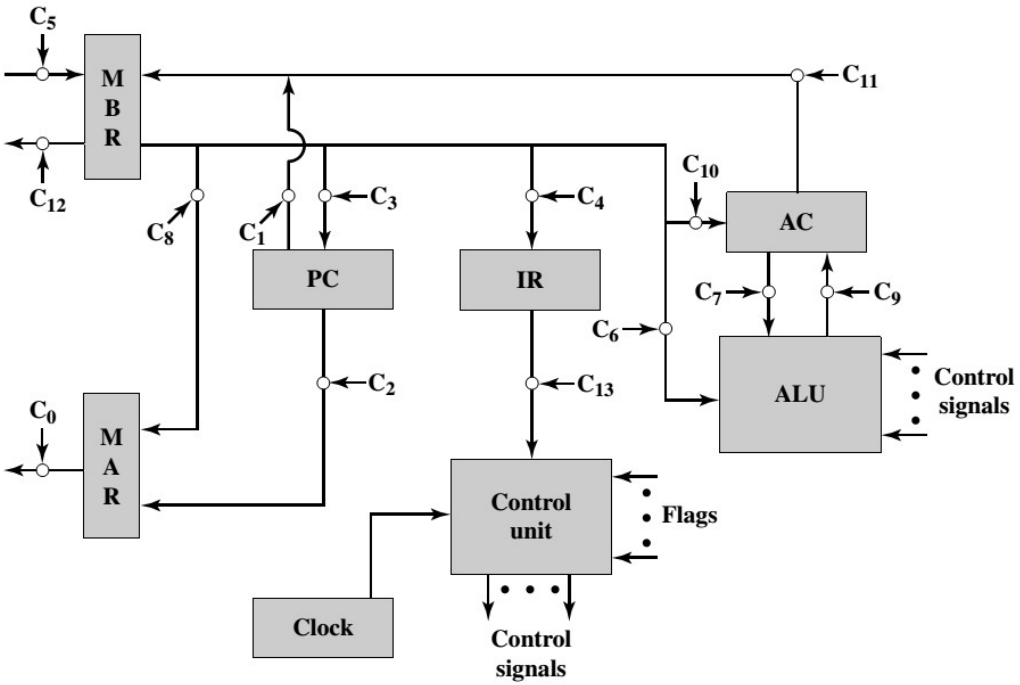
Is it that simple?

No. In a modern complex processor, the number of Boolean equations needed to define the control unit is very large.

	Micro-operations	Active Control Signals
Fetch:	t_1: MAR ← (PC)	C_2
	t_2: MBR ← Memory PC ← (PC) + 1	C_5, C_R
	t_3: IR ← (MBR)	C_4
Indirect:	t_1: MAR ← (IR(Address))	C_8
	t_2: MBR ← Memory	C_5, C_R
	t_3: IR(Address) ← (MBR(Address))	C_4
Interrupt:	t_1: MBR ← (PC)	C_1
	t_2: MAR ← Save-address PC ← Routine-address	
	t_3: Memory ← (MBR)	C_{12}, C_W

PQ = 00 Fetch Cycle
PQ = 11 Interrupt Cycle
PQ = 10 Execute Cycle
PQ = 01 Indirect Cycle

*Hence an efficient and simpler approach, known as **Flow chart/ delay element method** is usually used*



	Micro-operations	Active Control Signals
Fetch:	t_1: MAR ← (PC)	C_2
	t_2: MBR ← Memory	C_5, C_R
	PC ← (PC) + 1	
	t_3: IR ← (MBR)	C_4
Indirect:	t_1: MAR ← (IR(Address))	C_8
	t_2: MBR ← Memory	C_5, C_R
	t_3: IR(Address) ← (MBR(Address))	C_4
Interrupt:	t_1: MBR ← (PC)	C_1
	t_2: MAR ← Save-address	
	PC ← Routine-address	
	t_3: Memory ← (MBR)	C_{12}, C_W

•Referring to the figures above, depict the control signals required for the execution cycle of LOAD 200 and STORE 300. **[5]**

•Express the Boolean expression for each of the control signals. **[5]**

•Implement the Boolean expressions for each of the control signals using minimum number of gates. **[5]**

Load 200

Control sig

1st part

t_4: MAR ← IR(addr) ⟶ C_8

t_5: MBR ← Mem ⟶ C_5, C_R

t_6: AC ← MBR ⟶ C_{10}

Store 300

t_4: MAR ← IR(addr) ⟶ C_8

t_5: MBR ← AC ⟶ C_{11}

t_6: Mem ← MBR ⟶ C_{12}, C_W

$C_8, C_5, C_R, C_{10}, C_{11}, C_{12}, C_W$ —

$C_8 = \bar{P}Q \cdot t_2 + P\bar{Q} \cdot t_4 \cdot$ LOAD

$+ P\bar{Q} \cdot t_4 \cdot$ STORE

$C_5 = \bar{P}\bar{Q} \cdot t_2 + \bar{P}Q \cdot t_2 + P\bar{Q} \cdot t_5 \cdot$ LOAD ✓

$C_R = \bar{P}\bar{Q} \cdot t_2 + \bar{P}Q \cdot t_2 + P\bar{Q} \cdot t_5 \cdot$ LOAD ✓

$C_{10} = P\bar{Q} \cdot t_6 \cdot$ LOAD

$C_{11} = P\bar{Q} \cdot t_5 \cdot$ STORE

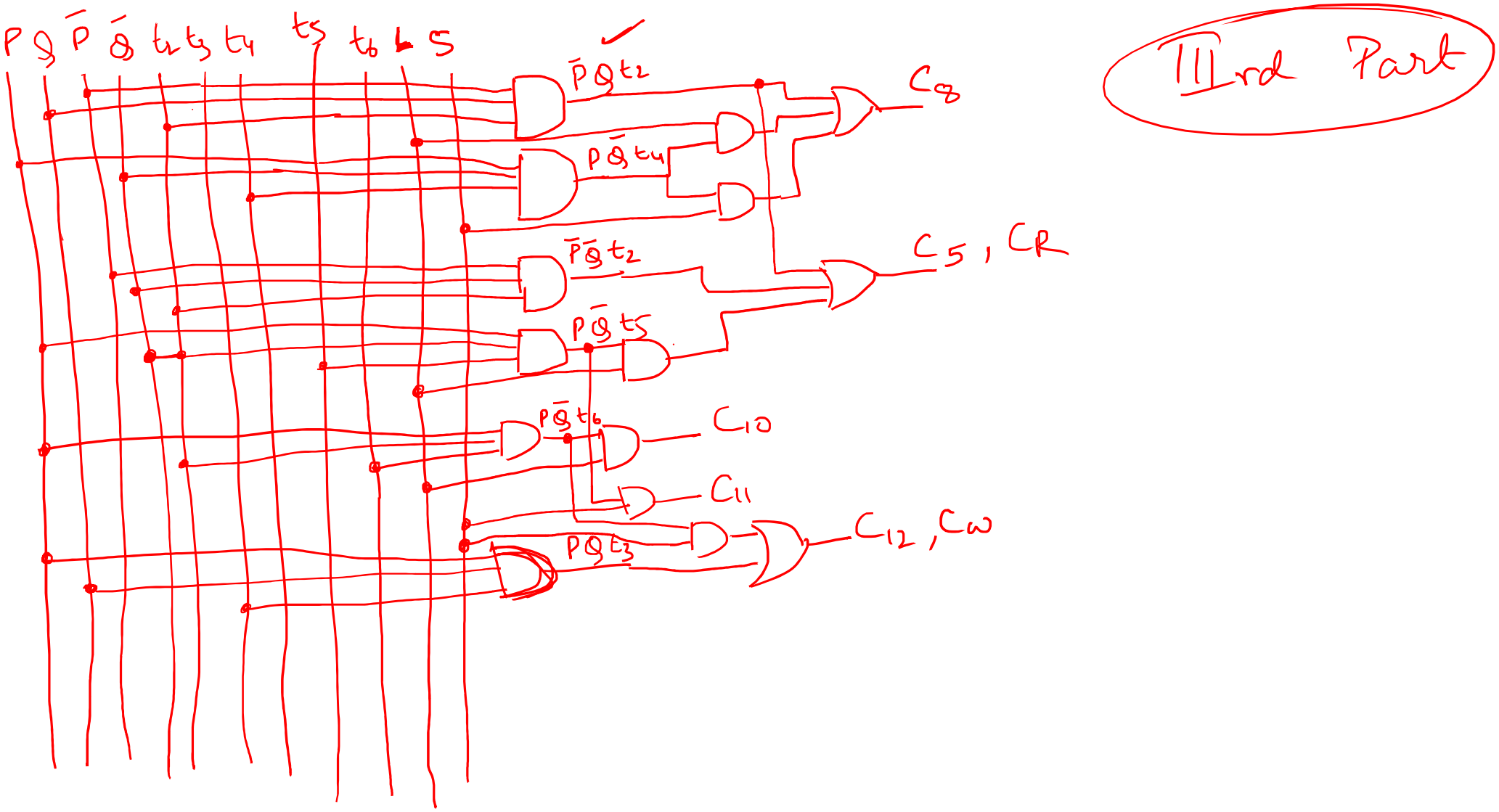
$C_{12} = PQ \cdot t_3 + P\bar{Q} \cdot t_6 \cdot$ STORE

$C_W = PQ \cdot t_3 + P\bar{Q} \cdot t_6 \cdot$ STORE

00 → Fetch
01 → 2nd
10 → Exc
11 → Int

II nd Part

$P, Q, \bar{P}, \bar{Q}, t_2, t_3, t_4, t_5, t_6$, LOAD, STORE

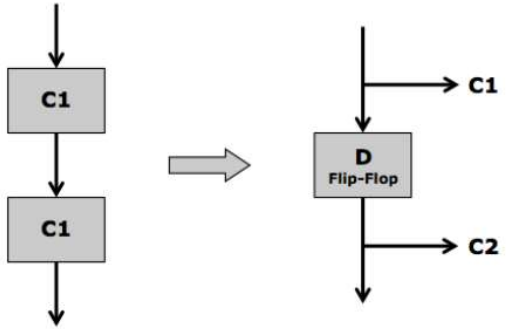

P Q̄ P̄ Q̄ t1 t3 t4 t5 t6 t5
P̄Q̄t2
C8
PQ̄t4
C5, CR
P̄Q̄t2
PQ̄t5
PQ̄t6
C10
C11
C12, Cω
PQt3

III rd Part

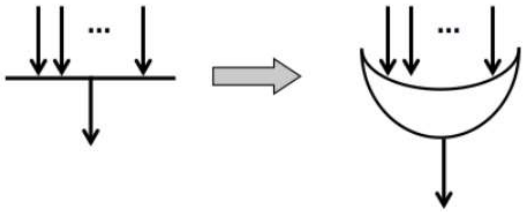
Flow chart/ Delay element method:

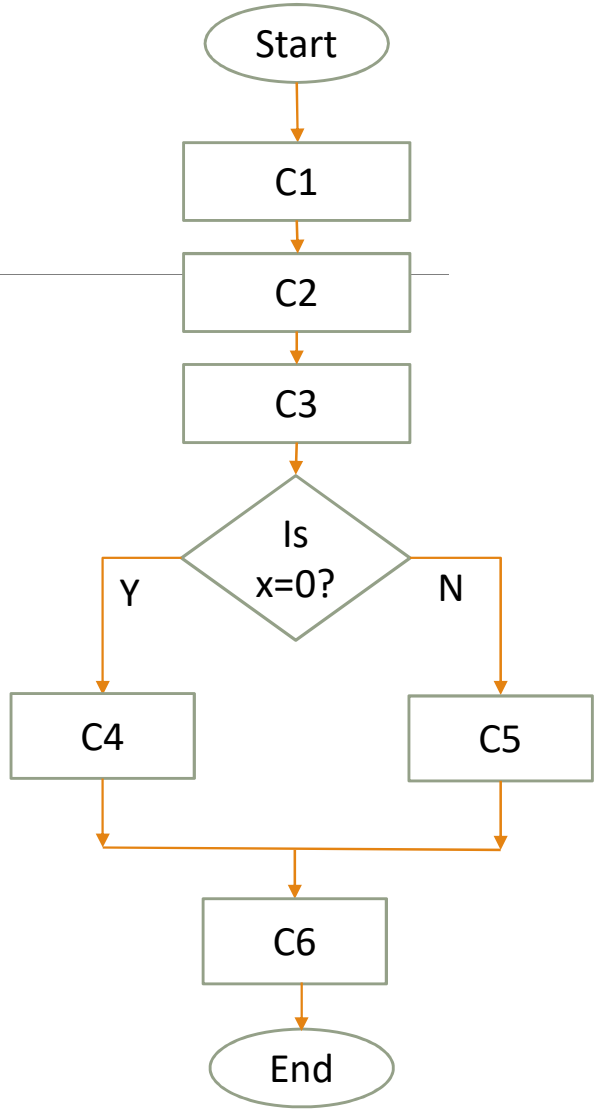
DELAY ELEMENT METHOD

1) Here the behavior of the control unit is represented in the form of a flowchart.
2) Each step in the flowchart represents a control signal to be produced.
3) Once all steps of a particular instruction, are performed, the complete instruction gets executed.
4) Control signals perform Micro-Operations, which require one T-states each.
5) Hence between every two steps of the flowchart, there must be a delay element.
6) The delay must be exactly of one T-state. This delay is achieved by D Flip-Flops.
7) These **D Flip-Flops** are **inserted** between every two **consecutive control signals**.

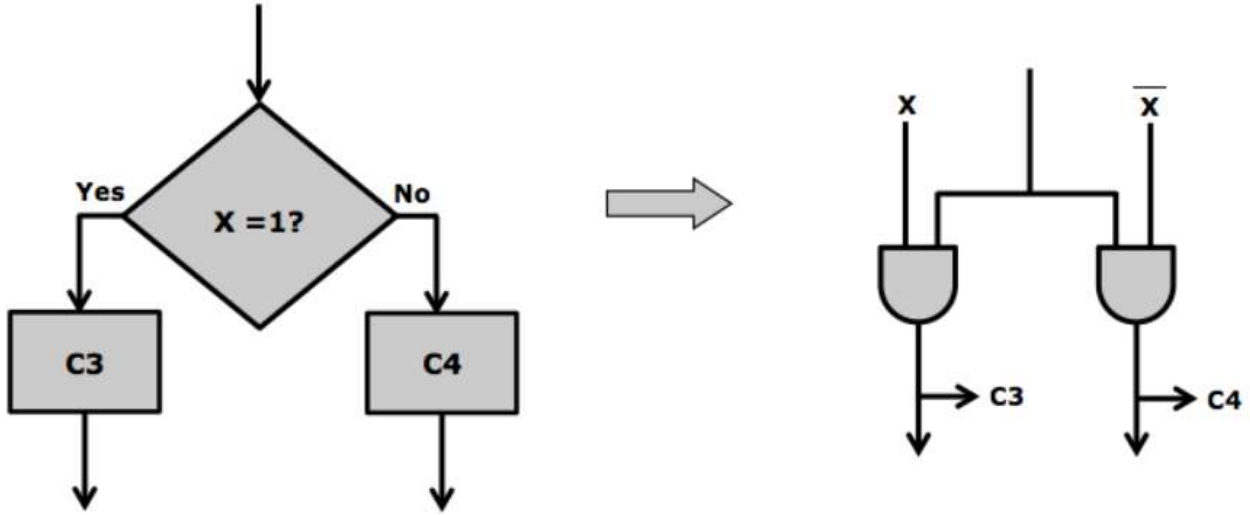


8) Of all D Flip-Flops only one will be active at a time. So the method is also called **"One Hot Method"**.
9) In a **multiple entry point**, to combine two or more paths, we use an **OR gate**.





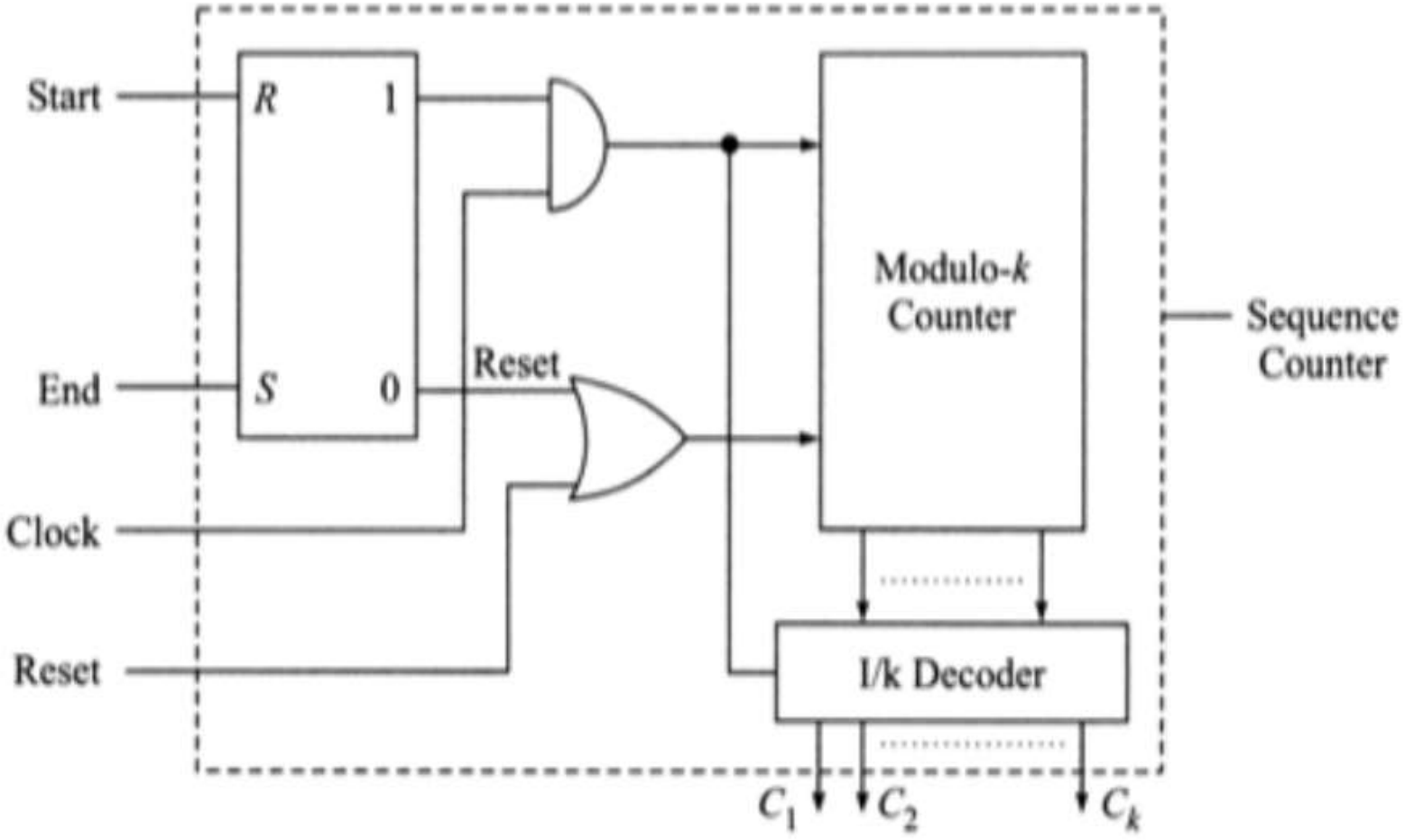
10)A **decision box** is replaced by a set of two complementing **AND gates**



ADVANTAGE:
As the method has a logical approach, it can **reduce the circuit complexity**.
This is done by re-utilizing common elements between various instructions.

DRAWBACK:
As the no of instructions increase, the number of **D Flip-Flops increase**, so the **cost increases**.
Moreover, **only one of those D Flip-Flops are actually active at a time.**

Sequence counter method



1) This is the **most popular** form of hardwired control unit.
2) It follows the same **logical approach of a flowchart**, like the Delay element method, but does not use all those unnecessary D Flip-Flops.
3) First a flowchart is made representing the behavior of a control unit.
4) It is then **converted into a circuit** using the same principle of AND & OR gates.
5) We need a **delay of 1 T-state** (one clock cycle) between every two **consecutive control signals**.
6) That is achieved by the above circuit.
7) If there are **"k" number of distinct steps** producing control signals, we employ a **"mod k"** and **"k" output decoder**.
8) The counter will **start counting** at the beginning of the instruction.
9) The "clock" input via an AND gate ensures each count will be generated **after 1 T-state**.
10) The count is given to the **decoder** which **triggers the generation of "k" control signals**, each after a delay of 1 T-state.
11) When the **instruction ends**, the **counter is reset** so that next time, it begins from the first count.

ADVANTAGE:

Avoids the use of **too many D Flip-Flops**.

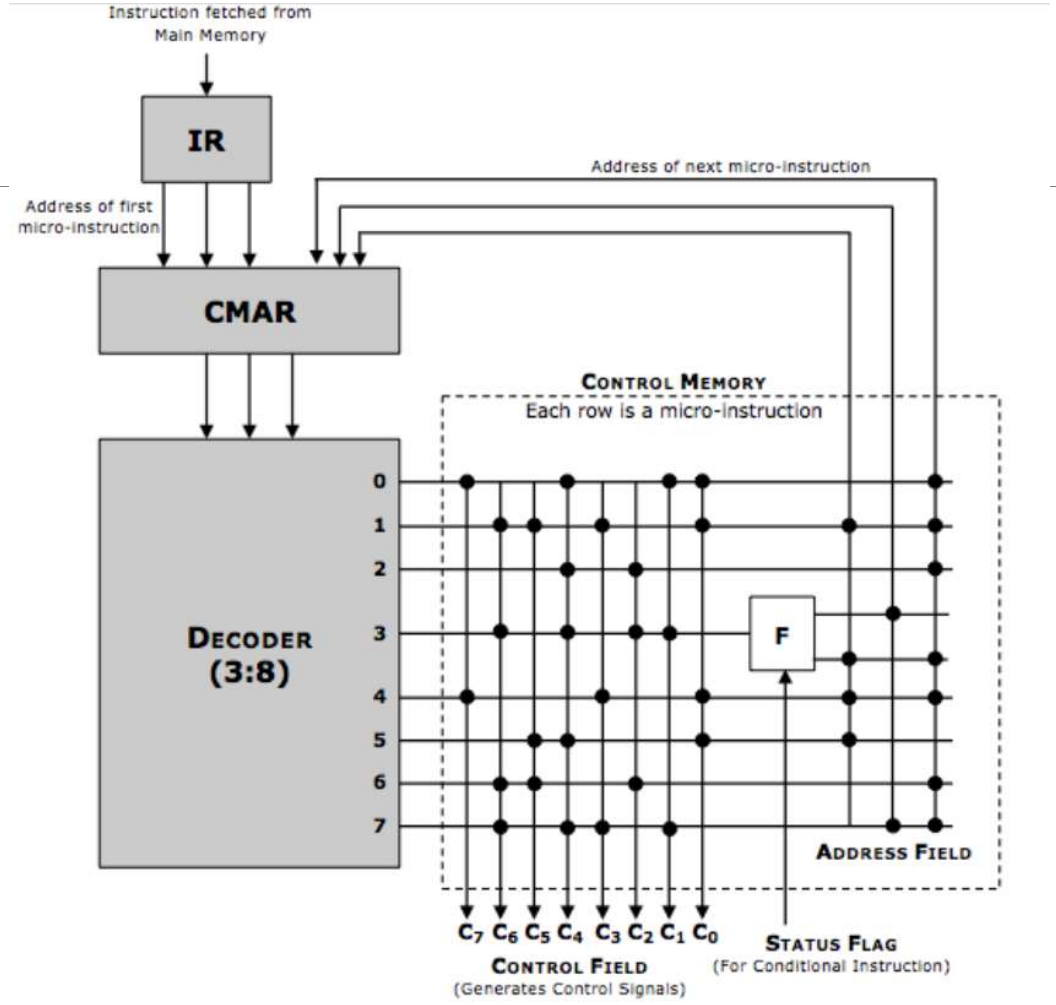
GENERAL DRAWBACKS OF A HARDWIRED CONTROL UNIT

1) Since they are based on hardware, as the instruction set increases, the **circuit becomes more and more complex**. For modern processors having hundreds of instructions, it is virtually impossible to create Hardwired Control Units.
2) Such large circuits are **very difficult to debug**.
3) As the **processor gets upgraded**, the entire Control Unit has to be **redesigned**, due to the **rigid nature** of hardware design.

Wilkes' Design for a Microprogrammed Control Unit

1) Microprogrammed Control Unit produces control signals by **software**, using **micro-instructions**
2) A program is a set of instructions.
3) An instruction requires a set of Micro-Operations.
4) **Micro-Operations are performed by control signals**.
5) Instead of generating these control signals by hardware, **we use micro-instructions**.
6) This means **every instruction requires a set of micro-instructions**
7) **This is called its micro-program**.
8) Microprograms for all instructions are **stored in a small memory called "Control Memory"**.
9) The Control memory is **present inside the processor**.
10) Consider an **Instruction** that is **fetched from the main memory** into the Instruction Register (**IR**).
11) The processor uses its unique **"opcode"** to identify the **address of the first micro-instruction**.
12) That address is loaded into **CMAR** (Control Memory Address Register).
13) CMAR passes the address to the **decoder**.
14) The decoder **identifies the corresponding micro-instruction** from the Control Memory.
15) A micro-instruction has **two fields**: a control filed and an address field.
16) **Control field:** Indicates the **control signals to be generated**.
17) **Address field:** Indicates the **address of the next micro-instruction**.
18) This address is further **loaded into CMAR** to **fetch the next** micro-instruction.
19) For a **conditional micro-instruction**, there are **two address fields**.
20) This is because, the address of the next micro-instruction **depends on the condition**.
21) The condition (true or false) is **decided by the appropriate control flag**.
22) The control memory is usually implemented using FLASH ROM as it is writable yet non volatile.

Wilkes' Design for Micro programmed CU



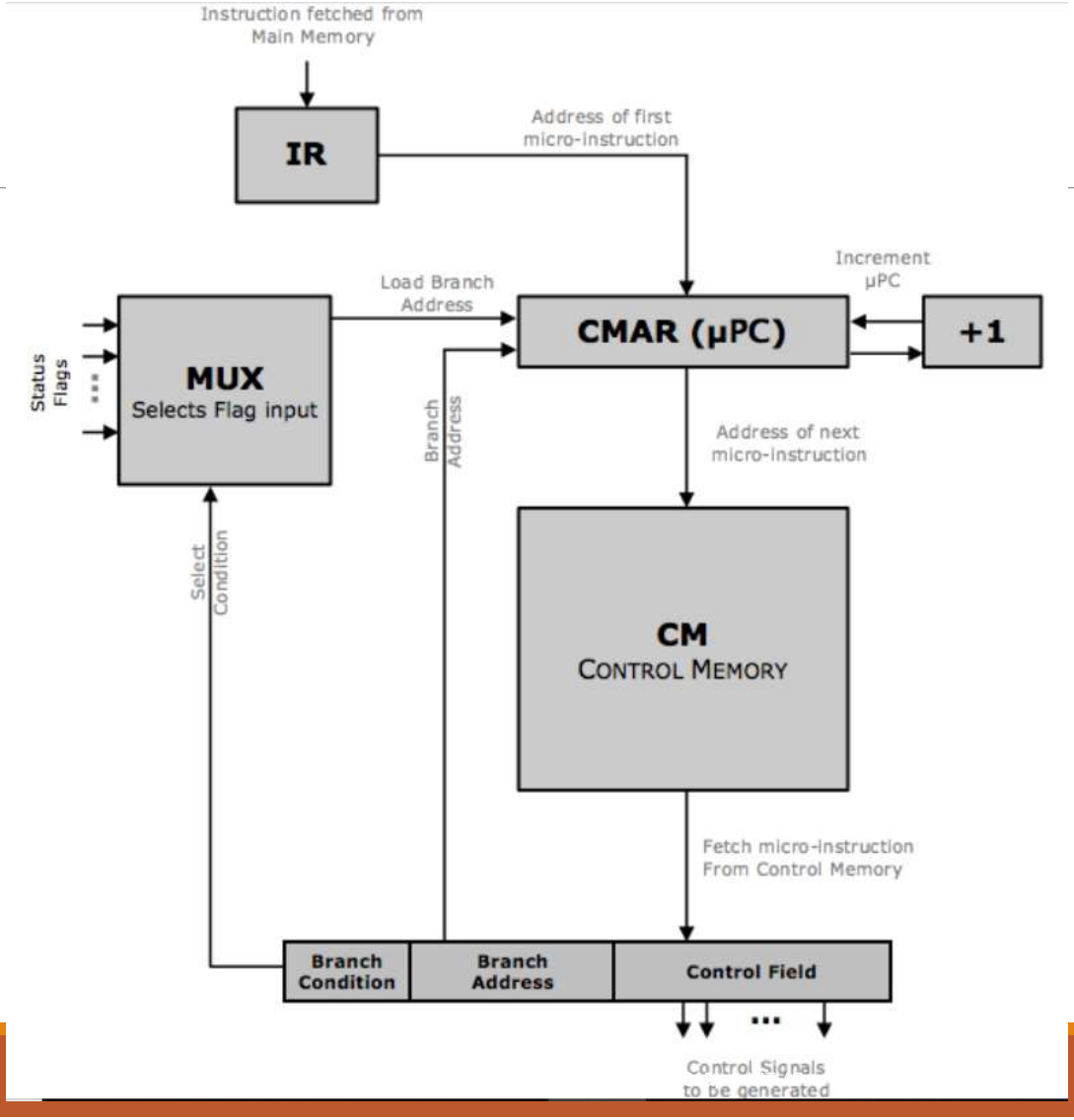
ADVANTAGES

1) The biggest advantage is **flexibility.**
2) Any **change** in the control unit can be performed by **simply changing the micro-instruction**.
3) This makes **modifications and up gradation** of the Control Unit **very easy**.
4) Moreover, software can be **much easily debugged** as compared to a large Hardwired Control Unit.

DRAWBACKS

1) **Control memory** has to be present **inside** the processor, **increasing its size.**
2) This also **increases the cost** of the processor.
3) The **address field** in every micro-instruction **adds more space** to the control memory. This can be easily **avoided** by proper **micro-instruction sequencing**.

Actual Microprogrammed control unit



TYPICAL MICROPROGRAMMED CONTROL UNIT

1) Microprogrammed Control Unit produces control signals by **software**, using **micro-instructions**.
2) A program is a set of instructions.
3) An instruction requires a set of Micro-Operations.
4) **Micro-Operations are performed by control signals**.
5) Here, these control signals are generated using **micro-instructions**.
6) This means **every instruction requires a set of micro-instructions**
7) **This is called its micro-program**.
8) Microprograms for all instructions are **stored in a small memory called "Control Memory"**.
9) The Control memory is **present inside the processor**.
10) Consider an **Instruction** that is **fetched from the main memory** into the Instruction Register (**IR**).
11) The processor uses its unique **"opcode"** to identify the **address of the first micro-instruction**.
12) That address is loaded into **CMAR** (Control Memory Address Register) also called **μIR**.
13) This address is **decoded** to **identify the corresponding μ-instruction** from the Control Memory.
14) There is a big **improvement over Wilkes' design**, to reduce the size of micro-instructions.
15) Most micro-instructions will **only have a Control field**.
16) The **Control field** Indicates the **control signals to be generated**.
17) Most micro-instructions **will not have an address field**.
18) Instead, **μPC will simply get incremented** after every micro-instruction.
19) This is as long as the μ-program is executed **sequentially**.
20) If there is a **branch μ-instruction** only then there will be an **address filed**.
21) If the branch is **unconditional**, the **branch address will be directly loaded into CMAR.**
22) For **Conditional branches**, the Branch condition will **check the appropriate flag**.
23) This is done using a **MUX** which has all flag inputs.
24) If the **condition is true**, then the **MUX will inform CMAR to load the branch address**.
25) If the **condition is false** CMAR will simply get **incremented**.
26) The control memory is usually implemented using **FLASH ROM** as it is **writable** yet **non volatile**.

ADVANTAGES

1) The biggest advantage is **flexibility.**
2) Any **change** in the control unit can be performed by **simply changing the micro-instruction**.
3) This makes **modifications and up gradation** of the Control Unit **very easy**.
4) Moreover, software can be **much easily debugged** as compared to a large Hardwired Control Unit.
5) Since most micro-instructions are executed sequentially, they don't **need for an address field**.
6) This significantly **reduces the size of micro-instructions**, and hence the **Control Memory**.

DRAWBACKS

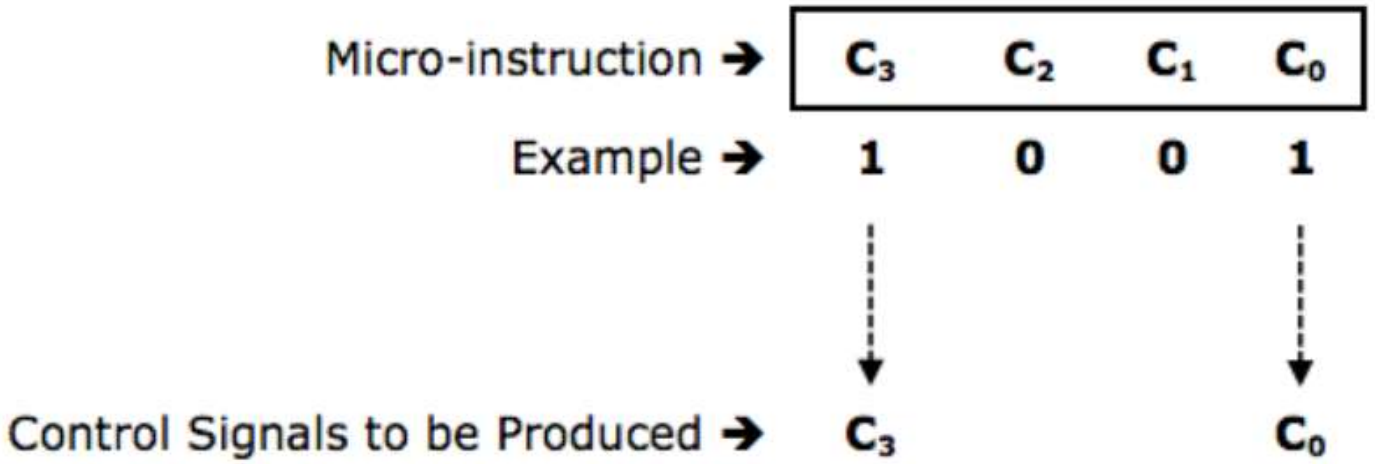
1) **Control memory** has to be present **inside** the processor, **increasing its size.**
2) This also **increases the cost** of the processor.

Micro-instruction Format

The main part of the micro-instruction is its control field.
It determines the control signals to be produced.
It can be of two different formats: Horizontal or Vertical.

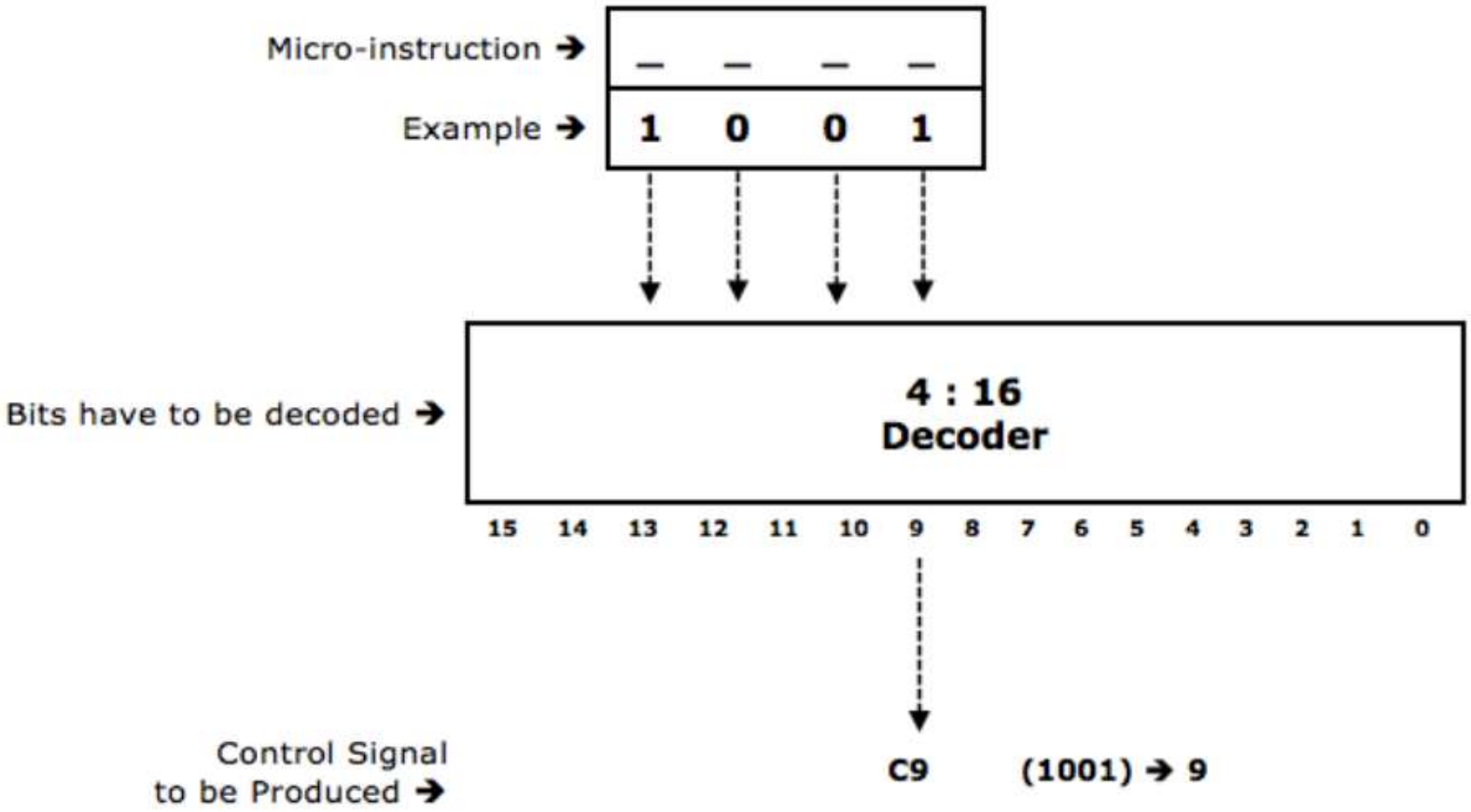
1) Horizontal Micro-Instruction

Here every bit of the micro-instruction corresponds to a control signal.
Whichever bit is "1", that particular control signal will be produced by the micro-instruction.



2) VERTICAL MICRO-INSTRUCTION

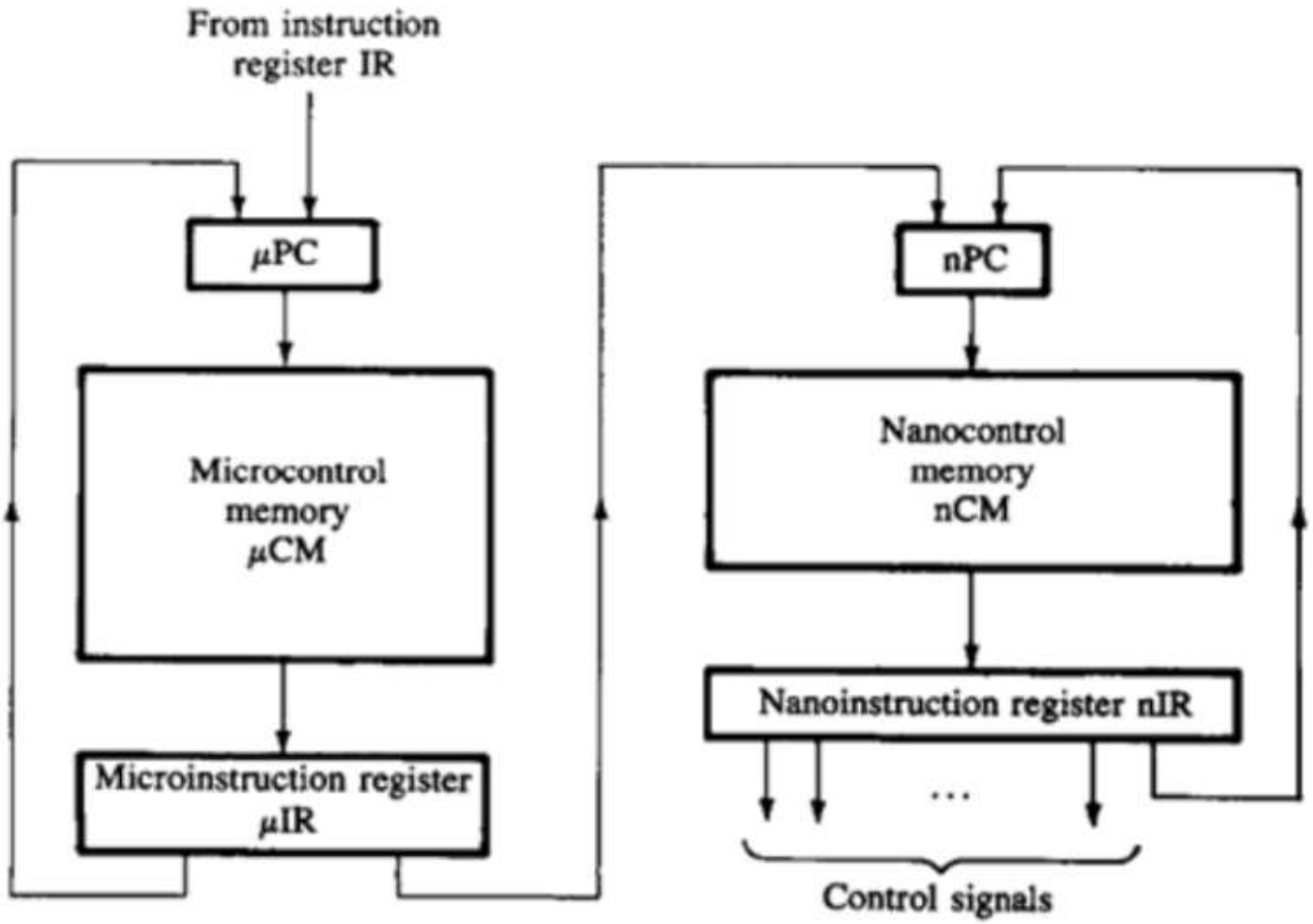
Here bits of the micro-instruction have to be decoded.
The decoded output decides the control signal to be produced.



	HORIZONTAL MICRO-INSTRUCTION	VERTICAL MICRO-INSTRUCTION
1	Every bit of the micro-instruction corresponds to a control signal.	Bits of the micro-instruction have to be decoded to produce control signals.
2	Does not require a decoder.	Needs a decoder.
3	N bits in the micro-instruction will totally produce N control signals.	N bits in the micro-instruction will totally produce 2^N control signals.
4	Multiple control signals can be produced by one micro-instruction.	Only one control signal can be produced by one micro-instruction.
5	As the control signals increase, the micro-instruction grows wider. Hence the Control Memory grows Horizontally.	To produce more control signals, more number of micro-instructions are needed. Hence the Control Memory grows Vertically.
6	Executes faster as no decoding needed.	Executes slower as decoding is needed.
7	Micro-instruction are very wide. Hence Control memory is large.	Micro-instruction are much narrower. Hence Control memory is small.
8	Circuit is simpler as a decoder is not needed.	Circuit is more complex as a decoder is needed.

NANO-PROGRAMMING

1) **Horizontal** µ-instructions can produce **multiple control signals** simultaneously, but are **very wide**.
2) This makes the Control Memory **very large in size**.
3) **Vertical** micro-instructions are **narrow**, **but on decoding** can **produce** only **one control signal**.
4) This makes the Control Memory **small** but the execution is **slow**.
5) Hence a **combination of both techniques** is needed called **Nano-Programming**.
6) Here we have a **two level control memory**.
7) The instruction is fetched from the **main memory into IR**.
8) Using its **opcode** we load **address of its first micro-instruction** into **µPC**,
9) Using this address we **fetch the micro-instruction** from µ-Control Memory (**µCM**) **into µIR**.
10) This is in **vertical form** and has to be decoded.
11) The decoded output **loads a new address** in a Nano program counter (**nPC**).
12) Using this address we **fetch the Nano-instruction** from Nano-Control Memory (**nCM**) into **nIR**.
13) This is in **horizontal form** and can **directly generate control signals**.
14) Such a combination **gives advantage of both techniques**.
15) The size of the Control Memory is **small** as µ-instructions are **Vertical**.
16) Multiple control signals can be **produced simultaneously** as Nano-instructions are **Horizontal**.


From instruction register IR
μPC
Microcontrol memory μCM
Microinstruction register μIR
nPC
Nanocontrol memory nCM
Nanoinstruction register nIR
...
Control signals

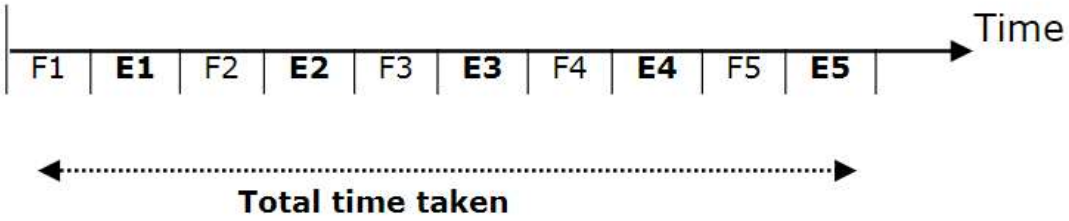
Overlapping different stages of an instruction is called pipelining.

An instruction requires several steps which mainly involve fetching, decoding and execution.
If these steps are performed one after the other, they will take a long time.
As processors became faster, several of these steps started to get overlapped, resulting in faster processing. This is done by a mechanism called pipelining.

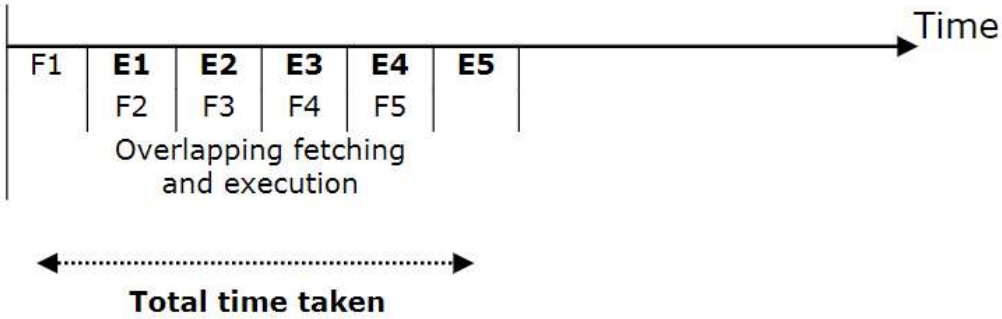
2 STAGE PIPELINING - 8086

Here the instruction process in divided into two stages of fetching and execution.
Fetching of the next instruction takes place while the current instruction is being executed. Hence instructions are being processed at any point of time.

NON-PIPELINED PROCESSOR EG: 8085

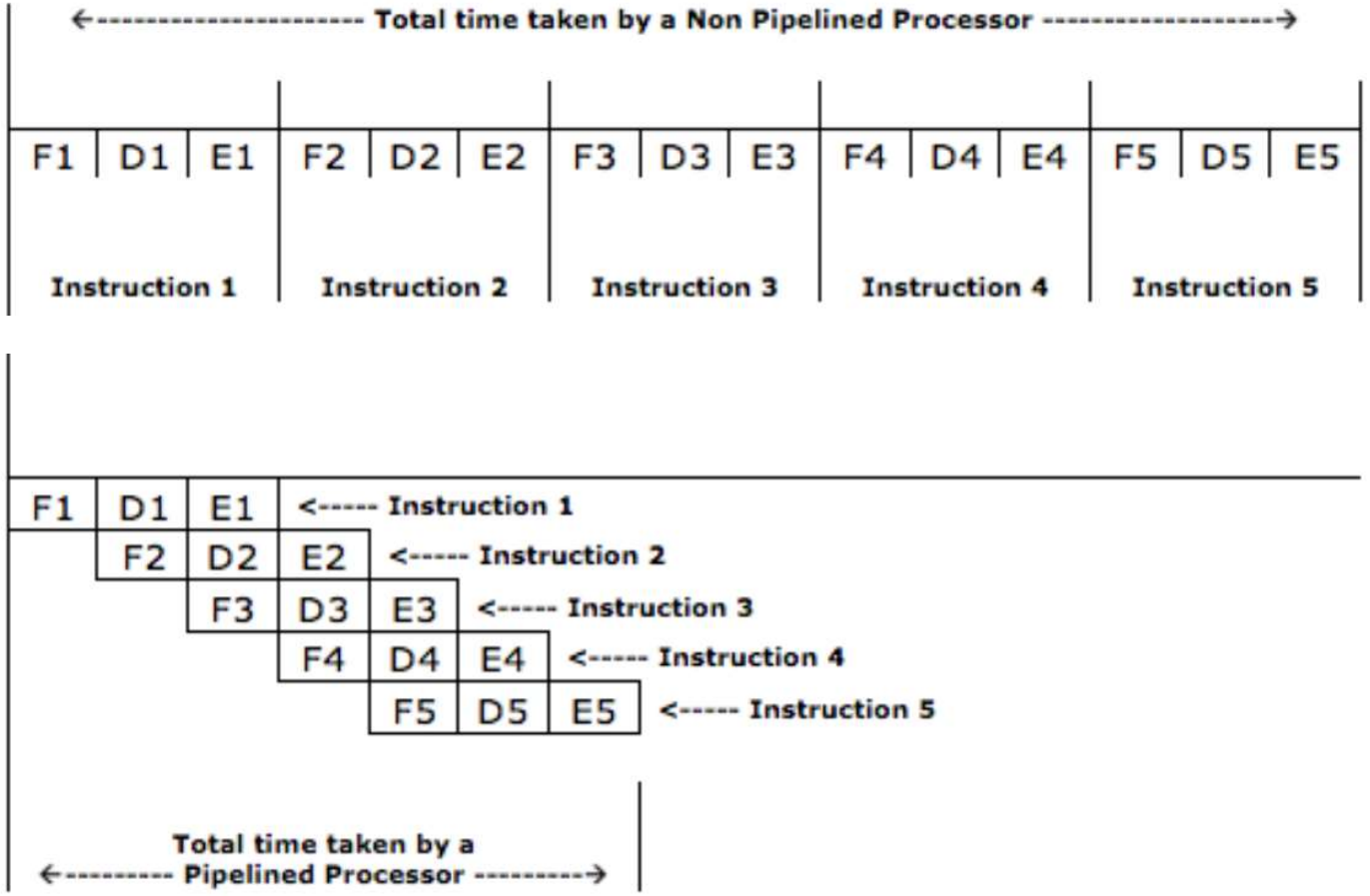


PIPELINED PROCESSOR EG: 8086



3 Stage Pipelining – 80386/ ARM 7

Here the instruction process in divided into three stages of **fetching, decoding and execution** and are overlapped. Hence **three instructions** are being processed at any point of time.



4 Stage Pipelining

Fetch, Decode, Execute, Store

5 Stage Pipelining - Pentium

Fetch, Decode, Address Generation, Execute, Store

6 Stage Pipelining – Pentium Pro

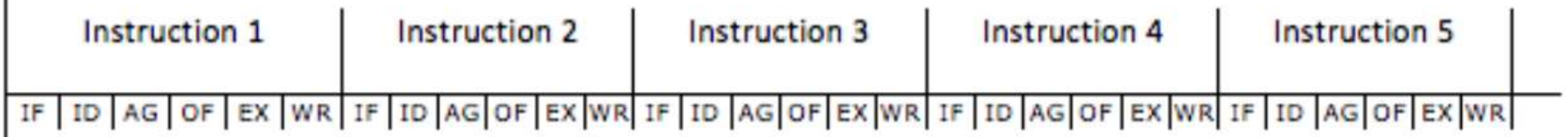
Instruction Fetch (IF):	Fetch the instruction
Instruction Decode (ID):	Decode the instruction.
Address Generation (AG):	Calculate address of Memory operand
Operand Fetch (OF):	Fetch memory operand
Execute (EX):	Execute the operation
Write Back (WR):	Write back/ Store the result

Non Pipelined Processor

K = No of stages = 6.

N = No of Instructions = 5.

Total cycles = K x N = 6 x 5 = 30 cycles.

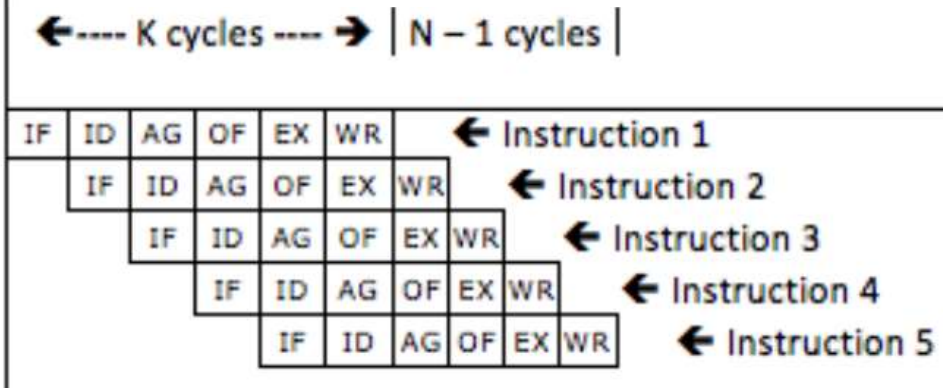


Non Pipelined Processor

K = No of stages = 6.

N = No of Instructions = 5.

Total cycles = K x N = 6 x 5 = 30 cycles.



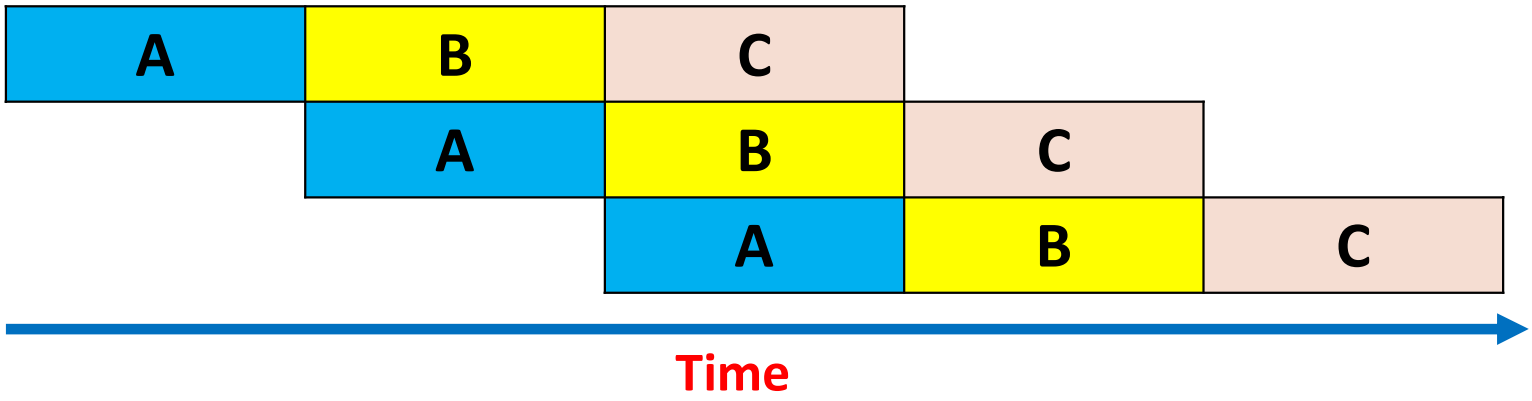
CONTENT

- ✓ **Problems on Pipeline**

PIPELINE

Non-pipeline

Pipeline



MODEL-1 : PIPELINE PROBLEM-1

Problem 1: Find the **number of clock cycles** required to execute 10 instructions with pipeline method and without pipeline method for the following instruction structure ?

I	F (2)	D (1)	E (1)

Fetch - 2 Clock cycle

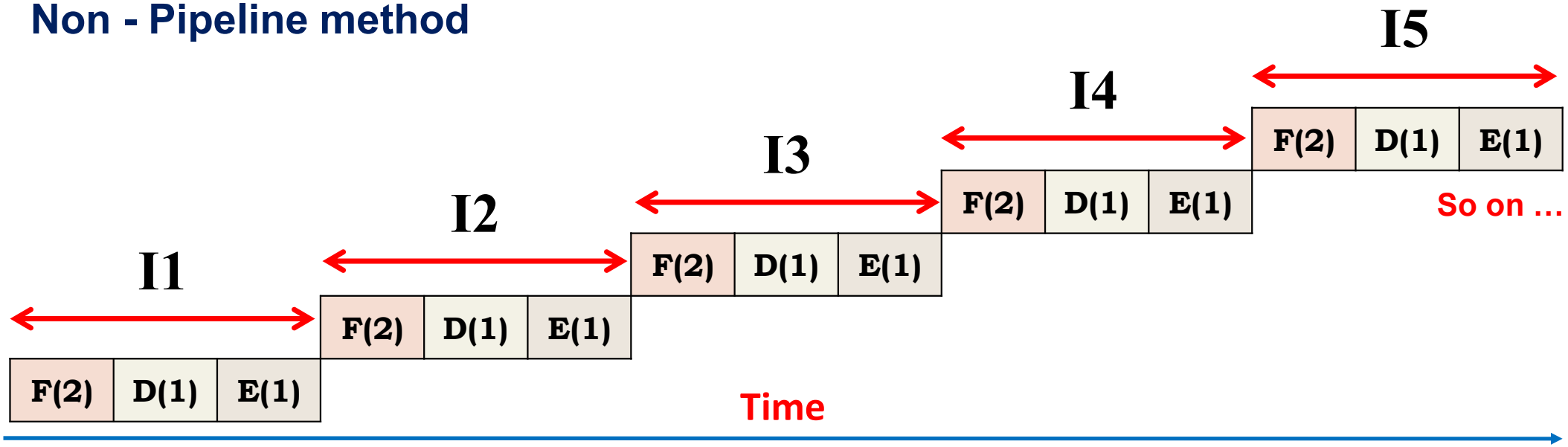
Decoding - 1 Clock cycle

Execution - 1 Clock cycle

I1
I2
I3
I4
I5
I6
I7
I8
I9
I10

MODEL-1 : PIPELINE PROBLEM-1

Non - Pipeline method



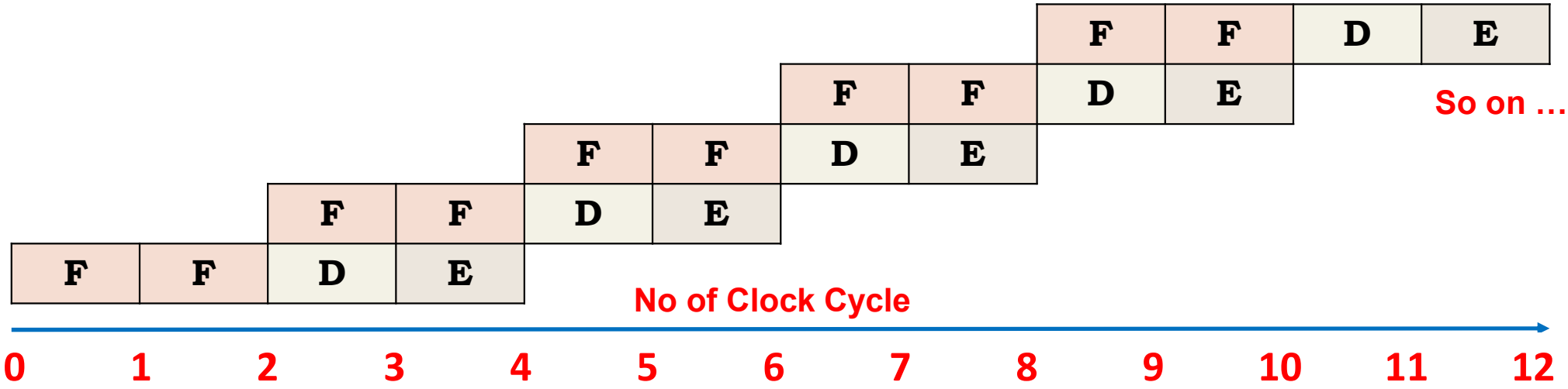
No of Clock Cycle required to execute 10 instructions is

= (No of instructions) x (Total no of required for single instruction)

= 10 x 4 = **40 Clock cycles**

MODEL-1 : PIPELINE PROBLEM-1

Pipeline method



No of Clock Cycle required to execute 10 instructions is

= (No of clocks required for 1st instruction)+ ((no of instruction -1) x (difference between two instruction))

= 4 + ((10-1) x 2) = 4 + (9 x 2) = 4 + 18 = **22 Clock cycles**

MODEL-1 : PIPELINE PROBLEM-2

Problem 2: Find the **number of clock cycles** required to execute 100 instructions with pipeline method and without pipeline method for the following instruction structure ?

I	F (2)	D (1)	E (3)

Fetch - 2 Clock cycle

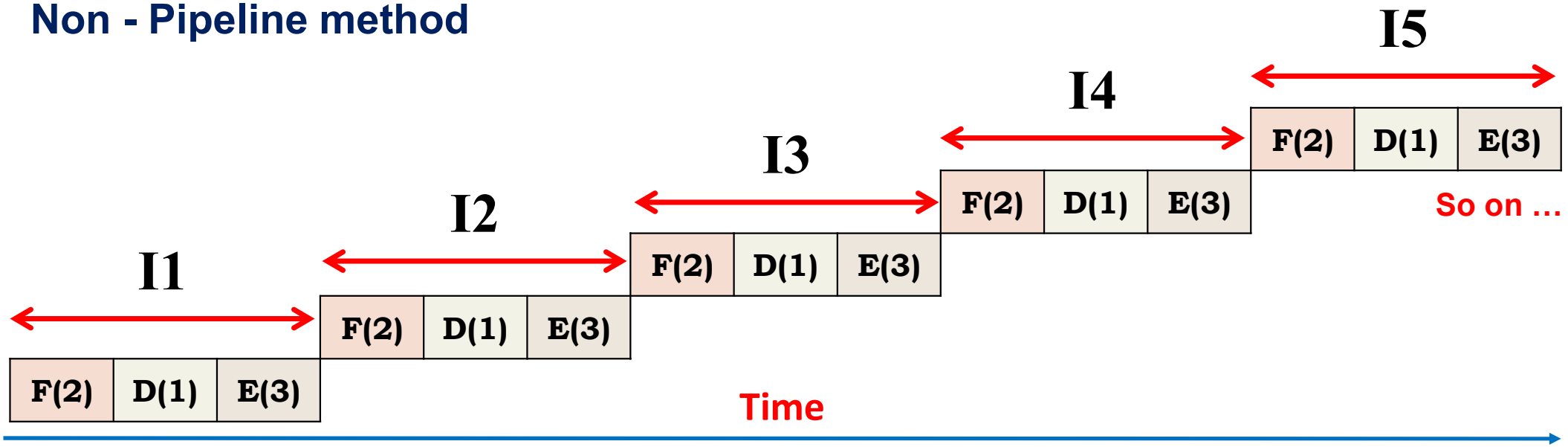
Decoding - 1 Clock cycle

Execution - 3 Clock cycle

I1
I2
I3
I4
I5
I6
I7
I8
I9
I10

MODEL-1 : PIPELINE PROBLEM-2

Non - Pipeline method



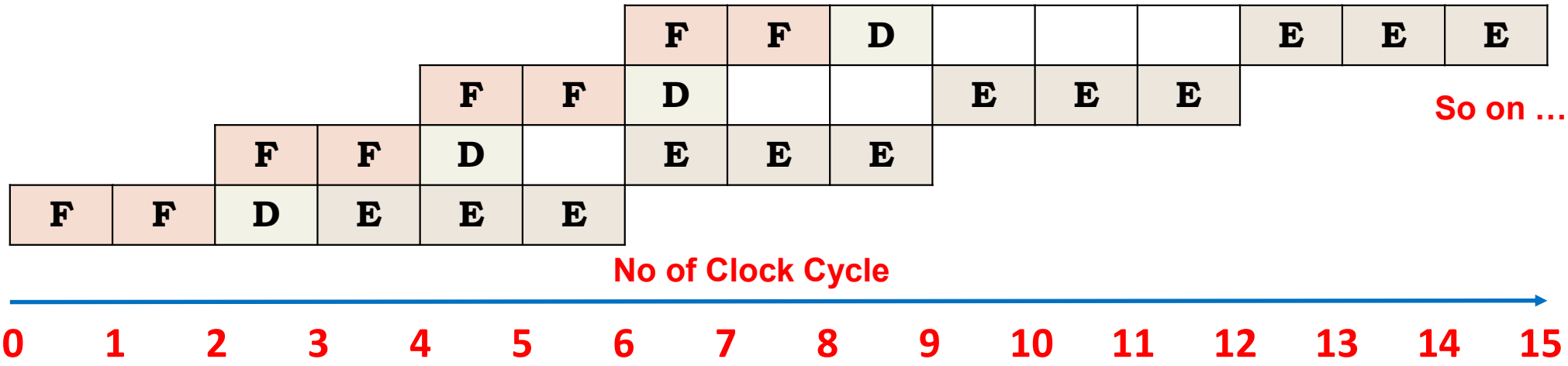
No of Clock Cycle required to execute 100 instructions is

= (No of instructions) x (Total no of cycles required for single instruction)

= 100 x 6 = **600 Clock cycles**

MODEL-1 : PIPELINE PROBLEM-2

Pipeline method



No of Clock Cycle required to execute 100 instructions is

= (No of clocks required for 1st instruction)+ ((no of instruction -1) x (difference between two instruction))

= 6 + ((100-1) x 3) = 6 + (99 x 3) = 6 + 297 = **303 Clock cycles**

MODEL-1 : PIPELINE PROBLEM ASSIGNMENT-1

Assignment - 1: Find the **number of clock cycles** required to execute 1000 instructions with pipeline method and without pipeline method for the following instruction structure ?

I	F (1)	D (2)	E (3)

Fetch - 1 Clock cycle

Decoding - 2 Clock cycle

Execution - 3 Clock cycle

I1
I2
I3
I4
I5
I6
I7
I8
I9
I10

MODEL-1 : PIPELINE PROBLEM ASSIGNMENT-1

Assignment - 1: Find the **number of clock cycles** required to execute 1000 instructions with pipeline method and without pipeline method for the following instruction structure ?

I	F (1)	D (2)	E (3)

Fetch - 1 Clock cycle

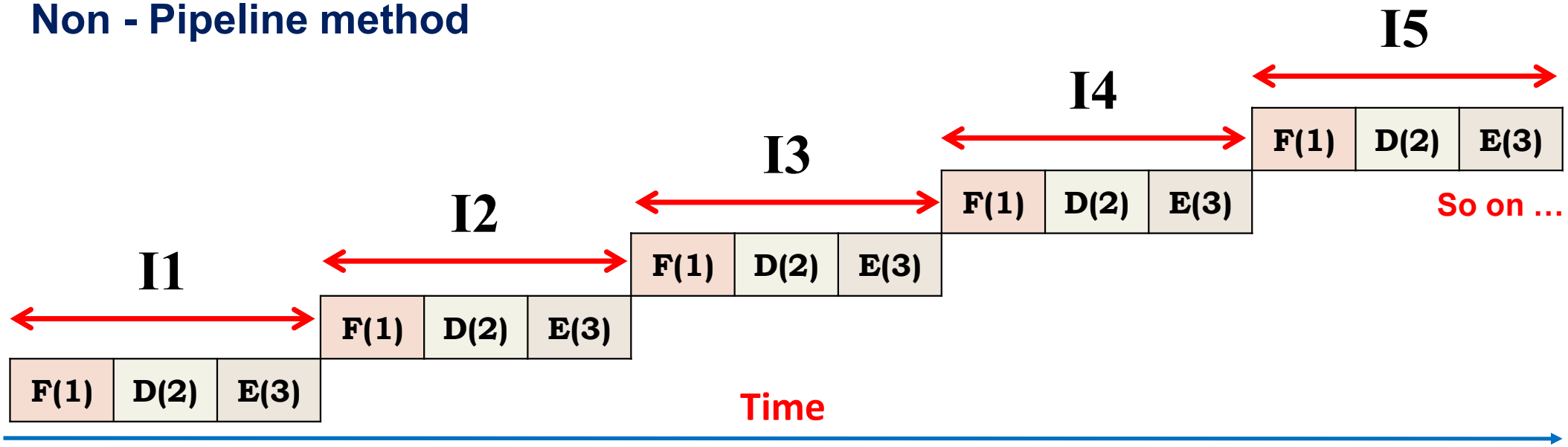
Decoding - 2 Clock cycle

Execution - 3 Clock cycle

I1
I2
I3
I4
I5
I6
I7
I8
I9
I10

MODEL-1 : PIPELINE PROBLEM ASSIGNMENT-1

Non - Pipeline method



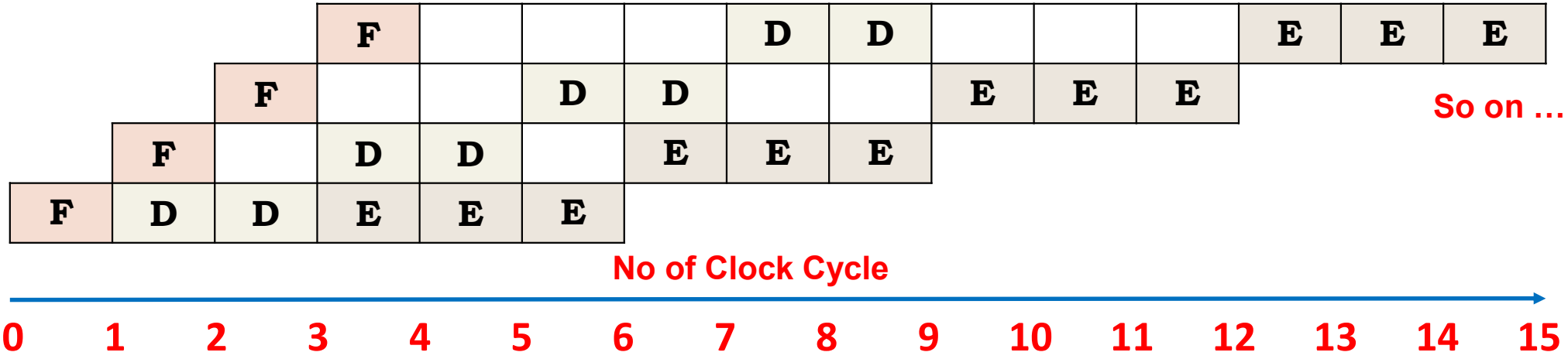
No of Clock Cycle required to execute 1000 instructions is

= (No of instructions) x (Total no of required for single instruction)

= 1000 x 6 = **6000 Clock cycles**

MODEL-1 : PIPELINE PROBLEM ASSIGNMENT-1

Pipeline method



No of Clock Cycle required to execute 100 instructions is

= (No of clocks required for 1st instruction)+ ((no of instruction -1) x (difference between two instruction))

= 6 + ((1000-1) x 3) = 6 + (999 x 3) = 6 + 2997 = **3003 Clock cycles**

MODEL-1 : PIPELINE PROBLEM-3

Problem 1: Find the **number of clock cycles** required to execute 1000 instructions with pipeline method and without pipeline method for the following instruction structure? **If microcontroller frequency is 1GHz then also find the max operating frequency**?

I	F (1)	D (1)	E (4)

Fetch - 1 Clock cycle

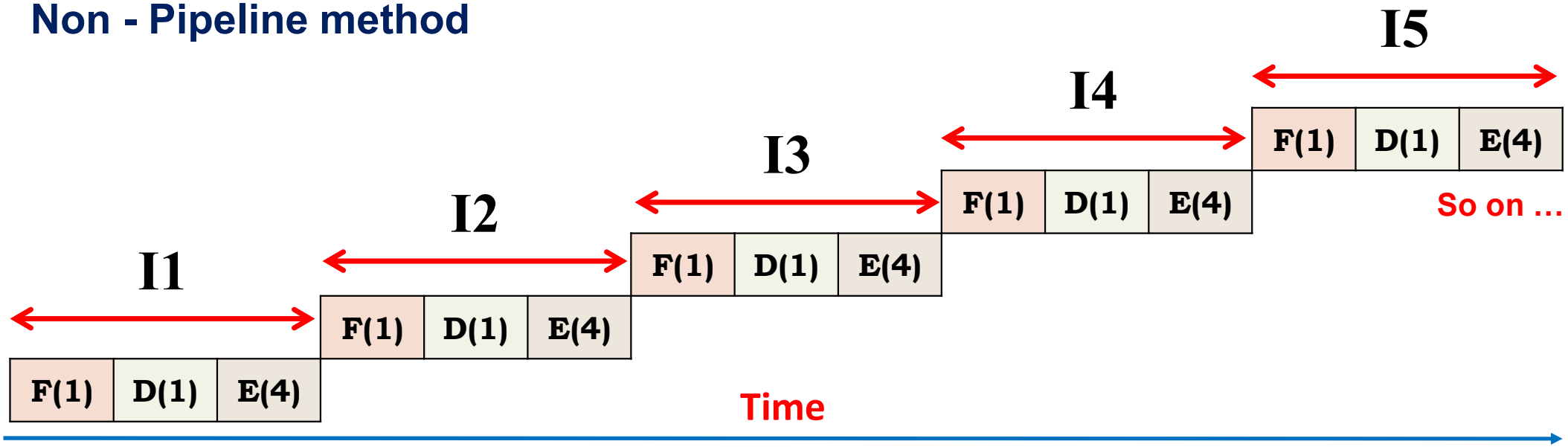
Decoding - 1 Clock cycle

Execution - 4 Clock cycle

I1
I2
I3
I4
I5
I6
I7
I8
I9
I10

MODEL-1 : PIPELINE PROBLEM-3

Non - Pipeline method



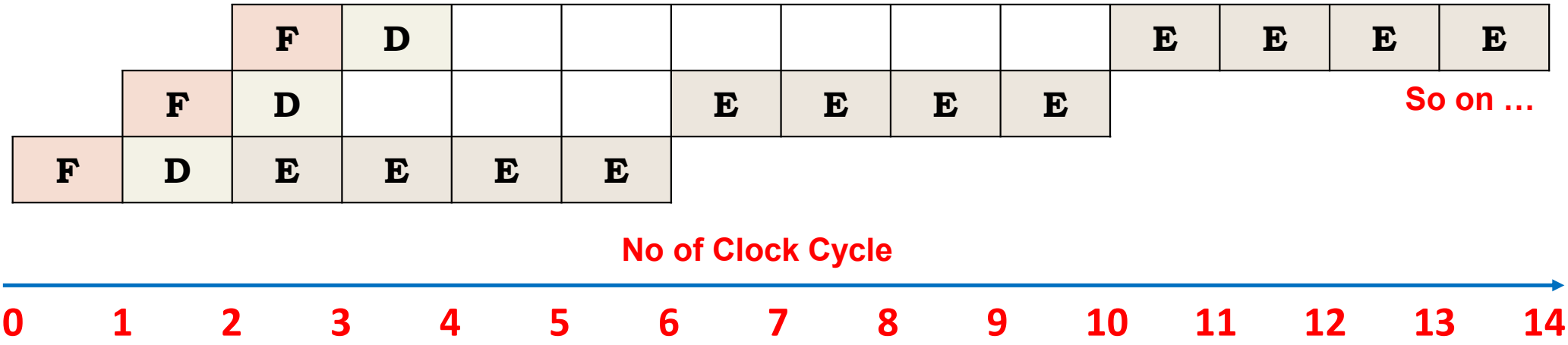
No of Clock Cycle required to execute 1000 instructions is

= (No of instructions) x (Total no of required for single instruction)

= 1000 x 6 = **6000 Clock cycles**

MODEL-1 : PIPELINE PROBLEM-3

Pipeline method



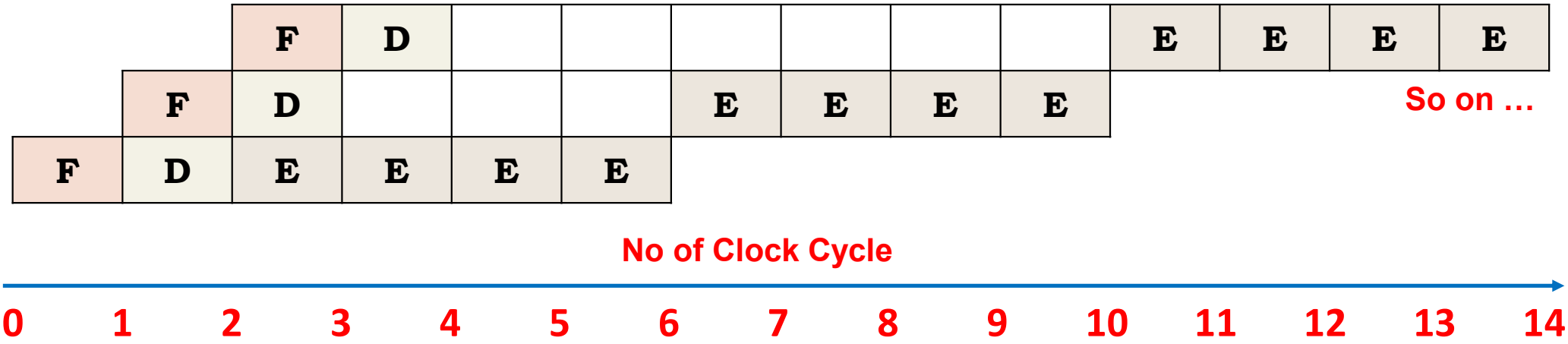
No of Clock Cycle required to execute 1000 instructions is

= (No of clocks required for 1st instruction)+ ((no of instruction -1) x (difference between two instruction))

= 6 + ((1000-1) x 4) = 6 + (999 x 4) = 6 + 3996 = **4002 Clock cycles**

MODEL-1 : PIPELINE PROBLEM-3

Pipeline method

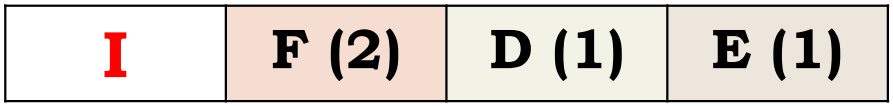


Maximum operating frequency is given by

$$f_{max.\ op} = \frac{f_{mc}}{\text{max. difference between two instructions}} = \frac{1\text{GHz}}{4} = 0.25\ \text{GHz}$$

MODEL-1 : PIPELINE PROBLEM-4

Problem 4: If microcontroller frequency is 1GHz then also find the max operating frequency?



Fetch - 2 Clock cycle

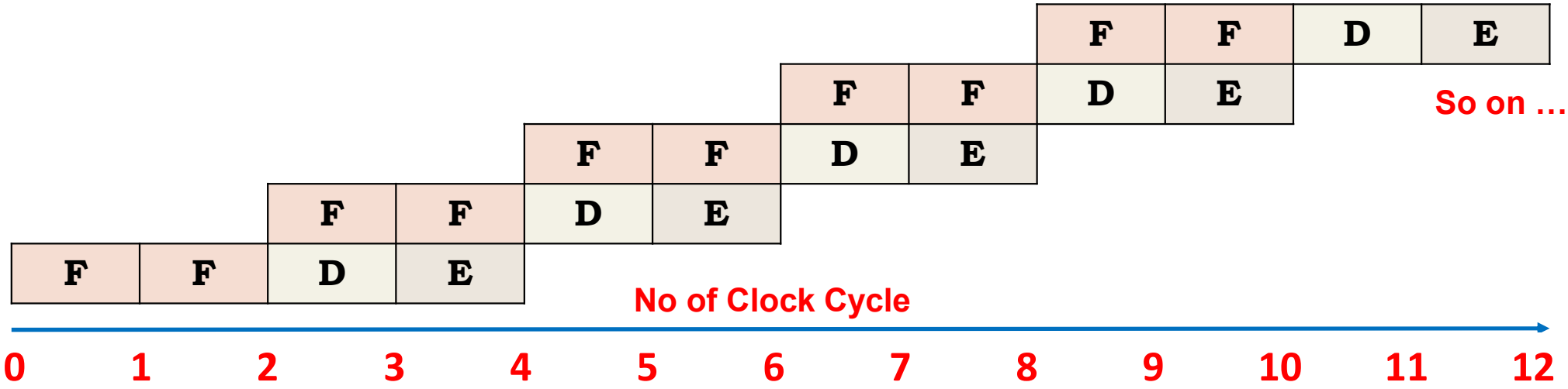
Decoding - 1 Clock cycle

Execution - 1 Clock cycle

I1
I2
I3
I4
I5
I6
I7
I8
I9
I10

MODEL-1 : PIPELINE PROBLEM-4

Pipeline method

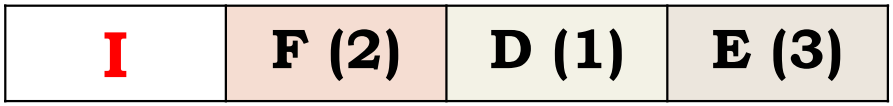


Maximum operating frequency is given by

$$f_{max.\ op} = \frac{f_{mc}}{\text{max. difference between two instructions}} = \frac{1\text{GHz}}{2} = 0.5\ \text{GHz}$$

MODEL-1 : PIPELINE PROBLEM-5

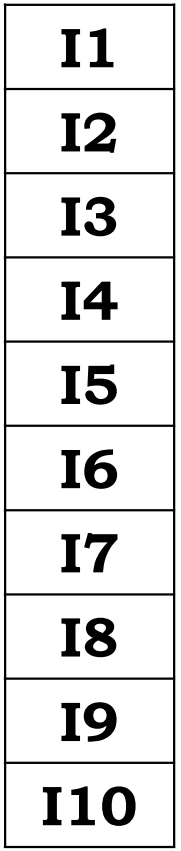
Problem 5: If microcontroller frequency is 1GHz then also find the max operating frequency ?



Fetch - 2 Clock cycle

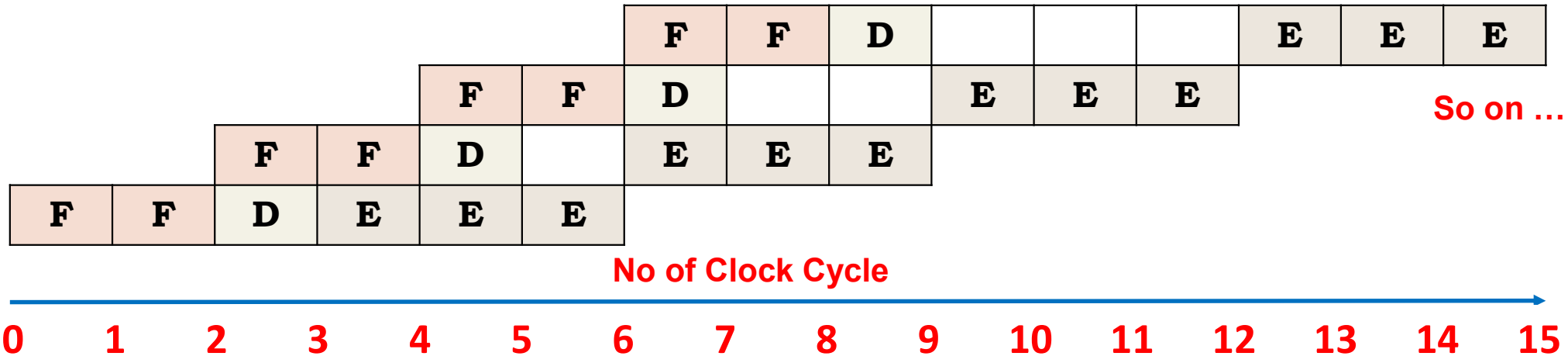
Decoding - 1 Clock cycle

Execution - 3 Clock cycle



MODEL-1 : PIPELINE PROBLEM-5

Pipeline method



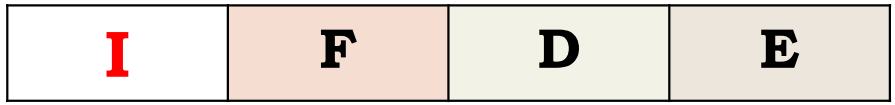
Maximum operating frequency is given by

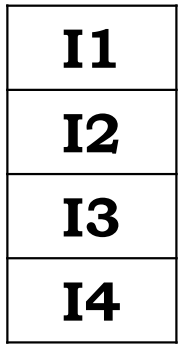
$$f_{max.\ op} = \frac{f_{mc}}{\text{max. difference between two instructions}} = \frac{1\text{GHz}}{3} = 0.33\ \text{GHz}$$

MODEL-2 PROBLEMS

MODEL-2 : PIPELINE PROBLEM-1

Problem 1: Find the **number of clock cycles** required to execute 4 instructions with pipeline method and without pipeline method for the following instruction structure? If microcontroller frequency is 1GHz then also find the max operating frequency?





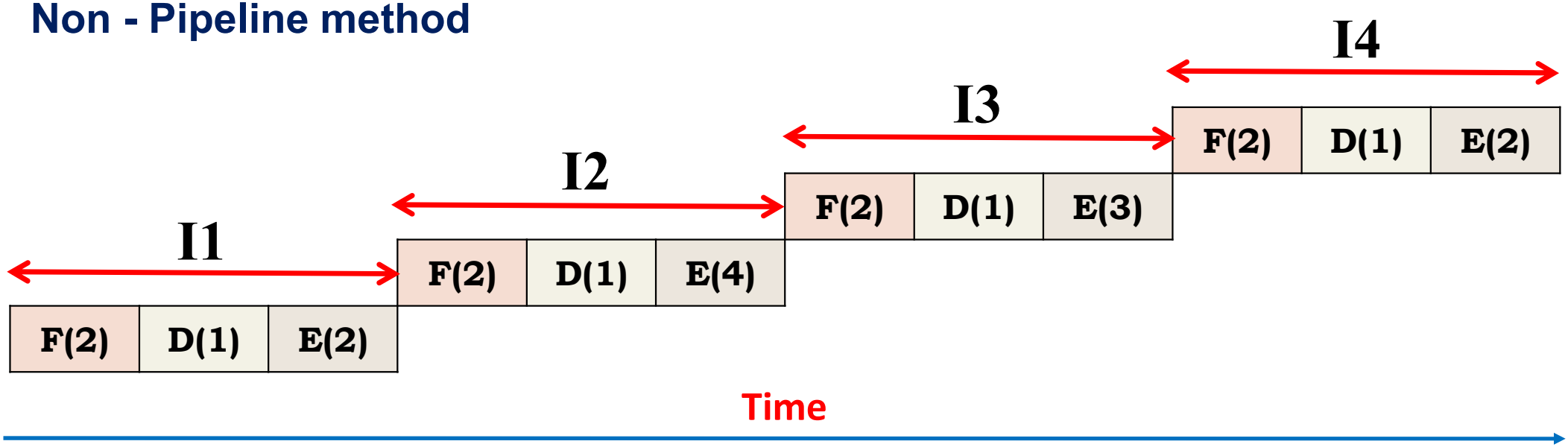
Fetch – 2 Clock cycle

Decoding – 1 Clock cycle

Execution – 2 (I1), 4 (I2), 3 (I3) and 2 (I4) Clock cycles

MODEL-2 : PIPELINE PROBLEM-1

Non - Pipeline method

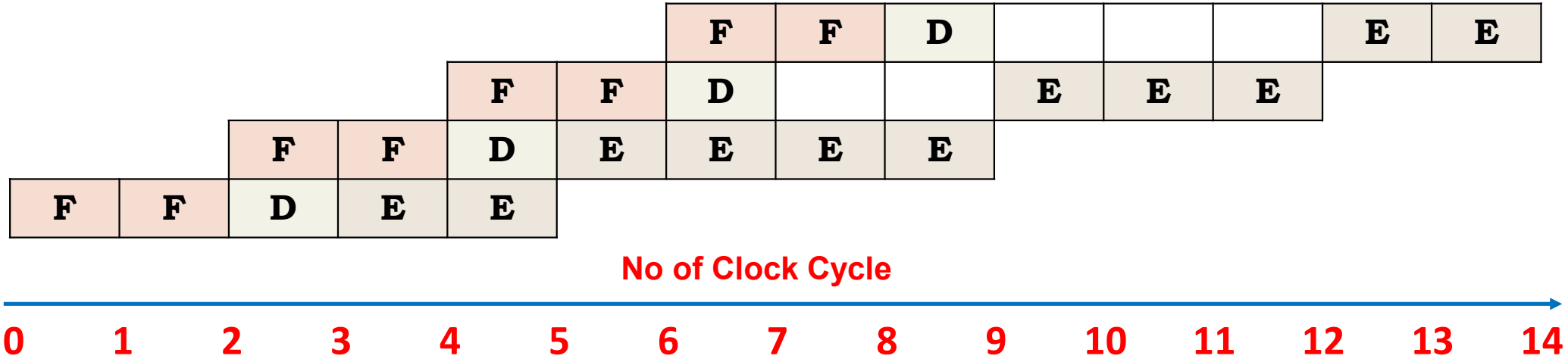


No of Clock Cycle required to execute 4 instructions is

= 5 + 7 + 6 + 5 = **23 Clock cycles**

MODEL-2 : PIPELINE PROBLEM-1

Pipeline method

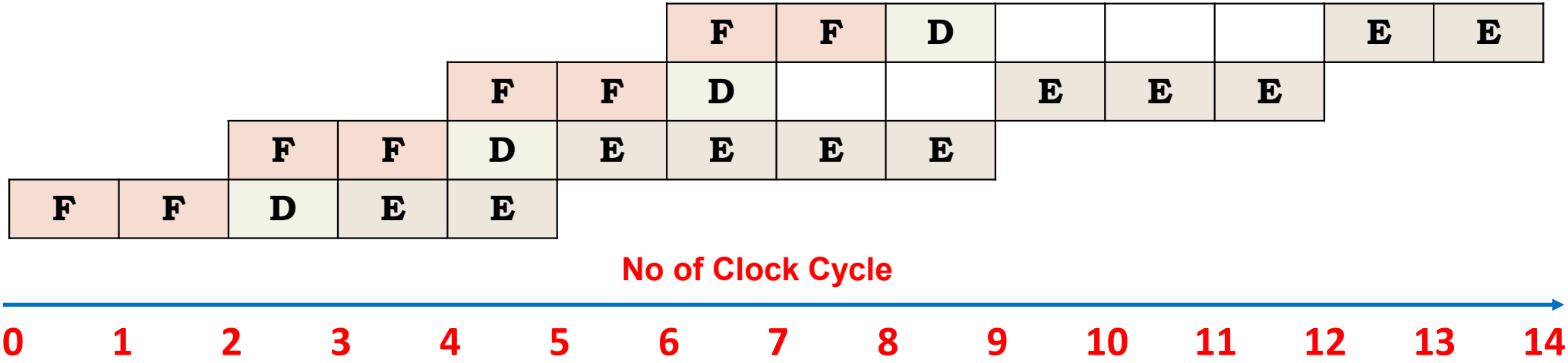


No of Clock Cycle required to execute 4 instructions is

= **14 Clock Cycles** (From diagram)

MODEL-2 : PIPELINE PROBLEM-1

Pipeline method

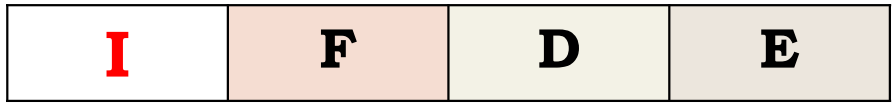


Maximum operating frequency is given by

$$f_{max.\ op} = \frac{f_{mc}}{\text{max. difference between two instructions}} = \frac{1\text{GHz}}{4} = 0.25\ \text{GHz}$$

MODEL-2 : PIPELINE PROBLEM-2

Problem 2: Find the **number of clock cycles** required to execute 4 instructions with pipeline method and without pipeline method for the following instruction structure? If microcontroller frequency is 1GHz then also find the max operating frequency?



I1
I2
I3
I4

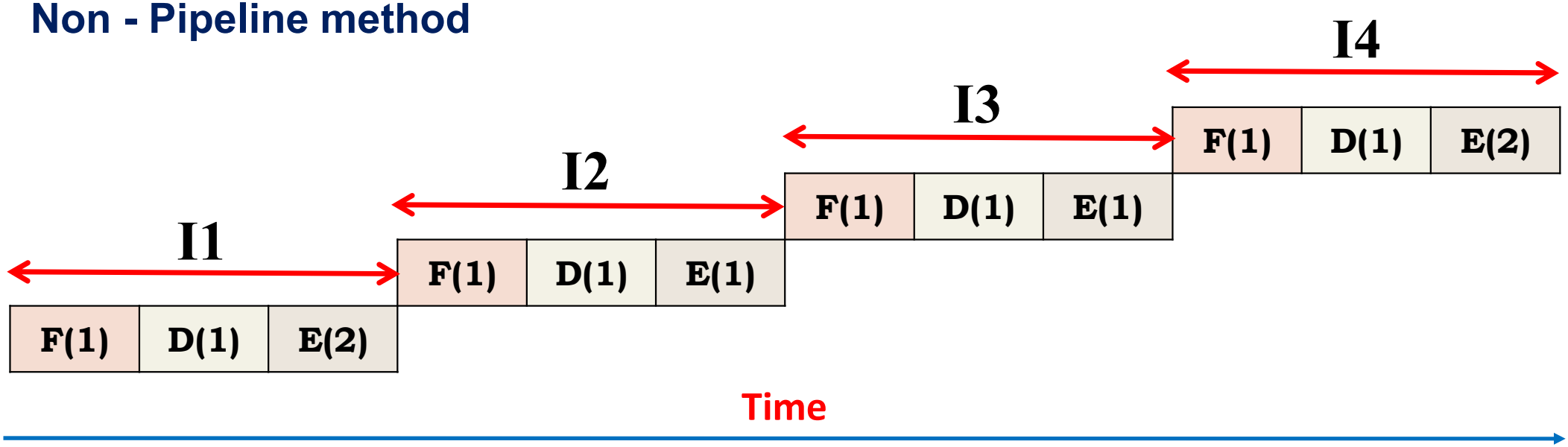
Fetch – 1 Clock cycle

Decoding – 1 Clock cycle

Execution – 2 (I1), 1 (I2), 1 (I3) and 2 (I4) Clock cycles

MODEL-2 : PIPELINE PROBLEM-2

Non - Pipeline method

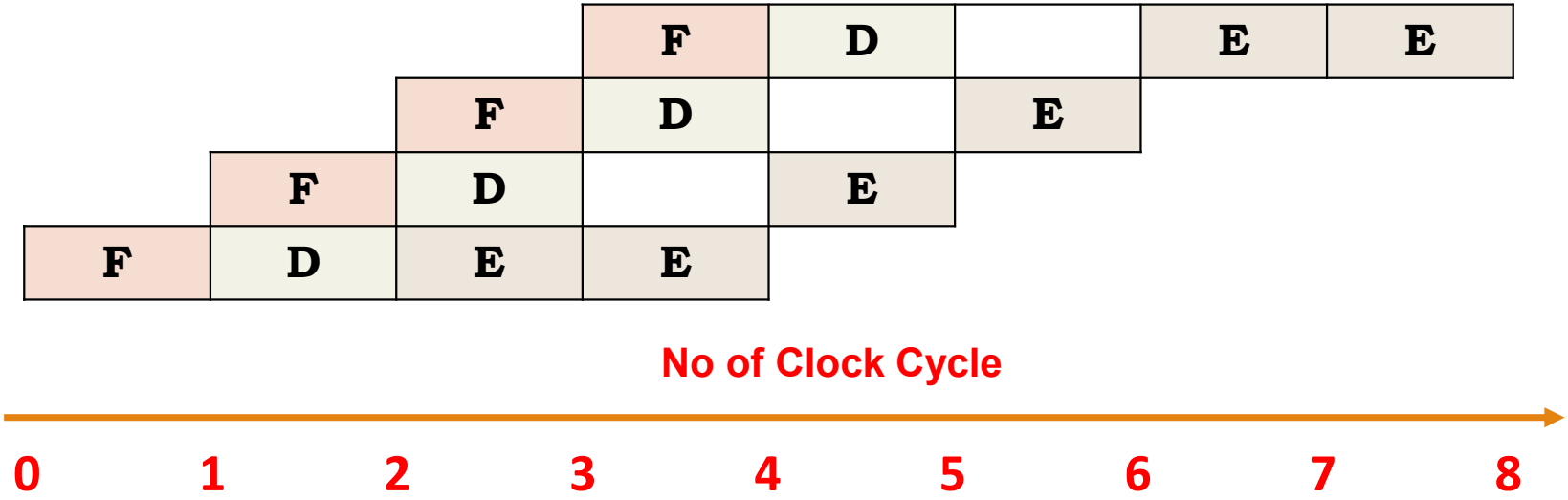


No of Clock Cycle required to execute 4 instructions is

= 4 + 3 + 3 + 4 = **14 Clock cycles**

MODEL-2 : PIPELINE PROBLEM-2

Pipeline method

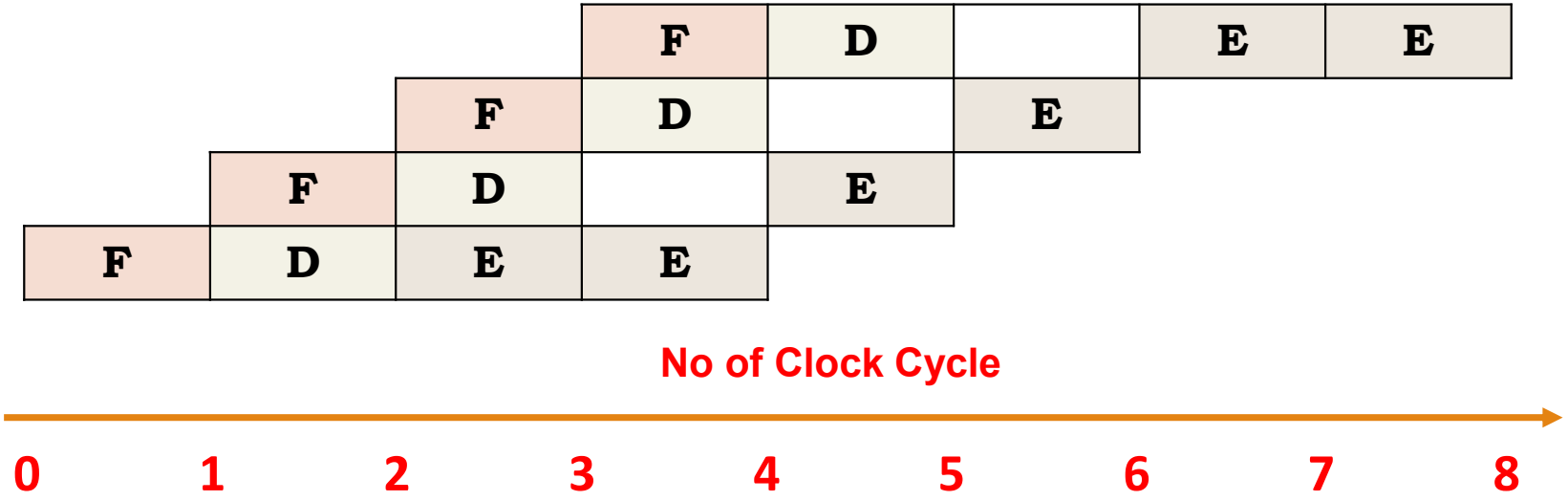


No of Clock Cycle required to execute 4 instructions is

= **8 Clock Cycles** (From diagram)

MODEL-2 : PIPELINE PROBLEM-2

Pipeline method

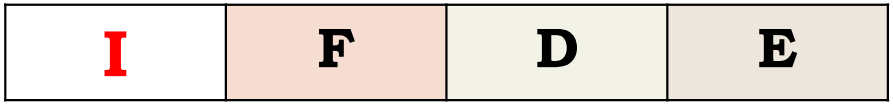


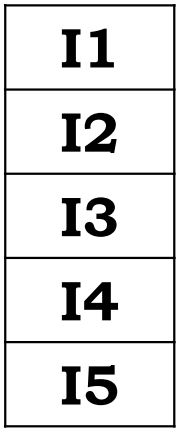
Maximum operating frequency is given by

$$f_{max.\ op} = \frac{f_{mc}}{\text{max. difference between two instructions}} = \frac{1\text{GHz}}{2} = 0.5\ \text{GHz}$$

MODEL-2 : PIPELINE PROBLEM ASSIGNMENT-1

Assignment - 1: Find the **number of clock cycles** required to execute 5 instructions with pipeline method and without pipeline method for the following instruction structure? If microcontroller frequency is 2 GHz then also find the max operating frequency?





Fetch – 1 Clock cycle

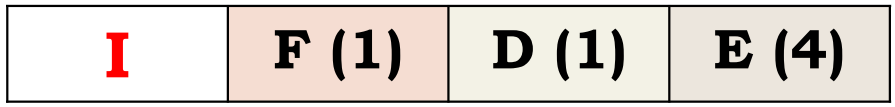
Decoding – 1 Clock cycle

Execution – 3 (I1), 4 (I2), 2 (I3), 1 (I4) and 2 (I5) Clock cycles

MODEL-3 PROBLEMS

MODEL-3 : PIPELINE PROBLEM-1

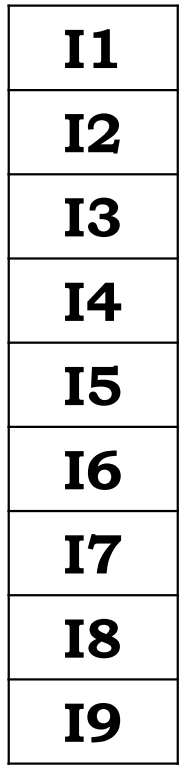
Problem - 1: If the pipeline is flushed for every 3 instructions then find the number of clock cycles required to execute 9 instructions with pipeline method?



Fetch - 1 Clock cycle

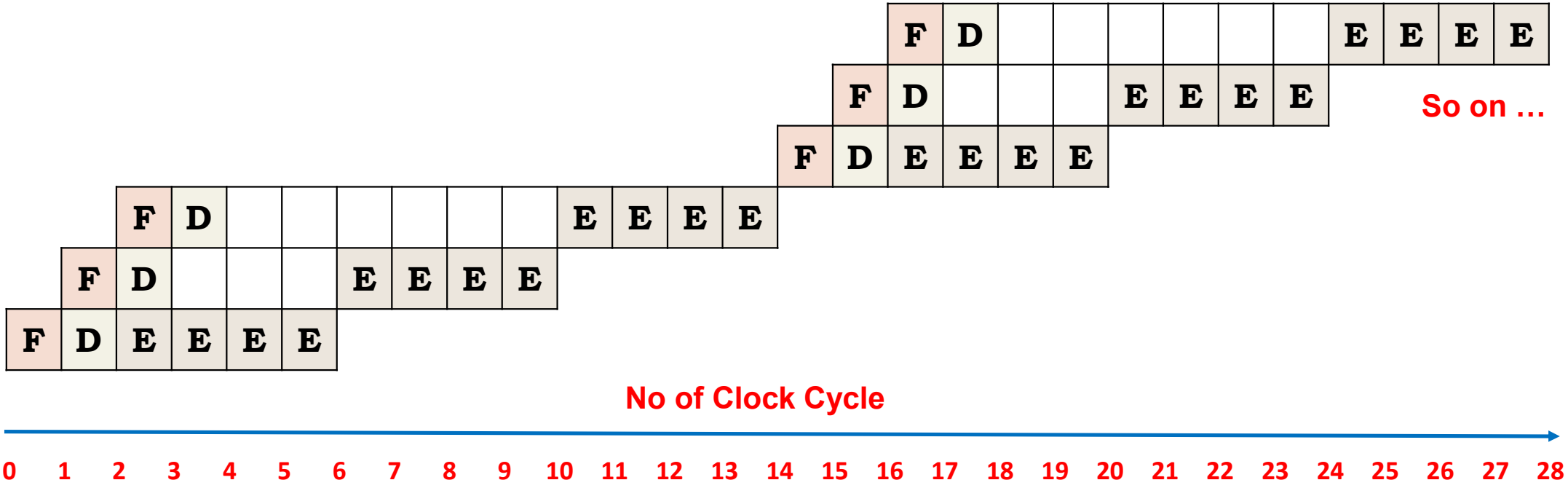
Decoding - 1 Clock cycle

Execution - 4 Clock cycle



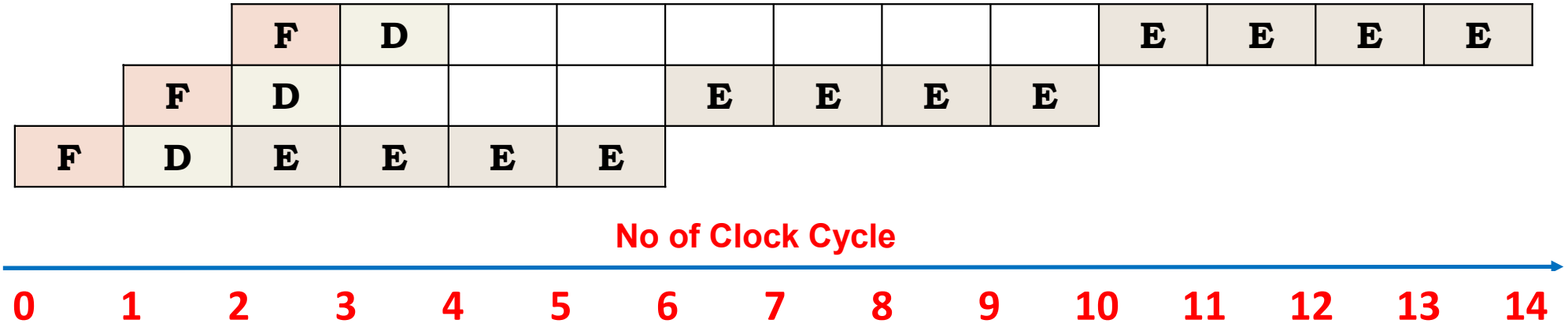
MODEL-3 : PIPELINE PROBLEM-1

Pipeline method



MODEL-3 : PIPELINE PROBLEM-1

Pipeline method



No of Clock Cycle required to execute 3 instructions is

= (No of clocks required for 1st instruction)+ ((no of instruction -1) x (difference between two instruction))

= 6 + ((3-1) x 4) = 6 + (2 x 4) = 6 + 8 = **14 Clock cycles**

Total = 14 + 14 + 14 = 42 Clock cycles

MODEL-3 : PIPELINE PROBLEM-2

Problem - 2: If the pipeline is flushed for every 10 instructions then find the number of clock cycles required to execute 40 instructions with pipeline method?

I	F (1)	D (1)	E (4)

Fetch - 1 Clock cycle

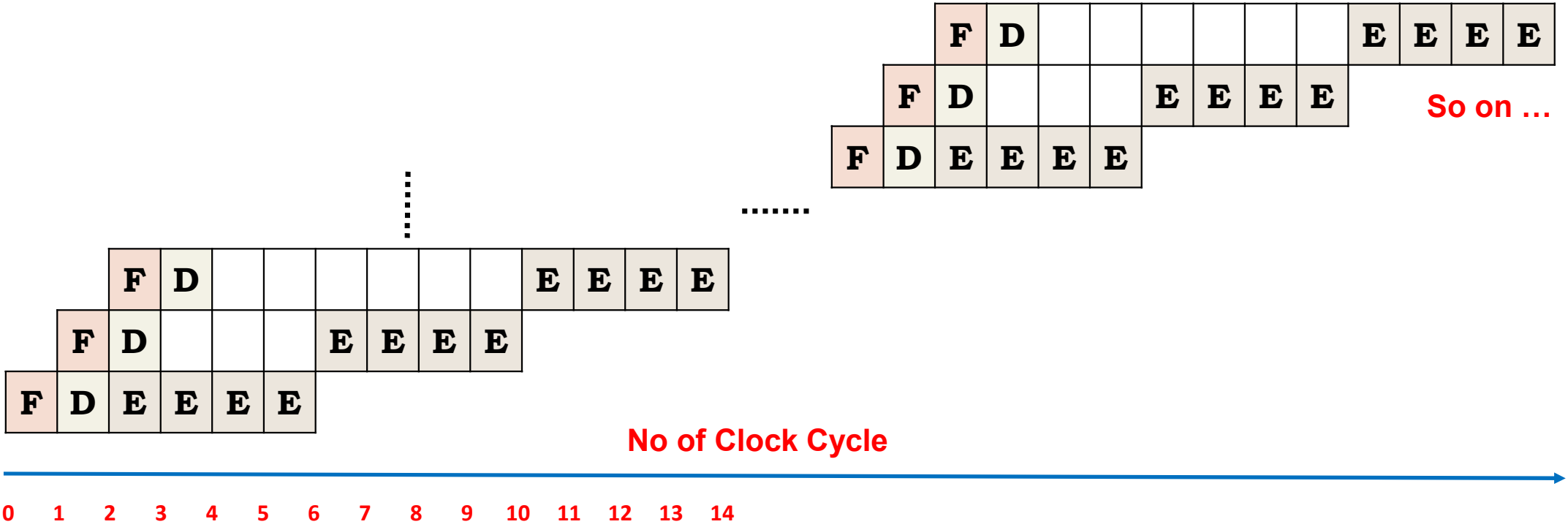
Decoding - 1 Clock cycle

Execution - 4 Clock cycle

I1
I2
I3
I4
I5
I6
I7
I8
I9
I10

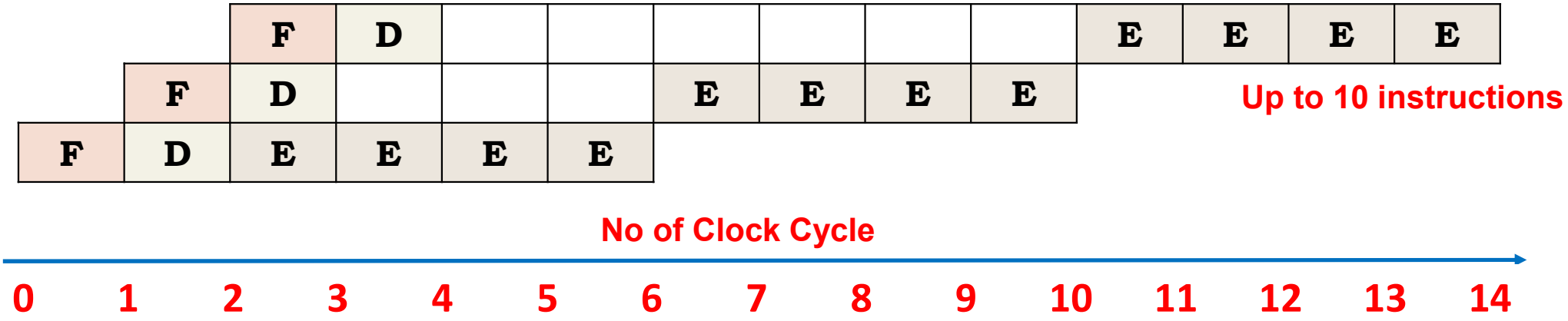
MODEL-3 : PIPELINE PROBLEM-2

Pipeline method



MODEL-3 : PIPELINE PROBLEM-2

Pipeline method



No of Clock Cycle required to execute 10 instructions is

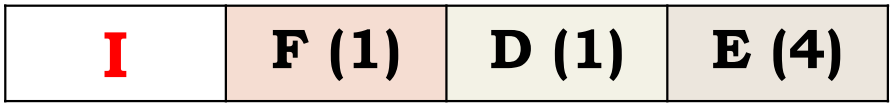
= (No of clocks required for 1st instruction)+ ((no of instruction -1) x (difference between two instruction))

= 6 + ((10-1) x 4) = 6 + (9 x 4) = 6 + 36 = **42 Clock cycles**

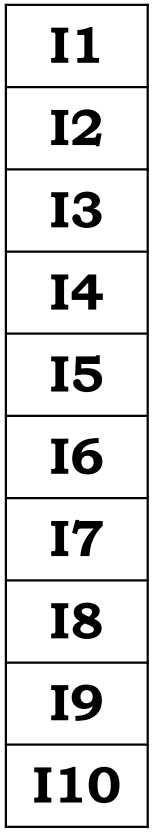
Total = 42 + 42 + 42 + 42 = 168 Clock cycles

MODEL-3 : PIPELINE PROBLEM-3

Problem - 3: If the pipeline is flushed for every 10 instructions then find the number of clock cycles required to execute 41 instructions with pipeline method?

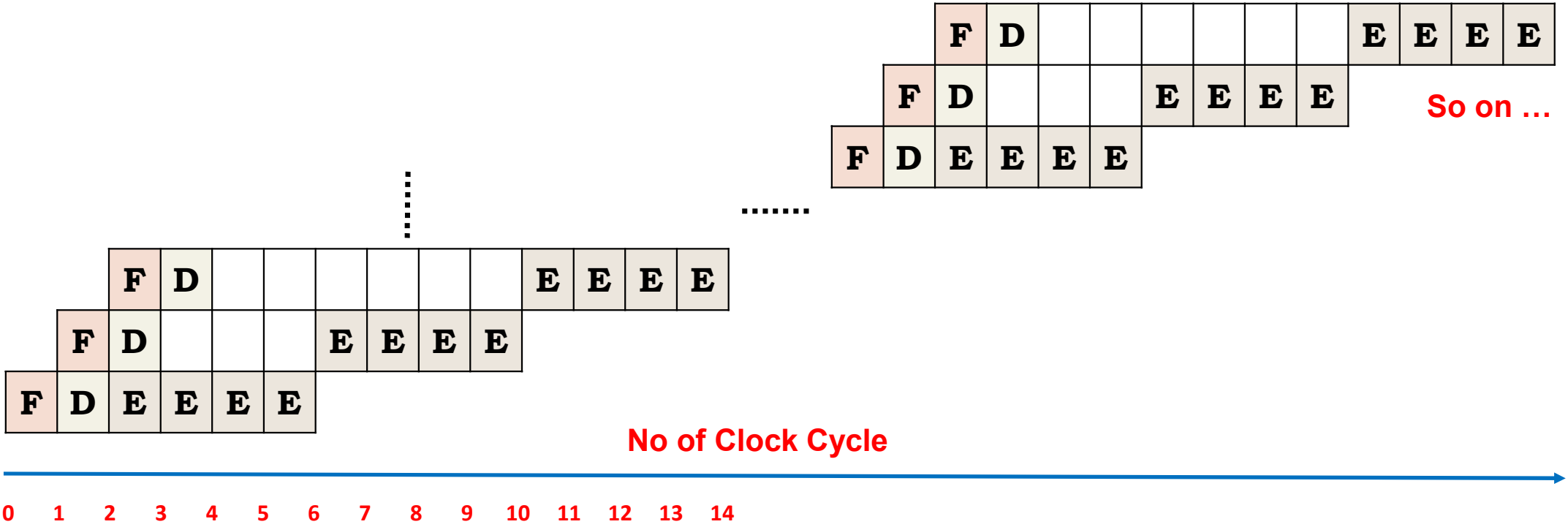


Fetch	- 1 Clock cycle
Decoding	- 1 Clock cycle
Execution	- 4 Clock cycle



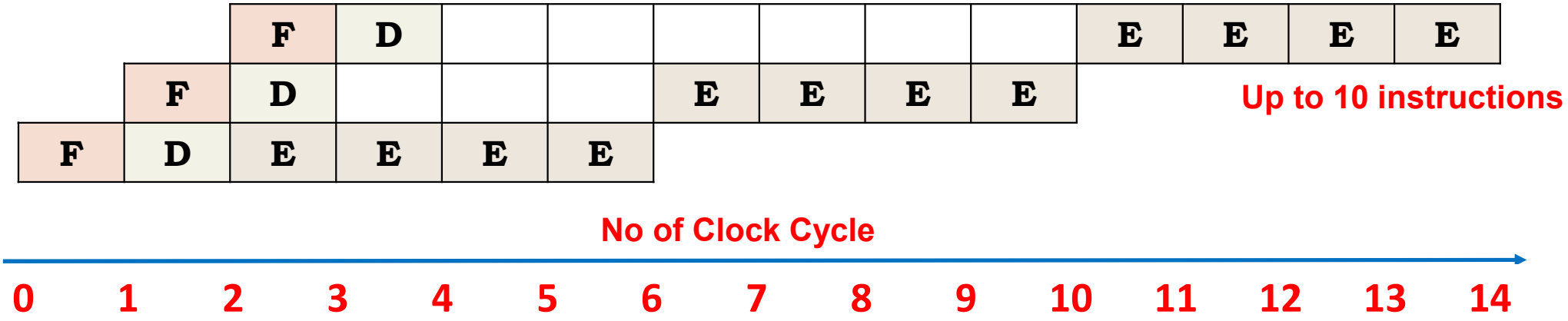
MODEL-3 : PIPELINE PROBLEM-3

Pipeline method



MODEL-3 : PIPELINE PROBLEM-3

Pipeline method



No of Clock Cycle required to execute 10 instructions is

= (No of clocks required for 1st instruction)+ ((no of instruction -1) x (difference between two instruction))

= 6 + ((10-1) x 4) = 6 + (9 x 4) = 6 + 36 = **42 Clock cycles**

Total = 4 x 42 + 6 = 168 + 6 = 174 Clock cycles

MODEL-3 : PIPELINE PROBLEM-4

Problem - 4: If the pipeline is flushed for every 10 instructions then find the number of clock cycles required to execute 141 instructions with pipeline method?

I	F (2)	D (1)	E (3)

Fetch - 2 Clock cycle

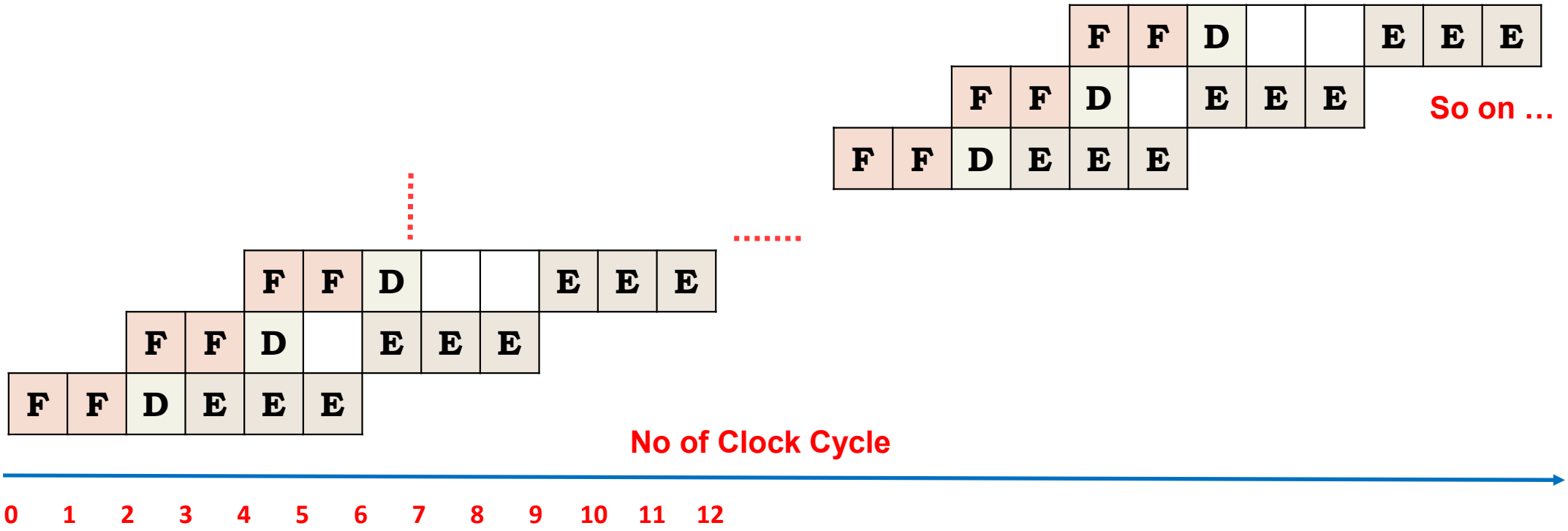
Decoding - 1 Clock cycle

Execution - 3 Clock cycle

I1
I2
I3
I4
I5
I6
I7
I8
I9
I10

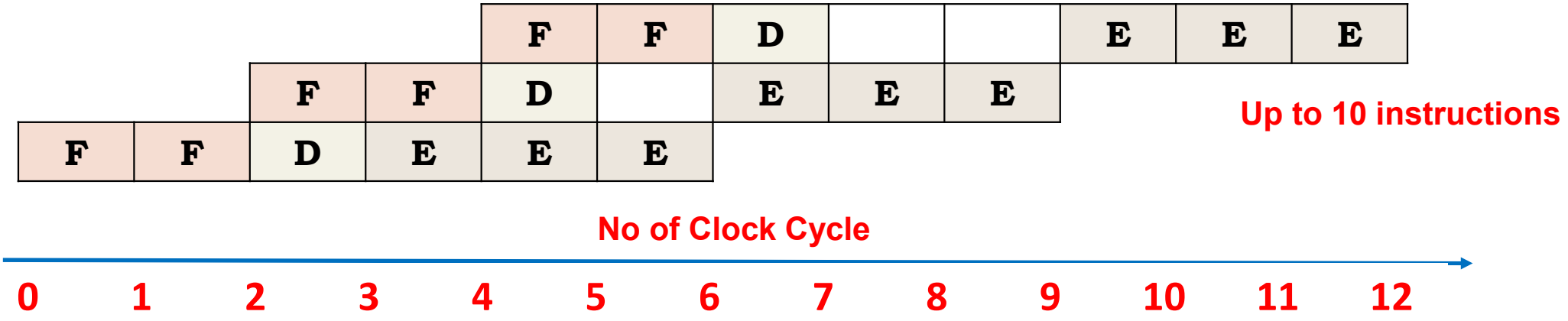
MODEL-3 : PIPELINE PROBLEM-4

Pipeline method



MODEL-3 : PIPELINE PROBLEM-4

Pipeline method



No of Clock Cycle required to execute 10 instructions is

= (No of clocks required for 1st instruction)+ ((no of instruction -1) x (difference between two instruction))

= 6 + ((10-1) x 3) = 6 + (9 x 3) = 6 + 27 = **33 Clock cycles**

No of Clock Cycle required to execute 141 instructions = 14 x 33 + 6 = 462 + 6 = **468 Clock cycles**

MODEL-3 : PIPELINE PROBLEM ASSIGNMENT-1

Problem - 1: If the pipeline is flushed for every 15 instructions then find the number of clock cycles required to execute 1501 instructions with pipeline method?

I	F (2)	D (1)	E (4)

Fetch - 2 Clock cycle

Decoding - 1 Clock cycle

Execution - 4 Clock cycle

I1
I2
I3
I4
I5
I6
I7
I8
I9
I10

MODEL-4 PROBLEMS

MODEL-4 : PIPELINE PROBLEM-1

Problem 1: Find the **number of clock cycles** required to execute 10 instructions with pipeline method and without pipeline method for the following instruction structure? **Improve the pipeline structure**.

I	F (1)	D (1)	E (4)

Fetch - 1 Clock cycle

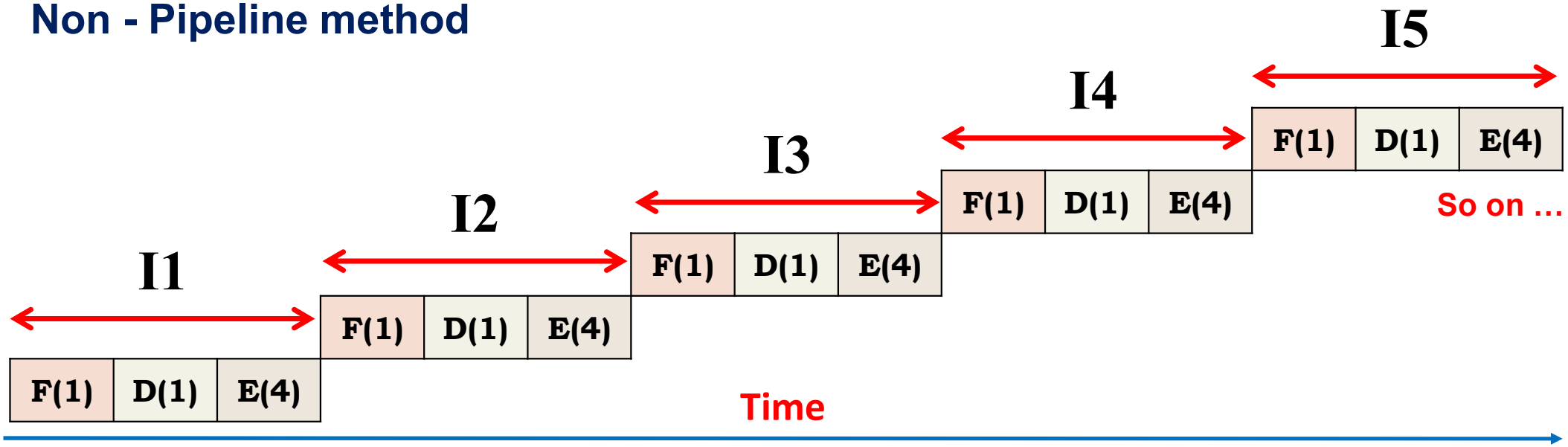
Decoding - 1 Clock cycle

Execution - 4 Clock cycle

I1
I2
I3
I4
I5
I6
I7
I8
I9
I10

MODEL-4 : PIPELINE PROBLEM-1

Non - Pipeline method



No of Clock Cycle required to execute 10 instructions is

= (No of instructions) x (Total no of required for single instruction)

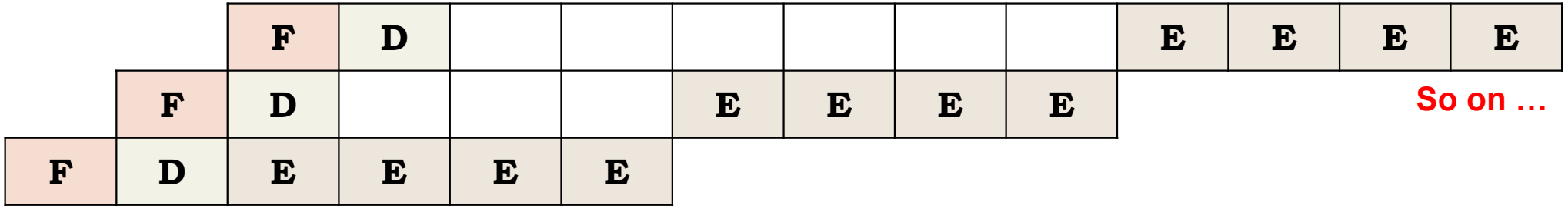
= 10 x 6 = **60 Clock cycles**

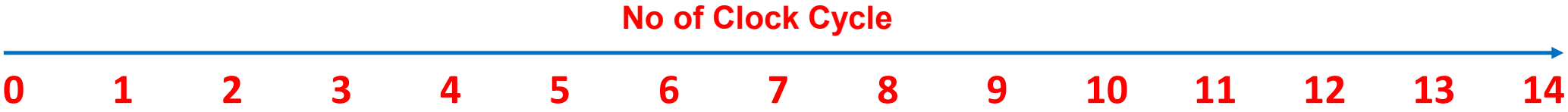
MODEL-4 : PIPELINE PROBLEM-1

Less efficient design of pipeline

Pipeline method

I	F (1)	D (1)	E (4)





No of Clock Cycle required to execute 10 instructions is

= (No of clocks required for 1st instruction)+ ((no of instruction -1) x (difference between two instruction))

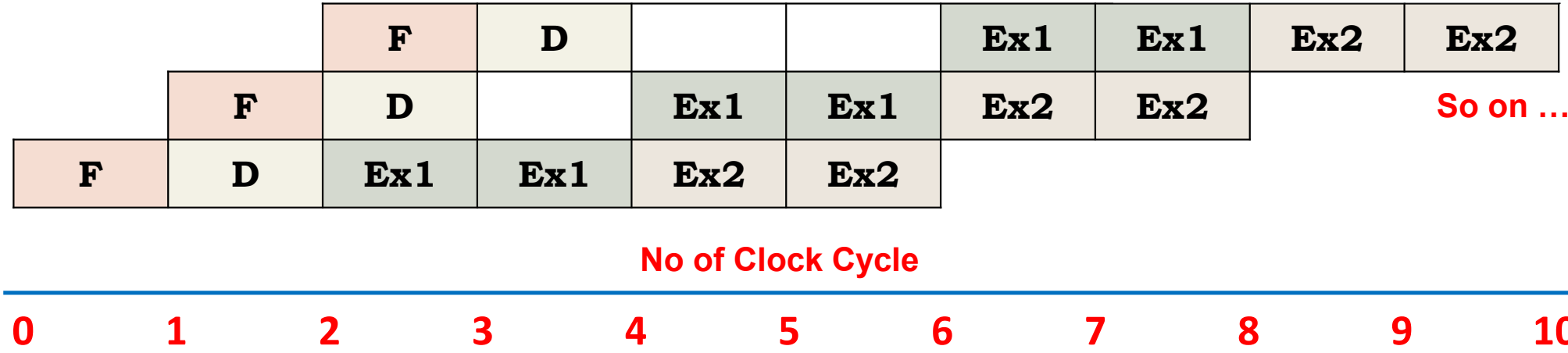
= 6 + ((10 - 1) x 4) = 6 + (9 x 4) = 6 + 36 = **42 Clock cycles**

MODEL-4 : PIPELINE PROBLEM-1

Pipeline method

I	F(1)	D(1)	Ex1 (2)	Ex2 (2)

More efficient design of pipeline: Break the execution in two parts of two cycle each



No of Clock Cycle required to execute 10 instructions is

= (No of clocks required for 1st instruction)+ ((no of instruction -1) x (difference between two instruction))

= 6 + ((10 - 1) x 2) = 6 + (9 x 2) = 6 + 18 = **24 Clock cycles**

MODEL-4 : PIPELINE PROBLEM-2

Problem 2: Find the **number of clock cycles** required to execute 100 instructions with pipeline method and without pipeline method for the following instruction structure? **Improve the pipeline structure**.

I	F (2)	D (1)	E (6)

Fetch - 2 Clock cycle

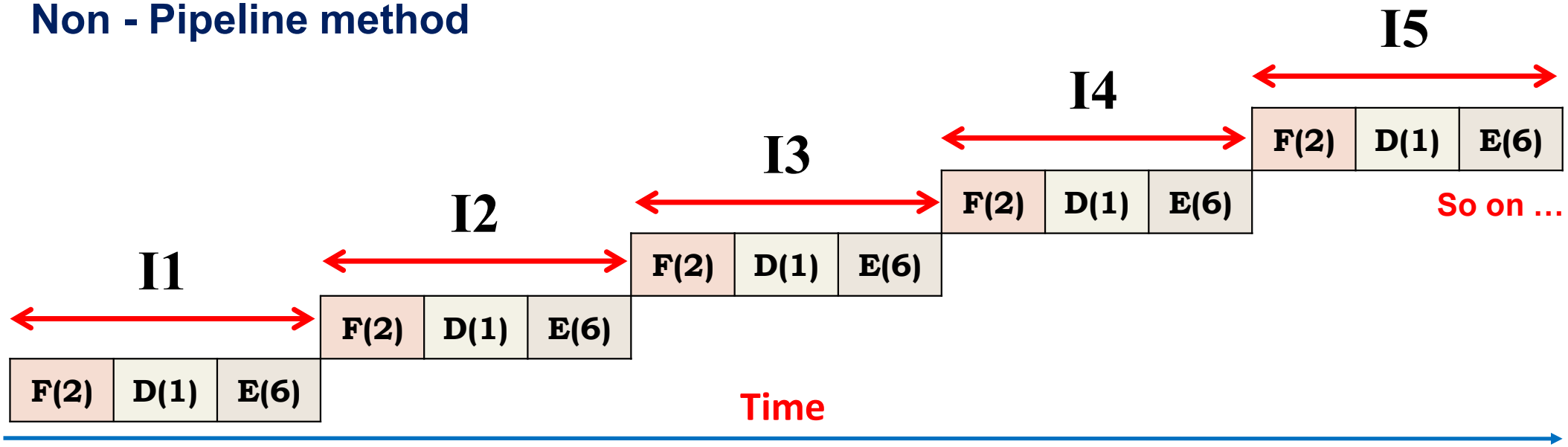
Decoding - 1 Clock cycle

Execution - 6 Clock cycle

I1
I2
I3
I4
I5
I6
I7
I8
I9
I10

MODEL-4 : PIPELINE PROBLEM-2

Non - Pipeline method



No of Clock Cycle required to execute 100 instructions is

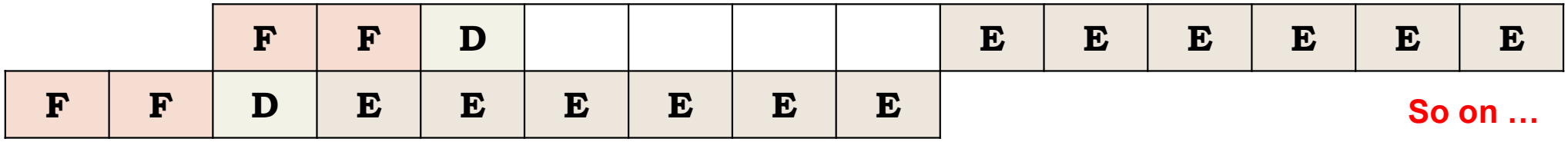
= (No of instructions) x (Total no of required for single instruction)

= 100 x 9 = **900 Clock cycles**

MODEL-4 : PIPELINE PROBLEM-2

Pipeline method

I	F (2)	D (1)	E (6)



No of Clock Cycle required to execute 100 instructions is

= (No of clocks required for 1st instruction)+ ((no of instruction -1) x (difference between two instruction))

= 9 + ((100 - 1) x 6) = 9 + (99 x 6) = 9 + 594 = **603 Clock cycles**

MODEL-4 : PIPELINE PROBLEM-2

Pipeline method

I	F(2)	D(1)	Ex1 (3)	Ex2 (3)

		F	F	D		Ex1	Ex1	Ex1	Ex2	Ex2	Ex2
F	F	D	Ex1	Ex1	Ex1	Ex2	Ex2	Ex2			

So on ...

No of Clock Cycle

0 1 2 3 4 5 6 7 8 9 10 11 12

No of Clock Cycle required to execute 100 instructions is

= (No of clocks required for 1st instruction)+ ((no of instruction -1) x (difference between two instruction))

= 9 + ((100 - 1) x 3) = 9 + (99 x 3) = 9 + 297 = **306 Clock cycles**

MODEL-4 : PIPELINE PROBLEM ASSIGNMENT-1

Problem - 1: Find the **number of clock cycles** required to execute 5432 instructions with pipeline method and without pipeline method for the following instruction structure ? **Improve the pipeline structure**.

I	F (2)	D (1)	E (8)

Fetch - 2 Clock cycle

Decoding - 1 Clock cycle

Execution - 8 Clock cycle

I1
I2
I3
I4
I5
I6
I7
I8
I9
I10

MODEL-5 PROBLEMS

Problem-:

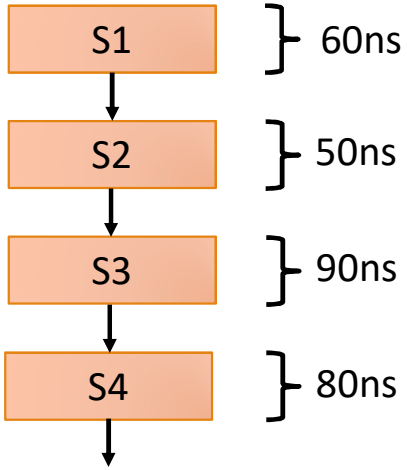
Consider a pipeline having 4 phases with duration 60, 50, 90 and 80 ns. Given latch delay is 10 ns. Calculate-

1. Pipeline cycle time
2. Non-pipeline execution time
3. Speed up ratio
4. Pipeline time for 1000 instructions
5. Sequential time for 1000 instructions
6. Throughput

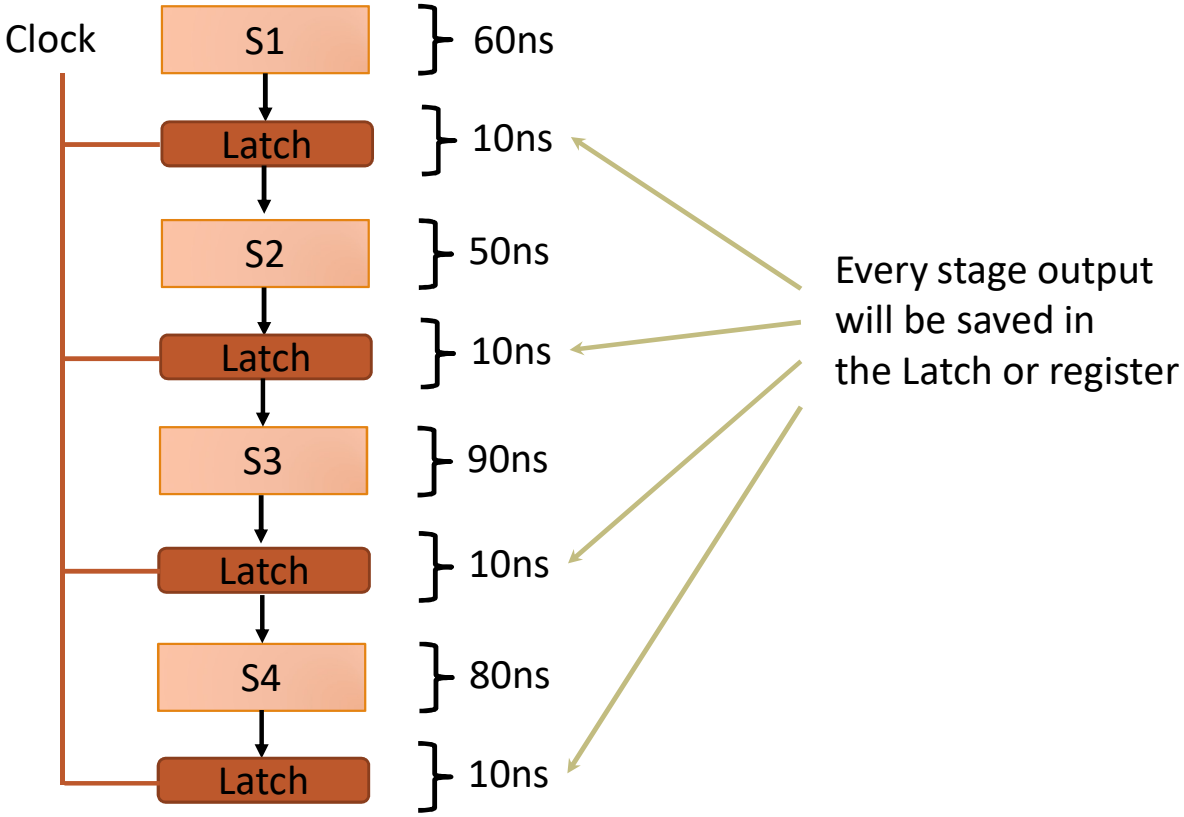
Solution:

Given-

- Four stage pipeline is used
- Delay of stages = 60, 50, 90 and 80 ns
- Latch delay or delay due to each register = 10 ns



Non-Pipelined Architecture



Pipelined Architecture

Note: In any stage of pipeline, the output of each stage will be moved to the next state after the 100 ns (max(60,50,90,80) + 10 ns)

Part-01: Pipeline Cycle Time-

Cycle time = Maximum delay due to any stage + Delay due to its register (Latch)
= Max { 60, 50, 90, 80 } + 10 ns
= 90 ns + 10 ns
= 100 ns

Part-02: Non-Pipeline Execution Time-

Non-pipeline execution time for one instruction = 60 ns + 50 ns + 90 ns + 80 ns
= 280 ns

Part-03: Speed Up Ratio-

Speed up = Non-pipeline execution time / Pipeline execution time
= 280 ns / Cycle time
= 280 ns / 100 ns
= 2.8

Part-04: Pipeline Time For 1000 Instructions-

Pipeline time for 1000 instructions

= Time taken for 1st instruction + Time taken for remaining 999 instructions

= 1 x 4 clock cycles + 999 x 1 clock cycle

= 4 x cycle time + 999 x cycle time

= 4 x 100 ns + 999 x 100 ns

= 400 ns + 99900 ns

= 100300 ns

Part-05: Sequential Time For 1000 Instructions-

Non-pipeline time for 1000 tasks

= 1000 x Time taken for one instruction

= 1000 x 280 ns

= 280000 ns

Part-06: Throughput-

Throughput for pipelined execution = Number of instructions executed per unit time

= 1000 instructions / 100300 ns

Drawbacks/ Hazards of Pipelining

There are various hazards of pipelining, which **cause a dip** in the performance of the processor. These hazards become even **more prominent** as the **number of pipeline stages increase**. They may occur due to the following reasons.

1) Data Hazard/ Data Dependency Hazard

Data Hazard is caused when **the result (destination) of one instruction becomes the operand (source) of the next instruction**.

Consider two instructions I1 and I2 (I1 being the first).

Assume I1: INC [4000H]
Assume I2: MOV BL , [4000H]

Clearly in I2, BL should get the incremented value of location [4000H].
But this can only happen once I1 has completely finished execution and also written back the result at [4000H].
In a multistage pipeline, I2 may reach execution stage before I1 has finished storing the result at location [4000H], and hence get a wrong value of data.
This is called **data dependency hazard**.
It is solved by inserting NOP (No operation) instructions between such data dependent instructions.

2) Control Hazard/ Code Hazard

Pipelining assumes that the program will always flow in a sequential manner.
Hence, it performs various stages of the forthcoming instructions before-hand, while the current instruction is still being executed.
While programs are sequential most of the times, it is not true always.
Sometimes, branches do occur in programs.
In such an event, all the forthcoming instructions that have been fetched/ decoded etc have to be flushed/ discarded, and the process has to start all over again, from the branch address. This causes pipeline bubbles, which simply means time of the processor is wasted.

Consider the following set of instructions:

```
Start:
        JMP Down
        INC BL
        MOV CL, DL
        ADD AL, BL
        …
        …
        …

Down:   DEC CH
```

JMP Down is a branch instruction.
After this instruction, program should jump to the location "Down" and continue with DEC CH instruction.

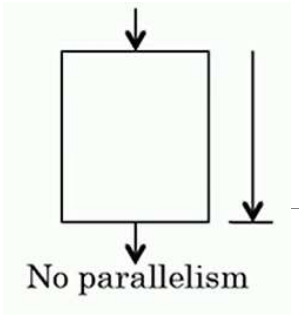
But, in a multistage pipeline processor, the sequentially next instructions after JMP Down have already been fetched and decoded. These instructions will now have to be discarded and fetching will begin all over again from DEC CH. This will keep several units of the architecture idle for some time. This is called a pipeline bubble.

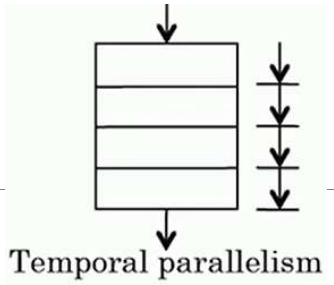
The **problem of branching is solved** in higher processors by a method called "**Branch Prediction Algorithm**". It was introduced by **Pentium** processor. It relies on the **previous history** of the instruction as most programs are repetitive in nature. It then **makes a prediction** whether branch will be **taken or not** and hence puts the correct instructions in the pipelines.

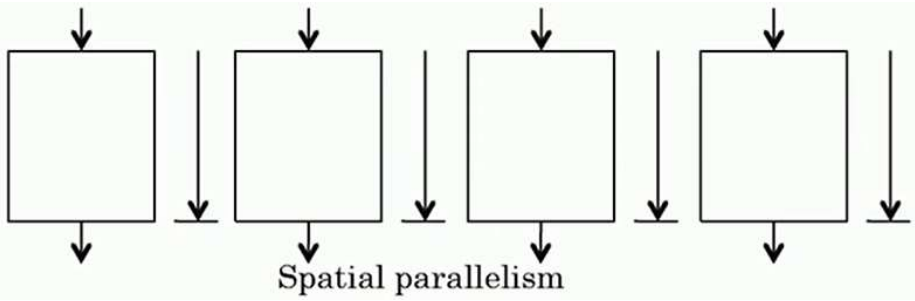
3) STRUCTURAL HAZARD

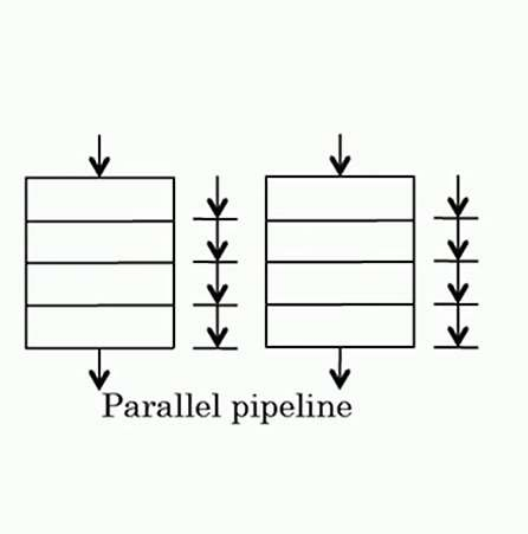
Structural hazards are caused by **physical constraints in the architecture like the buses**. Even in the most basic form of pipelining, we want to execute one instruction and fetch the next one. Now as long as execution only involves registers, pipelining is possible. But **if execution requires to read/ write data from the memory, then it will make use of the buses, which means fetching cannot take place at the same time**. So the fetching unit will have to wait and hence a pipeline bubble is caused. This problem is solved in complex Harvard architecture processors, which use separate memories and separate buses for programs and data. This means fetching and execution can actually happen at the same time without any interference with each other.
E.g.: PIC 18 Microcontroller.

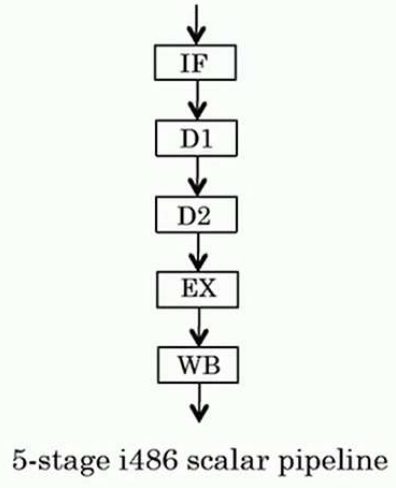
Scalar to Superscalar pipeline

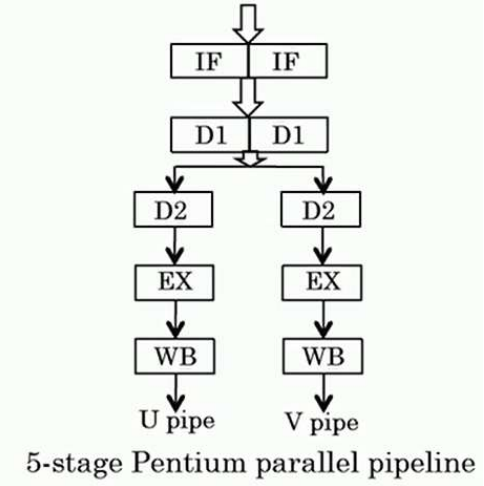








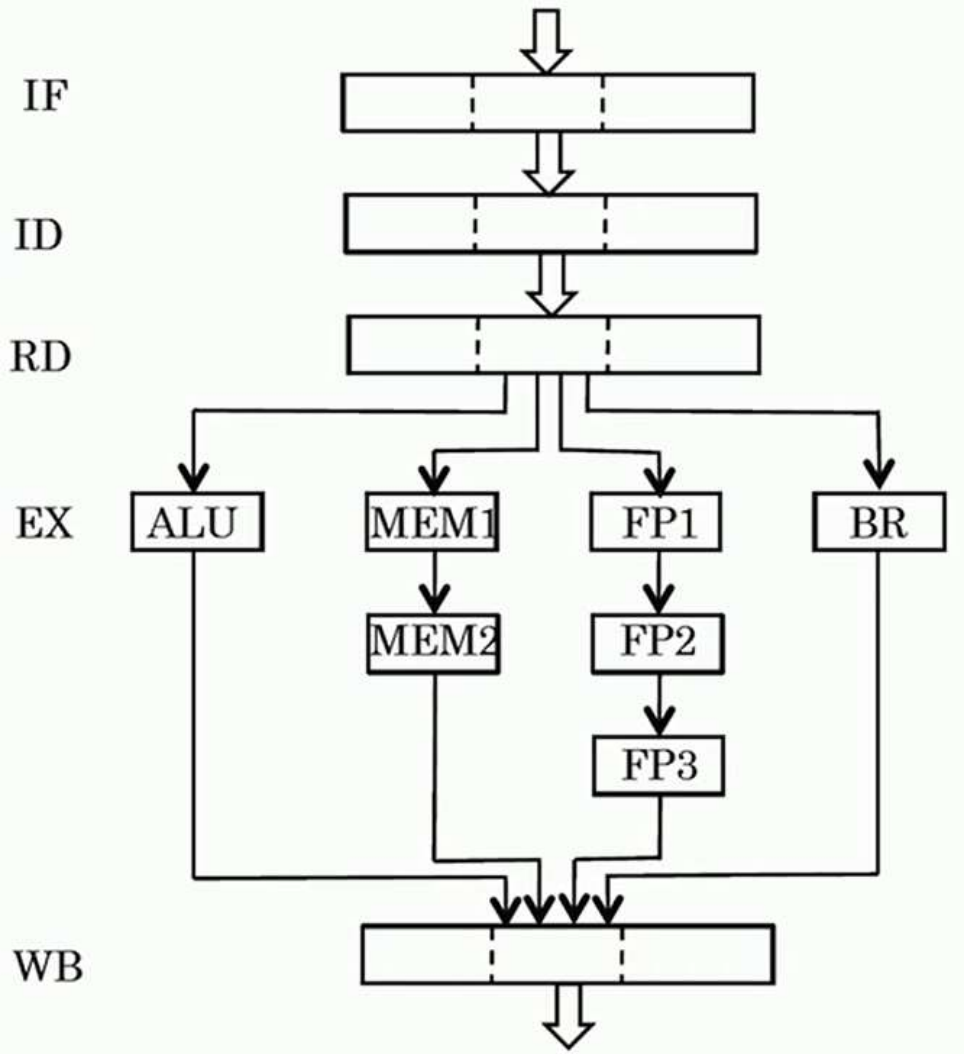




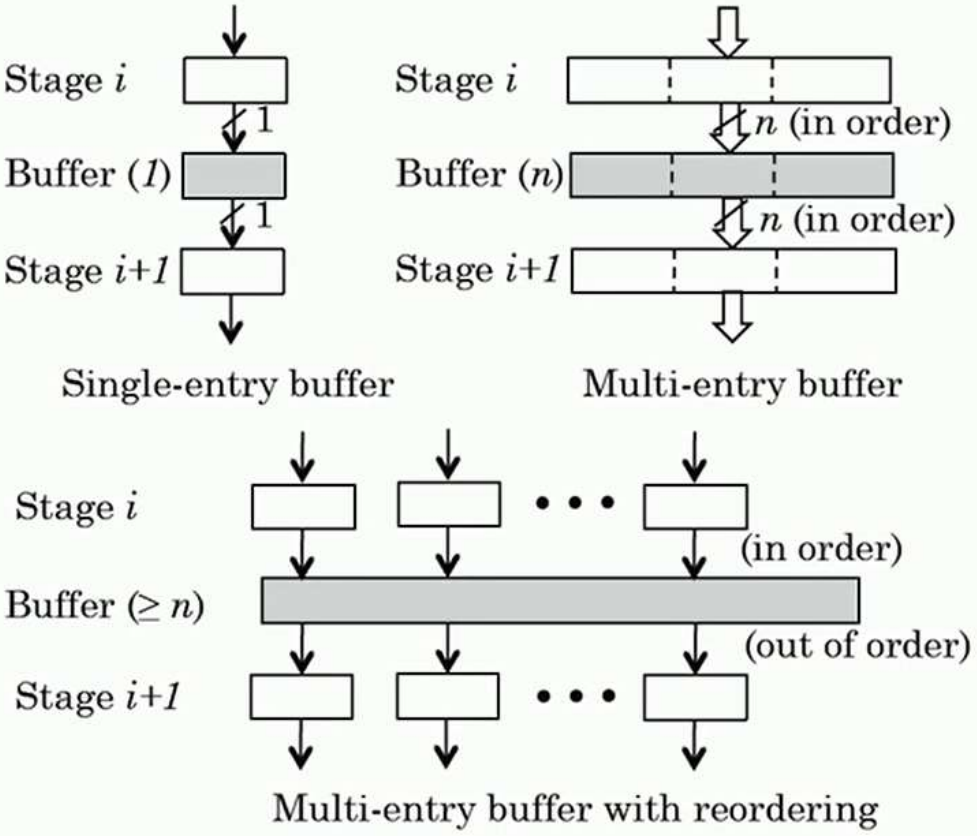
Parallel Pipelines

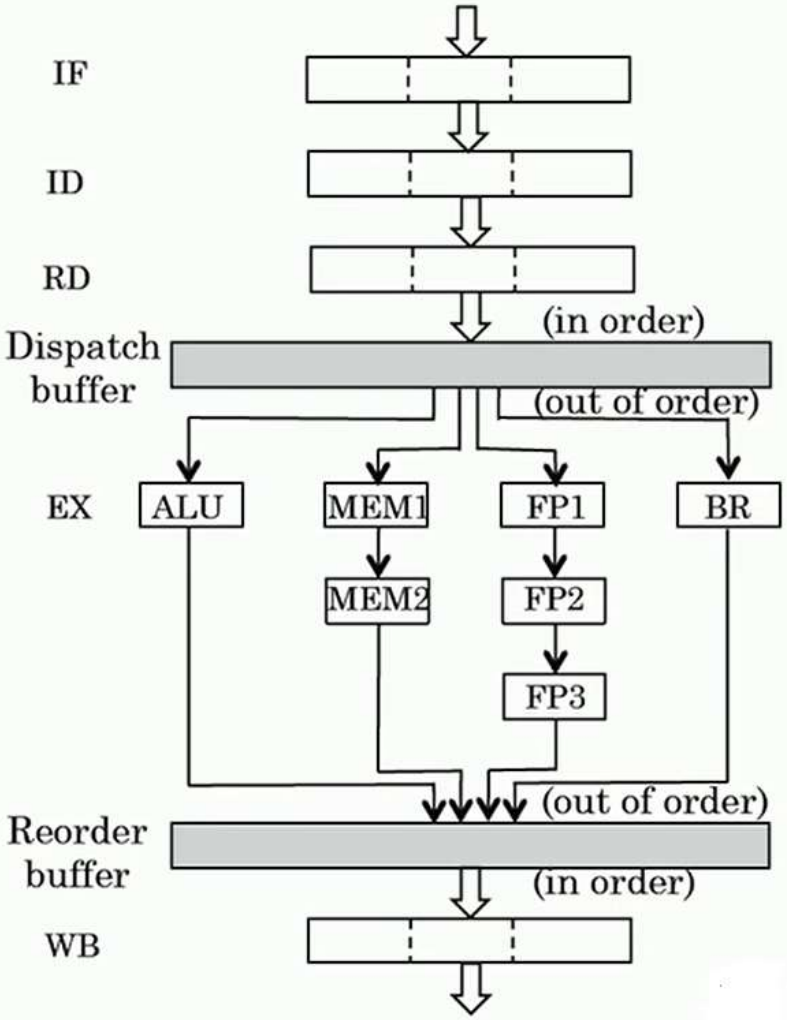
- Speedup of a scalar pipeline is determined by the *depth* of the pipeline
- Speedup of a parallel pipeline is determined by the *width* of the pipeline

Diversified Pipeline



Dynamic pipelines

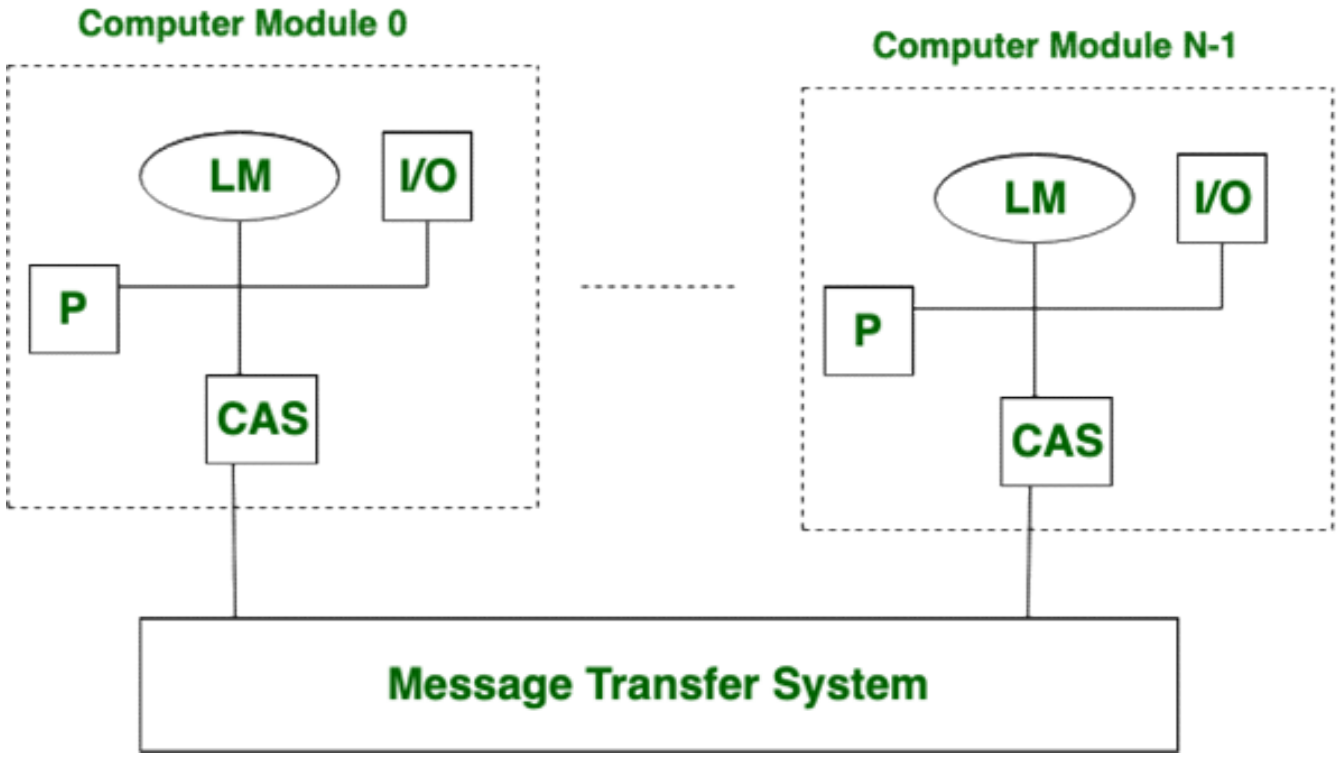




Loosely Coupled Multiprocessor system

- It is a type of multiprocessing system in which, There is distributed memory instead of shared memory.
- In loosely coupled multiprocessor system, data rate is low rather than tightly coupled multiprocessor system.
- In loosely coupled multiprocessor system, modules are connected through MTS (Message transfer system) network.

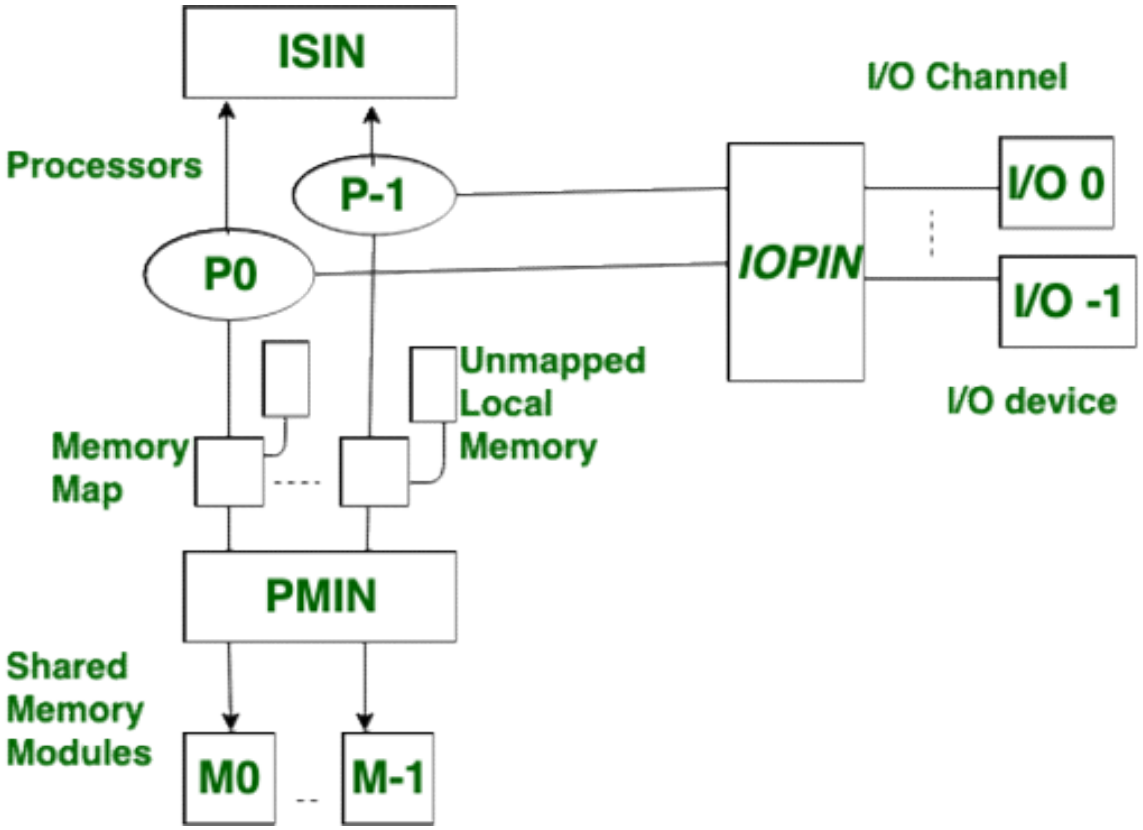
Loosely Coupled Multiprocessor system



Tightly Coupled Multiprocessor system

- It is a type of multiprocessing system in which there is shared memory.
- In tightly coupled multiprocessor system, data rate is high rather than loosely coupled multiprocessor system.
- In tightly coupled multiprocessor system, modules are connected through PMIN, IOPIN and ISIN networks.

Tightly Coupled Multiprocessor system



Differences

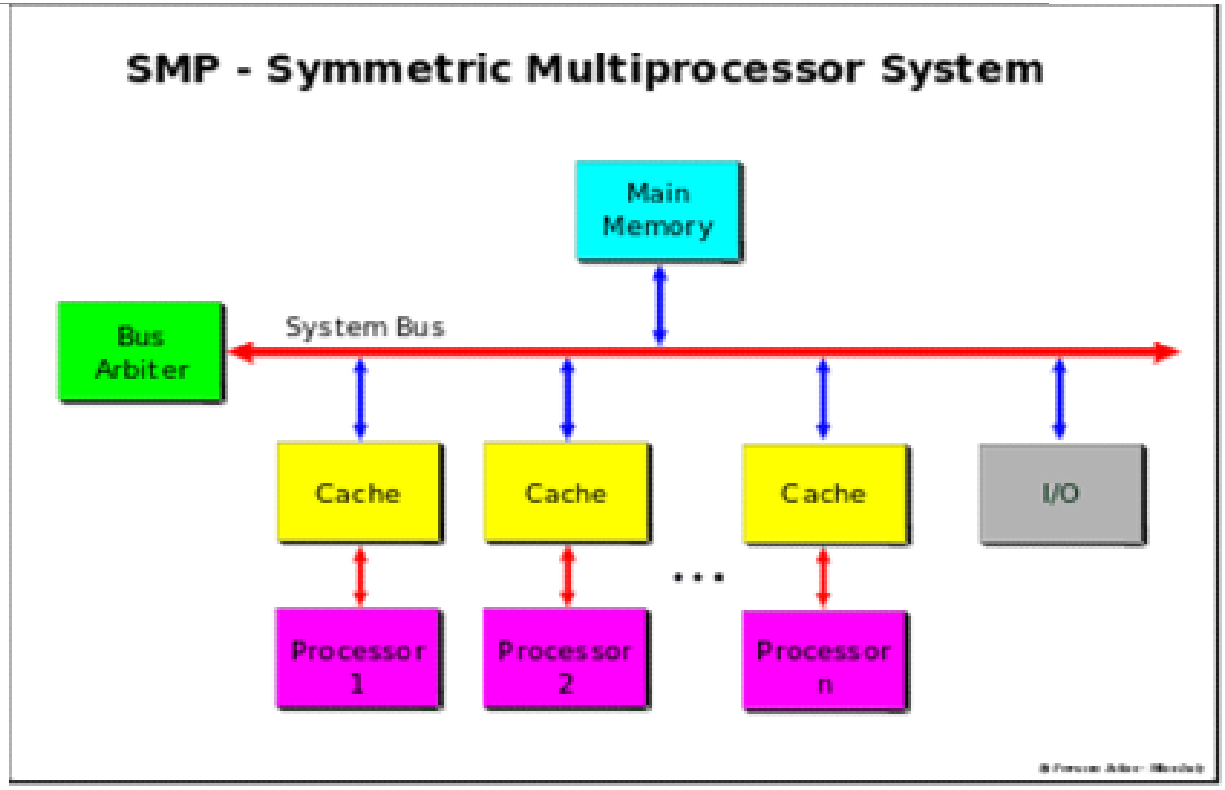
Loosely coupled	Tightly coupled
There is distributed memory in loosely coupled multiprocessor system	There is shared memory, in tightly coupled multiprocessor system
Has low data rate	Has high data rate
The cost of this system is less	It is more costly
Modules are connected through Message transfer system network	While there is PMIN, IOPIN and ISIN networks
Memory conflicts don't take place	This system have memory conflicts
It has low degree of interaction between tasks	It has high degree of interaction between tasks
there is direct connection between processor and I/O devices	IOPIN helps connection between processor and I/O devices
Applications of loosely coupled multiprocessor are in distributed computing systems	Applications of tightly coupled multiprocessor are in parallel processing systems

Symmetric Multiprocessor system

- SMP systems have centralized shared memory called *main memory* (MM) operating under a single operating system with two or more homogeneous processors.
- Usually each processor has an associated private high-speed memory known as cache memory (or cache) to speed up the main memory data access and to reduce the system bus traffic.
- Processors may be interconnected using buses, crossbar switches or on-chip mesh networks.
- The bottleneck in the scalability of SMP using buses or crossbar switches is the bandwidth and power consumption of the interconnect among the various processors, the memory, and the disk arrays.
- Mesh architectures avoid these bottlenecks, and provide nearly linear scalability to much higher processor counts at the sacrifice of programmability

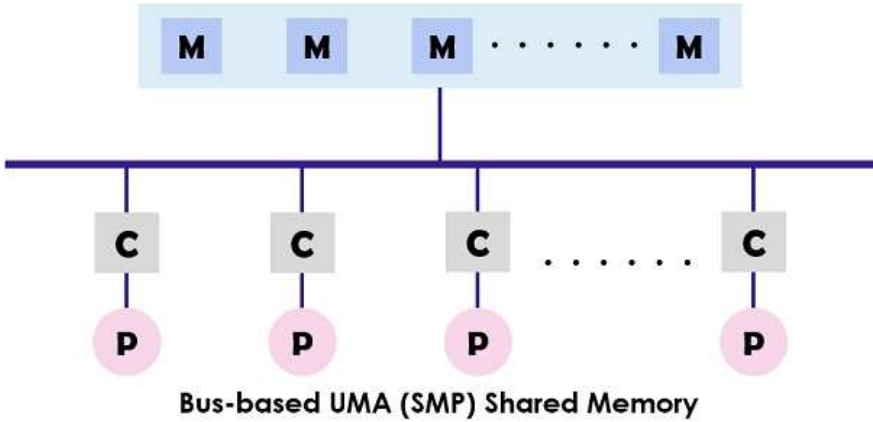
Symmetric Multiprocessor system

- SMP systems allow any processor to work on any task no matter where the data for that task is located in memory, provided that each task in the system is not in execution on two or more processors at the same time.
- With proper operating system support, SMP systems can easily move tasks between processors to balance the workload efficiently.



UMA

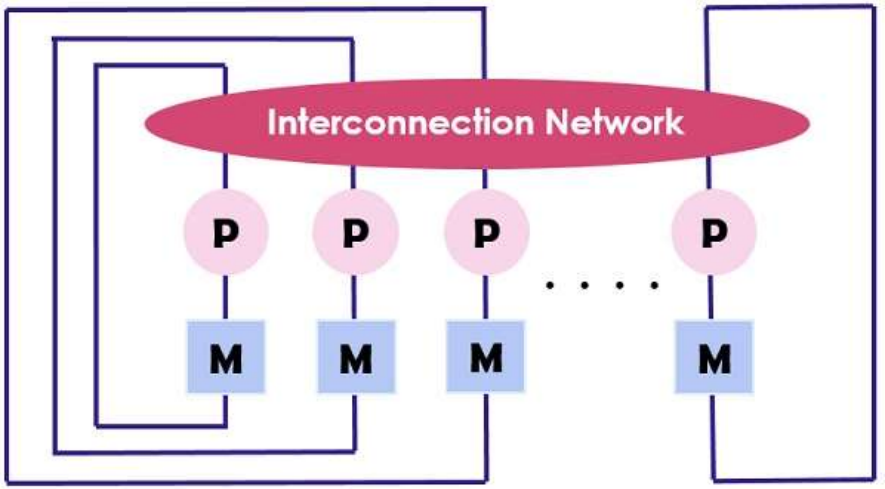
- UMA stands for Uniform Memory Access; it is a shared memory architecture for the multiprocessors.
- Single memory controller is used and accessed by all the processors with the help of the interconnection network.
- Each processor has equal memory accessing time (latency) and access speed.
- It can employ either of the single bus, multiple bus or crossbar switch.
- As it provides balanced shared memory access, it is also known as **SMP (Symmetric multiprocessor)** systems.
- Uniform Memory Access is slower than non-uniform Memory Access.
- Uniform Memory Access has limited bandwidth.
- Uniform Memory Access is applicable for general purpose applications and time-sharing applications.
- In uniform Memory Access, memory access time is balanced or equal.



Bus-based UMA (SMP) Shared Memory

NUMA

- NUMA stands for Non-Uniform Memory Access; it is a multiprocessor model in which each processor is connected with a dedicated memory.
- However, these small parts of the memory combine to make a single address space.
- Unlike UMA, the access time of the memory relies on the distance where the processor is placed → which means varying memory access time.
- It allows access to any of the memory location by using the physical address.
- NUMA is intended to increase the available bandwidth to the memory and for which it uses multiple memory controllers.
- It combines numerous machine cores into "**nodes**" where each core has a memory controller.
- To access the local memory in a NUMA machine the core retrieves the memory managed by the memory controller by its node.
- While to access the remote memory which is handled by the other memory controller, the core sends the memory request through the interconnection links.



NUMA Shared Memory System

Memories: Specification

How much?

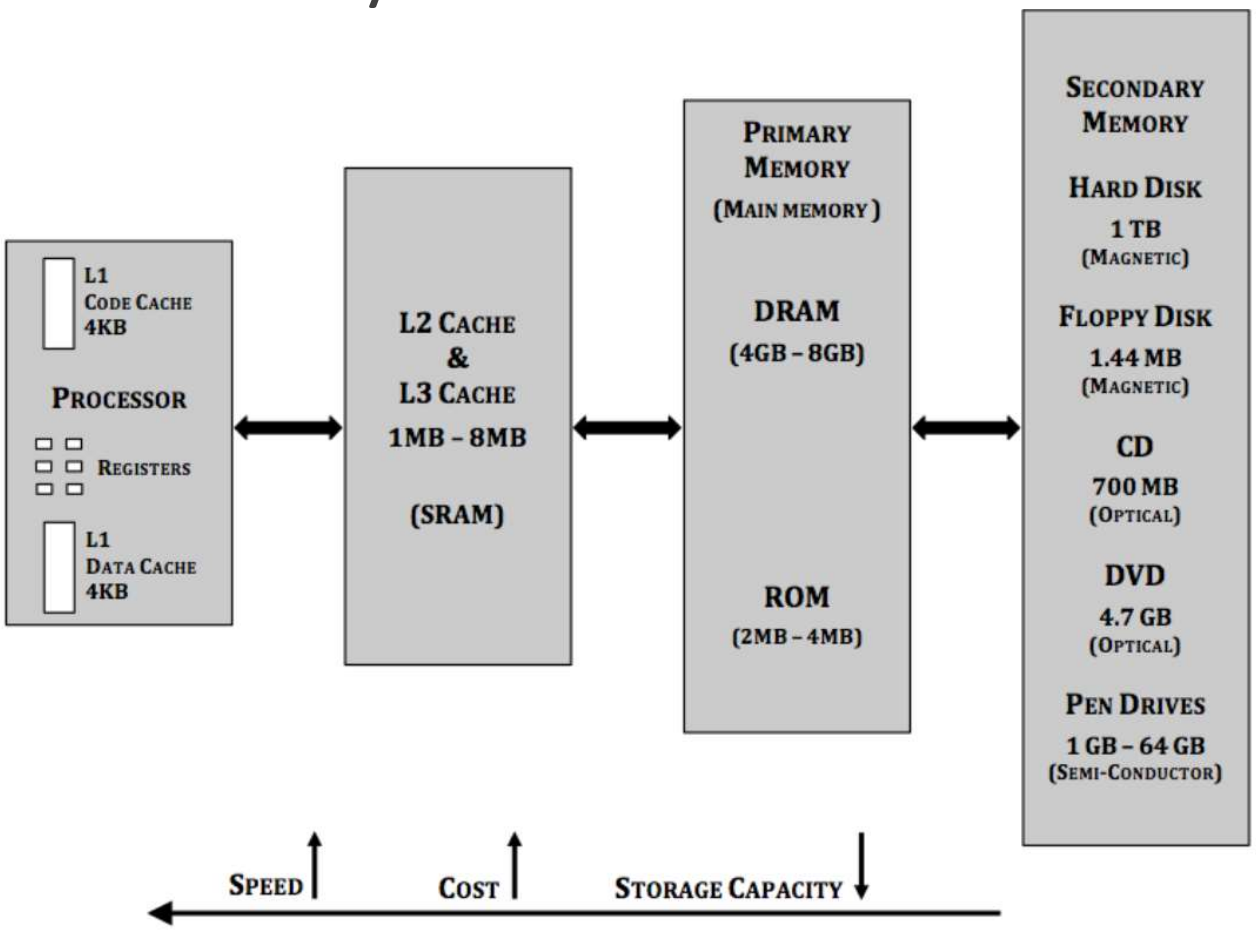
- Capacity

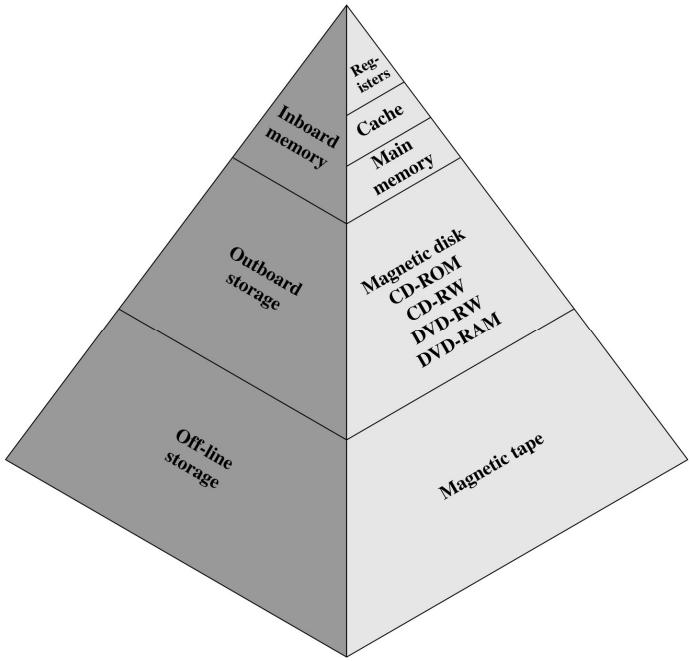
How fast?

- Time is money

How expensive?

Hierarchy List





REGISTERS

1) Registers are **present inside the processor**.
2) They are basically a **set of flip-flops**.
3) They store **data and addresses** and can **directly take part** in **arithmetic and logic operations**.
4) They are very small in size typically **just a few bytes**.

PRIMARY MEMORY

1) It is the original form of memory also called as **Main memory**.
2) It comprises of **RAM and ROM**, both are **Semi-Conductor** memories. (chip memories)
3) **ROM is non-volatile**.
   It is used is storing permanent information like the **BIOS program**.
   It is typically of **2 MB - 4 MB** in size.
4) **RAM is writable** and hence is used for **day-to-day operations**.
   Every file that we access from secondary memory, is **first loaded into RAM**.
   To provide large amount of working space RAM is **typically 4 GB - 8 GB**.

SECONDARY MEMORY

1) The main purpose of Secondary Memory is to **increase the storage capacity, at low cost**.
2) Its biggest component is the **Hard Disk**.
   This is where all the files inside a computer **are stored**.
3) It is **writeable as well as non-volatile**.
4) Typical size of a **HD is 1 TB**.
5) Disk memories are much **slower than chip memories** but are also **much cheaper**.

PORTABLE SECONDARY MEMORY

1) These are required to **physically transfer files** between computers.
2) **Floppy Disk**: It is a **magnetic form** of storage. Typical **Size is 1.44 MB**.
3) **CD**: It is an **optical form** of storage. Typical **Size is 700 MB**.
4) **DVD**: It is an **optical form** of storage. Typical **Size is 4.7 MB**.
5) **Pen Drives & Memory Cards**: It is a **semi-conductor form** of storage.
   It is composed of FLASH ROM.
   It's a special type of ROM that's writable as well as non-volatile.
   Typical **Size ranges form 1 GB - 64 GB** depending upon the cost.

CACHE MEMORIES

1) It is the **fastest form of memory** as it uses SRAM (Static RAM).
2) The Main Memory uses DRAM (Dynamic RAM).
3) **SRAM uses flip-flops and hence is much faster than DRAM which uses capacitors.**
4) But SRAM is also **very expensive** as compared to DRAM.
5) Hence **only the current portion of the file** we need to access is copied from Main Memory (DRAM) to Cache memory (SRAM), to be directly accessed by the processor.
6) This gives **maximum performance and yet keeps the cost low**.
7) Typical size of Cache is around **2 MB – 8MB**.
8) If **code and data** are in the **same cache** then it is **unified cache** else its **called split cache**.
9) Depending upon the location of cache, it is of three types: L1, L2 and L3.
10) **L1** cache is **present inside the processor** and is a **split cache** typically **4-8 KB**.
11) L2 is present on the **same die as the processor** and is a **unified cache** typically **1 MB**.
12) L3 is present **outside the processor**. It is also **unified** and is typically of **2-8 MB**.

MEMORY CHARACTERISTICS

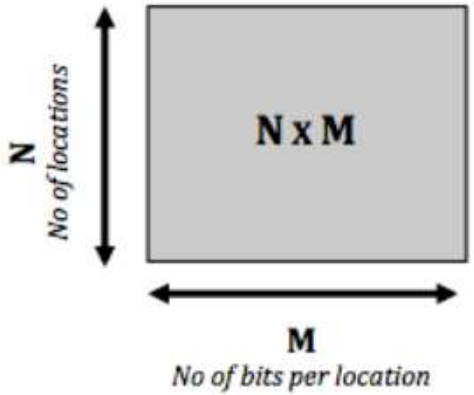
1) Location

Based on its physical location, memory is classified into three types.

- **On-Chip:** This memory is present **inside the CPU**. E.g.:: Internal Registers and **L1 Cache**.
- **Internal:** This memory is present **on the motherboard**. E.g.:: **RAM**.
- **External:** This memory is **connected to the motherboard**. E.g.:: **Hard disk**.

2) Storage Capacity

This indicates the **amount of data stored** in the memory.
Obviously it should be **as large as possible**.
It is represented as N x M.
Here,
N = **Number of memory locations** (no of words)
M = **Number of pits per memory location** (word size)
E.g.:: (4K x 8) means there are 4K locations of 8-bits each.



3) Transfer Modes

Data can be accessed from memory in two different ways.

- **Word Transfer:** Here, if CPU needs some data, it will transfer only that amount of data.
  E.g.:: Data accessed from **L1 Cache.**
- **Block Transfer:** Here, if CPU needs some data, it will transfer an entire block containing that data. This makes further access to remaining data of this block much faster. This is based on Principle of Spatial Locality. A processor is most likely to access data near the current location being accessed.
  E.g.:: On a **cache miss**, processor goes to **main memory** and copies a **block** containing that data.

4) Access Modes

Memories can allow data to be accessed in two different ways.

- **Serial Access:** Here locations are accessed one by one in a **sequential manner**.
  The access time depends on how far the target location is, from the current location.
  Farther the location, **more** will be its **access time**.
  E.g.:: Magnetic tapes.
- **Random Access:** Here **all locations** can be directly accessed in any **random order**.
  This means **all locations** have the **same access time** irrespective of their address.
  E.g.:: Most modern memories like RAM.

5) Physical Properties

There are various Physical attributes to memory.

- **Writeable: Contents** of the memory **can be altered**. E.g.:: **RAM**
- **Non-Writeable: Contents** of the memory **cannot be altered**. E.g.:: **ROM**
- **Volatile: Contents** of the memory are **lost** when power is **switched off**. E.g.:: **RAM**
- **Non-Volatile: Contents** of the memory are **retained** when power is **switched off**. E.g.:: **ROM**

Most secondary memories like Hard disk are Writable as well as non-volatile.

6) Access Time (t_A)

It is the time taken between **placing the request** and **completing the data transfer**.
It should be as **less as possible**.
It is also known as **latency**.

7) Reliability

It is the **time** for which the memory is expected to **hold the data without any errors**.
It is measured as **MTTF: Mean Time To Failure**.
It should be as **high as possible**.

8) Cost

This indicates the **cost of storing data** in the memory.
It is expressed as **Cost/bit**.
It must be **as low as possible**.

9) Average Cost

It is the total cost per bit, for the entire memory storage.
Consider a system having **two memories M_1 (RAM) & M_2 (ROM)**
If **C_1** is the cost of memory M_1 of size **S_1**
& **C_2** is the cost of memory M_2 of size **S_2**

Then the average cost of the memory is be calculated as:
$$C_{AVG} = (C_1 S_1 + C_2 S_2)/ (S_1 + S_2)$$

Small sizes of **expensive** memory and **large** size of **cheaper** memory **lowers** the **average cost**.

10) Hit Ratio (H)

Consider two memories M_1 and M_2.

M_1 is **closer** to the processor E.g.:: **RAM**, than **M_2** E.g.:: **Hard disk**.

If the **desired data is found in M_1**, then it is called a **Hit**, else it is a **Miss**.

Let **N_1** be the number of **Hits** and **N_2** the number of **Misses**.

The **Hit Ratio** H is defined as **number of hits divided by total attempts**.

$H = (N_1) / (N_1 + N_2)$

It is expressed s a percentage.

H can never be 100%. In most computers it is maintained around 98%.

From the above discussion it is clear that no single memory can satisfy all the characteristics, **hence we need a hierarchy of memories**.

Cache memories are the **fastest** but also the **most costly**.

Hard disk is **writeable** as well as **non volatile** and is also very **inexpensive**, but is much **slower**.

CD/DVD etc. are needed for **portability**.

ROM is **nonvolatile**, and is used for **storing BIOS**.

DRAM is **writable**, **faster than hard disk** and **cheaper than SRAM** hence forms **most part of Main Memory**.

So you want fast?

It is possible to build a computer which uses only static RAM (see later)

This would be very fast

This would need no cache

- How can you cache cache?

This would cost a very large amount

Locality of Reference

During the course of the execution of a program, memory references tend to cluster

e.g. loops

Cache

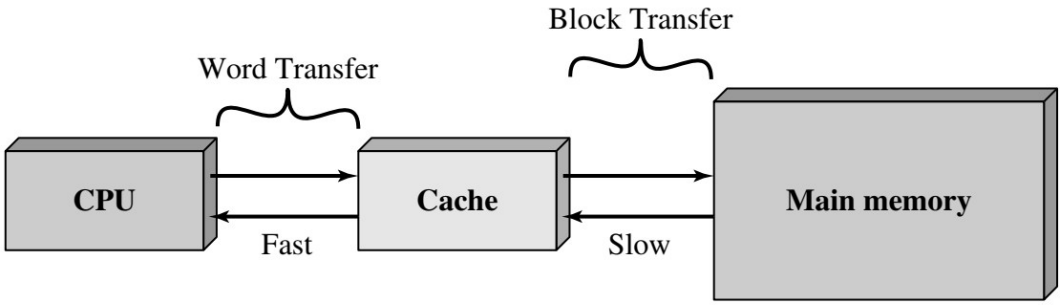
Small amount of fast memory

Sits between normal main memory and CPU

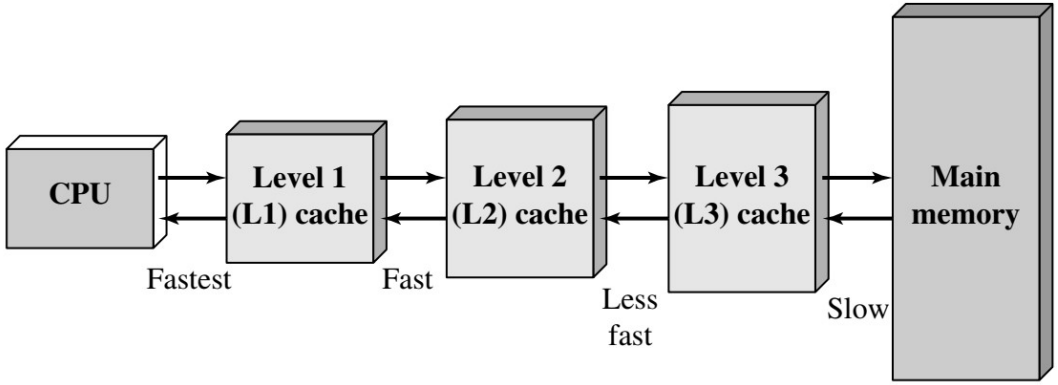
May be located on CPU chip or module

Cache memory

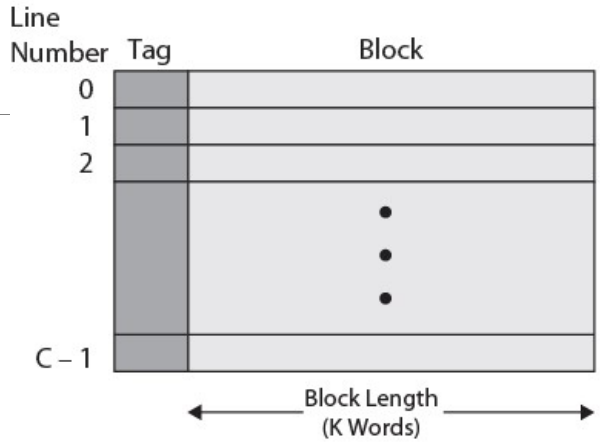
- Cache memory is intended to give memory speed approaching that of the fastest memories available
- At the same time provide a large memory size at the price of less expensive types of semiconductor memories.



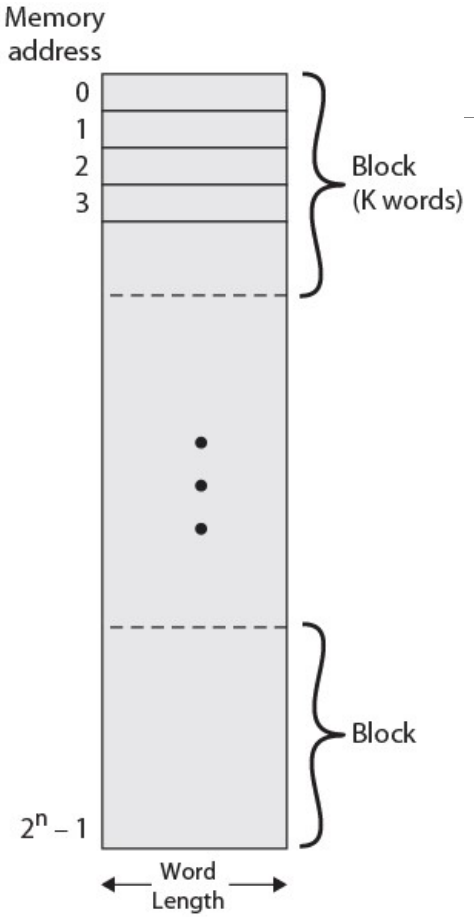
(a) Single cache



Cache/Main Memory Structure



(a) Cache



(b) Main memory

CACHE MAPPING TECHNIQUES

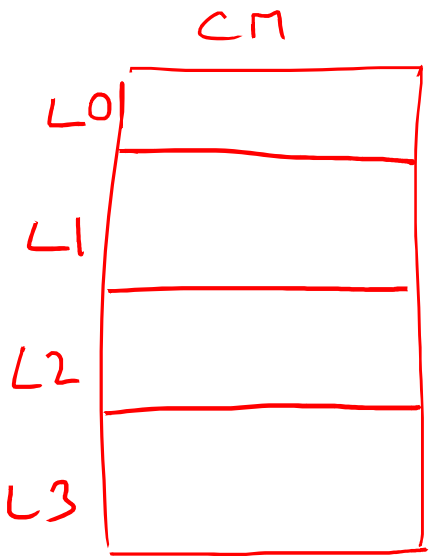
Blocks are loaded from Main Memory to Cache Memory.
Cache Mapping decides which block of Main Memory comes into which block of Cache Memory.
There are several mapping techniques trying to balance between Hit Ratio, Search-time and Tag size.
Every cache block has a **Tag** indicating which block of Main Memory is mapped into that block.
A collection of such tags is called the **cache directory**, very similar to a page table.
Cache directory is a part of the cache.
Since Cache Memory is **very expensive**, we need the cache directory to be as small as possible.
That means the **Tag must be of minimum size**.
The Main Memory address, issued by the processor contains the desired block number.
This is compared to the Tag of a Cache Block, which gives the Block number that is present.
If they are equal, **it's a Hit**. If Not, the search may have to be repeated for several other blocks.
It is obvious to understand, **the number of searches must be as low as possible.**
Finally, the Mapping technique must yield maximum utilization of the Cache memory space, so that the **Hit ratio remains as High as possible**.
There are three popular Cache Mapping Techniques:

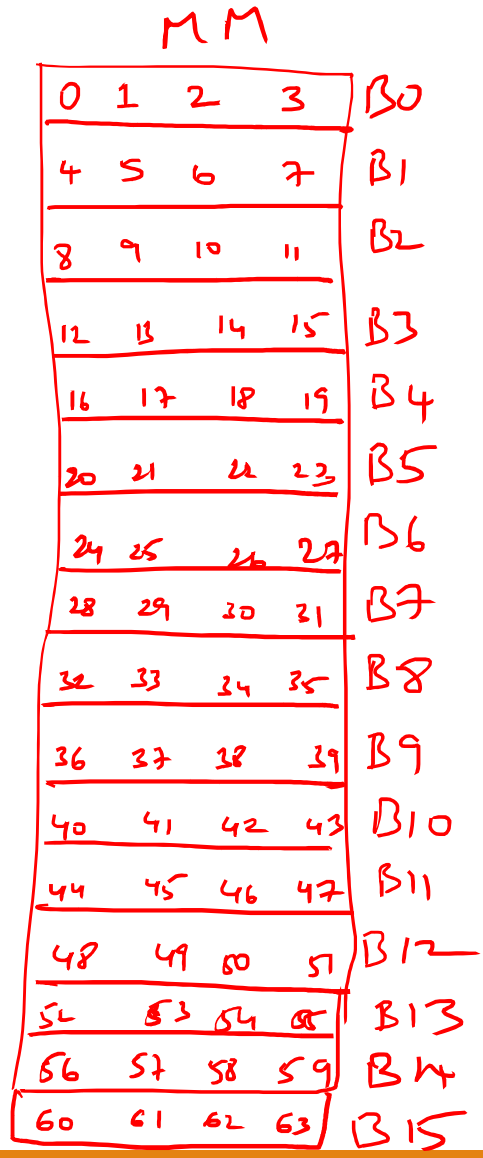
1) **Associative Mapping** also called Fully Associative Mapping
2) **Direct Mapping** also called One-Way Set Associative Mapping
3) **Set Associative Mapping** also called Two-Way Set Associative Mapping

CM = 16B

MM = 64B

Block size = 4B



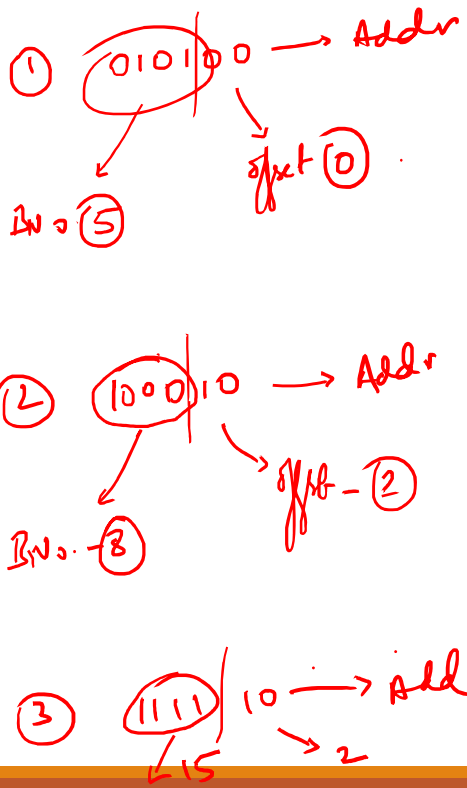


CM = 16B

MM = 64B

Block size = 4B

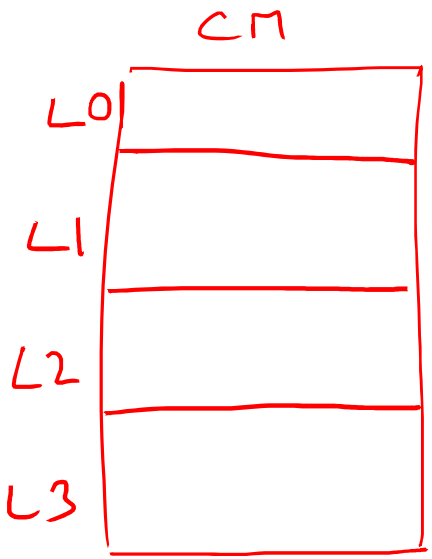
Associative Mapping

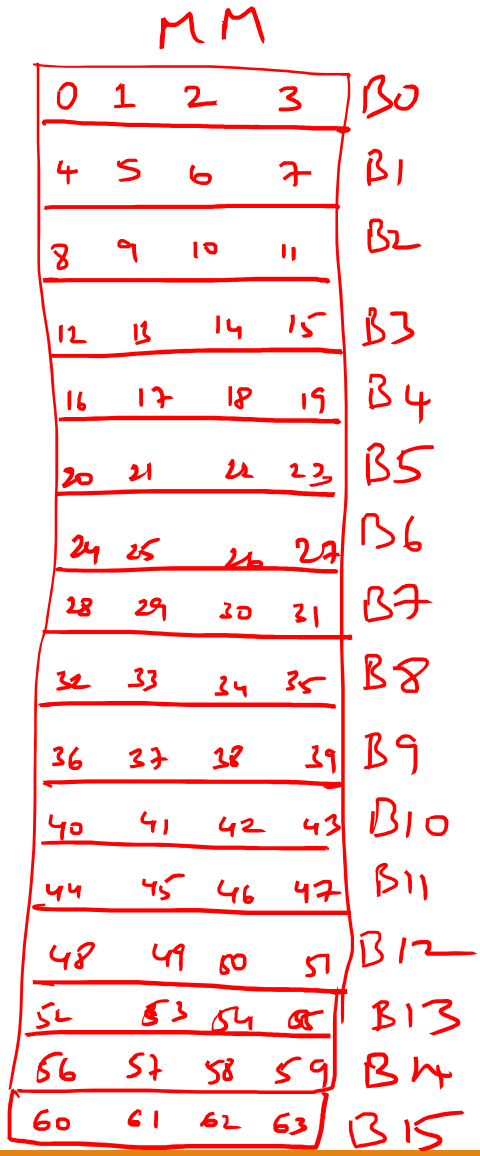


CM = 16B

MM = 64B

Block size = 4B



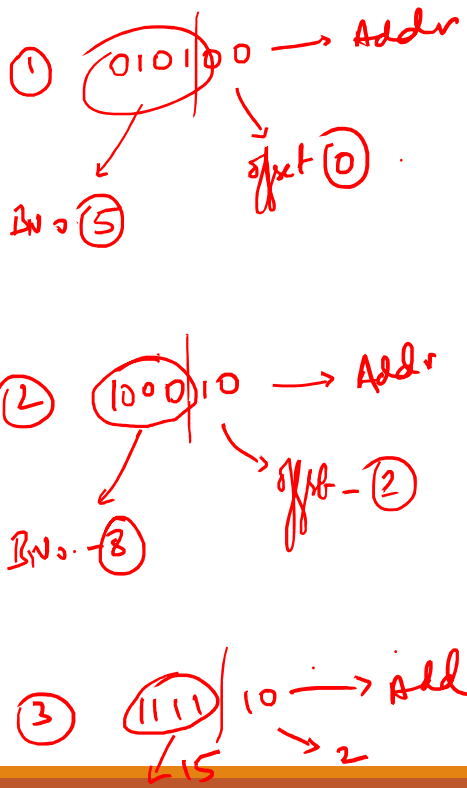


CM = 16B

MM = 64B

Block size = 4B

Associative Mapping



Associative Mapping (Fully Associative Mapping)

During memory operations, Blocks are loaded from Main Memory to Cache Memory.
Cache Mapping decides which block of Main Memory comes into which block of Cache Memory.
Fully Associative Mapping technique states:
Any block of Main Memory can be mapped at Any available block of Cache Memory.
There are no rules restricting the mapping at all.
This means the Full Cache is available for mapping hence the name Fully Associative.

Consider Pentium Processor Cache

Size of Main Mem:	**4GB** = 2^{32}
Size of Cache Mem:	**8KB** = 2^{13}
Size of Cache Block (Line):	**32** bytes (words) = 2^5
No. of Blocks in Main Mem:	Size of Main Memory (2^{32}) ÷ Size of Block (2^5) = 2^{27}
No. of Blocks in Cache Mem:	Size of Cache Memory (2^{13}) ÷ Size of Block (2^5) = 2^8 = **256**
Main Mem address:	**32 bits** (because main Mem is of **4GB** = 2^{32})

Tag Size:
A block of Cache Memory can **contain any block** of main memory out of a possible 2^{27} **blocks**.
Hence, the **Tag** next to every block in Cache Memory must be of **27 bits**.

Searches:
A block of main Memory can be **mapped into any block** of Cache Memory out of **256 Blocks**.
Hence, we need to do **256 searches** in Cache Memory.

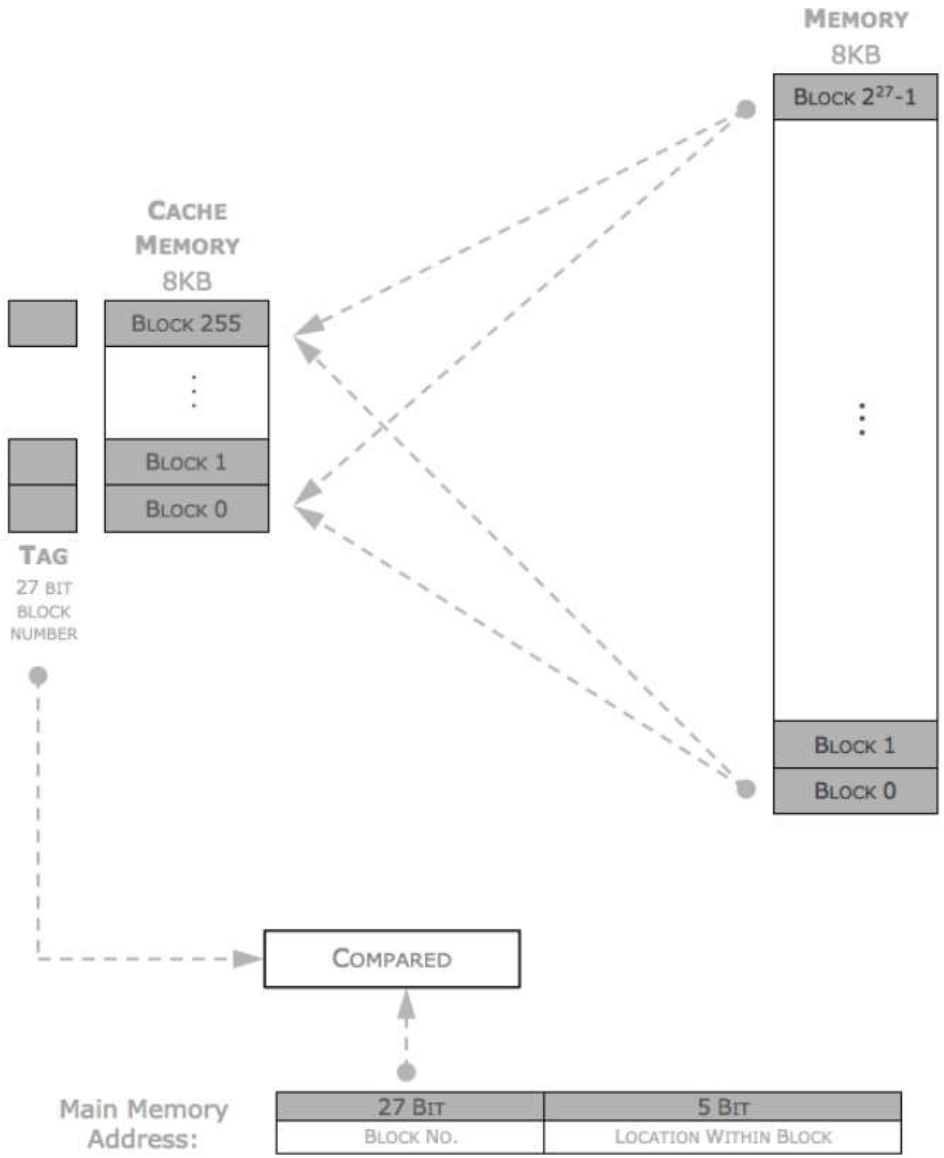
Method of Searching:
The Processor issues a 32 bit Main Memory address. It can be divided as:

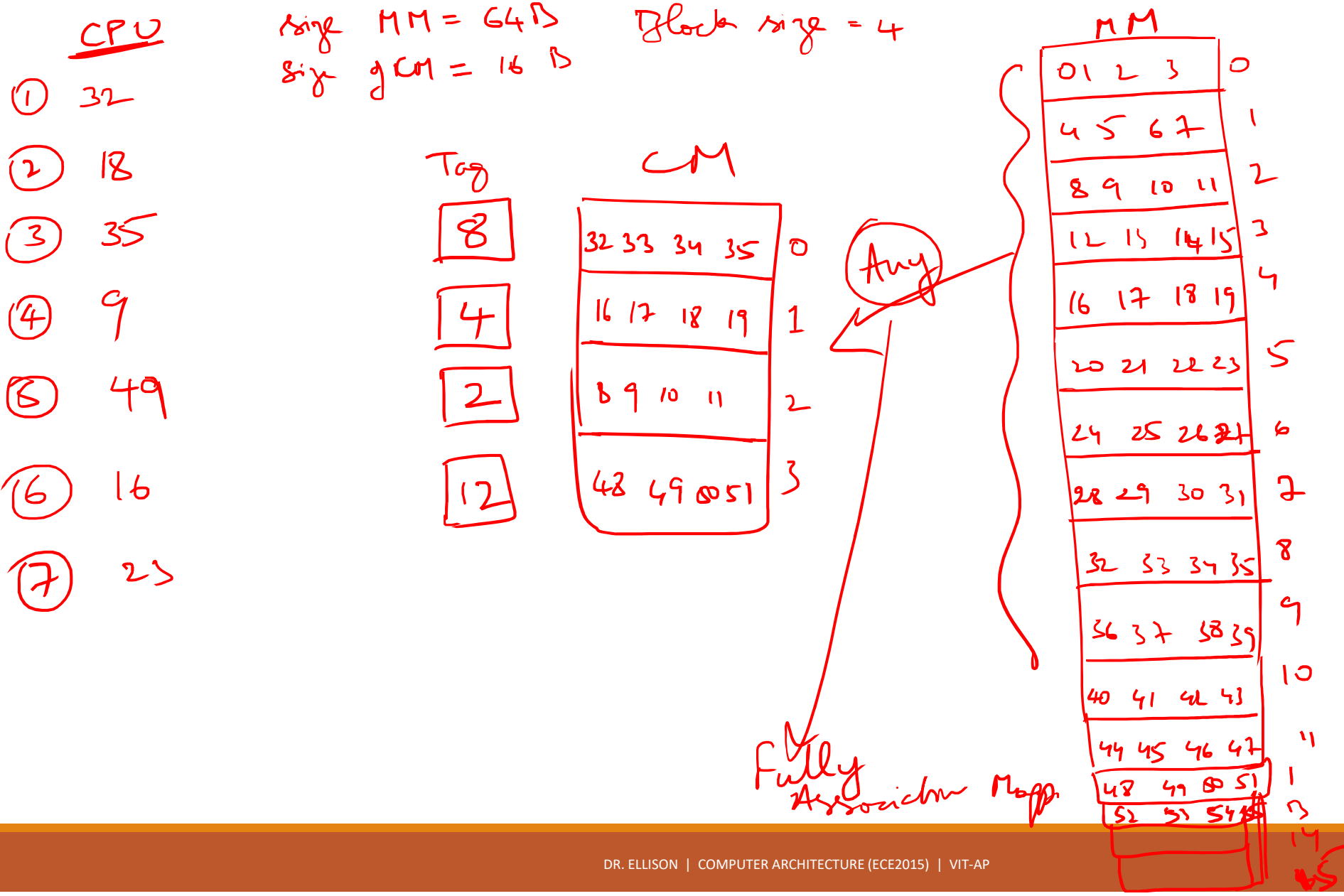
Main Memory Address:	27 BIT	5 BIT
	Block No.	Location Within Block

This 27 bit Block number is the block number we need to search.
The Tag of each cache block also contains a 27-bit block number.
This is the block number that's present in that respective cache block.
These two block numbers are compared. **If they are equal, it's a HIT.**
If not equal, the search is repeated with the Tag of the next cache block.
This is done a total of 256 times as there are 256 blocks in Cache Memory.
If none of them match with the block number we are searching, then it's a Miss.

Advantage
Since the full cache is available for mapping, it causes maximum utilization of Cache Memory hence gives the **Best Hit Ratio.**

Drawback
Tag Size too big: **27bits.**
Searches are too many: **256.**


MEMORY
8KB
BLOCK 2^{27}-1
CACHE
MEMORY
8KB
BLOCK 255
BLOCK 1
BLOCK 0
TAG
27 BIT
BLOCK
NUMBER
BLOCK 1
BLOCK 0
COMPARED
Main Memory
Address:
27 BIT
BLOCK NO.
5 BIT
LOCATION WITHIN BLOCK


CPU
1 32
2 18
3 35
4 9
5 49
6 16
7 23
size MM = 64B
size CM = 16 B
Block size = 4
Tag
CM
8
4
2
12
32 33 34 35
16 17 18 19
8 9 10 11
48 49 50 51
0
1
2
3
Any
Fully Associative Mapp.
MM
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
16 17 18 19
20 21 22 23
24 25 26 27
28 29 30 31
32 33 34 35
36 37 38 39
40 41 42 43
44 45 46 47
48 49 50 51
52 53 54 55
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Exercise

Consider the main memory size is 8GB and cache memory is 64MB. Let the size of each block be 4KB. Perform fully associative mapping with suitable diagrams assuming that the cache is full with values from the RAM. If CPU is trying is access the memory location 4357, find out whether there will be a cache hit or miss.

Direct Mapping (One Way Set Associative Mapping)

Direct Mapping technique states:
Any block of Main Memory can only be mapped at ONE block of Cache Memory.
Since there is only one way of Mapping, its also called One Way Set Associative Mapping.
We treat the entire Cache as One Set.
The Main Memory is divided into Sets which are then subdivided into Blocks.
A Block of Main Mem. (of any set), can only be mapped into the same Block No. in Cache Mem.
This means, Block 0 of Main Memory (of any set), can only be mapped into Block 0 of Cache Memory.
In other words, Block 0 of Cache Memory can only contain Block 0 of Main Memory but of any Set.

Consider Pentium Processor Cache

Size of Main Mem:	**4GB** = 2^{32}
Size of Cache Mem:	**8KB** = 2^{13} ... this is treated as One Set
Hence Size of Set:	**8KB** = 2^{13}
Size of Cache Block (Line):	**32** bytes (words) = 2^5
No. of Blocks in a set:	Size of Set (2^{13}) ÷ Size of Block (2^5) = 2^8 = **256**
No. of Sets in Main Mem:	Size of Main Mem (2^{32}) ÷ Size of Set (2^{13}) = 2^{19}
No. of Sets in Cache Mem:	1
Main Mem address:	**32 bits** (because main Mem is of **4GB** = 2^{32})

Tag Size:
Since, Block 0 of Cache Memory can only contain Block 0 of Main Memory but of any Set, the Tag has to only indicate the Set No. of Main Memory, from which the block is present.
As Main Memory has 2^{19} sets, the **Tag** size is **19 bits**.

Searches:
If we need Block 0 if Main Memory, we only need to search Block 0 of Cache Memory.
Hence, we need to do only **1 search** in Cache Memory to know if it is a Hit or a Miss.

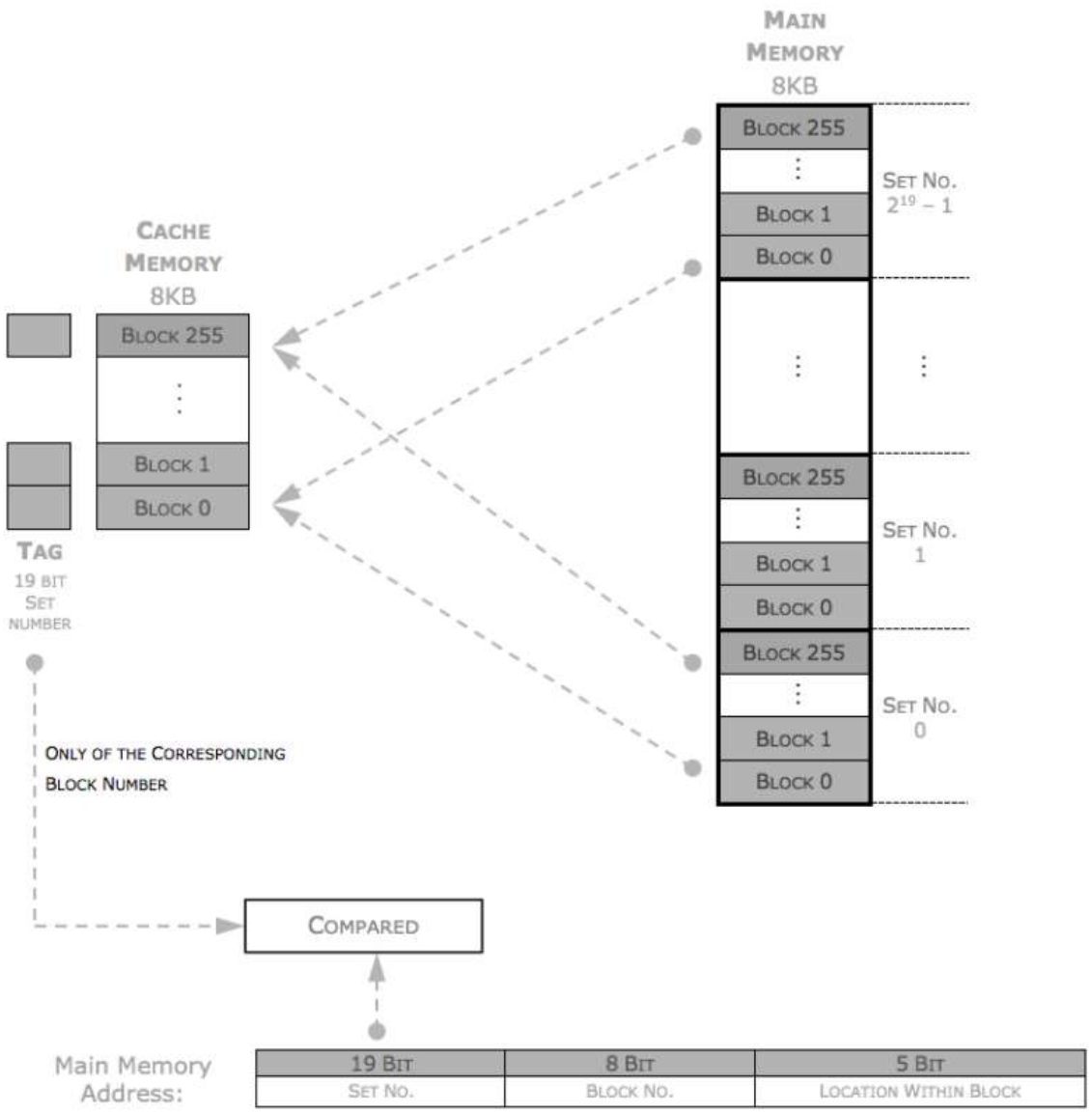
Method of Searching:
The Processor issues a 32 bit Main Memory address. It can be divided as:

Main Memory Address:	19 BIT	8 BIT	5 BIT
	Set No.	Block No.	Location Within Block

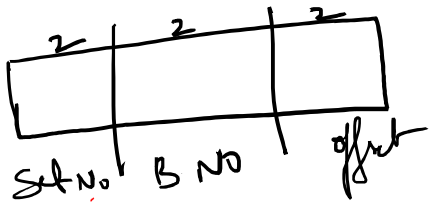
First we look at the block no. we need, to know where we need to search.
We then look at the Set No. that we need and compare it with the Tag of the corresponding Block no in the Cache Memory.
Assume the Main Memory address is 5:0:6. This means we need location 6 of Block 0 of set 5.
We go to Block 0 of Cache Memory.
It has a Tag, which gives the Set No of Main Memory whose Block 0 is present in Cache Memory.
These two set numbers are compared. **If they are equal, it's a HIT, else it's a Miss.**

Advantage
In **1 Search** we know if it is a Hit or a Miss. **Tag Size = 19 bits**.

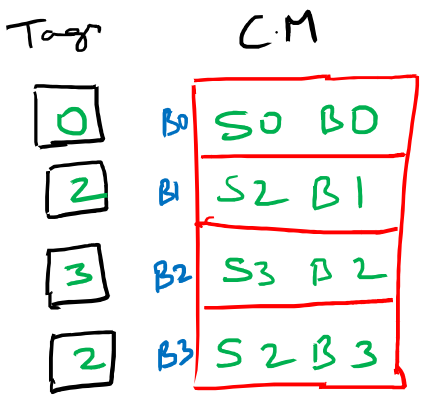
Drawback
Since the method is **very rigid**, the **Hit Ratio drops tremendously.**

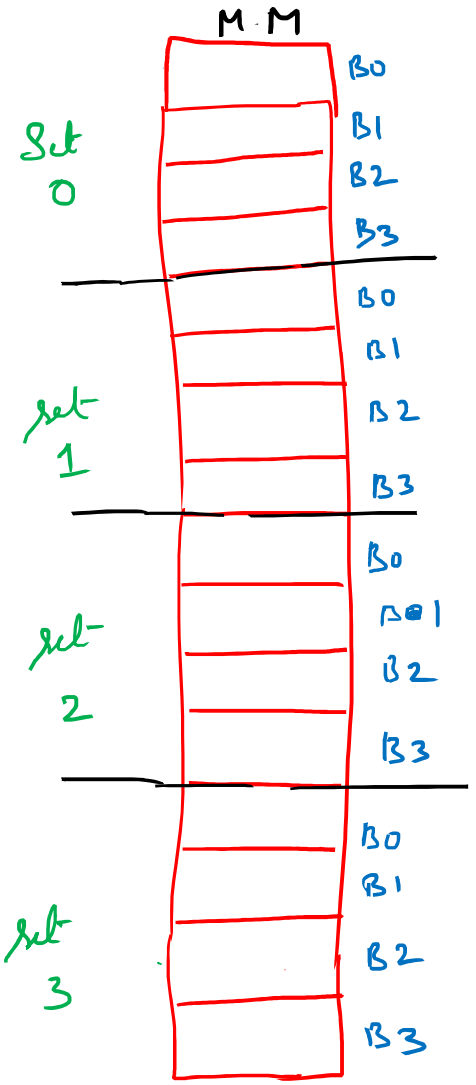

Main Memory
8KB
Block 255
Block 1
Block 0
Set No.
$2^{19} - 1$
Block 255
Block 1
Block 0
Set No.
1
Block 255
Block 1
Block 0
Set No.
0
Cache Memory
8KB
Block 255
Block 1
Block 0
Tag
19 bit
Set
number
Only of the Corresponding
Block Number
Compared
Main Memory
Address:
19 Bit
Set No.
8 Bit
Block No.
5 Bit
Location Within Block

MM Addr = 6 bits



1. 31
2. 47
3. 58
4. 39
5. 3
6. 57





Direct Mapping (One Way Set Associative Mapping)

Direct Mapping technique states:
Any block of Main Memory can only be mapped at ONE block of Cache Memory.
Since there is only one way of Mapping, its also called One Way Set Associative Mapping.
We treat the entire Cache as One Set.
The Main Memory is divided into Sets which are then subdivided into Blocks.
A Block of Main Mem. (of any set), can only be mapped into the same Block No. in Cache Mem.
This means, Block 0 of Main Memory (of any set), can only be mapped into Block 0 of Cache Memory.
In other words, Block 0 of Cache Memory can only contain Block 0 of Main Memory but of any Set.

Consider Pentium Processor Cache

Size of Main Mem:	**4GB** = 2^{32}
Size of Cache Mem:	**8KB** = 2^{13} ... this is treated as One Set
Hence Size of Set:	**8KB** = 2^{13}
Size of Cache Block (Line):	**32** bytes (words) = 2^5
No. of Blocks in a set:	Size of Set (2^{13}) ÷ Size of Block (2^5) = 2^8 = **256**
No. of Sets in Main Mem:	Size of Main Mem (2^{32}) ÷ Size of Set (2^{13}) = 2^{19}
No. of Sets in Cache Mem:	1
Main Mem address:	**32 bits** (because main Mem is of **4GB** = 2^{32})

Tag Size:
Since, Block 0 of Cache Memory can only contain Block 0 of Main Memory but of any Set, the Tag has to only indicate the Set No. of Main Memory, from which the block is present.
As Main Memory has 2^{19} sets, the **Tag** size is **19 bits**.

Searches:
If we need Block 0 if Main Memory, we only need to search Block 0 of Cache Memory.
Hence, we need to do only **1 search** in Cache Memory to know if it is a Hit or a Miss.

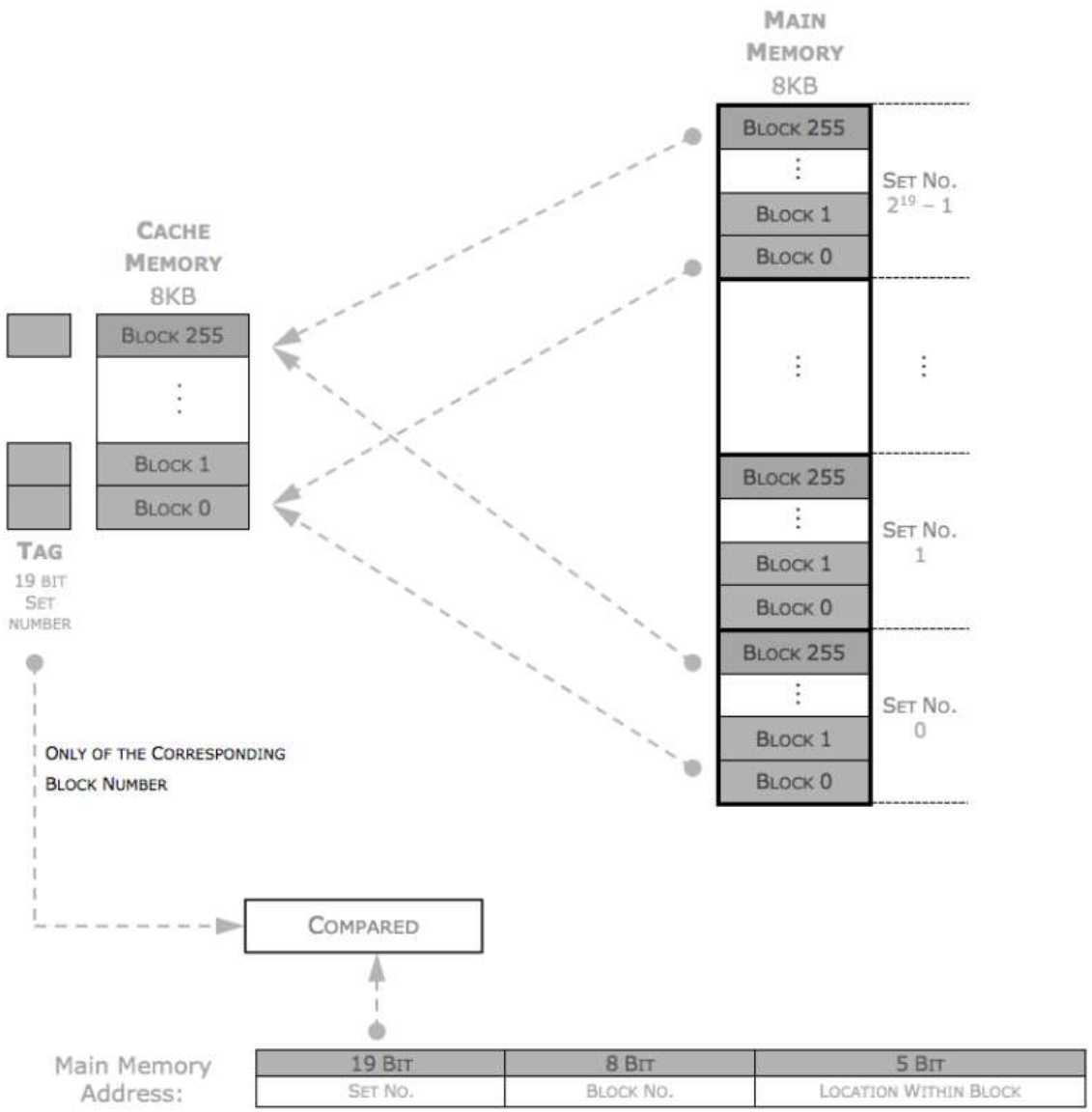
Method of Searching:
The Processor issues a 32 bit Main Memory address. It can be divided as:

Main Memory Address:	19 BIT	8 BIT	5 BIT
	Set No.	Block No.	Location Within Block

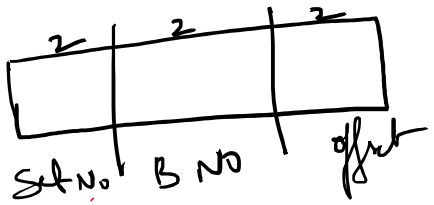
First we look at the block no. we need, to know where we need to search.
We then look at the Set No. that we need and compare it with the Tag of the corresponding Block no in the Cache Memory.
Assume the Main Memory address is 5:0:6. This means we need location 6 of Block 0 of set 5.
We go to Block 0 of Cache Memory.
It has a Tag, which gives the Set No of Main Memory whose Block 0 is present in Cache Memory.
These two set numbers are compared. **If they are equal, it's a HIT, else it's a Miss.**

Advantage
In **1 Search** we know if it is a Hit or a Miss. **Tag Size = 19 bits**.

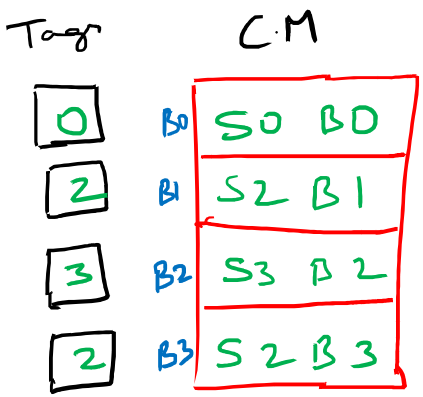
Drawback
Since the method is **very rigid**, the **Hit Ratio drops tremendously.**

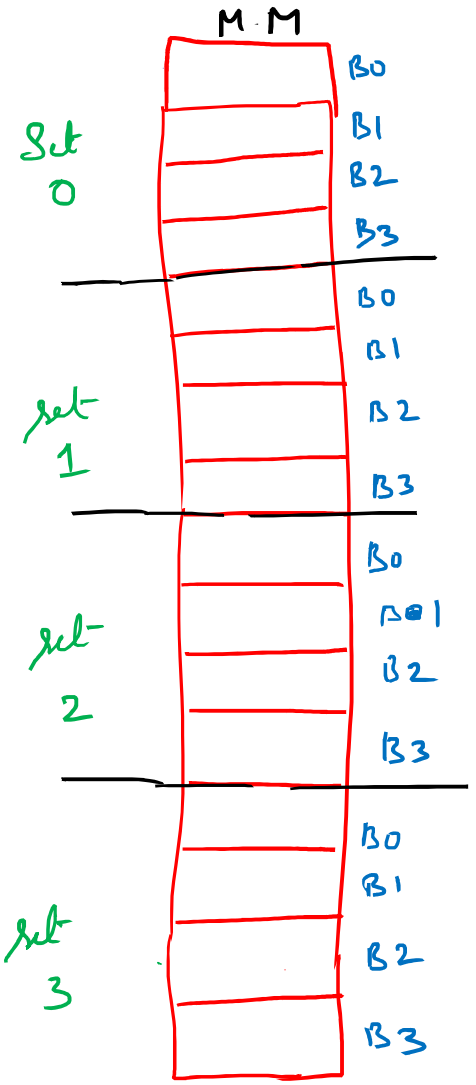

Main Memory
8KB
Block 255
Block 1
Block 0
Set No. $2^{19} - 1$
Block 255
Block 1
Block 0
Set No. 1
Block 255
Block 1
Block 0
Set No. 0
Cache Memory
8KB
Block 255
Block 1
Block 0
Tag
19 bit
Set
number
Only of the Corresponding
Block Number
Compared
Main Memory
Address:
19 Bit
Set No.
8 Bit
Block No.
5 Bit
Location Within Block

MM Addr = 6 bits



1. 31
2. 47
3. 58
4. 39
5. 3
6. 57





Exercise

Consider the main memory size is 32GB and cache memory is 16MB. Let the size of each block be 1MB. Perform direct mapping with suitable diagrams assuming that the cache is full with values from the RAM. If CPU is trying is access the memory location 9893, find out whether there will be a cache hit or miss.

Set Associative Mapping (Two Way Set Associative Mapping)

Two way Set Associative Mapping technique states:
A block of Main Memory can only be mapped into the same corresponding Block No. of Cache Memory, in any of the two sets.
Since there are two ways of Mapping, its called Two Way Set Associative Mapping.
We treat the entire Cache as **Two Sets**. The Main Memory is divided into Sets, subdivided into Blocks.
A Block of Main Mem. (of any set), can only be mapped into the same Block No. in Cache Memory again of any set. This means, Block 0 of Main Memory (of any set), can only be mapped into Block 0 of Cache Memory, into one of its two sets.
In other words, Block 0 of Cache Memory can only contain Block 0 of Main Memory but of any Set.

Consider Pentium Processor Cache (This is Actually how Pentium's Cache is implemented)

Size of Main Mem:	**4GB** = 2^{32}
Size of Cache Mem:	**8KB** = 2^{13} ... this is treated as Two Sets
Hence Size of Set:	**4KB** = 2^{12}
Size of Cache Block (Line):	**32** bytes (words) = 2^5
No. of Blocks in a set:	Size of Set (2^{12}) ÷ Size of Block (2^5) = 2^7 = **128**
No. of Sets in Main Mem:	Size of Main Mem (2^{32}) ÷ Size of Set (2^{12}) = 2^{20}
No. of Sets in Cache Mem:	2
Main Mem address:	**32 bits** (because main Mem is of **4GB** = 2^{32})

Tag Size:
Since, Block 0 of Cache Memory can only contain Block 0 of Main Memory but of any Set, the Tag has to only indicate the Set No. of Main Memory, from which the block is present.
As Main Memory has 2^{20} sets, the **Tag** size is **20 bits**.

Searches:
If we need Block 0 if Main Memory, we only need to search Block 0 of Cache Memory, but in any of the two sets. Hence, we need **2 searches** in Cache Memory to know if it is a Hit or a Miss.

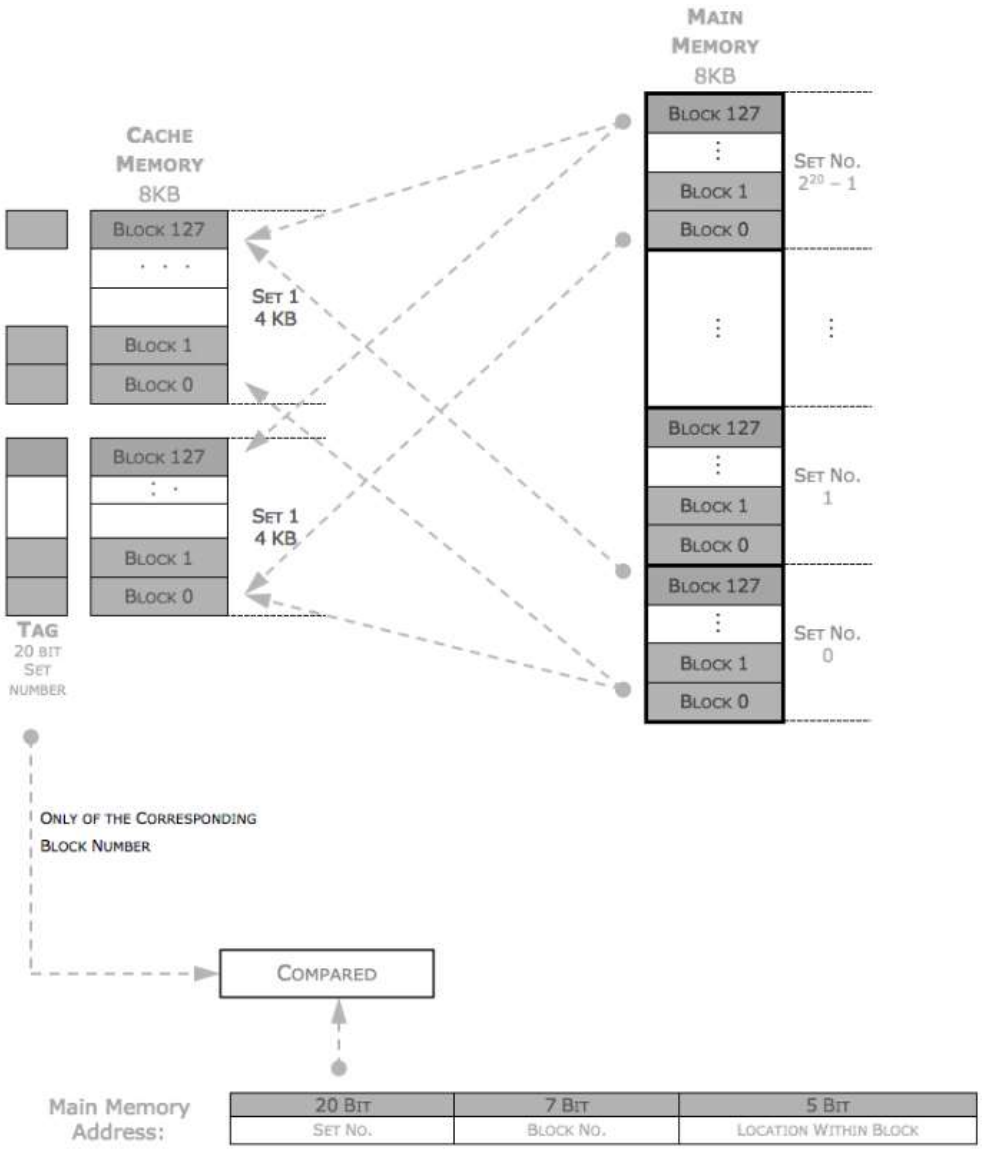
Method of Searching:
The Processor issues a 32 bit Main Memory address. It can be divided as:

Main Memory Address:	20 BIT	7 BIT	5 BIT
	Set No.	Block No.	Location Within Block

First we look at the block no. we need, to know where we need to search.
We then look at the Set No. that we need and compare it with the Tags of the corresponding Block no in the Cache Memory, in any of the two sets.
Assume the Main Memory address is 5:0:6. This means we need location 6 of Block 0 of set 5.
We go to Block 0 of Cache Memory, in both sets.
It has a Tag, which gives the Set No of Main Memory whose Block 0 is present in Cache Memory.
These required Set no. is compared with the two Tags. **If found in any of the 2, it's a HIT, else Miss.**

Advantage
In **2 Searches** we know if it is a Hit or a Miss. **Tag Size = 20 bits**.

Drawback
Since the method is **flexible**, it significantly **increases** the **Hit Ratio.**


Main Memory
8KB
Cache Memory
8KB
Block 127
Block 1
Block 0
Set 1
4 KB
Set No.
$2^{20} - 1$
Set No.
1
Set No.
0
Tag
20 bit
Set
number
Only of the Corresponding
Block Number
Compared
Main Memory
Address:
20 Bit
Set No.
7 Bit
Block No.
5 Bit
Location Within Block

Size of MM [2-way] = 4KB

Block size = 16B

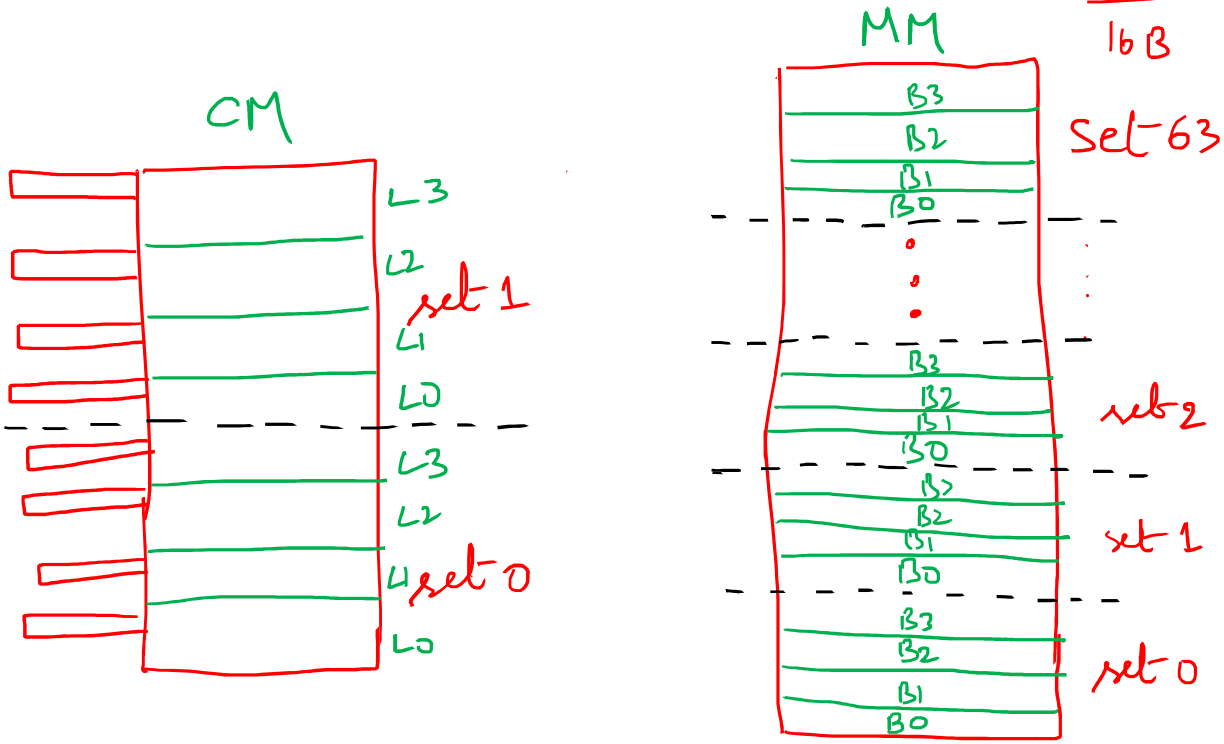
Size of CM = 128B

Size of set = 64B // No of sets in MM = $\frac{256}{4}$ = 64 sets // No of set in CM = 2

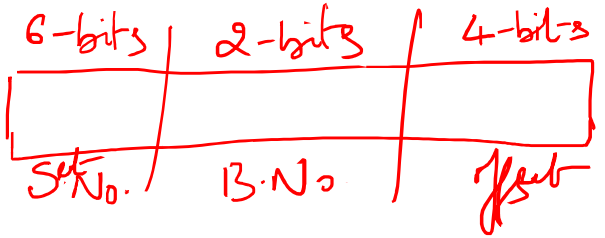
No of Blocks in MM = $\frac{4KB}{16B} = \frac{2^2 \times 2^{10}}{2^4} = 2^8$ = 256 Blocks

No of Blocks in CM = $\frac{128B}{16B} = \frac{2^7}{2^4} = 2^3$ = 8 Blocks

No. of Block in each set = $\frac{64B}{16B} = \frac{2^6}{2^4}$ = 4 Blocks in each set



MM Addr (12-bits)



Expanding the logic of set associative cache further, we can derive the following conclusion:

No of Ways	No of Searches	Tag Size	No of Blocks in a Set
2 Way	2	20 bits	128
4 Way	4	21 bits	64
8 Way	8	22 bits	32
16 Way	16	23 bits	16
32 Way	32	24 bits	8
64 Way	64	25 bits	4
128 Way	128	26 bits	2
256 Way	256	27 bits	1
This becomes exactly the same as Fully Associative: 256 Searches, 27-bit Tag			

WEB Materials-1

Direct Mapping

https://www.youtube.com/watch?v=VePK5TNgQU8&t=1579s

Fully Associative Mapping

https://www.youtube.com/watch?v=3eriC-pIQKg

Set-Associative Mapping

https://www.youtube.com/watch?v=mCF5XNn_xfA

https://www.youtube.com/watch?v=j5PUJllPPVE&t=893s

https://www.youtube.com/watch?v=1J_DhymCJok

WEB Materials-2

Direct Mapping

https://www.youtube.com/watch?v=QcAaP5V2Gpc&authuser=1

Fully Associative Mapping

https://www.youtube.com/watch?v=vWxtmci1Nko&authuser=1

Set-Associative Mapping

https://www.youtube.com/watch?v=vWxtmci1Nko&authuser=1

4-way

Size of MM = 1KB

Size of CM = 64B

Block size = 4B

No of sets in CM = 4

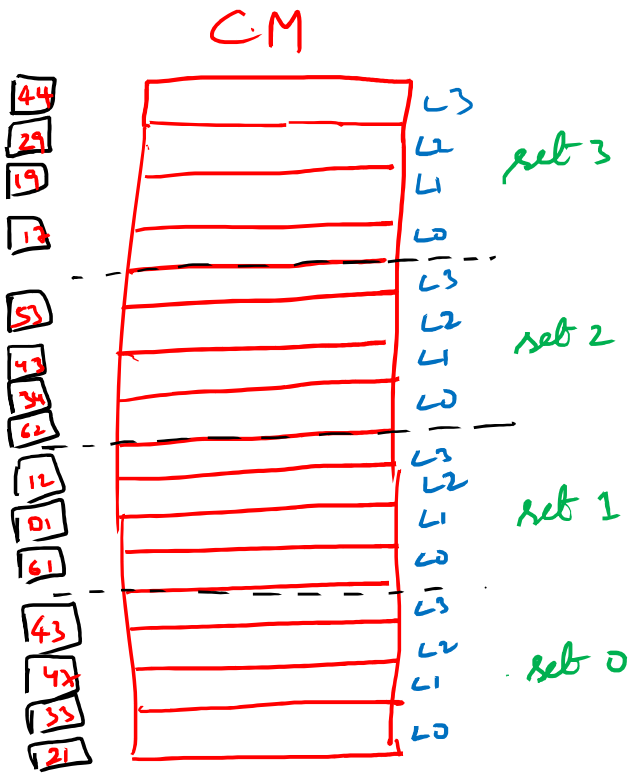
Set size = 16B

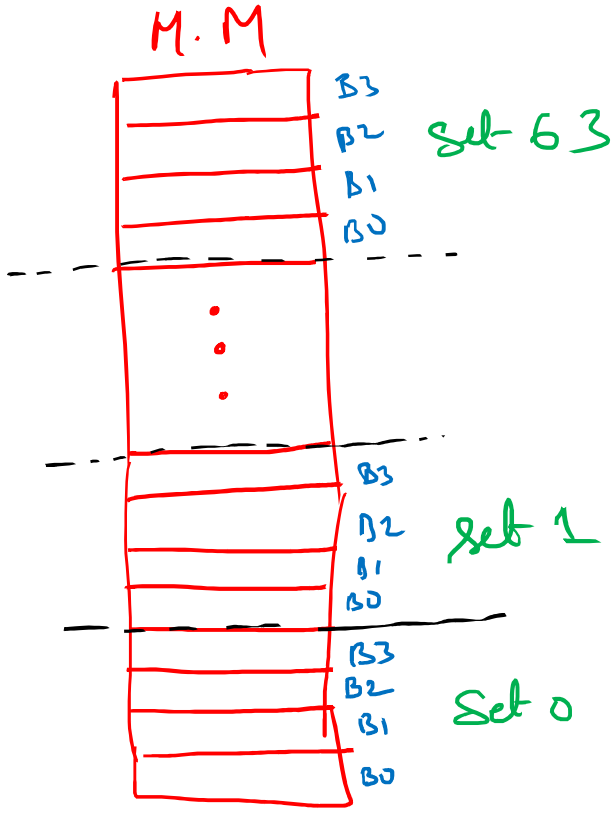
No of Blocks in MM = 256

No of Blocks in CM = 16

No of sets in MM = 64

No of blocks in each set = 4





Size of MM = 16KB
Size of CM = 256B
Block size = 32B

(i) Show direct mapping and let the CPU is trying to access the address "1AD6".

(ii) Show 4-way set associative mapping and try to access the address "169E".

(iii) Show Fully associative mapping & try to access the address "34BC"

(iii)

CM

L0	B127
L1	B501
L2	B468
L3	B197
L4	B309
L5	B317
L6	B409
L7	B23

(ii)

C4

L0	S212 B0	Set 0
L1	S127 B1	
L0	S56 B0	Set 1
L1	S97 B1	
L0	S250 B0	set-2
L1	S168 B1	
L0	S63 B0	set 3
L1	S32 B1	

(i)

L0	B72
L1	B25
L2	B170
L3	B99
L4	B116
L5	B237
L6	B182
L7	B319