Date / /
Page No.

Distinction between Computer Architecture and Computer Organization

Computer Architecture

- It refers to those attributes of system that have a direct impact on a logical execution of a program
- architectural attributes
  - Instruction set
  - no. of bits used to represent various data types
  - I/O mechanisms
  - techniques for addressing memory

Computer Organization

- It refers to operational units and their inter-connections that realize architectural specifications
- Organizational attributes
  - hardware details transparent to programmer
    - control signals
    - interfaces b/w computer and peripherals
    - memory technology used

Example: Architectural design issue → whether a computer will have a multiply instruction

Organizational issue → whether that instruction will be implemented by a special multiply unit or by a mechanism

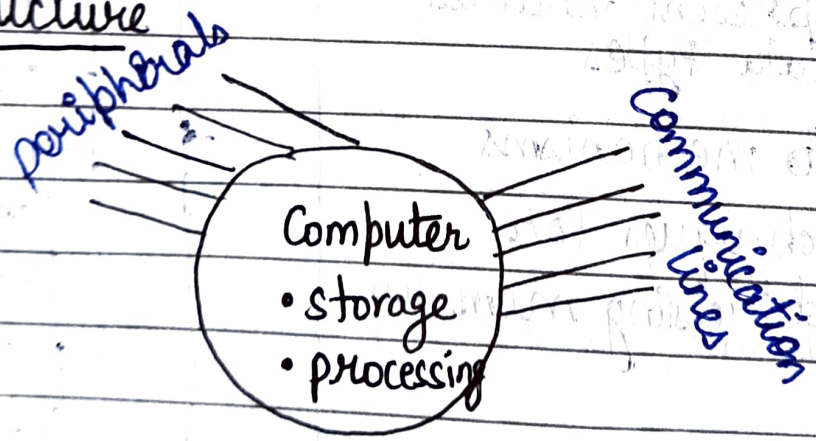
that makes repeated use of add unit of the system.

Computer Structure and Function

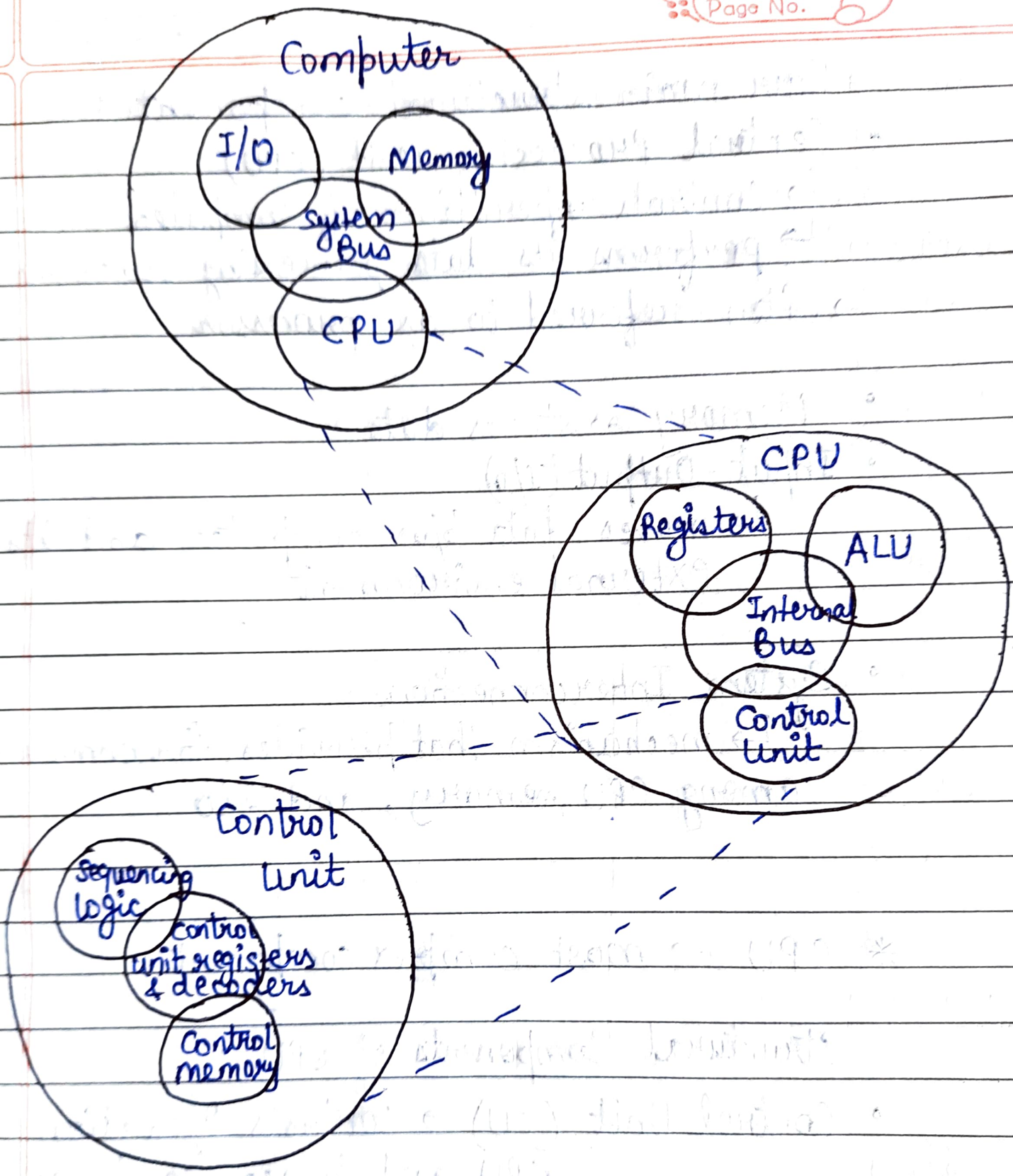
Structure → way in which components are interrelated

Function → operation of each individual component as part of the structure

(I). Structure



Interaction with external environment → peripheral devices or communication lines



Computer Structure

four main structural components:

- Central Processing Unit (CPU)
  - ↳ Controls operation of computer
  - ↳ performs its data processing functions

  → often referred to as processor

- Memory → stores data
- Input - Output (I/O)
  - ↳ moves data b/w computer and its external environment

- System Interconnection
  - ↳ mechanism that provides for communication among CPU, memory, and I/O.

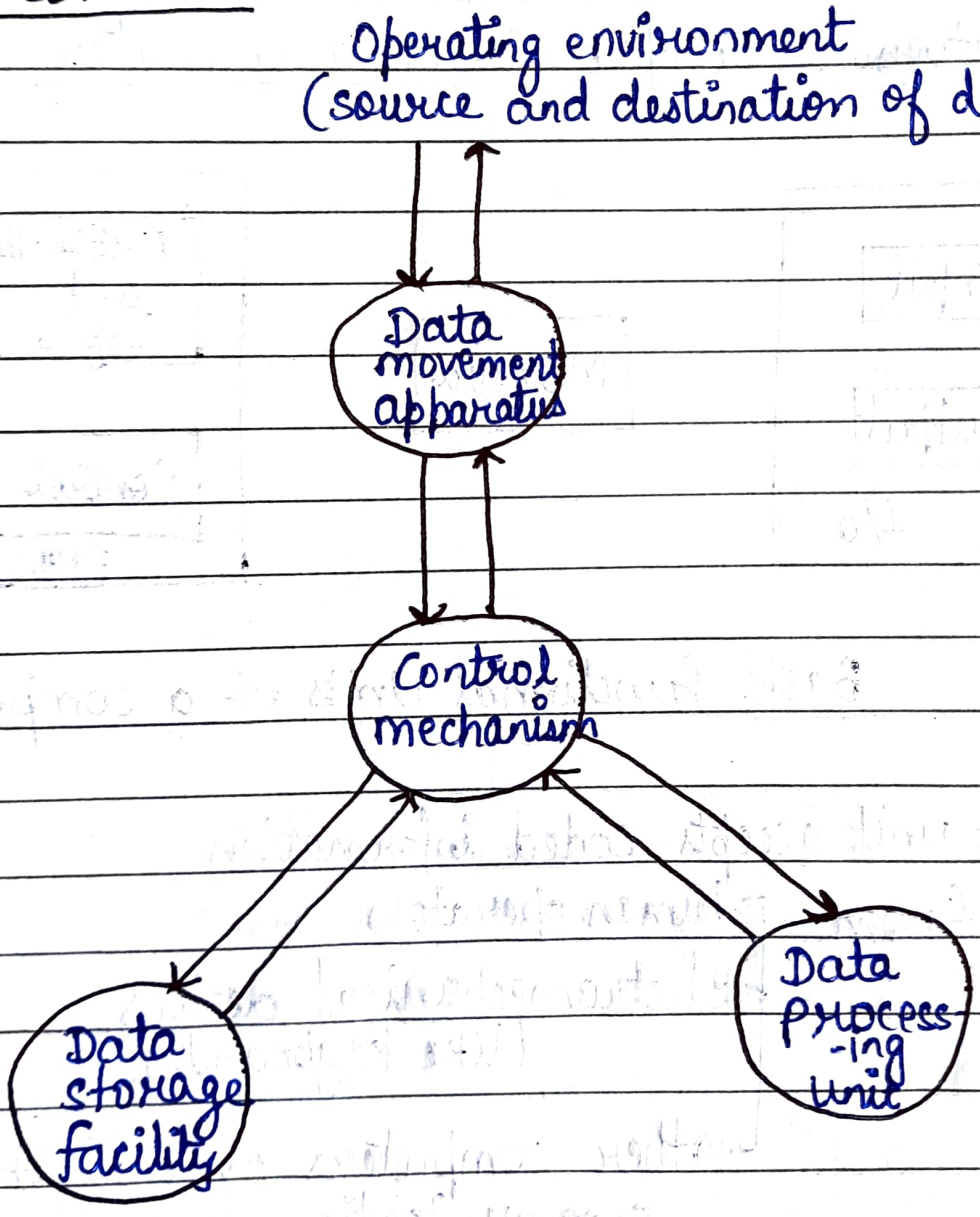
* CPU → most complex component

Structural components of CPU:

- Control Unit (CU) → controls Operation of CPU and hence computer
- Arithmetic and logic unit (ALU)
  - ↳ performs computer's data processing functions

- Registers → provides storage internal to CPU.
- CPU interconnection
  ↳ mechanism that provides for communication among control unit, ALU, and registers

(II) Function



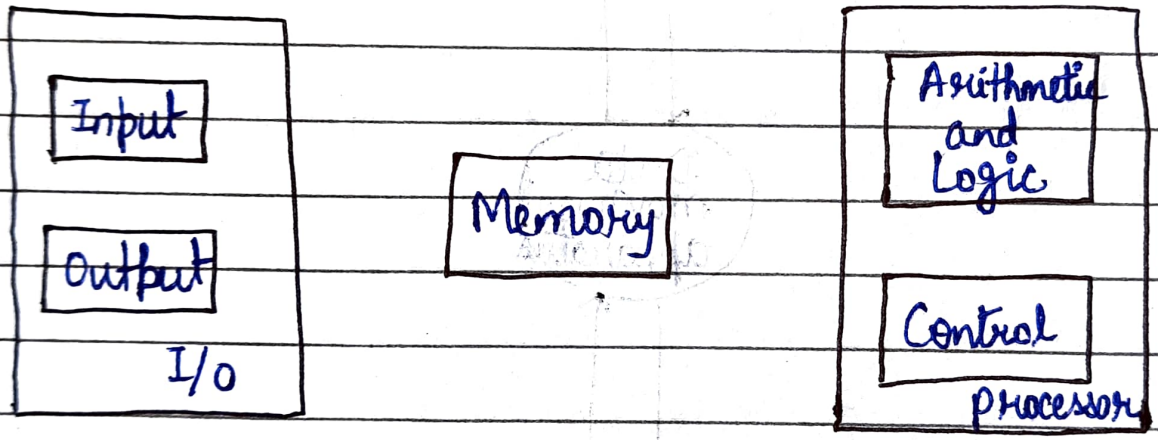
Functional View of Computer

Date / /
Page No.

Basic functions of a computer :

* Data processing
* Data Storage
* Data movement
* Control

Functional Components of a Computer



Basic functional Units of a computer

- Input unit accepts coded information from
  - human operators
  - electromechanical devices (like keyboards)
  - other computers over digital communication lines

- Information received is either stored in computer's memory for later reference or immediately used by ALU to perform desired operations
- Processing steps are determined by a program stored in the memory.
- Finally, results are sent back to outside world through output unit.
- All of these actions are coordinated by control unit
- There are connections among functional units too.

Information handled by a computer
- Instructions
- Data

* Instructions, or machine instructions are commands that
  - govern transfer of information within a computer as well as between computer and its I/O devices
  - specify arithmetic and logic operations to be performed

Program → list of instructions that performs a task (usually stored in memory)

→ Processor fetches instructions (that make up program) from the memory, one after another, and perform desired operations.

→ Computer is completely controlled by stored program, except for possible external interruption by an operator or by I/O devices connected to the machine.

Data → numbers and encoded characters used as operands by instructions

↓

any digital information

→ An entire program may be considered as data if it is to be processed by another program.

e.g. High-level language source program is input data to compiler program that

translates source program into a machine language program (object program).

Information handled by a computer must be encoded in a suitable format.

- binary
- Binary coded decimal (BCD)
- ASCII (American Standard Code for Information Interchange)
- EBCDIC (Extended Binary Coded Decimal Interchange Code)

(i) Input Unit

coded information is accepted by input unit

e.g. Keyboard, joysticks, trackballs, and mouses

Microphones → used to capture audio input

(ii) Memory Unit → to store programs and data

Two classes of storage
- primary
- secondary

⇒ Primary storage (essential & expensive) → fast memory that operates at electronic speeds.

Date / /
Page No.

Programs must be stored in memory while they are being executed.

memory → large no. of semiconductor storage cells (each capable of storing one bit of information)

→ These cells are processed in groups of fixed size called words.

→ To provide easy access to any word in the memory, a distinct address is associated with each word location.

Addresses are numbers that identify successive locations.

- A given word is accessed by specifying its address and issuing a control command that starts storage or retrieval process.

Word length of computer (range from 16 to 64 bits)
↳ no. of bits in each word

- Instructions and data can be written into memory or read out under control of processor.

Random Access Memory (RAM)
↳ any location can be reached in a short and fixed amount of time after specifying its address.

memory access time → time required to access one word
↓
fixed, independent of location of word being accessed

Small, fast, RAM units → caches
↓
- tightly coupled with processor
- often contained on same integrated chip to achieve high performance

Largest and slowest unit → main memory

⇒ Additional, cheaper, secondary storage is used

when large amount of data and many programs have to be stored, particularly for information that is accessed infrequently

e.g. magnetic disks and tapes
optical disks (CD-ROMs)

(iii) Arithmetic and Logic Unit (ALU)

Any arithmetic or logic operation is initiated by bringing required operands into processor

↓

Operation is performed by ALU in processor

→ When operands are brought into processor, they are stored in high-speed storage elements called registers

↓

each can store one word of data.

* Access times to registers are faster than access times to the fastest cache unit in memory hierarchy.

→ A single processor control a number of external devices.

(iv) Output Unit

↳ counterpart of input unit

e.g. printer, graphic displays

both input and output function

(v). Control Unit

↳ to coordinate different units of computer

- Actual timing signals that govern transfers are generated by control circuits.
- Data transfers b/w processor and memory are also controlled by control unit through timing signals.

→ A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.

Date / /
Page No.

Historical Development in Computers

- History of computer is considered with generations of a computer from first generation to fifth generation.

Charles Babbage → "father of Computer"
↳ designed Analytical Engine in 19th century that became basic framework of today's computer.

Imp. • Evolution of computers has been characterized by
  - increasing processor speed
  - decreasing component size
  - increasing memory size
  - increasing I/O capacity and speed

- factor responsible for greate increase in processor speed → shrinking size of microprocessor components
  (this reduces distance b/w components & hence increases speed)

Date / /
Page No.

- Improvement in speed is also because of heavy use of pipelining and parallel execution techniques

- Critical issue in computer system design is
  ↳ balancing performance of the various elements so that gains in performance in one area are not handicapped by a lag in other areas.

  e.g processor speed has increased more rapidly than memory access time

  ↳ variety of techniques used to compensate this mismatch
  (like including caches, wider data paths from memory to processor, and more intelligent memory chips)

Generations of Computers

	Technology used
1. First Generation (1946-1957)	Vacuum tube
2. Second Generation (1958-1964)	Transistor
3. Third Generation (1965-1971)	Integrated circuit
4. fourth Generation (1972-1980)	VLSI microprocessor
5. Fifth Generation (1980 onwards)	ULSI microprocessor

Date / /
Page No.

1. First Generation Computers (1946-1957)

→ used vacuum tubes as basic components for memory and circuitry for CPU.

→ These tubes produced a lot of heat, therefore, installations used to fuse frequently.
  ↳ very expensive (affordable by large organizations only)

→ Mainly batch processing operating system used.

→ Input & output devices → Punch cards
  → Paper tape
  → magnetic tape

→ Used machine code as programming language

Main features of 1st generation

- Vacuum Tube technology
- Unreliable
- Supported machine language only
- Very costly
- Generates lot of heat
- Slow input and output devices

- Huge size
- Need of AC
- Non-portable
- Consumes lot of electricity

Some computers of this generation were :

- ENIAC
- EDVAC
- UNIVAC
- IBM-701
- IBM-750

2. Second Generation Computers (1958-1964)

→ Transistors used
  - ↳ cheaper
  - ↳ consumed less power
  - ↳ more compact in size
  - ↳ more reliable
  - ↳ faster

  } as compared to vacuum tubes used in 1st generation

→ • Magnetic cores as primary memory
• magnetic tape and magnetic disks as secondary storage devices

→ Assembly language and high-level programming languages like FORTRAN, COBOL used

→ used batch processing and multiprogramming operating system.

Main features of 2nd generation

- Use of transistors
- More reliable
- Smaller size
- Generate less heat
- Consumed less electricity
- faster

} as compared to first generation computers

- Still very costly
- AC required
- Supported machine and assembly languages.

Some computers of this generation were:

- IBM 1620
- IBM 7094
- CDC 1604
- CDC 3600
- UNIVAC 1108

3. Third Generation Computers (1965-1971)

→ used Integrated Circuits (ICs)

↓ many transistors, resistors, and capacitors along with associated circuitry on a chip

Date / /
Page No.

IC $\xrightarrow{\text{invented by}}$ Jack Kilby

→ This led to computers smaller in size, reliable, and efficient.

→ Remote processing, time-sharing, multi-programming operating system were used.

→ High-level languages (FORTRAN-II to IV, COBOL, PASCAL PL/1, BASIC, ALGOL-68 etc.) used.

Main features of 3rd generation

- IC used
- More reliable in comparison to previous generations
- Smaller size
- Generated less heat
- Faster
- Lesser maintenance
- Costly
- AC required
- Consumed lesser electricity
- Supported high-level language

acitors
chip

Some computers of this generation were :

- IBM-360 series
- Honeywell - 6000 series
- PDP (Personal Data Processor)
- IBM-370/168
- TDC-316

4. Fourth Generation Computers (1972-1980)

→ Used Very Large Scale Integrated (VLSI) Circuits.

↓

about 5000 transistors and other circuit elements with their associated circuits on a single chip

↓

∴ possible to have microcomputer

→ more powerful, compact, reliable, and affordable

↳ Personal computer (PC) revolution

→ Time sharing, real time networks, distributed operating system used.

→ All high-level languages like C, C++, DBASE etc. used

Date / /
Page No.

Main features of 4th generation

- VLSI technology used
- Very cheap
- Portable and reliable
- Use of PCs
- Very small size
- Pipeline processing
- No AC required
- Concept of internet was introduced
- Great developments in the fields of networks
- Computers became easily available.

Some computers of this generation were :

- DEC 10
- STAR 1000
- PDP 11
- CRAY-1 (Super Computer)
- CRAY-X-MP (Super Computer)

5. Fifth Generation Computers (1980 onwards)

→ Ultra large Scale Integration (ULSI) technology used

↓

resulting in production of microprocessor chips having ten million electronic components

→ based on parallel processing hardware and AI (Artificial Intelligence) software.

↓

interprets means and methods of making computers think like human beings

→ All the high-level languages like C and C++, Java, .Net etc. used.

Main features of 5th Generation

- ULSI technology
- Development of true artificial intelligence
- Development of Natural language processing
- Advancement in Parallel Processing
- Advancement in Supercomputer technology
- More user-friendly interfaces with multimedia features
- Availability of very powerful and compact computers at cheaper rates

Some computers types of this generation:

- Desktop
- Laptop
- Notebook
- Ultrabook
- Chromebook

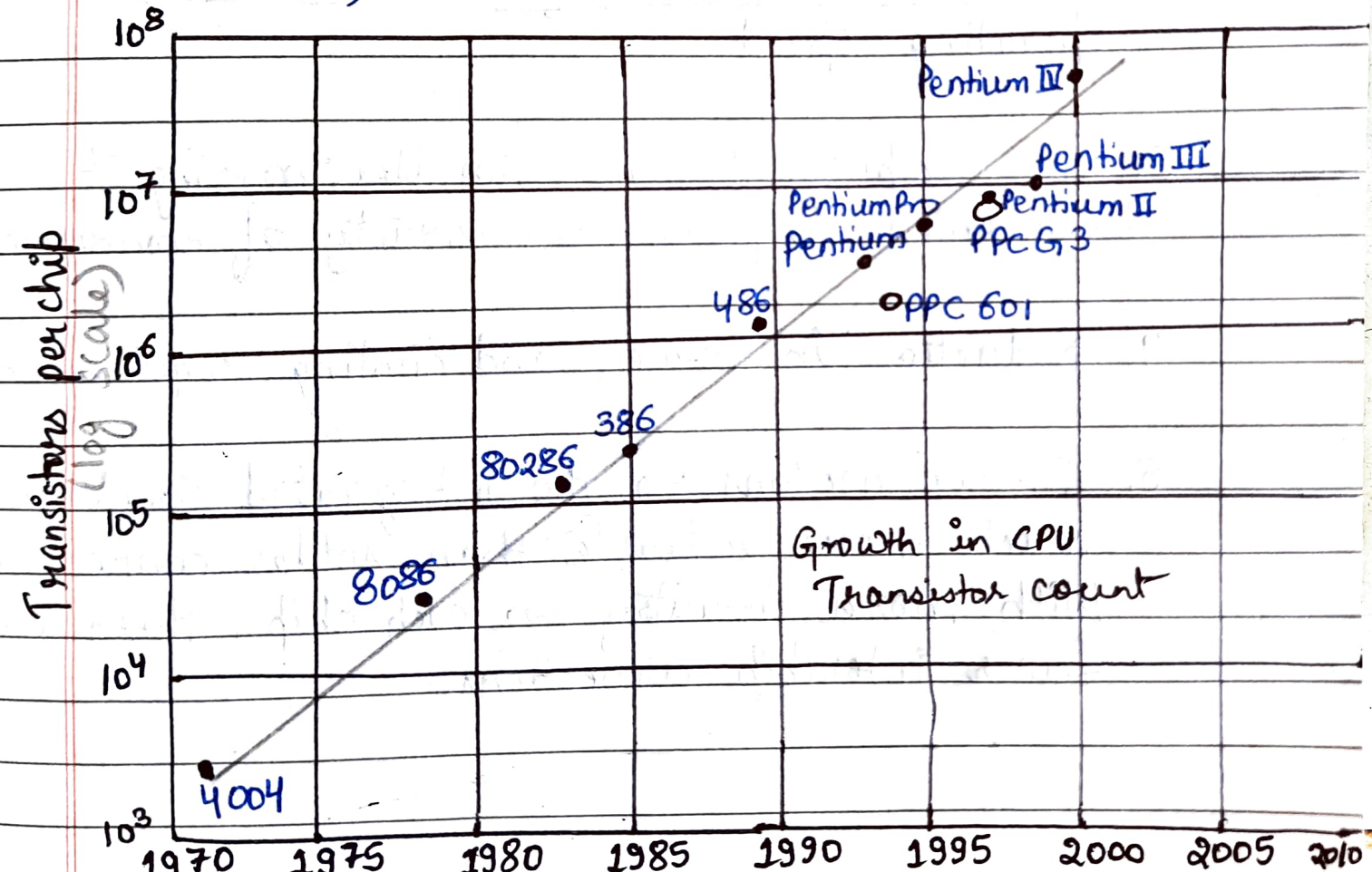
Date / /
Page No.

Moore's Law

→ Given by Gordon Moore, cofounder of Intel, in 1965.

→ Moore observed that number of transistors that could be put on a single chip was doubling every [redacted] years, though cost of computers is halved and correctly predicted that this pace would continue into near future.

→ The pace slowed to a doubling every 18 months in 1970s, but has sustained that rate ever since



Date / /
Page No.

Consequences of Moore's law

1. The cost of a chip has remained virtually unchanged during this period of rapid growth in density. This means cost of computer logic and memory circuitry has fallen at a dramatic rate.

2. Because logic and memory elements are placed closer together on more densely packed chips, the electrical path length is shortened, increasing operating speed.

3. The computer becomes smaller, making it more convenient to place in a variety of environments

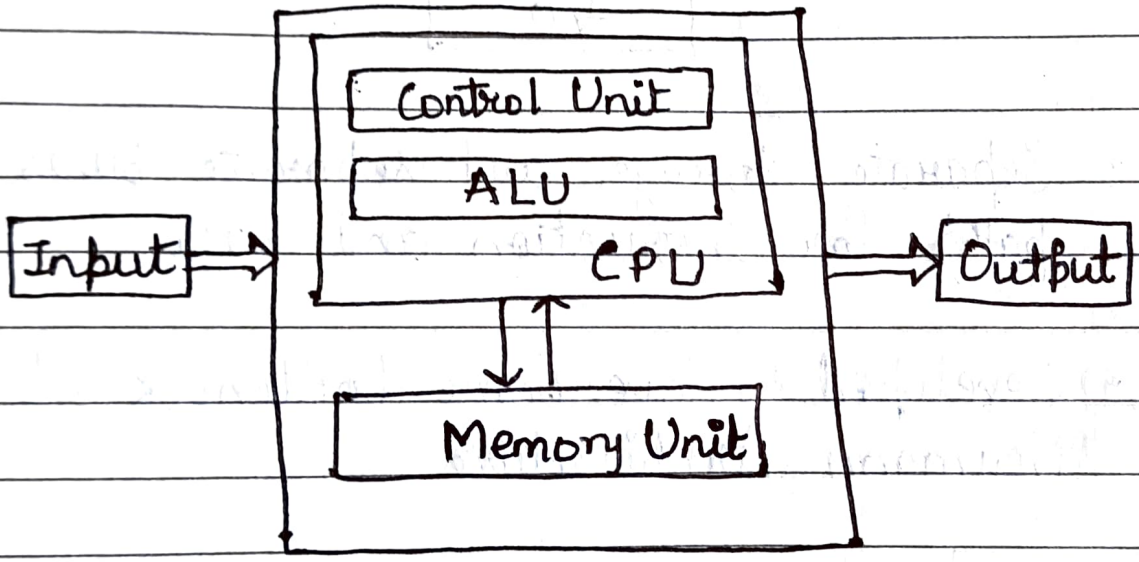
4. Reduction in power and cooling requirements

5. Interconnections on the integrated circuit are much more reliable than solder connections. With more circuitry on each chip, there are fewer interchip connections.

Classification of processor architectures based on their memory usage technique

- Von Neumann Architecture
- Non Von Neumann (Harvard) Architecture

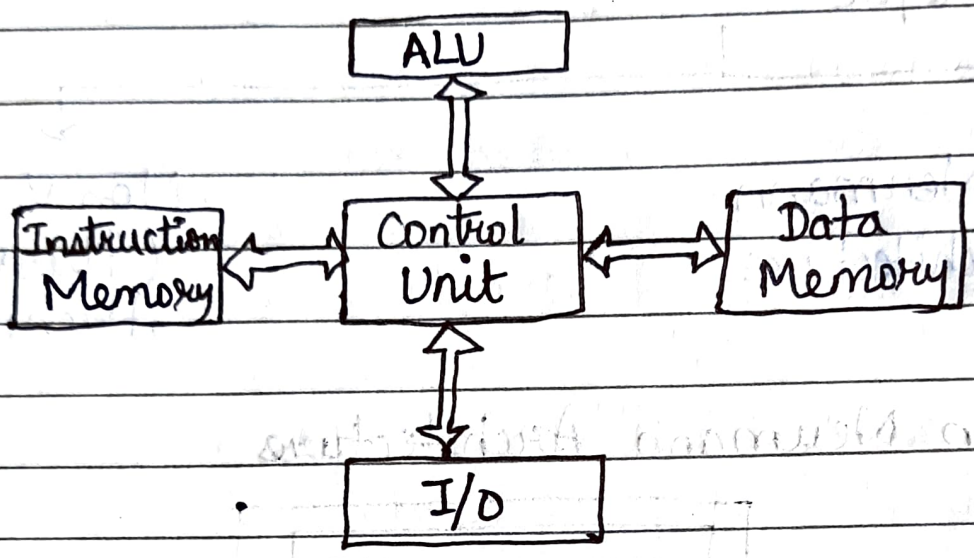
① Von Neumann Architecture



- Based on concept of stored program
- Program data and instruction data are stored in same memory.
- Developed by John Von Neumann in 1945

Date / /
Page No.

② Harvard Architecture



- Separate storage and separate buses (signal path) for instruction and data.
- Developed to overcome bottleneck of Von Neumann Architecture.
- Strict separation between data and code thus able to run a program and access data independently, and simultaneously.

Comparison between Von Neumann and Harvard Architecture

Point of Comparison	Von Neumann Architecture	Harvard Architecture
① Arrangement	No separate data & program memory. Single memory connection given to CPU	CPU is connected with both data memory (RAM) and program memory (ROM), separately
② Hardware requirements	Less hardware since a common memory needs to be accessed	More hardware required (separate data and address bus for each memory)
③ Space requirements	Less space	More space
④ Speed of execution	slower because processor cannot fetch data & instructions at same time	faster as processor fetches data and instructions simultaneously

Date / /
Page No 5

Point of Comparison	Von Neumann	Harvard
⑤. Clock cycles required for execution	An instruction is executed in two cycles	Single clock cycle is required to execute single instruction
⑥. Space usage	Space is not wasted because space of data memory can be utilized by instructions memory and vice-versa	It results in wastage of space since if space is left in data memory then instructions memory cannot use that space of data memory & vice-versa
⑦. Controlling	simpler controlling as either data or instructions are to be fetched at a time	Complex controlling since data and instructions are to be fetched simultaneously
⑧. Applications	Used in personal computers and small computers	Used in microcontrollers and signal processing

Evolution of Intel Microprocessors / x86 Architecture

- **8080**: 1st general purpose microprocessor
  8-bit machine with 8-bit data path to memory
  used in 1st personal computer, the Altair
- **8086**: → more powerful 16-bit machine
  → Larger registers
  → used an instruction queue that prefetches few instructions before they are executed
  → its variant 8088 used in IBM's 1st PC
  → 1st appearance of x86 architecture
- **80286**: extension of 8086 enabled addressing a 16 MByte memory instead of just 1MByte
- **80386**: Intel's 1st 32-bit machine
  1st Intel processor to support multitasking (run multiple programs at same time)
- **80486**: use of more powerful cache technology & instruction pipelining
  → also offered a built-in math coprocessor, offloading complex math operations from main CPU.
- **Pentium**: use of superscalar techniques allowing multiple instructions to execute in parallel
- **Pentium Pro**: superscalar organization with use of register renaming, branch prediction, data flow analysis & speculative execution
- **Pentium II**: incorporated Intel MMX technology designed to process video, audio & graphics data efficiently

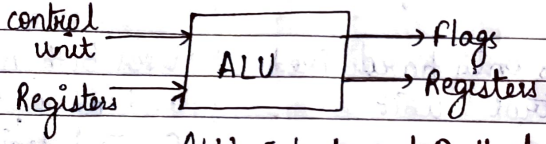
- Pentium III: additional floating-point instructions to support 3D graphics
- Pentium 4: additional floating-point & other enhancements for multimedia
- Core: 1st Intel x86 microprocessor with dual core, referring to implementation of two processors on a single chip.
- Core 2: extends architecture to 64 bits.
  → Core 2 Quad provides four processors on a single chip

Processor	No. of Transistors	Clock Speed	Address Bus	Data Bus	Addressable memory
8080	6000	2MHz	16-bit	8-bit	64K
8085	6500	5MHz	16-bit	8-bit	64K
8086	29000	5MHz	20-bit	16-bit	1M
8088	29000	5MHz	20-bit	8-bit*	1M
80286	134000	8MHz	24-bit	16-bit	16M
80386	275000	16MHz	32-bit	32-bit	4G
80486	1.2M	25MHz	32-bit	32-bit	4G
Pentium	3.1M	60MHz	32-bit	32/64 bit	4G
Pentium Pro	5.5M	150MHz	36-bit	32/64-bit	64G
Pentium II	8.8M	233MHz	36-bit	64-bit	64G
Pentium III	9.5M	650MHz	36-bit	64-bit	64G
Pentium 4	42M	1.4GHz	36-bit	64-bit	64G

* External 8-bit and internal 16-bit data bus

Arithmetic and Logic Unit (ALU)

↳ part of computer that actually performs arithmetic and logical operations on data



ALU Inputs and Outputs

ALU and all electronic components in computer are based on use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations.

Addition and Subtraction of Signed Numbers

Let X and Y be two numbers to be added

Let x_i and y_i be equally weighted bits in X and Y

Let c_i be carry-in to i^{th} stage or carry out from $(i-1)^{st}$ stage

S_i = Sum of i^{th} stage

C_{i+1} = carry out of i^{th} stage

C_{i+1} S_i C_i → i^{th} stage

Date / /
Page No. 6

Truth Table

x_i	y_i	Carry in c_i	Sum S_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

K-map simplification for S_i

x_i \ $y_i c_i$	00	01	11	10
0		1		1
1	1		1	

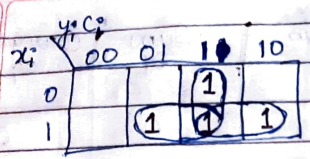
$$S_i = \bar{x}_i y_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i$$

$$= c_i(\bar{x}_i \bar{y}_i + x_i y_i) + \bar{c}_i(\bar{x}_i y_i + x_i \bar{y}_i)$$

$$S_i = c_i(\overline{x_i \oplus y_i}) + \bar{c}_i(x_i \oplus y_i)$$

$$\therefore \boxed{S_i = x_i \oplus y_i \oplus c_i}$$

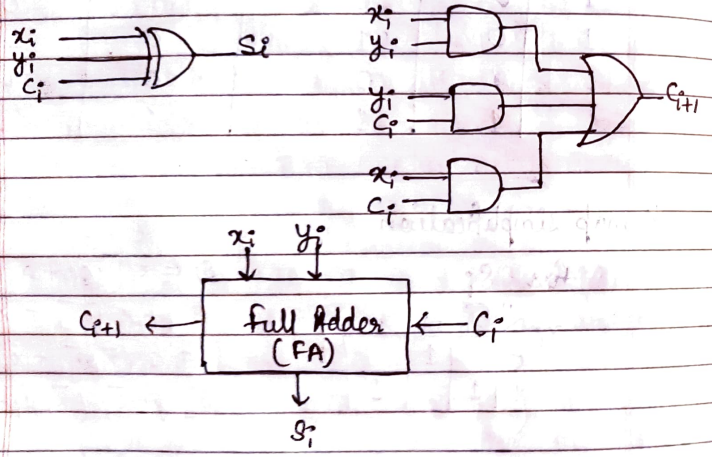
for C_{i+1}



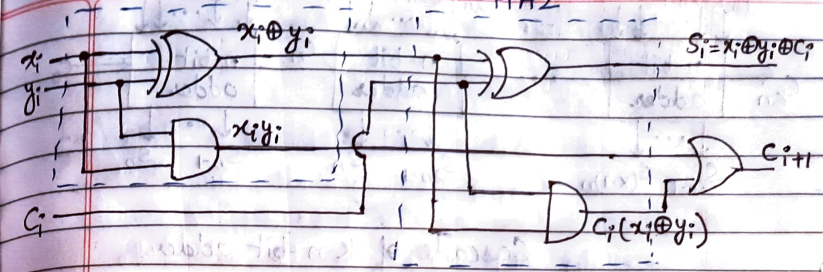
$$C_{i+1} = x_i y_i + y_i c_i + x_i c_i$$

$$C_{i+1} = x_i y_i + c_i (x_i \oplus y_i)$$

Single stage of addition is done using full adder (FA) circuit



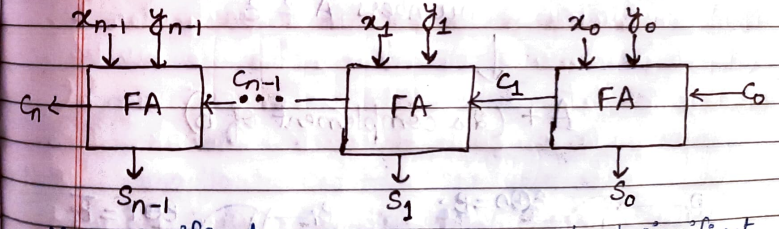
Logic for a single stage (using full Adder)



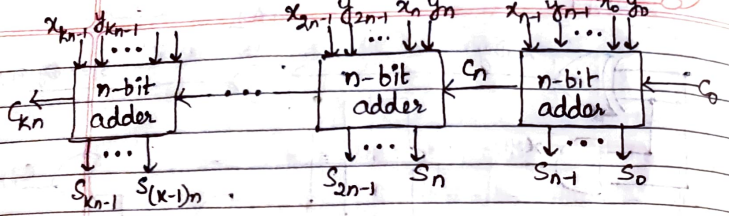
full adder circuit using two half adders and one OR gate

n-bit ripple-carry adder

n full adder blocks are cascaded to form an n-bit ripple-carry adder used to add two n-bit numbers.



An n-bit ripple-carry adder



Cascade of k n-bit adders

↓

Capable of handling input numbers that are kn bits long

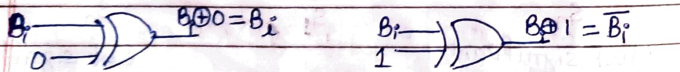
Addition/Subtraction Logic Unit

A − B

→ Subtraction on 2's complement numbers A & B

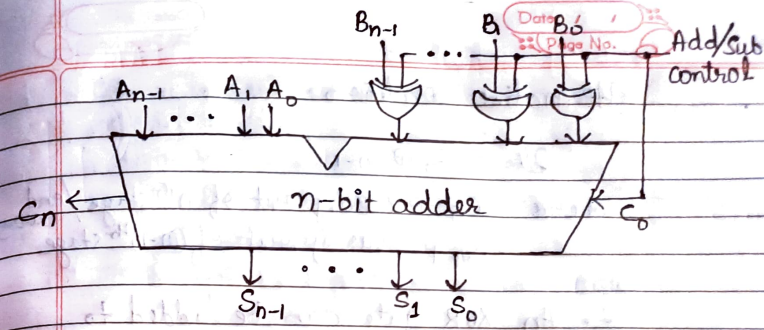
↓

A + (2's complement of B)



Acting as buffer

Acting as inverter



Binary addition-subtraction logic network

when Add/sub = C_0 = 0 ⇒ A + B

Add/sub = C_0 = 1 ⇒ A + (1's complement of B) + 1

(1's complement of B) → 2's complement

∴ A + (−B)

→ in 2's complement form

Note: Overflow condition (occur when sign of operand are same but that of result is different)

$$Ov = A_{n-1} B_{n-1} \overline{S}_{n-1} + \overline{A}_{n-1} \overline{B}_{n-1} S_{n-1}$$

here A_{n-1} & B_{n-1} are sign bits of A & B

S_{n-1} → sign bit of sum

Alternative condition

$OV = C_n \oplus C_{n-1}$

where C_n = carry out of n^{th} stage / end carry
C_{n-1} = carry out of $(n-1)^{th}$ stage

∴ An XOR gate can be added to binary addition-subtraction logic to detect overflow condition $C_n \oplus C_{n-1}$.

Disadvantage of Ripple-Carry Adder
↳ too much delay in developing its outputs, S_0 through S_{n-1} and C_n

→ Delay through any combinational logic network constructed from gates in a particular technology is determined by adding up the number of logic-gate delays along longest signal propagation path through network.

→ In case of n-bit ripple-carry adder, the longest path is from i/p's x_0, y_0 and c_0 at LSB position to outputs C_n and S_{n-1} at most significant bit (MSB) position.

C_{n-1} is available in $2(n-1)$ gate delays

S_{n-1} is available in $2(n-1)$ gate delays + 1 XOR gate delay

final carry out C_n = $2n$ gate delays

∴ If a ripple-carry adder is used to implement the addition/subtraction unit, all sum bits are available in $2n$ gate delays, including delay through XOR gates on B i/p

Using implementation $C_n \oplus C_{n-1}$ for overflow this indicator is available after $2n+2$ gate delays.

Two approaches to reduce delay in adders:
(i) Use of fastest possible electronic technology in implementing ripple-carry logic design or variations of it
(ii) Use of an augmented logic gate network structure that is larger than ripple carry adder.

Fast Adder

Carry Look Ahead Addition

For fast addition → carry signals generation must speed up

$$S_i = x_i \oplus y_i \oplus c_i$$

and $$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$
$$= x_i y_i + (x_i \oplus y_i) c_i$$

$$\therefore \quad c_{i+1} = G_i + P_i c_i$$
$$S_i = P_i \oplus c_i$$

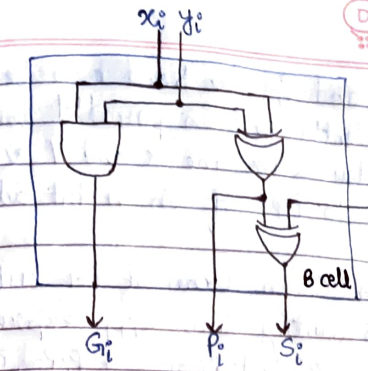
where $G_i = x_i y_i$ and $P_i = x_i \oplus y_i$

G_i: generate function for stage i

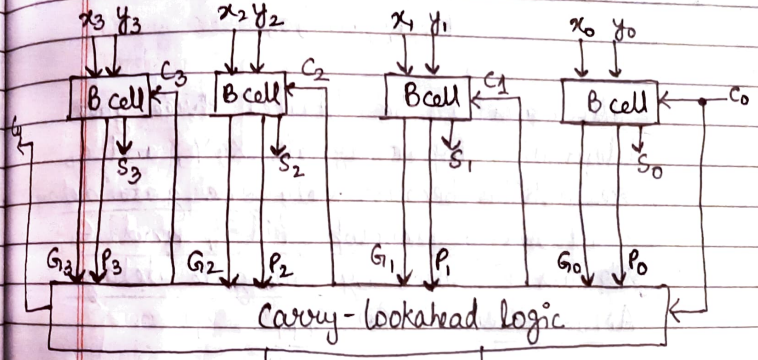
P_i: propagate function for stage i

x_i	y_i	c_i	c_{i+1}	Condition
0	0	0	0	no carry generation
0	0	1	0	
0	1	0	0	
0	1	1	1	Carry propagate
1	0	0	0	
1	0	1	1	
1	1	0	1	carry generate
1	1	1	1	

from c_i to c_{i+1}



ⓐ. Bit stage cell



$$G_0^I = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

$$P_0^I = P_3 P_2 P_1 P_0$$

ⓑ. 4-bit adder

4-bit carry-lookahead adder

→ All G_i and P_i functions can be formed independently and in parallel in one logic gate delay after the X and Y vectors are applied to i/ps of an n-bit adder.

Expanding C_i in terms of i-1 subscripted variables and substituting into C_{i+1} expression,

$$C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} C_{i-1}$$

$$C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \ldots + P_i P_{i-1} \ldots P_1 G_0 + P_i P_{i-1} \ldots P_0 C_0$$

∴ All carries can be obtained three gate delays after input signals X, Y, and C_0 are applied because only one gate delay is needed to develop all P_i and G_i signals followed by two gate delays in AND-OR circuit for C_{i+1}.

→ After a further XOR gate delay, all sum bits are available.

∴ Independent of n, the n-bit addition process requires only four gate delays.

for 4-bit adder

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

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

$$C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$

The carries are implemented in the block labeled carry-lookahead logic.

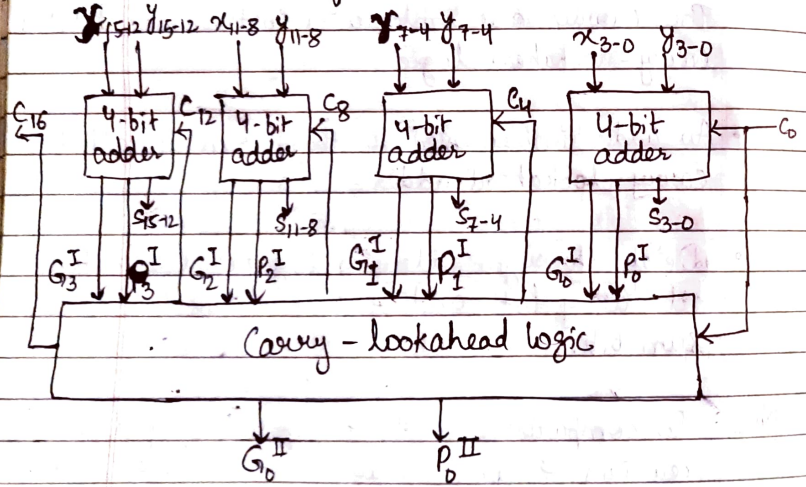
- An adder implemented in this form is called a carry-lookahead adder.
- Delay through adder is 3 gate delays for all carry bits and 4 gate delays for all sum bits.

Note: In comparison, a 4-bit ripple carry adder requires 7 gate delays for S_3 and 8 gate delays for C_4.

→ Extending carry-lookahead adder for longer operands is difficult due to problem of gate fan-in constraints

→ However a number of 4-bit adders can be cascaded to build longer adders

four
~~Eight~~ 4-bit carry-lookahead adders can be connected to form a 16-bit adder.



16-bit carry-lookahead adder built from 4-bit adders

$$C_{16} = G_3^I + P_3^I G_2^I + P_3^I P_2^I G_1^I + P_3^I P_2^I P_1^I G_0^I + P_3^I P_2^I P_1^I P_0^I C_0$$

Delay in developing carries produced by carry-lookahead circuits is two gate delays more than the delay needed to develop G_K^I and P_K^I functions

G_K^I and P_K^I require two gate delays and one gate delay, respectively, after generation of G_i and P_i

∴ all carries produced by carry-lookahead circuits are available 5 gate delays after X, Y, and c_0 are applied as inputs.

S_{15} is generated inside high-order 4-bit block in two gate delays after C_{12}, followed by S_{15} in one further gate delay.

∴ S_{15} is available after 8 gate delays

Note • for 16-bit carry lookahead adder

delay in C_{16} = 9 gate delays
delay in S_{15} = 8 gate delays

• For 16-bit ripple carry adder

delay in C_{16} = 32 gate delays
delay in S_{15} = 31 gate delays

Multiplication of Positive Numbers

e.g

```
    1 1 0 1      (13)10  Multiplicand M
  X 1 0 1 1      (11)10  Multiplier Q
  ---------
    1 1 0 1  --> P1
  1 1 0 1    --> P2
  0 0 0 0    --> P3
1 1 0 1      --> P (product)
---------------
1 0 0 0 1 1 0 1   (143) Product P
```

Manual multiplication algorithm

• Product of two n-bit numbers can be accomodated in 2n digits

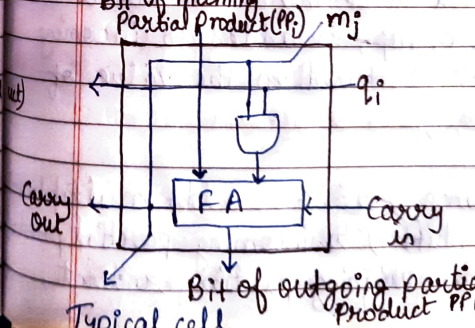
• multiplication of multiplicand by one bit of multiplier is done in steps

* If multiplier bit = 1 ⇒ multiplicand entered in appropriate position to be added to partial product

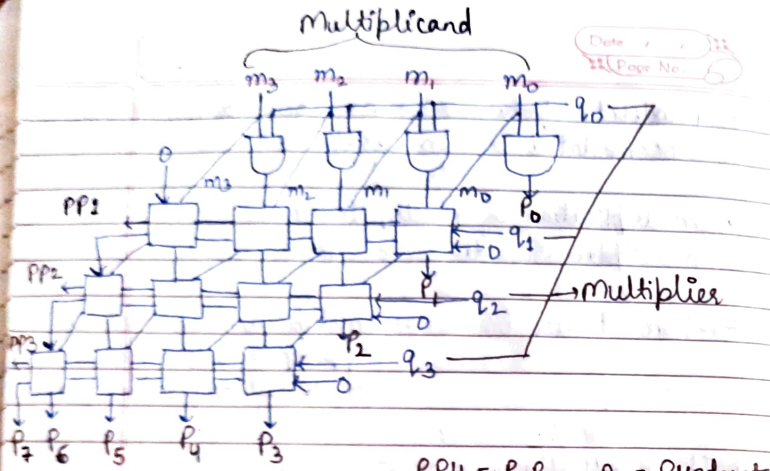
* If multiplier bit = 0 ⇒ 0s are entered

initial partial product is 0 (P_0)

Hardware → combinational 2-D logic array



FA → full adder
m_j → multiplicand bit
q_i → multiplier bit



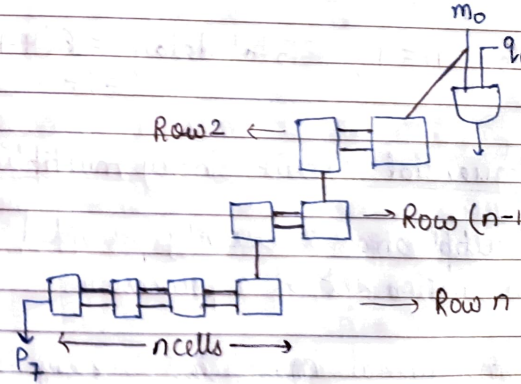
$PP4 = P_7P_6 \ldots P_0$ = product

Array implementation for multiplication of positive binary numbers (4-bit numbers)

- AND gate in each cell determines whether a multiplicand bit, m_j, is added to incoming partial product bit, based on the value of multiplier bit, q_i.
- Each row i, where $0 \le i \le 3$, adds multiplicand (appropriately shifted) to incoming partial product PP_i, to generate outgoing partial product, $PP(i+1)$, if $q_i = 1$.
- If $q_i = 0$, PP_i is passed vertically downward unchanged.

Initial partial product PP_0 is all 0s
PP_4 is desired product

The worst case signal propagation delay path is from upper right corner of array to the high-order product bit output at the bottom left corner of array



Assume there are two gate delays from i/p to o/p of FA

Each of two FA blocks in row 2 through n-1 introduces 2 gate delays, for a total of 4(n-2) gate delays.

Row n introduces 2n gate delays

Adding in initial AND gate delay for row 1 and other cells,

$$\text{total delay} = 4(n-2)+2n+1$$
$$= 6n-8+1$$
$$\boxed{\text{total delay} = 6(n-1)-1}$$

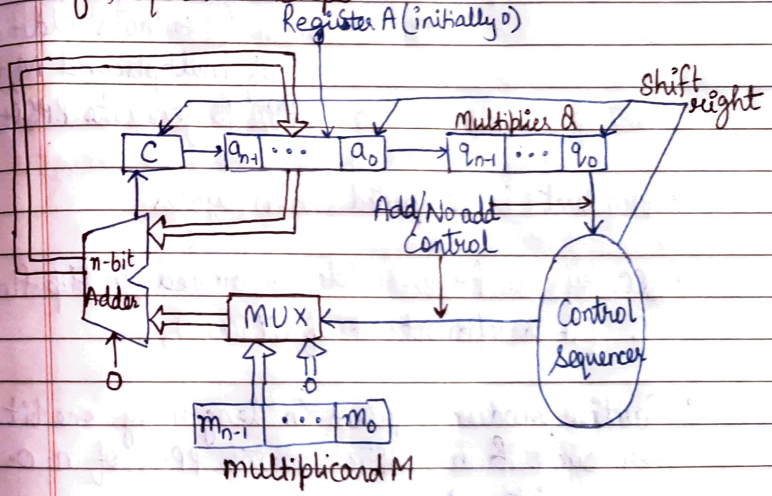
for $n=4$; Total delay $= 6(4-1)-1$
$= 17$

Sequential circuit binary multiplier

- Multiplication is usually provided in machine instruction set of a processor.
- Combinational multiplier is easy to understand but it uses many gates for multiplying numbers of practical size (32- or 64-bit numbers)

∴ Multiplication can also be performed using a mixture of combinational array techniques and sequential techniques requiring less combinational logic.

Simplest way to perform multiplication is to use adder circuitry in ALU for a number of sequential steps



Register configuration for sequential circuit binary multiplier

This circuit performs multiplication by using a single n-bit adder n times to implement spatial

addition performed by n rows of ripple carry adders.

- Register A and Q (combined) → hold PP_i value
- multiplier bit q_i → generates Add/No add signal (controls addition of multiplicand M, to PP_i to generate $PP_{(i+1)}$)
- Product is computed in n cycles.
- Carry out from adder is stored in flip-flop C (it is leftmost bit of $PP_{(i+1)}$).
- Partial product grows in length by one bit per cycle from initial vector PP_0, of n 0s in register A.
- At end of each cycle, C, A, and Q are shifted right one bit position to allow for growth of partial product as multiplier is shifted out of register Q.

M = 1101

C	A	Q		
0	0000	1011		Initial configuration
0	1101	1011	Add	First cycle
0	0110	1101	Shift	
1	0011	1101	Add	2nd cycle
0	1001	1110	Shift	
0	1001	1110	No add	3rd cycle
0	0100	1111	Shift	
1	0001	1111	Add	4th cycle
0	1000	1111	Shift	

Product

Multiplication Example

- After they are used, multiplier bits are discarded by right-shift operation.
- After n cycles, high-order half of product is held in register A and low-order half is in register Q.

∴ Multiply instruction takes much more time to execute than an Add instruction.

→ Several techniques have been used to speed up multiplication.

Signed - Operand Multiplication

↳ multiplication of 2's-complement signed operands generating double length product

General strategy → to accumulate partial products by adding versions of multiplicand as selected by multiplier bits

e.g. positive multiplier $= (+11)_{10}$
negative multiplicand $= (-13)_{10}$

```
      1 0 0 1 1        (-13)10
    X 0 1 0 1 1        (11)10
  ---------------
  1 1 1 1 1 0 0 1 1
  1 1 1 1 1 0 0 1
  0 0 0 0 0 0 0
  1 1 1 0 0 1 1
  0 0 0 0 0 0
 ---------------------
  1 1 0 1 1 1 0 0 0 1    (-143)10
```

• While adding a negative multiplicand to a partial product, sign-bit value of multiplicant must be extended to left as far as product will extend.

∴ Hardware discussed earlier can be used for negative multiplicands if it provides for sign extension of partial products.

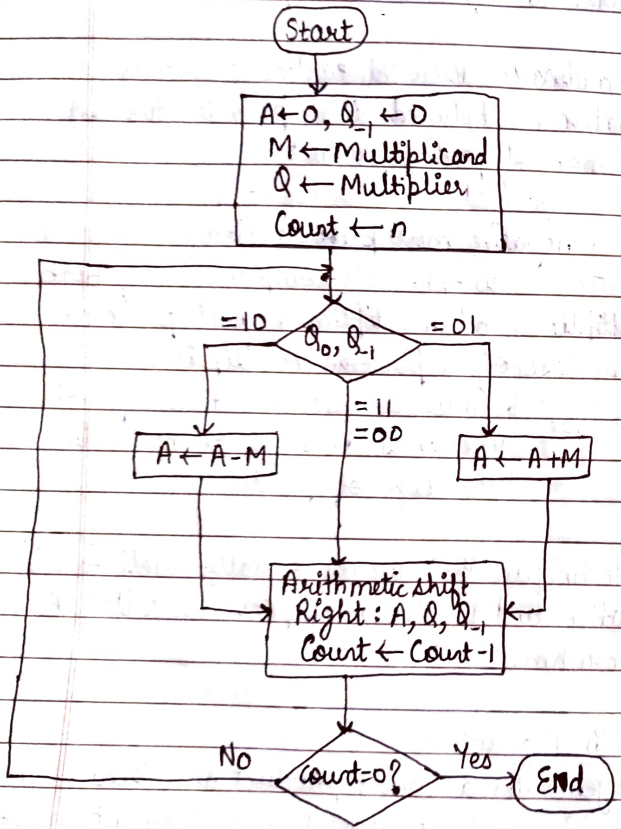
• For a negative multiplier, a straightforward solution is to form 2's-complement of both multiplier and multiplicand and proceed as in case of a positive multiplier.
↳ possible because complementation of both operands does not change value or the sign of product

A technique that works equally well for both negative and positive multipliers is called Booth algorithm.

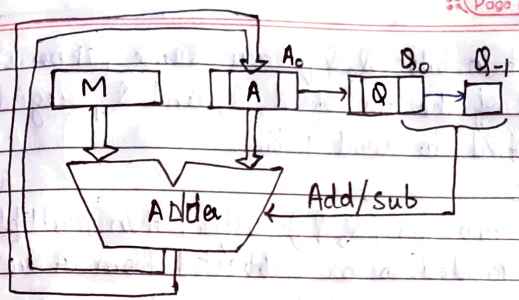
Booth Algorithm

↳ generates a 2n-bit product and treats both positive and negative 2's-complement n-bit operands uniformly.

- This algorithm has benefit of speeding up multiplication process



Booth's Algorithm for 2's Complement Multiplicatio



Hardware Structure Implementing Booth's Algorithm

- Multiplier and multiplicand are placed in Q and M registers, respectively.
- Q_{-1} → 1-bit register placed logically to right of LSB (Q_0)
- Result of multiplication will appear in A and Q registers.
- A and Q_{-1} are initialized to 0.

Control logic scans bits of multiplier one at a time. As each bit is examined, bit to its right is also examined.

If two bits ($Q_0 Q_{-1}$) are same (11 or 00), then all of the bits of A, Q, and Q_{-1} registers are shifted to right 1 bit.

If two bits ($Q_0 Q_{-1}$) differ, then multiplicand is added to or subtracted from A register, depending on whether two bits are 01 or 10. Following addition or subtraction, right shift occurs.

Right shift is such that leftmost bit of A, i.e., A_{n-1}, not only is shifted into A_{n-2}, but also remains in A_{n-1}. This is required to preserve sign of number in A and Q. It is known as an arithmetic shift, as it preserves sign bit.

e.g. (i). Multiplicand (M) $= (7)_{10} = (0111)_2$
Multiplier (Q) $= (3)_{10}$
operation to be performed $\longrightarrow 7 \times 3$

$$(-M) = (1001)_2$$

	Multiplier			Multiplicand	
n	A	Q	Q_{-1}	M	Initial Values
04	0000	0011	0	0111	
04	1001	0011	0	0111	$A \leftarrow A-M$ (Arithmetic) } First cycle
03	1100	1001	1	0111	Arithmetic Shift }
02	1110	0100	1	0111	Arithmetic Shift } 2nd cycle
02	0101	0100	1	0111	$A \leftarrow A+M$ (Arithmetic) } 3rd cycle
01	0010	1010	0	0111	Arithmetic Shift }
00	0001	0101	0	0111	Arithmetic Shift } 4th cycle

product

Note: Efficiency of algorithm is good. Blocks of 1s or 0s are skipped over, with an aver of only one addition or subtraction per bl

e.g. (ii). $M = (-7)_{10} = (1001)_2$; $(-M) = (0111)$
$Q = (+3)_{10}$

n	A	Q	Q_{-1}	M	
4	0000	0011	0	1001	→ Initial value
	0111	0011	0	1001	$A \leftarrow A-M$ } 1
3	0011	1001	1	1001	ASR AQQ_{-1} }

n	A	Q	Q_{-1}	M	
2	0001	1100	1	1001	ASR AQQ$_{-1}$ } 2nd cycle
	1010	1100	1	1001	A ← A+M } 3rd cycle
1	1101	0110	0	1001	ASR AQQ$_{-1}$ }
0	1110	1011	0	1001	ASR AQQ$_{-1}$ } 4th cycle

product

product = 11101011
= $(-21)_{10}$

eg (iii) $M = (7)_{10} = (0111)_2$
$Q = (-3)_{10} = (1101)_2$

$(-M) = (1001)_2$

n	A	Q	Q_{-1}	M	
4	0000	1101	0	0111	⟶ Initial values
	1001	1101	0	0111	⟶ A ← A-M } 1st cycle
3	1100	1110	1	0111	ASR AQQ$_{-1}$ }
	0011	1110	1	0111	A ← A+M } 2nd cycle
2	0001	1111	0	0111	ASR AQQ$_{-1}$ }

n	A	Q	Q_{-1}	M	
	1010	1111	0	0111	A ← A-M } 3rd cycle
1	1101	0111	1	0111	ASR AQQ$_{-1}$ }
0	1110	1011	1	0111	ASR AQQ$_{-1}$ } 4th cycle

product

product = AQ = 11101011
= $(-21)_{10}$

eg (iv) $M = (-7)_{10} = (1001)_2$
$Q = (-3)_{10} = (1101)_2$

$(-M) = (0111)_2$

n	A	Q	Q_{-1}	M	
4	0000	1101	0	1001	⟶ Initial values
	0111	1101	0	1001	A ← A-M } 1st cycle
3	0011	1110	1	1001	ASR AQQ$_{-1}$ }
	1100	1110	1	1001	A ← A+M } 2nd cycle
2	1110	0111	0	1001	ASR AQQ$_{-1}$ }
	0101	0111	0	1001	A ← A-M } 3rd cycle
1	0010	1011	1	1001	ASR AQQ$_{-1}$ }

	A	Q	Q_{-1}	M	
0	0000	0101	1	1001	
	product				ASR AQQ_{-1} } 4th cycle

product = AQ = $(00010101)_2$
= $(21)_{10}$

Integer Division

```
13 | 274 | 21
   26
   ---
    14
    13
    --
     1
```

```
          10101
1101 | 100010010
       01101
       -----
       10000
        1101
        ----
        1110
        1101
        ----
           1
```

Longhand division examples

Ckt implementing this longhand method operates as follows.

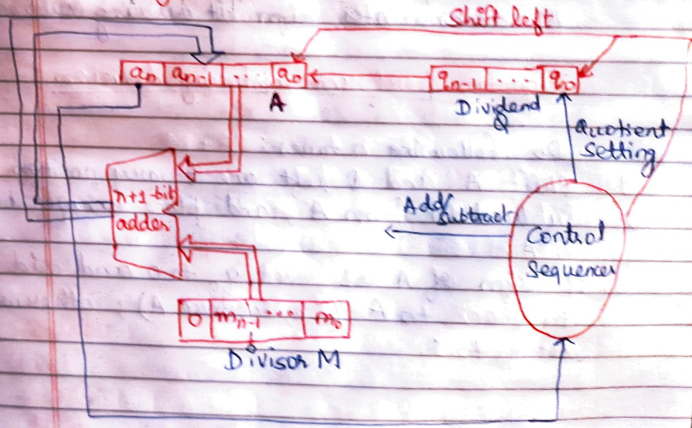
- It positions divisor appropriately with respect to dividend & performs a subtraction

Case 1: If remainder is zero or +ve
→ quotient bit = 1

remainder is extended by another bit of dividend, divisor is repositioned & another subtraction is performed.

Case 2: If remainder is -ve
↳ quotient bit = 0
dividend is restored by adding back divisor and divisor is repositioned for another subtraction.

① Restoring division



M ← n-bit +ve divisor

Q ← n-bit +ve dividend

Reg. A set to 0 initially

After division, n-bit quotient → reg. Q
remainder → reg. A

→ Required subtractions are facilitated by using 2's-complement arithmetic

Extra bit position at left end of both A & M accomodates sign bit during subtraction

Algo to perform restoring division

Do following n times:

1. Shift A and Q left one binary position
2. Subtract M from A, and place answer back in A
3. If sign of A is 1, set q_0 to 0 and add M back to A (i.e, restore A); otherwise set q_0 to 1.

restoring division example

$$\begin{array}{r} 1 \\ 11 \overline{)1000} \\ 11 \\ \hline 10 \end{array}$$

	A	Q	M	
Initially	0 0 0 0 0	1 0 0 0	0 0 0 1 1	1st cycle
Shift left	0 0 0 0 1	0 0 0 □		
Subtract	1 1 1 0 1			← using 2's complement
Set q_0	1 1 1 1 0			
Restore	0 0 0 0 1	0 0 0 0		(bcoz remainder -ve)
Shift	0 0 0 1 0	0 0 0 □		2nd cycle
Subtract	1 1 1 0 1			
Set q_0	1 1 1 1 1			
Restore	0 0 0 1 0	0 0 0 0		
Shift	0 0 1 0 0	0 0 0 □		3rd cycle
Subtract	1 1 1 0 1			
Set q_0	0 0 0 0 1			
[illegible]		0 0 0 1		(bcoz remainder +ve)
Shift	0 0 0 1 0	0 0 1 1		yth cycle
Subtract	1 1 1 0 1	0 0 1 □		
Set q_0	1 1 1 1 1			
Restore				

000 10 0 1 0

Remainder Quotient

Nonrestoring Division

Restoring division algorithm can be improved by avoiding need for restoring A after an unsuccessful subtraction

↓ -ve result

In restoring algo, sequence of operations after subtraction operation

- If A is +ve → shift left and subtract M
  i.e., 2A - M
- If A is -ve → restore A by performing A+M and then shift it left and subtract M
  i.e., 2(A+M) - M = 2A + M

q_0 bit is appropriately set to 0 or 1 after correct operation has been performed.

This can be summarized in following algorithm for nonrestoring division

Step 1: Do the following n times:

1. If sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
2. Now if sign of A is 0, set q_0 to 1; otherwise set q_0 to 0.

Step 2: If sign of A is 1, add M to A.

Step 2 is needed to leave proper positive remainder in A at the end of n cycles of Step 1.

Note: Restore operations are no longer needed, and exactly one Add or Subtract operation is performed per cycle.

-M = 11101

non-restoring division example

	A	Q	M	
Initially	00000	1000	00011	
shift	00001	000□		1st cycle
Subtract	11101			
Set q_0	(1)1110	0000		
shift	11100	00□□		2nd cycle
Add	00011			
Set q_0	(1)1111	00□□		

```
Shift   1 1 1 1 0      0 0 0 0   } 3rd cycle
Add     0 0 0 1 1
Set q0  (0) 0 0 0 1    0 0 0 1 ↑

Shift   0 0 0 1 0      0 0 1 0   } 4th cycle
Subtract 1 1 1 0 1
Set q0  (1) 1 1 1 1    0 0 1 0
                       Quotient

Add     1 1 1 1 1      } Restore
        0 0 0 1 0        Remainder
        0 0 0 1 0
        Remainder
```

Floating Point Numbers

Representing fractional numbers

↳ binary no. with fractional part

$$B = b_{n-1} b_{n-2} \cdots b_1 b_0 \cdot b_{-1} b_{-2} \cdots b_{-m}$$

corresponds to decimal numbers

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

- Also called fixed-point numbers because position of radix point is fixed.

If radix point is allowed to move, it is called a floating point representation.

Some Examples

$1011.1 \longrightarrow 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 + 1\times2^{-1} = 11.5$

$101.11 \longrightarrow 1\times2^2 + 0\times2^1 + 1\times2^0 + 1\times2^{-1} + 1\times2^{-2} = 5.75$

$10.111 \longrightarrow 1\times2^1 + 0\times2^0 + 1\times2^{-1} + 1\times2^{-2} + 1\times2^{-3} = 2.875$

Some observations

- Shift right by 1 bit means divide by 2
- Shift left by 1 bit means multiply by 2
- Numbers of form $0.111111\ldots_2$ has a value less than 1.0.

Limitations of Representation

- In fractional part, only nos of form $\frac{x}{2^k}$ can be represented exactly
  * other numbers have repeating bit represent (i.e never converge)

e.g. $3/4 = 0.11$

$7/8 = 0.111$

$5/8 = 0.101$

$1/3 = 0.1010101010101\ [01]\ldots$

$1/5 = 0.00110011001100\ [0011]\ldots$

$1/10 = 0.0001100110011\ [0011] \ldots$

- More no. of bits, more accurate is the representation

* processor having specified finite no of bits for representation may have error.

  sometimes $\left(\frac{1}{3}\right) * 3 \neq 1$

Floating-point Number Representation
(IEEE-754)

- In fixed point representation for representing number with fractional parts, fractional point is assumed somewhere in between number
  → n bits is integer part
  → m bits is fraction part
  → lacks flexibility
  * Cannot be used to represent very small or very large numbers.

Sol" use floating-point number representation

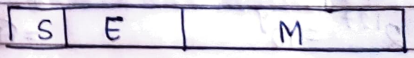
- Number F is represented as a triplet $<S, M, E>$ such that

$$F = (-1)^S M \times 2^E$$

- S is the sign bit indicating whether number is negative (=1) or positive (=0).
- M is called mantissa, and is normally a fraction in range $[1.0, 2.0)$.
- E is called exponent, which weights number by power of 2.

Encoding:

- Single-precision numbers: total 32 bits
  E → 8 bits
  M → 23 bits
- Double-precision numbers: total 64 bits
  E → 11 bits
  M → 52 bits



Note: • Number of significant digits depends on number of bits in M
→ 7 significant digits for 24-bit mantissa (23 bits + 1 implied bit)

1. .... → ∵ M is in range [1.0, 2.0)
23 bits
↓
implied bit (always starts with 1)
↳ (so no need to represent it separately)

∴ 24 bit mantissa represented in 23 bits.

How many significant digits?

$2^{24} = 10^x$

$24 \log_{10} 2 = x \log_{10} 10$

$\boxed{x = 7.2}$ → 7 significant decimal places

- Range of number depends on number of bits in E.

→ 10^{38} to 10^{-38} for 8-bit exponent
↳ in 2's complement representation
$= -2^{8-1}$ to $+(2^{8-1}-1)$
$= -128$ to $+127$
Range

Range of exponent?

$2^{127} = 10^y$

$127 \log_{10} 2 = y \log_{10} 10$

$\boxed{y = 38.1}$ → maximum exponent value 38 (in decimal)

Normalized Representation

→ how E and M are actually encoded

→ Assume that actual exponent of number is EXP (i.e., number is $M \times 2^{EXP}$)

→ Permissible range of E: $1 \leq E \leq 254$
(all-0 and all-1 patterns are not allowed)

- **Encoding of exponent E**

The exponent is encoded as a biased value

$E = EXP + BIAS$

where BIAS = 127 i.e. $(2^{8-1}-1)$ for single precision

BIAS = 1023 i.e. $(2^{11-1}-1)$ for double precision

EXP: −126 to +127 (leaving all-0 & all-1 patterns)

if BIAS = 127

$E = 1$ to 254

unsigned +ve no.

- **Encoding of Mantissa M**

→ Mantissa is coded with an implied leading (i.e. 24 bits)

$M = 1.xxx\ldots x$

↳ normalized

if M = 0.xx... ↳ not normalized

→ Here xxxx...x denotes bits that are actually stored for mantissa
Extra leading bit is for free

- When xxxx...x = 0000...0, M is minimum (= 1.0)
- When xxxx...x = 1111...1, M is maximum (= 2.0 − ϵ)

Encoding Example

(1)

- Consider number F = 15335

$15335_{10} = 11101111100111_2$
$= 1.1101111100111 \times 2^{13}$

- Mantissa will be stored as:
$M = 11011111001110000000000_2$

- Here, EXP = 13, BIAS = 127
$E = 13 + 127 = 140 = 10001100_2$

0	10001100	11011111001110000000000

466F9C00 in hex

(2) Consider number F = −3.75
$-3.75_{10} = -11.11_2 = -1.111 \times 2^1$

Mantissa will be stored as:
$M = 11100000000000000000000_2$

Here EXP = 1 ; BIAS = 127
$E = 1 + 127 = 128 = 10000000_2$

1	10000000	11100000000000000000000

C0700000 in hex

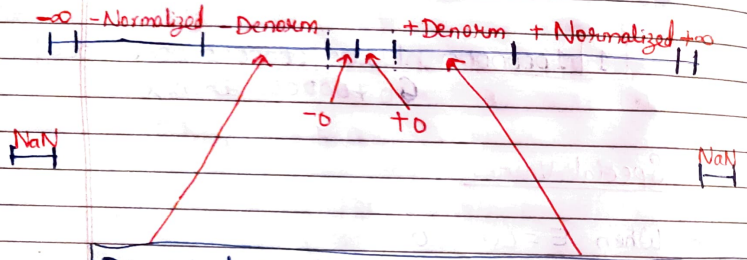
Special Values

- When E = 000...0
  → M = 000...0 represents value 0
  → M ≠ 000...0 represents numbers very close to 0
  - Also referred to as de-normalized number
  - zero is represented by all-zero string

- When E = 111...1
  → M = 000...0 represents value ∞ (infinity)
  → M ≠ 000...0 represents Not-a-Number (NaN)

→ NaN represents cases when no numeric value value can be determined like uninitialized values, $\infty * 0$, $\infty - \infty$, square root of a negative number, etc.

Summary of Number Encodings



Denormal numbers have very small magnitudes (close to 0) such that trying to normalize them will lead to an exponent that is below minimum possible value.

- Mantissa with leading 0's and exponent field equal to 0
- Number of significant digits gets reduced in the process

Rounding

- Assume addition of two numbers (say in single precision)
  - → add mantissa values after shifting one of them right for exponent alignment.
  - → take first 23 bits of sum and discard the residue R (beyond 32 bits)

- IEEE-754 format supports four rounding modes:
  - (a) Truncation
  - (b) Round to $+\infty$ (similar to ceiling function)
  - (c) Round to $-\infty$ (similar to floor function)
  - (d) Round to nearest

e.g.

$1.100 \times 2^2 \rightarrow 1.100 \times 2^2$

$1.011 \times 2^1 \Rightarrow 0.1011 \times 2^2$

10.0011×2^2

- To implement rounding, two temporary bits are maintained:
  - → Round Bit (r): equal to MSB of residue R
  - → Sticky Bit (s): logical OR of rest of bits of residue R

- Decisions regarding rounding can be taken based on these bits:

(a) $R > 0$: If $r + s = 1$

(b) $R = 0.5$: if $r.s' = 1$

(c) $R > 0.5$: if $r.s = 1$

// '+' → logical OR
'.' → logical AND

- Renormalization after Rounding:

→ If the process of rounding generates a result that is not in normalized form, then renormalization of result is needed.

Floating Point Arithmetic

Floating Point Addition/Subtraction

- Two numbers: $M_1 \times 2^{E1}$
  $M_2 \times 2^{E2}$
  where $E1 > E2$

- Basic steps

→ Select number with smaller exponent (i.e., E2) and shift its mantissa right by (E1 − E2) positions

→ Set exponent of result equal to larger exponent (i.e., E1)

→ Carry out $M_1 \pm M_2$, and determine sign of the result.

→ Normalize resulting value, if necessary.

Addition Example

- Two numbers to be added:

  F1 = 270.75
  F2 = 2.375

$F1 = (270.75)_{10} = (100001110.11)_2 = 1.00001110111 \times 2^8$

$F2 = (2.375)_{10} = (10.011)_2 = 1.0011 \times 2^1$

- Shift mantissa of F2 right by 8 − 1 = 7 positions and add:

```
1 000  0111 0110 0000 0000 0000
       1 0011 0000 0000 0000 0000 000
1 000  1000 1001 0000 0000 0000 0000 000
```

take first 24 bits of result (rounding using truncation)

Residue

Result: 1.00010001001×2^8

Subtraction Example

- Two numbers to be subtracted: F1 - F2

  F1 = 270.75 and F2 = 224

$$F1 = (270.75)_{10} = (100001110.11)_2 = 1.00001110111 \times 2^8$$
$$F2 = (224)_{10} = (11100000)_2 = 1.110 \times 2^7$$

- Shift mantissa of F2 right by 8-7=1 position and subtract:

```
1000 0111 0110 0000 0000 0000
 111 0000 0000 0000 0000 0000 000
0001 0111 0110 0000 0000 0000 000
```

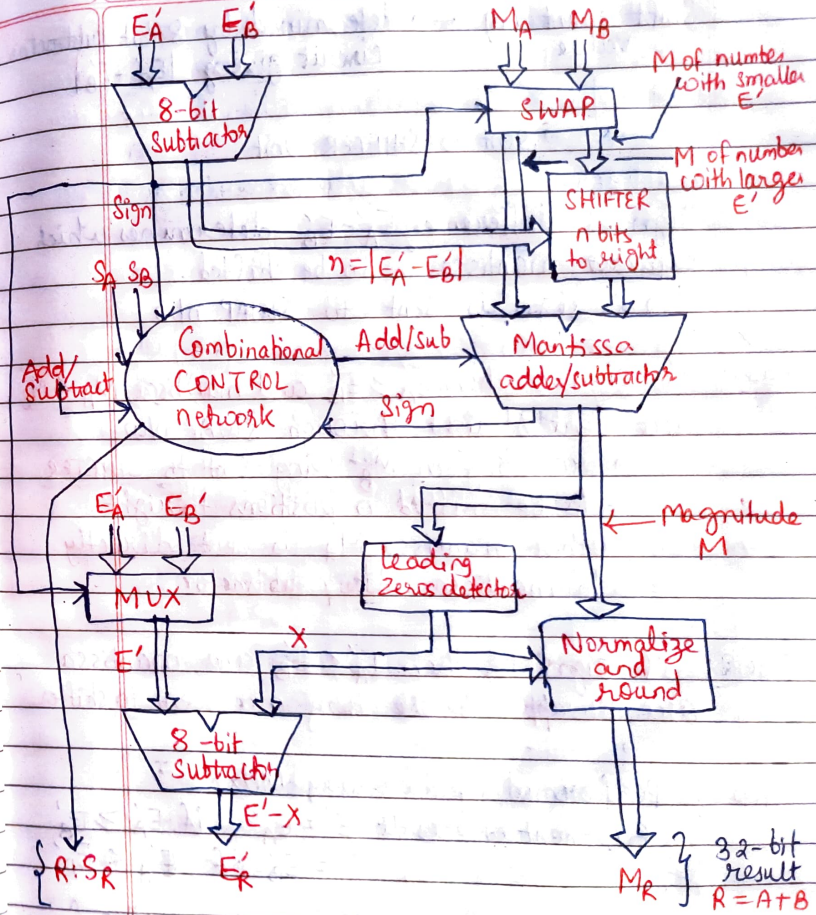
- for normalization, shift mantissa left 3 position and decrement E by 3.

Result: 1.01110110×2^5

Implementation of floating point operations (Add/sub)

Step1. Compare exponents to determine how far to shift mantissa of no. with smaller exponent

32-bit operands { A: S_A, E_A', M_A ; B: S_B, E_B', M_B }



Floating point addition-subtraction unit

Shift count (n) value → determined by 8-bit subtractor circuit giving $|E_A' - E_B'|$

↳ sent to SHIFTER unit.

Sign of difference $E_A' - E_B'$ determines which mantissa is to be shifted.

↳ i.e. sign is sent to SWAP n/w

Case 1: if sign = 0 then $E_A' \geq E_B'$ and mantissa M_A & M_B are sent straight through SWAP n/w

↳ resulting in M_B being sent to SHIFTER to be shifted n positions to right.

Other mantissa M_A is sent directly to mantissa adder / subtractor.

Case 2: If sign = 1, then $E_A' < E_B'$ and mantissa are swapped before they are sent to shifter.

Step 2: Performed by two-way MUX.

Exponent of result $E' = E_A'$ if $E_A' \geq E_B'$

" $E' = E_B'$ if $E_A' < E_B'$

↳ based on sign of difference resulting from comparing exponents in step 1.

Step 3: Uses Mantissa adder / subtractor.

CONTROL logic determines whether mantissa are to be added or subtracted based on signs of operands (S_A and S_B) and operation (Add or Subtract) that is to be performed on operands

CONTROL logic also determines sign of result S_R.

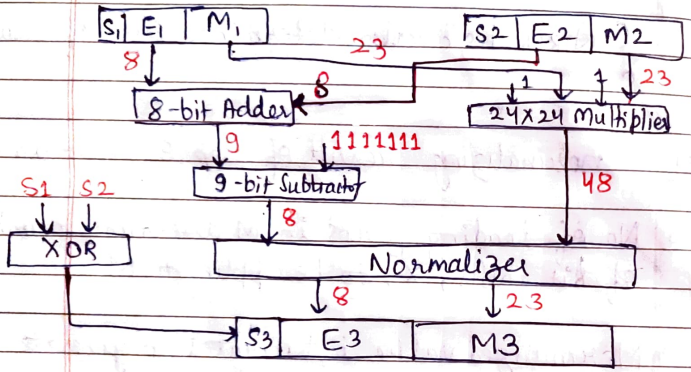
Step 4: normalizing result of step 3, mantissa M

- No. of leading zeros in M determines number of bit shifts, X, to be applied to M
- Normalized value is truncated to generate 24-bit mantissa, M_R, of result.
- The value X is also subtracted from tentative result exponent E' to generate true result exponent, E_R'.

Floating point Multiplication

1. Add exponents and subtract 127
2. Multiply the mantissa & determine the sign of result
3. Normalize the resulting value, if necessary.

Subtraction of 127 results from using excess 127 notation for exponents



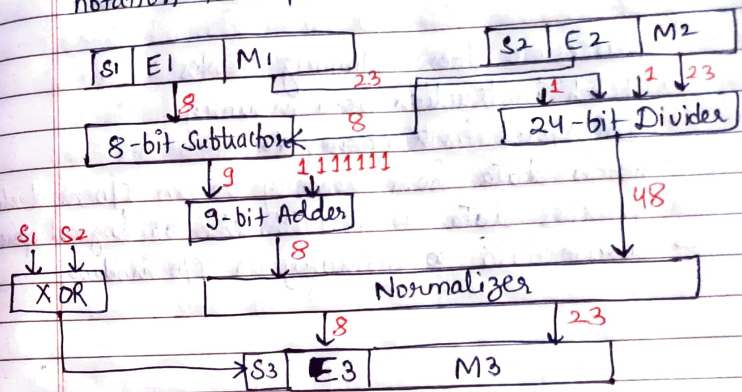
Floating point Division

Two numbers: $M_1 \times 2^{E1}$ and $M_2 \times 2^{E2}$

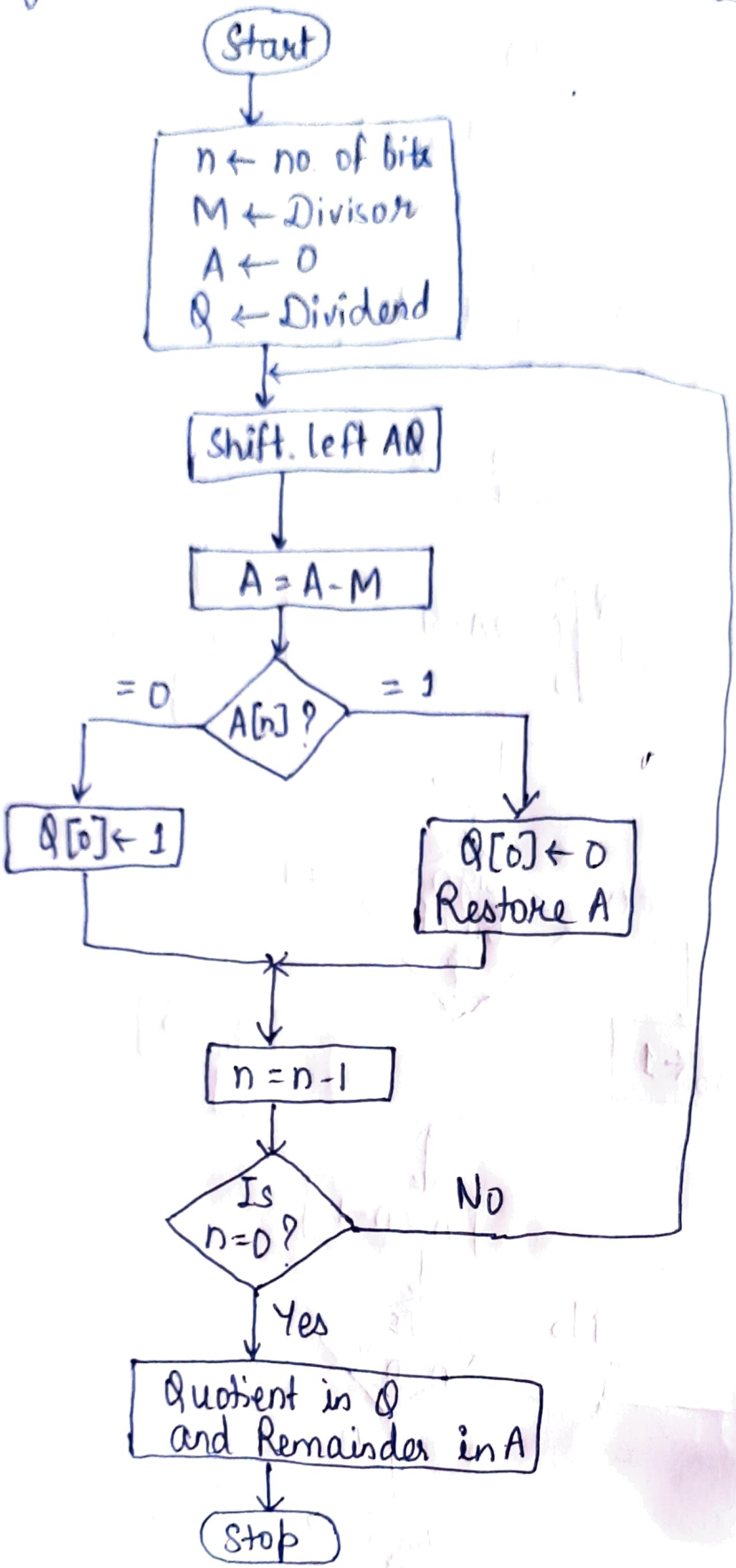
Basic steps

1. Subtract exponents E1 and E2 and add BIAS
2. Divide M1 by M2 and determine sign of result
3. Normalize resulting value, if necessary.

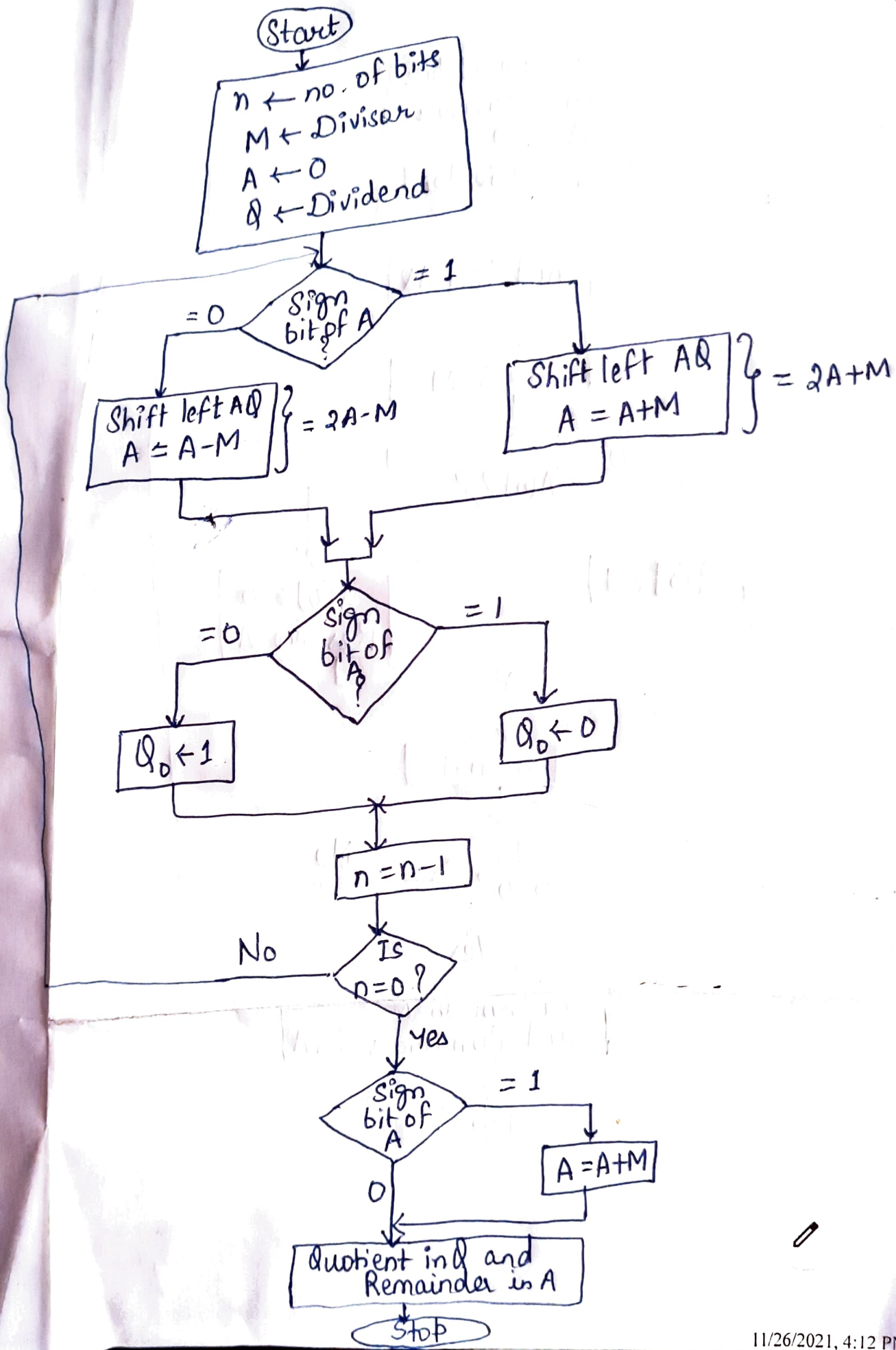
Addition of 127 results from using excess-127 notation for exponents.



Restoring Algorithm for Division of Unsigned No's



Non - Restoring Algorithm for Division of Unsigned Integers



Unit 2: Machine Instructions

- Computer Instruction
  - Opcode (specifies operation to be performed)
  - source & destination operand references (specify input and output locations for operation)
  - next instruction reference (usually implicit)

- General categories of opcodes specified operations:
  - → Arithmetic and logic operations
  - → movement of data
    - between two registers
    - register and memory
    - two memory locations
  - → I/O
  - → Control

- Operand references specify a register or memory location of operand data

Type of data
- addresses
- numbers
- characters
- logical data

Date / /
Page No.

- Common architectural feature in processor
  ↳ use of stack
  (may or may not be visible to programmer)

Stacks are used to manage procedure calls and returns
↳ may be provided as an alternative form of addressing memory.

Stack operations
- PUSH
- POP
- operations on top one or two stack locations

* Stacks typically are implemented to grow from higher addresses to lower addresses.

- Byte-addressable processors may be categorized as
  - big endian
  - little endian
  - bi-endian

Big endian → Multibyte numerical value stored with most significant byte in lowest numerical address

Little endian → stores the most significant byte in the highest numerical address.

Bi-endian → can handle both styles.

Memory Locations and Addresses

Memory $\xrightarrow{\text{consists of}}$ many millions of storage cells

↓

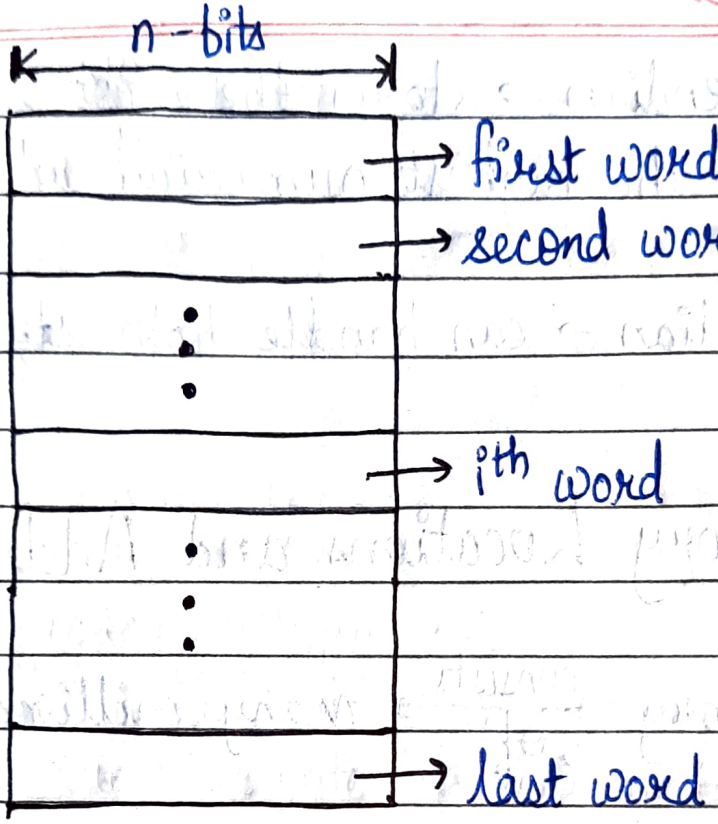
each of which can store a bit of information (0 or 1)

Memory is organized so that a group of n bits can be stored or retrieved in a single, basic operation.

Each group of n bits ⟶ a word of information

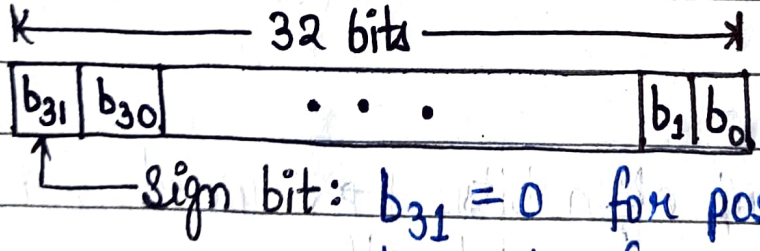
n → word length

∴ memory of a computer $\xrightarrow{\text{represented as}}$ collection of words

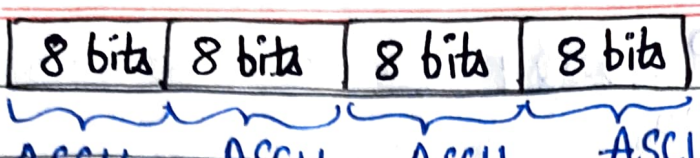


Memory Words

- Modern computers have word lengths range from 16 to 64 bits.



ⓐ. Signed integer



⑥. Four characters

Examples of encoded information in a 32-bit word

unit of 8 bits → byte

- Distinct names or addresses for each item locatio is required for accessing memory to store or retrieve a single item of information
  - ↳ use numbers from 0 through $2^k - 1$, for suitable value of k, as addresses of successive locations in memory.

2^k addresses → address space of computer

e.g. 24-bit address generates address space of 2^{24} (16,777,216) locations.
↳ $= 2^4 \times 2^{20} = 16\,M$
($\because 2^{20} = 1M$)

$1k = 2^{10} = 1024$

$1M = 2^{20}$

$1G = 2^{30}$

$1T = 2^{40}$

- **Byte Addressability**

Most practical assignment
↳ successive addresses referring to successive byte locations in memory

Byte locations have addresses 0, 1, 2 . . . .

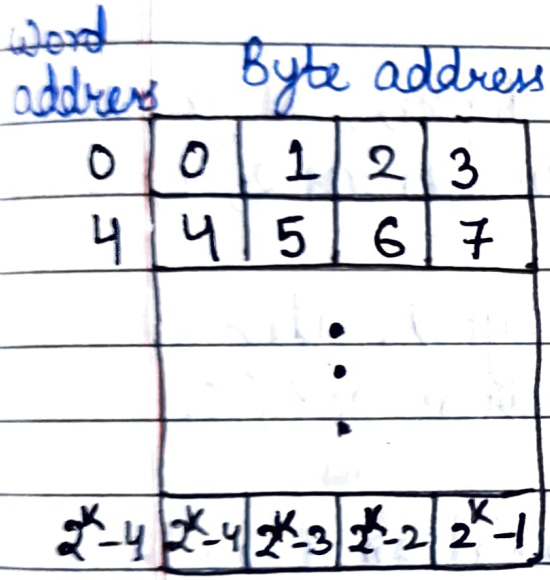
∴ if word length of machine = 32 bits
successive words are located at addresses 0, 4, 8, . . . , with each word consisting of four bytes.

- **Big-Endian and Little-Endian Assignments**

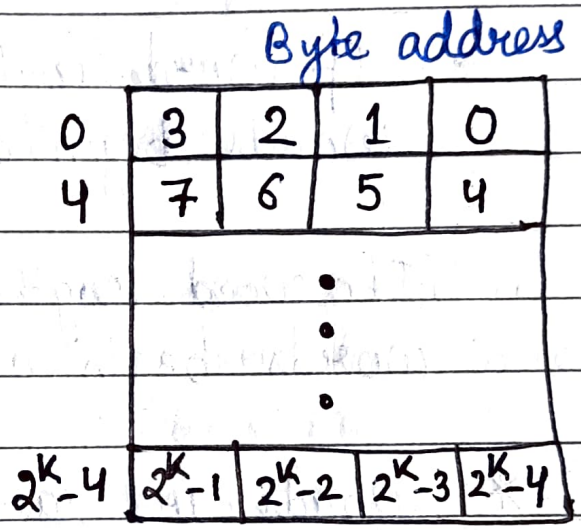
Two ways of assigning byte addresses across words
- Big-Endian
- Little Endian

Big-Endian → when lower byte addresses are used for more significant bytes (the leftmost bytes) of the word.

Little-Endian → when lower byte addresses are used for less significant bytes (the rightmost bytes) of the word.



ⓐ. Big-Endian assignment



ⓑ. Little-Endian assignment

Byte and word addressing

* Ordering used for labeling bits within a byte is $b_7, b_6, \ldots, b_0$, from left to right.

→ Some computers, however, use reverse ordering

Date / /
Page No.

- **Word Alignment**

* Words are said to be aligned in memory if they begin at a byte address that is a multiple of number of bytes in a word.

Generally, no. of bytes in a word is a power of 2.

∴ If word length is 16 (2 bytes), aligned words begin at byte addresses 0, 2, 4, ...

If word length is 64 (2^3 bytes), aligned words begin at byte addresses 0, 8, 16, ...

* Words are said to have unaligned addresses if words begin at an arbitrary byte address.

- **Accessing numbers, characters, and character strings**

→ A number usually occupies one word.
↳ can be accessed in memory by specifying its word address.

→ Similarly, individual characters can be accessed by their byte address.

→ In many applications, character strings of variable length are required.

- Beginning of string is indicated by giving address of byte containing its first character.
- Successive byte locations contain successive characters of the string

Two ways to indicate length of string
- Special control character with meaning "end of string" can be used as last character in string.
- a separate memory word location or processor register can contain a number indicating length of string in bytes

Memory operations
- Load (or Read or Fetch)
- Store (or Write)

Both program instructions and data operands } are stored in the memory

∴ To execute an instruction
- Processor control circuits must cause word (or words) containing instruction to be transferred from memory to processor.
- Operands and results must also be moved between memory and processor

Two memory operations :

1. Load operation transfers a copy of contents of a specific memory location to the processor
   * Memory contents remain unchanged

- To start Load operation, processor sends address of desired location to the memory and requests that its contents be read
- Memory reads the data stored at that address and sends them to the processor.

Date / /
Page No.

② Store operation transfers an item of information from processor to a specific memory location, destroying former contents of that location.

- The processor sends address of the desired location to the memory, together with data to be written into that location.

→ An information item of either one word or one byte can be transferred between the processor and the memory in a single operation.

* Processor contains a small number of registers (each capable of holding a word)

→ These registers are either the source or the destinatio of a transfer to or from the memory.

When a byte is transferred, it is usually located in the low-order (rightmost) byte position of the register.

Date / /
Page No. 6

Operands

Machine instructions operate on data (operands)

- Important general categories of data are:
  - Addresses (unsigned integers)
  - Numbers
  - Characters
  - Logical data

(i). Numbers

- All machine languages include numeric data types.

- Even in non numeric data processing, numbers are needed to act as counters, field widths, and so forth.

- Numbers stored in computers are limited
  - limit to magnitude of nos representable on a machine
  - in floating point numbers, limit to their precision

Types of numerical data

- Integer or fixed point
- floating point
- Decimal

- Although machine understands binary, human users of system deal with decimal numbers.

∴ Necessity to convert from decimal to binary on i/p and from binary to decimal on output.

- for simple computation, it is preferable to store and operate on numbers in decimal form.

Most common representation for this is packed decimal (Binary coded decimal (BCD))

↓

each decimal digit represented by a 4-bit code

∴ 0 = 0000, 1 = 0001, ..., 8 = 1000, and 9 = 1001

Note: 0 to 9 → valid BCD

(only 10 of 16 possible 4-bit values are used)

→ To form numbers, 4-bit codes are strung together, usually in multiples of 8-bits.

e.g. $(24\ 6)_{10} = (0010\ 0100\ 0110)_{BCD}$

0010 → 0100 → 0110

(less compact)

BCD code has more bits as compared to straight binary representation, but it avoids conversion overhead.

* Negative numbers can be represented by including a 4-bit sign digit at either left or right end of a string of packed decimal digits.

e.g. 1111 (code) → -ve no. (minus sign)

(ii). Characters

Common form of data → text or character strings

- As data processing and communications systems are designed for binary data, thus, a number of codes have been devised by which characters are represented by a sequence of bits.

Earliest common example → Morse Code

Nowadays commonly used character code is ASCII (American Standard Code for Information Interchange)

→ Each character in this code is represented by a unique 7-bit pattern; thus, 128 different characters can be represented.

Apart from printable characters, some of the patterns represent control characters

- for controlling printing of characters on a page
- Concerned with Communication procedures

→ These encoded characters are almost always stored and transmitted using 8 bits per character

- 8th bit may be set to 0 or used as a parity bit for error detection.

- For parity based case, bit is set such that total no. of binary 1s in each octet is always odd (odd parity) or always even (even parity).

* Another code used to encode characters
  ↳ Extended Binary Coded Decimal Interchange Code (EBCDIC)
  ↓
  Used on IBM S/390 machines (8-bit code)

- Both ASCII and EBCDIC is compatible with packed decimal

(iii) Logical Data

- Normally, each word or other addressable unit (byte, halfword, and so on) is treated as a single unit of data

Sometimes, however, n-bit unit $\xrightarrow{\text{considered as}}$ n 1-bit items of data

(each item having value 0 or 1)

↓

when data are viewed this way, they are considered to be logical data.

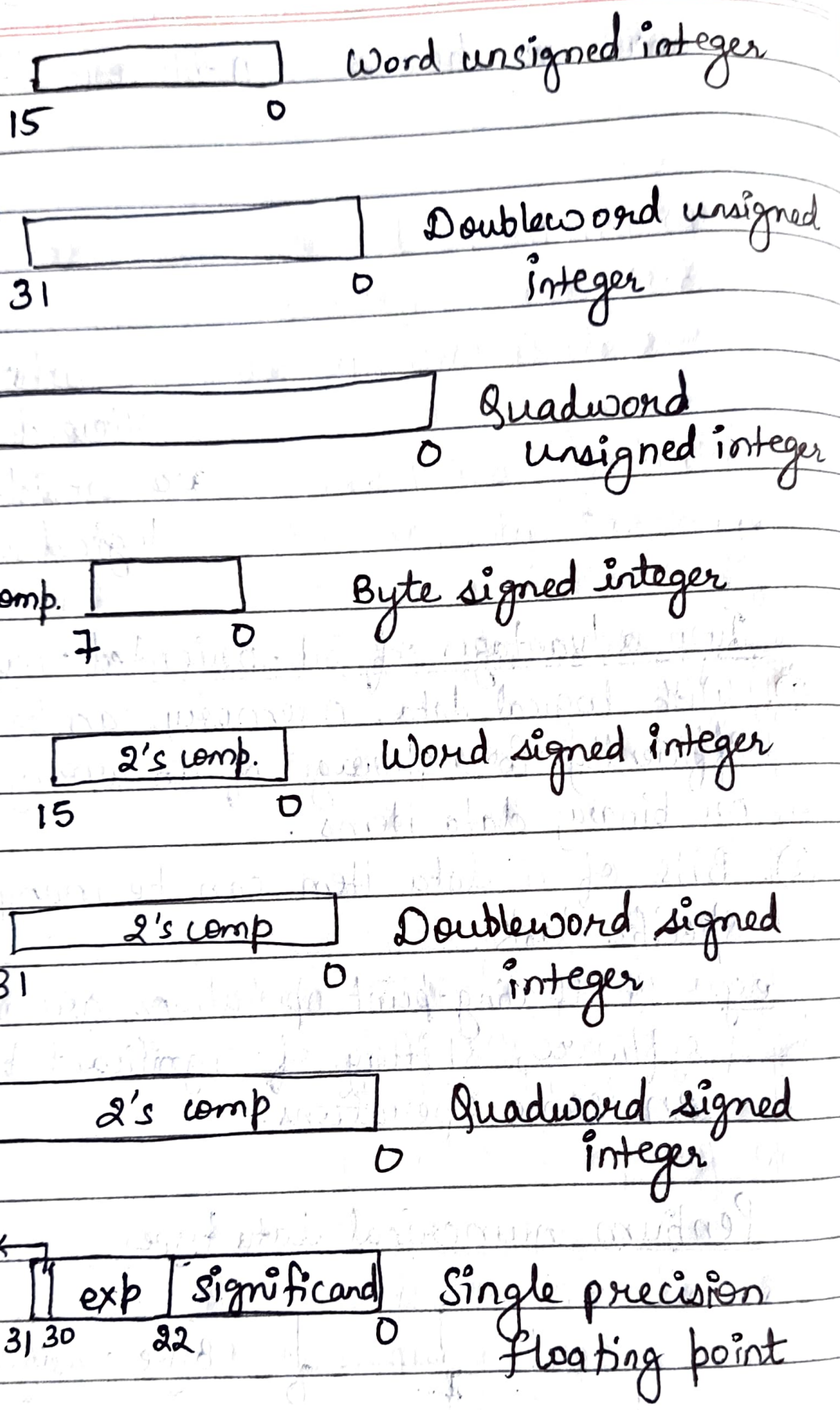
Two advantages of bit-oriented view

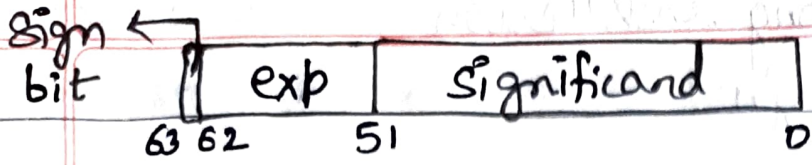
1. With logical data, memory can be used most efficiently for storage of an array of Boolean or binary data items.
2. Bits of a data item can be manipulated for specific task.

eg. If floating-point operations are implemented in software, shifting of significant bits is required in some operations.

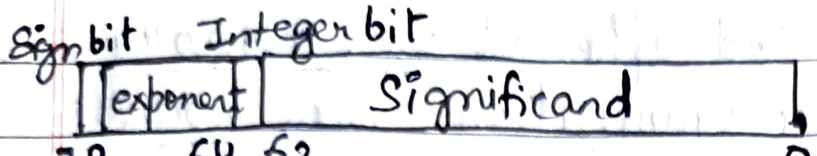
Pentium numerical data types

Byte unsigned integer


Word unsigned integer
15
0
Doubleword unsigned integer
31
0
Quadword unsigned integer
63
0
2's comp.
Byte signed integer
7
0
2's comp.
Word signed integer
15
0
2's comp
Doubleword signed integer
31
0
2's comp
Quadword signed integer
63
0
sign bit
exp
significand
Single precision floating point
31
30
22
0



Double precision floating point



Double extended precision floating point

Instruction formats

- Instruction format defines layout of bits of instruction, in terms of its constituent fields.

Instruction format
- opcode
- zero or more operands (implicitly or explicitly)

Explicit operand —referenced→ using one of the addressing modes

- Format must, implicitly or explicitly, indicate addressing mode for each operand.
- for most instruction sets, more than one instruction format is used.

Factors affecting instruction format design

(i) Instruction length

↳ affects and affected by, memory size, memory organization, bus structure, processor complexity, and processor speed

→ This determines richness and flexibility of machine as seen by assembly-language programmer

trade-off — powerful instruction set (or to save space)
- more opcodes
- more operands
- more addressing modes
- greater address range

powerful instruction set → shorter programs written to accomplish given tasks (easier for programmer)

→ All of these (opcodes, operands, addressing modes, address range) require bits and thus leads to longer instruction lengths.

Longer instruction length may be wasteful.

another trade-off → • Either instruction length should be equal to memory-transfer length (in a bus-system, data-bus length)

Or

• should be a multiple of other

Issue

Memory transfer rate has not kept up with increases in processor speed.

Memory can become a bottleneck if processor can execute instructions faster than it can fetch them.

Solution

• use of cache memory
• use shorter instructions

∴ 16-bit instructions can be fetched at twice rate of 32-bit instructions but probably can be executed less than twice rapidly.

→ Instruction length should be a multiple of

character length (usually 8 bits) and of length of fixed-point numbers.

Otherwise, there are wasted bits in each word when storing multiple characters or a character will have to spread out irregularly from word boundary.

(ii). Allocation of Bits

Trade-off between number of opcodes and power of addressing capability.

- More opcodes ⟶ more bits in opcode field

→ for instruction format of a given length, this reduces no. of bits available for addressing

Issue

Use of variable-length opcodes
↳ fewer bits for addressing

∴ used for those instructions that require fewer operands and/or less powerful addressing.

Date / /
Page No.

factors determining use of addressing bits :

* No. of addressing modes
  addressing mode can be
  - indicated implicitly
  - explicitly (one or more mode bits needed)

* No. of operands
  - typically two operands

Each operand address in instruction might require its own mode indicator, or use of a mode indicator could be limited to just one of the address fields.

* Register versus memory
  Register ⟶ used to store data brought into processor for processing

With a single user-visible register (usually called accumulator), one operand address is implicit & consumes no instruction bits.

- disadv: requires many instructions

∴ more registers can be used for operand references, as only fewer bits are needed to specify register

It is observed that a total of 8 to 32 user-visible registers is desirable

→ Modern architectures have at least 32 registers

* No. of register sets

Most contemporary machines have one set of general-purpose registers, with typically 32 or more registers in the set.

registers → used to store data
- addresses for displacement addressing

Some architectures, including that of x86, have a collection of two or more specialized sets (such as data and displacement)

advantage → for a fixed no. of registers, functional categorization requires fewer bits to be used in instruction.

eg. With two sets of 8 registers
→ only 3 bits are required to identify a register

→ Opcode or mode register will determine which set of registers is being referenced.

* Address range
  ↳ related to no. of address bits
  ↓
  It results in a severe limitation,
  ∴ direct addressing is rarely used

Sol^n

Displacement addressing
  ↳ range is opened up to length of address register.
  ↳ used where relatively large number of address bits are required in instruction

* Address granularity
  ↳ important factor for addresses that reference memory rather than register

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

→ Byte addressing is convenient for character manipulation but requires, for a fixed size memory, more address bits.

Instruction types based on operations

- Data transfers b/w memory and processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

① Register Transfer Notation

Possible locations involved in information transfers are →
- memory locations
- processor registers
- registers in I/O subsystem

- Generally a location is identified by a symbolic name standing for its hardware binary address.

e.g. Names for addresses of memory locations → LOC, PLACE, A, VAR2

Processor register names → R0, R5

I/O register names → { DATAIN, OUTSTATUS

- contents of a location —denoted by→ placing square brackets around name of the location

e.g. R1 ← [LOC]

↳ contents of memory location LOC are transferred into processor register R1

R3 ← [R1] + [R2]

↳ contents of registers R1 and R2 are added, and then sum is placed into register R3

This type of notation is known as Register Transfer Notation (RTN).

Note: Right hand side of an RTN expression always denotes a value, and the left-hand side is the name of a location where the value is to be placed, overwriting old contents of that location.

② Assembly language Notation
↳ to represent machine instructions and programs

e.g. MOVE LOC, R1
↳ transfer contents of memory location LOC to processor register R1

→ Contents of LOC are unchanged by execution of this instruction
→ Old contents of register R1 are overwritten.

Add R1, R2, R3
↳ adding two numbers contained in processor registers R1 and R2 and placing their sum in R3

Date / /
Page No.

Basic Instruction Types

Adding two numbers

$C = A + B$

Command in high level language

(add variables A and B and store sum to third variable C)

When program containing this command is compiled, three variables A, B, and C, are assigned to distinct locations in memory.

$\therefore [C] \leftarrow [A] + [B]$

contents of memory locations A and B are fetched from memory and transferred into the processor where their sum is computed. This result is then sent back to the memory and stored in location C.

Single machine instruction to accomplish this action

Add A, B, C

↳ three-address instruction

Operation Source 1, Source 2, Destination

operands A and B → source operands
C → destination operands
Add → opcode
(operation to be performed on operands)

Note: If K-bits $\xrightarrow{\text{needed to}}$ specify memory address of each operand then encoded form of above instruction $\xRightarrow{\text{contain}}$ 3k bits for addressing purposes

→ Apart from 3k bits, some bits are needed to denote Add operation.

for modern processor with a 32-bit address space, a 3-address instruction is too large to fit in one word for a reasonable word length. Thus multiple words for a single instruction would be needed.

Alternative approach → use of sequence of simples instructions to perform same task, with each instruction having only one or two operands

Two-address instruction — Operation Source, Destination

```
Add A, B    // B ← [A] + [B]    B is both source & destination
Move B, C   // C ← [B]
```

copy contents of location B to C, leaving contents of location B unchanged

→ In all instructions given above, source operands are specified first, followed by destination.

Some computers are there in which order of source and destination operands is reversed

Operation Destination, Source

Imp Even two-address instructions will not normally fit into one word for usual word lengths and address sizes.

Alternative approach → machine instructions that specify only one memory operand.

When a second operand is needed, it is understood implicitly to be in a unique locatio

↓

processor register, usually called accumulator used for this purpose

e.g.

one-address instruction

Add A

↓

Add contents of memory location A to contents of accumulator register and place sum back into accumulator

Load A

↳ copies contents of memory location A into accumulator.

Store A

↳ copies contents of accumulator into memory location A.

∴ using only one-address instructions, operation C ← [A] + [B] can be performed by executing sequence of instructions

```
Load A      // Accumulator ← [A]
Add B       // Accumulator ← [Acc.] + [B]
Store C     // C ← [Accumulator]
```

→ Some early computers were designed around a single accumulator structure.

- Modern computers have a number of general purpose processor registers (typically 8 to 32) and even considerably more in some cases.

Imp
Access to data in these registers is much faster than to data stored in memory locations because registers are inside processor.

→ As no. of registers is relatively small, only a few bits are needed to specify which register takes part in an operation.

e.g. for 32 registers → only 5 bits needed
(less than no. of bits needed to give address of a location in memory)

Use of registers
↓ allows
faster processing
↓ results in
shorter instructions

∴ registers are used to store data temporarily in processor during processing

e.g. Let R_i → general-purpose register

Load A, R_i
Store R_i, A
Add A, R_i

→ R_i performs function of accumulator for single accumulator case

- Even when only one memory address is directly specified in an instruction, instruction may not fit into one word.

Alternative Approach: When a processor has several general-purpose registers, many instructions involve only operands that are in registers.

→ In many modern processors, computations can be performed directly only on data held in processor registers

e.g.
- Add R_i, R_j // $[R_j] \leftarrow [R_i] + [R_j]$
- Add R_i, R_j, R_k // $[R_k] \leftarrow [R_i] + [R_j]$

↳ Such instructions where only register names are contained in instruction, will normally fit into one word

To transfer data b/w processor registers — Move Source, Destination

e.g. Move A, R_i

∴ In processors where arithmetic operations are allowed only on operands that are in processor registers, C = A+B task can be performed by instruction sequence

```
Move A, Ri
Move B, Rj
Add Ri, Rj
Move Ri, C
```

In processor where one operand may be in memory but other must be in a register, an instruction sequence for required task $C = A + B$ would be

```
Move A, Ri
Add  B, Ri
Move Ri, C
```

- Speed with which a given task is carried out depends on (i) time it takes to transfer instructions from memory into processor (ii) time taken to access operands referenced by these instructions.

As transfers involving memory are much slower than transfers within processor

↓

∴ A substantial increase in speed is achieved when several operations are performed in succession on data in processor registers without need to copy data to or from memory.

When machine language programs are generated by compilers from high-level languages, it is important to minimize frequency with which data is moved back and forth between memory and processor registers

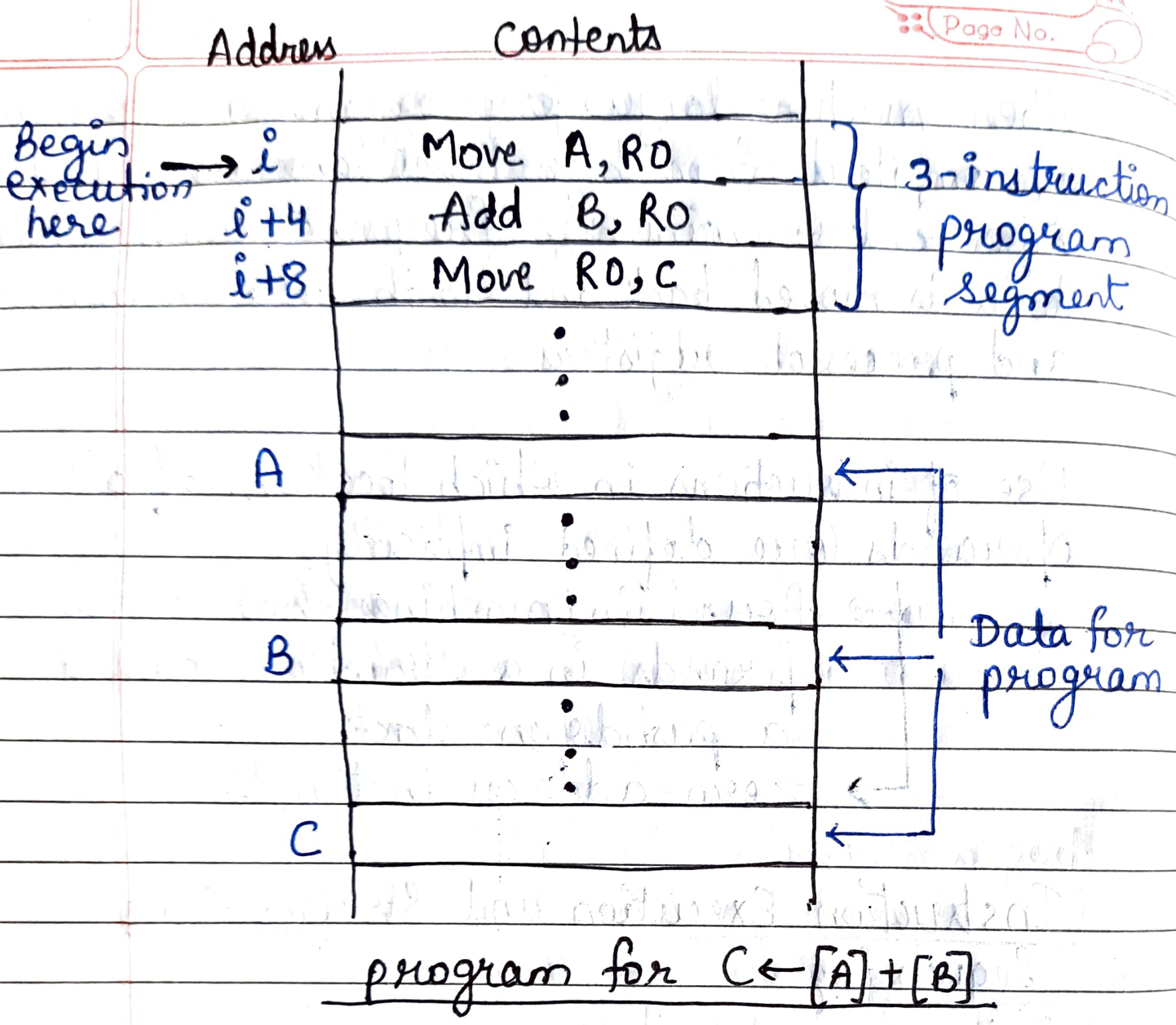
Use of instructions in which locations of all operands are defined implicitly
- found in machines that store operands in a structure called a pushdown stack
- zero-address instructions

Instruction Execution and Straight-Line Sequencing

Consider task $C \leftarrow [A] + [B]$

Assume that computer allows one memory operand per instruction and has a number of processor registers.

Assume Word length = 32 bits
Memory is byte addressable



program for C ← [A] + [B]

→ Three instructions of program are in successive word locations, starting at location i.

→ As each instruction is 4 bytes (or 32 bits) long, second and third instruction addresses start at i+4 and i+8.

Date / /
Page No.

Program Execution

Processor has register called program counter (PC) that holds address of instruction to be executed next.

∴ initially [PC] = i

→ Processor control circuits use information in PC to fetch and execute instructions, one at a time, in order of increasing addreses

↓

straight-line sequencing

→ During execution of each instruction, PC is incremented by 4 to point to next instruction

∴ after Move instruction at location i+8 is executed, PC contains value i+12, which is address of first instruction of next program segment.

Execution of given instruction ⟶ two-phase procedure

Phase 1 (Instruction fetch)

↳ instruction is fetched from memory location whose address is in PC.

↓

instruction is placed in instruction register (IR) in processor.

Phase 2 (Instruction Execute)

↳ instruction in IR is examined to determine which operation is to be performed.

↓

Specified operation is then performed by the processor

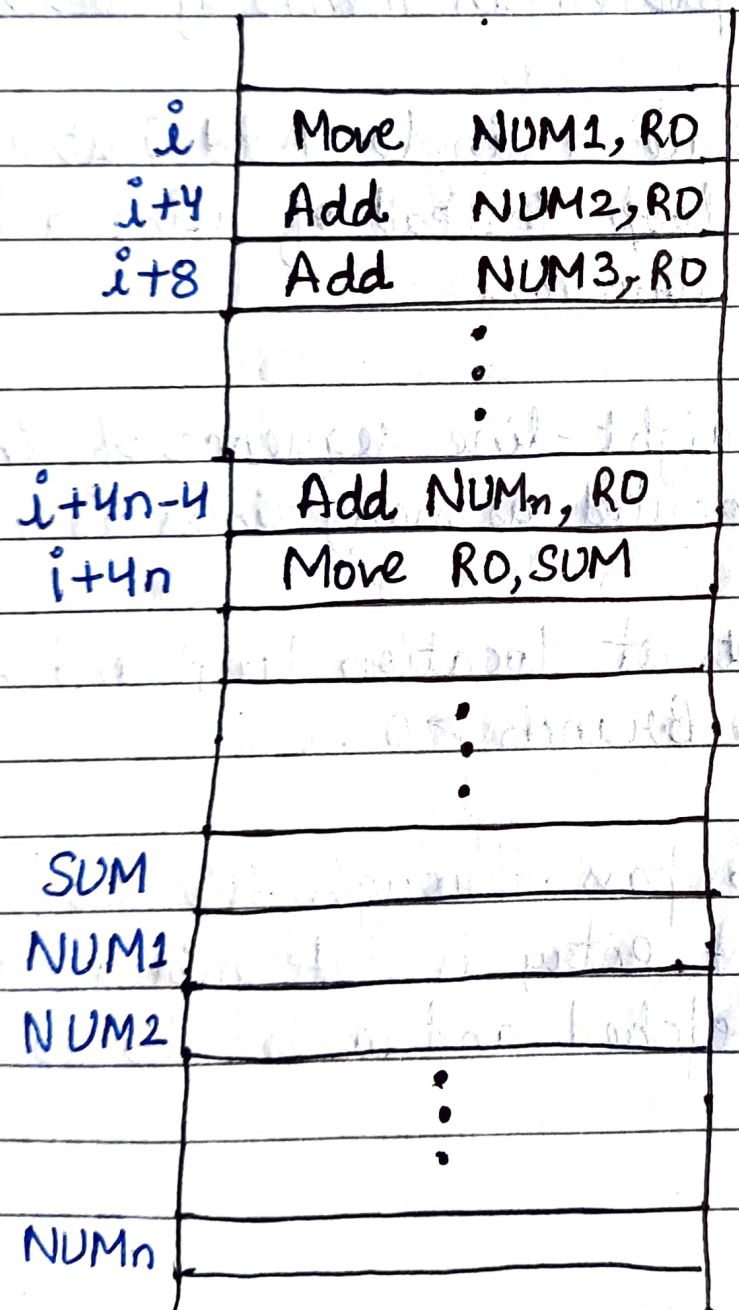
{ fetching operands from memory or from processor registers } + { performing an arithmetic or logic operation } + { storing result in destination location }

→ At some point during this two-phase procedure contents of PC are advanced to point to next instruction.

- After completion of execution phase of instruction PC contains address of next instruction, and a new instruction fetch phase can begin.

Branching

task : adding a list of n numbers



Straight-line program for adding n numbers

→ addresses of memory locations storing n numbers ⟶ NUM1, NUM2, ..., NUMn

→ Add instruction ⟶ to add each number to contents of register R0

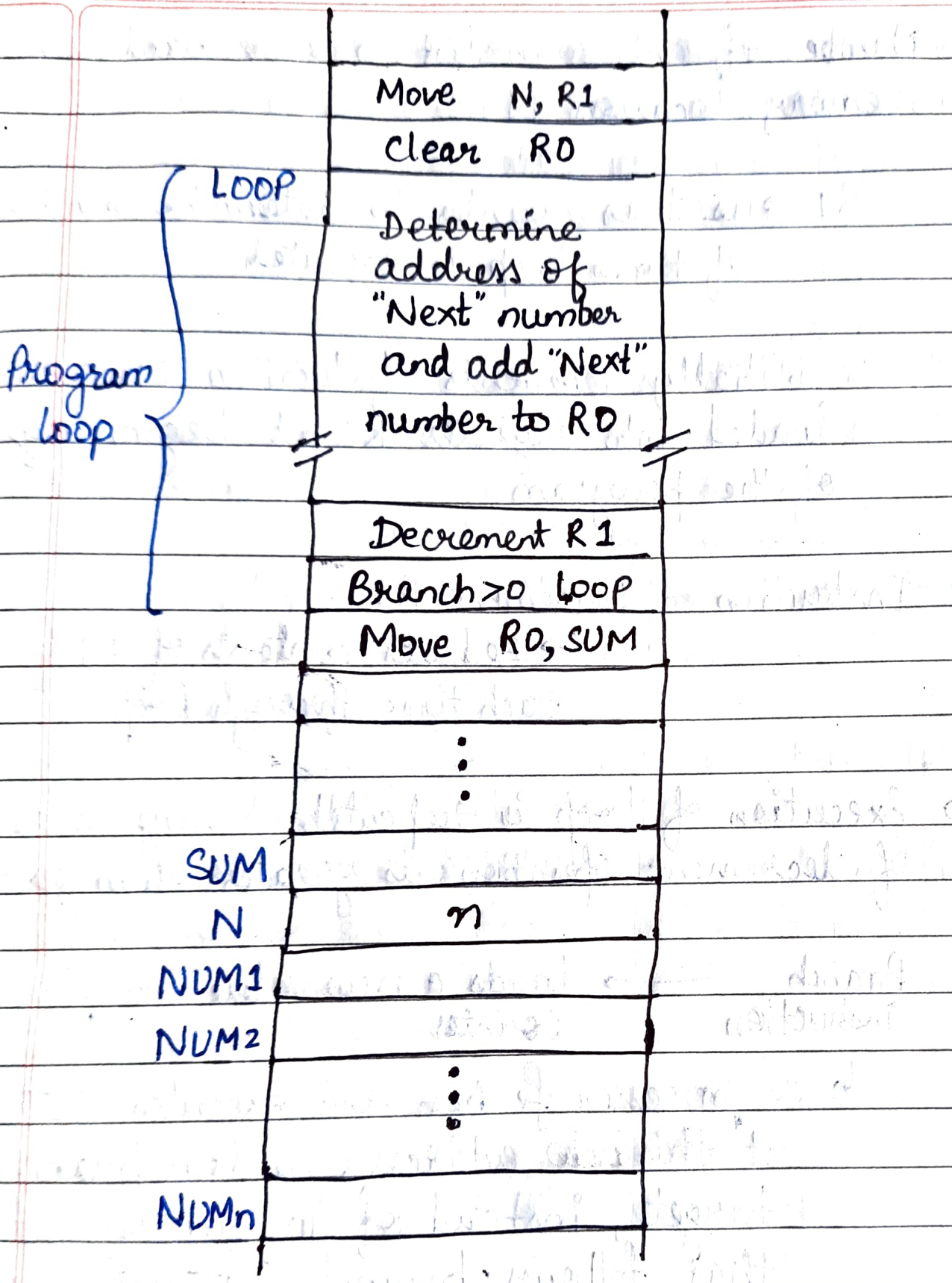
→ Result is placed in memory location SUM

• Possible to place a single Add instruction in a program loop instead of using a long list of Add instructions

Loop → Straight-line sequence of instructions executed as many times as needed

* Loop starts at location Loop and ends at instruction Branch>0.

During each pass through this loop, address of next list entry is determined, and that entry is fetched and added to R0.



Using a loop to add n numbers

Number of entries in list, n, is stored in memory location N

R1 ⇒ used as counter to determine number of times loop is executed

∴ initially contents of location N are loaded into register R1 at beginning of the program.

- Instruction Decrement R1
  ↳ reduces contents of R1 by 1 each time through loop

→ Execution of loop is repeated as long as result of decrement operation is greater than zero.

Branch instruction ⟶ loads a new value into program counter

↳ ∴ processor fetches and executes instruction at this new address, called branch target, instead of instruction at location that follows branch instruction in sequential address order.

Date / /
Page No.

Conditional Branch instruction

- causes a branch only if a specified condition is satisfied
- If condition is not satisfied, PC is incremented in normal way, and next instruction is sequential order is fetched and executed.

Branch >0 LOOP

- branch if greater than 0 is a conditional branch instruction that causes a branch to location LOOP if result of immediately preceding instruction, which is decrement value in R1, is greater than zero.

∴ loop is repeated as long as there are entries in list that are yet to be added to R0.

At the end of nth pass through loop, Decrement instruction produces a value of zero, &, hence branching does not occur.

∴ Move instruction is fetched and executed.
↳ final result from R0 is moved into memory location SUM

Condition Codes

- Processor keeps track of information about the results of various operations for use by subsequent conditional branch instructions
  - ↳ achieved by recording required information in individual bits (condition code flags)
    - ↓
    - Usually grouped together in a special processor register called condition code register or status register

- Individual condition code flags are set to 1 or cleared to 0, depending on the outcome of operation performed.

four commonly used flags are

- N (negative) → set to 1 if result is -ve
  → cleared to 0 otherwise
- Z (zero) → Set to 1 if result is 0
  → otherwise cleared to 0
- V (overflow) → set to 1 if arithmetic overflow occurs (result out of range)
  → otherwise cleared to 0
- C (carry) → Set to 1 if carry-out results from operation
  → otherwise cleared to 0

∴ conditional branch instructions are used to enable a variety of conditions to be tested.

Addressing Modes

- An operand reference in an instruction either contains
  - actual value of operand (immediate)
    or
  - reference to address of operand

→ The different ways in which location of an operand is specified in an instruction are reffered to as addressing modes

Most common addressing techniques

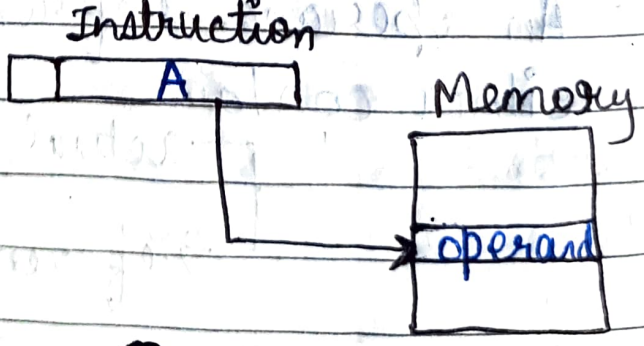
- Immediate
- Direct
- Indirect
- Register
- Register indirect
- Displacement
- Stack

→ Instruction format (defining layout fields in instructions)

depends on

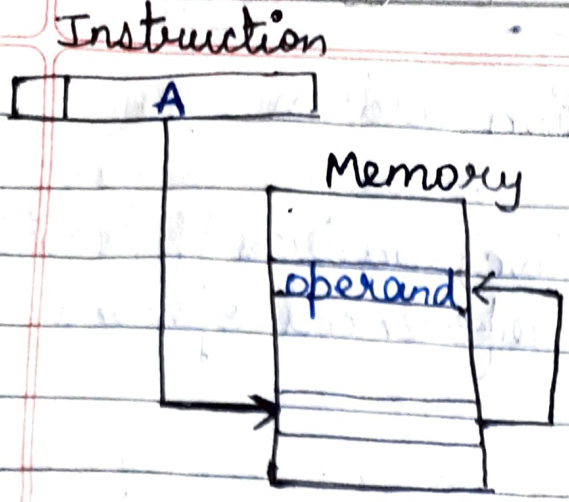
- instruction length → fixed / variable
- No. of bits assigned to opcode and each operand references
- how addressing mode is determined

ⓐ. Immediate

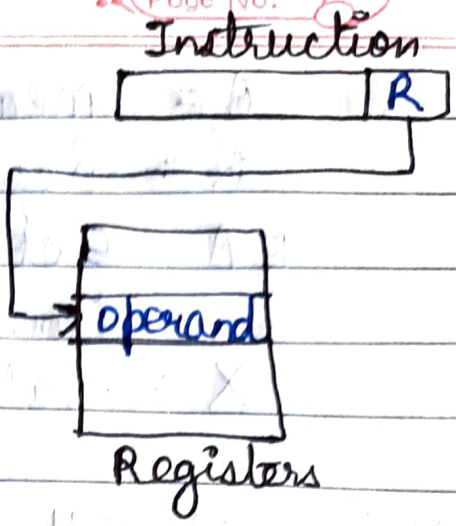


ⓑ. Direct

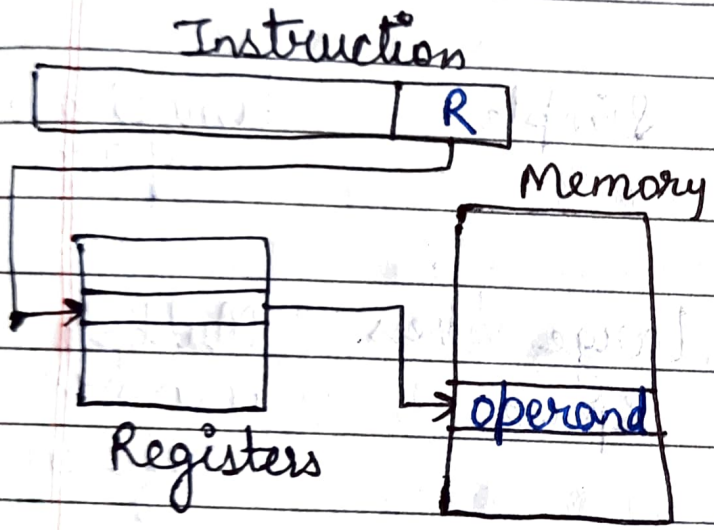
Addressing Modes



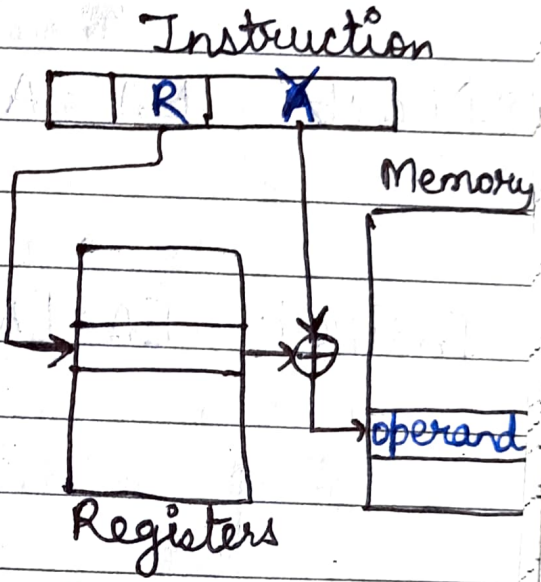
ⓒ. Indirect



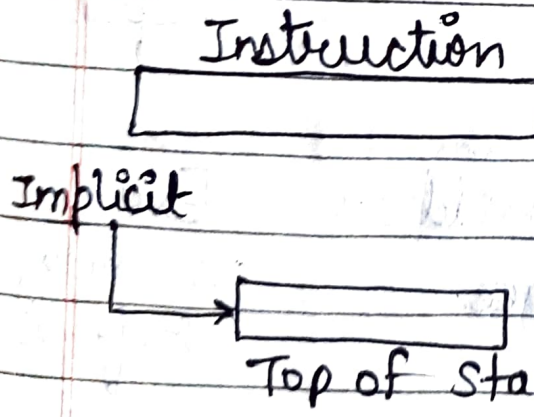
ⓓ. Register



ⓔ. Register indirect



ⓕ. Displacement



ⓖ. Stack

A = memory location address
R = Register
EA = actual (effective) address of location containing referenced operand.
X = value

Mode	Algorithm	Advantage	Disadvantage
① Immediate	Operand given in instruction only	No memory reference	limited operand magnitude
② Direct	EA = A	Simple	limited address space
③ Indirect	EA = [A]	Large address space	Multiple memory references
④ Register	EA = R	No memory reference	limited address space
⑤ Register Indirect	EA = [R]	Large address space	Extra memory reference

Mode	Algorithm	Advantage	Disadvantage
6. Displacement	EA = X + [R]	flexibility	Complexity
7. Stack	EA = top of Stack	No memory reference	Limited applicability

1. Immediate Addressing
   ↳ Operand value is given directly in instruction

e.g. MOVE #200, R0 // [R0] = 200

- used to define and use constants or set initial values of variables.

Adv no memory reference other than instruction fetch is required to obtain operand, ∴ saving one memory or cache cycle in instruction cycle.

Disadv Size of number is restricted to size of address field (generally small as compared to word length).

Date / /
Page No.

2. Direct Addressing
   ↳ operand is in memory location and address of that location is give directly in instruction.

e.g. Move LOC, R2 // R2 ← [LOC]

LOC = memory locat address

- used to represent global variables in a program.

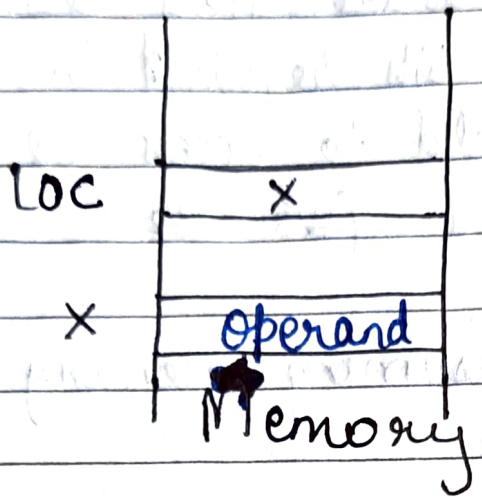
Adv. Requires only one memory reference & no special calculation.

Disadv Limited address space.

3. Indirect Addressing
   ↳ used to resolve limited address range of direct addressing
   ↳ Effective address of operand is contents of a memory location whose address appears in instruction

EA = [A]

e.g. Move [LOC], R1 // R1 ← [[LOC]]



Adv word length = N
then, address space available = 2^N

Disadv Requires two memory reference to fetch operand; one to get its address and second to get its value.

4. Register Addressing
↳ operand is the content of a processor register.
↳ Name of register is given in the instruction

e.g. Move R1, R2 // R2 ← [R1]

Adv

- Only small address field needed in instruction
  e.g. 5 bits required in address field to access 32 registers ($2^5 = 32$)
- No time consuming memory references are required.

* Used when operand in register remains in use for multiple operations
  e.g. intermediate result in a calculation

5. Register Indirect Addressing
   ↳ effective address of operand is contents of a register mentioned in instruction

e.g. Move [R2], R1 // R1 ← [[R2]]

R1, R2 ⇒ registers

Adv
- uses one less memory reference than indirect addressing.
- address space available = 2^N if word length = N

Disadv

Extra memory reference as compared to immediate & register addressing.

6. Displacement Addressing

$$EA = X + [R]$$

→ requires that instruction have two address fields, at least one of which is explicit.

→ value contained in one address field (value = X) is used directly.

→ other address field (or an implicit reference based on opcode) refers to a register whose contents are added to X to produce effective address.

Types of displacement addressing →
- Relative addressing
- Base-register addressing
- Indexing

(ii) Relative Addressing (PC relative addressing)

→ Effective address of operand is generated by adding a constant value to contents of program counter (PC).

→ This mode can be used to access data operands.

But most common use is to specify target address in branch instructions

e.g. Branch >0 LOOP

↳ program execution go to branch target location identified by name LOOP if branch condition is satisfied

EA of this location can be computed by specifying it as an offset from current value of PC.

$$EA = [PC]_{\text{current value}} + \underbrace{LOOP}_{\text{offset}}$$

Imp As branch target may be either before

Date / /
Page No.

or after the branch instruction, offset is given as a signed number.

another representation

$$X\,[PC]$$

$$EA = X + [PC]$$

X → offset or constant value

Adv

If most memory references are relatively near to the instruction being executed, then use of relative addressing saves address bits in the instruction.

(ii) Base-register Addressing

→ Referenced register contains a main memory address and address field contains a displacement (usually an unsigned integer representation) from that address.

Register reference may be explicit or implicit.

Notation

$$X\,[R_B]$$

$$EA = X + [R_B]$$

X → displacement

R_B → Referenced or base register

→ Base-register addressing is a convenient mean of implementing Segmentation

- In some implementations, a single segment base register is employed & is used implicitly.
- In some other, programmer may choose a register to hold base address of a segment, and the instruction must reference it explicitly
  ↳ In this case, if length of address field = k

No. of possible registers = N

then one instruction can reference any one of N areas of 2^k words

Segmentation
↳ way in which addressable memory can be subdivided
↳ visible to programmer
↳ convenient method for organizing programs and data

→ Segmentation allows programmer to view memory as consisting of multiple address spaces or segments.

Segments ⟶ variable size

- Data and program are assigned to different segments

(iii) Indexing

↳ effective address of operand is generated by adding a constant value to contents of a register

↳ referred to as index register

Index mode indicated symbolically as

$X[R_i]$

where X = constant value contained in instruction

R_i = index register

∴ effective address, $EA = X + [R_i]$

Imp Contents of index register are not changed in process of generating effective address

Use of indexing

↳ provide an efficient mechanisms for performing iterative operations

e.g. list of numbers stored starting at location A

→ Suppose user wants to add 1 to each element on list

∴ fetch each value, add 1 to it, and store it back

Sequence of EA required → A, A+1, A+2 ..... upto last location on list

using indexing: Value A is stored in the instruction's address field

→ while chosen register, called an index register is initialized to 0.

→ After each operation, index register is incremented by 1.

Because index registers are commonly used for such iterative tasks, it is typical that there is a need to increment or decrement index register after each reference to it.

Autoindexing: system automatically do this as part of same instruction cycle.

Autoindexing using increment

$$EA = A + [R]$$
$$[R] \leftarrow [R] + 1$$

- In some machines, both indirect addressing and indexing are provided, and it is possible to employ both in same instruction.

Two possibilities — indexing performed before } the indirection
indexing " after }

Case 1: Indexing performed after indirection
↳ postindexing

$$EA = [A] + [R]$$

First, contents of address fields are used

to access a memory location containing a direct address.

→ This address is then indexed by register value.

- This technique is useful for accessing one of a number of blocks of data of a fixed format.

<u>case 2</u>: Indexing performed before indirection
↳ preindexing

$$EA = [A + [R]]$$

↳ address calculated using simple indexing

→ calculated address contains not the operand but address of operand.

- Use of this technique is to construct a multiway branch table.

* Typically, an instruction set will not include both preindexing and postindexing.

7. Stack Addressing

Stack → linear array of locations

↓

Sometimes known as pushdown list or last-in-first out queue

→ Stack is reserved block of locations

Associated with stack is a pointer whose value is address of top of the stack

↳ Stack pointer

→ It is a form of implied addressing. The machine instructions need not include a memory reference but implicitly operate on top of stack.

Instruction Set Architectures
- RISC
- CISC

RISC (Reduced Instruction Set Computer)

CISC (Complex Instruction Set Computer)

Reduced Instruction Set Computer Architecture (RISC)

↳ Simple hardware as instruction set composed of a few basic steps for loading, evaluating, and storing operations

→ Reduce cycles per instruction at the cost of number of instructions per program

Characteristics of RISC

- uses small and limited number of instructio
- Simpler instruction, hence simple instruction decoding
- Instruction is of uniform fixed length and comes under size of one word.
- Instruction takes a single clock cycle to get executed.
- More number of general-purpose registers
- Simple Addressing modes

- Less data types
- Mostly uses hardwired control unit

e.g. RISC processors : IBM RS6000, MC88100

Complex Instruction Set Computer Architecture (CISC)

↳ a single instruction does all loading, evaluating, and storing operations like a multiplication command.

→ ∴ it is complex

→ attempts to minimize number of instructions per program but at the cost of increase in number of cycles per instruction

Characteristics of CISC → • large no. of different & complex instructions

- Complex instruction, hence complex instruction decoding
- Instructions are larger than one-word size
- Instructions may take more than a single clock cycle to get executed.
- Limited number of registers as operation get performed in memory itself.

- use of complex addressing modes
- More data types
- Mostly uses micro-program control unit. Sometimes hardwired control unit is also used.

e.g. CISC processors : Intel 386, 486, Pentium

Example: Suppose two 8-bit numbers are to be added.

RISC approach: Write command to load data in registers
↓
then use suitable operator
↓
finally store result in desired location.

CISC approach: Single command or instruction will perform task (load, operate & store)

Date / / Page No. 6

Differences b/w RISC and CISC

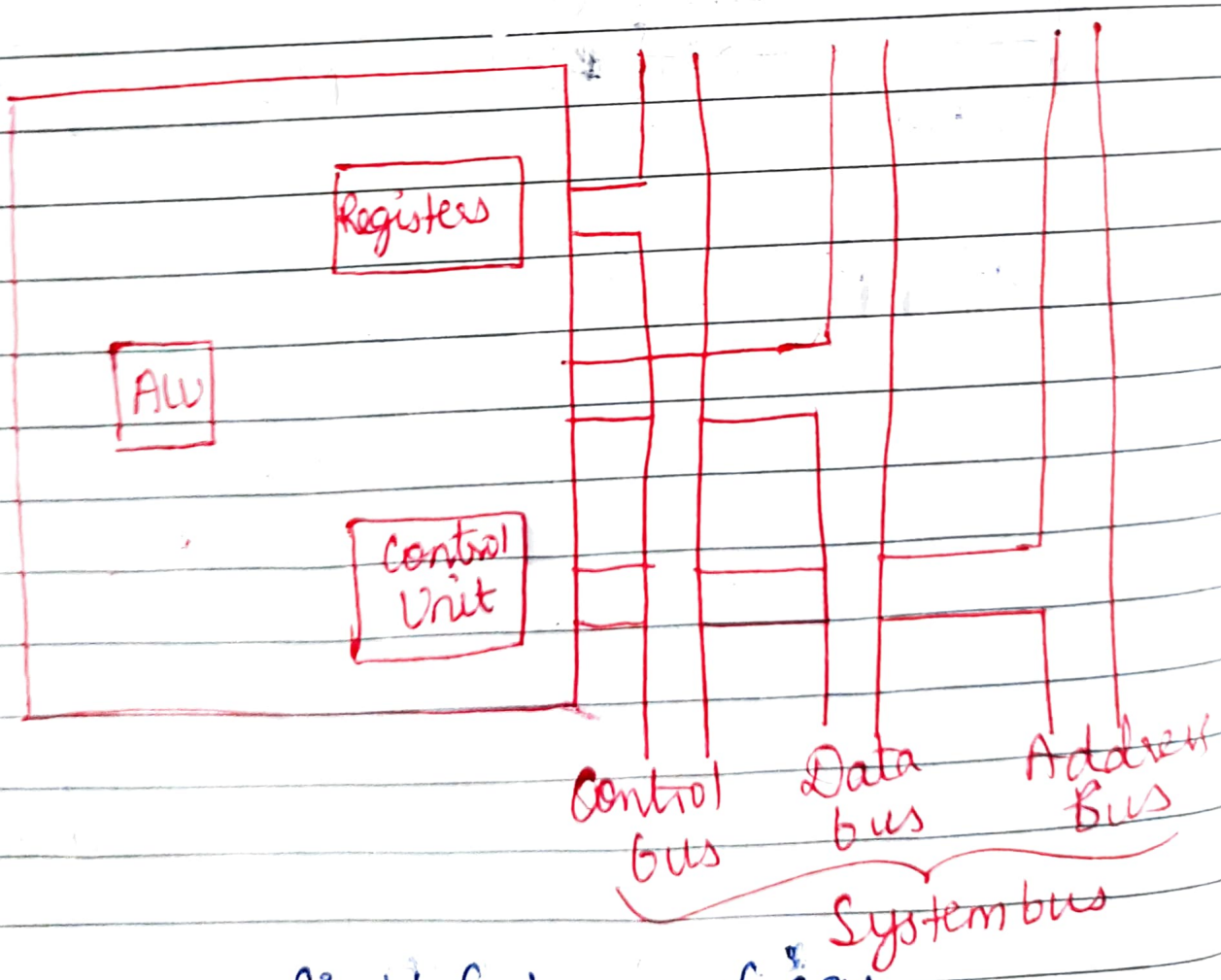
RISC	CISC
• focus on software	• focus on hardware
• Uses only hardwired control unit	• Uses both hardwired and microprogrammed control unit
• fixed sized instructions	• Variable sized instructions
• Can perform only Register to register arithmetic operations	• Can perform Reg to Reg or Reg to Mem or Mem to Mem operations
• Requires more number of registers	• Requires less number of registers
• Code size is large	• Code size is small
• Single clock cycle used to execute an instruction	• More than one clock cycle used to execute an instruction
• Instruction fit in one word	Instruction requires more than one word space

Unit 4: Processing Unit

Processor Organization

Processor does following tasks :

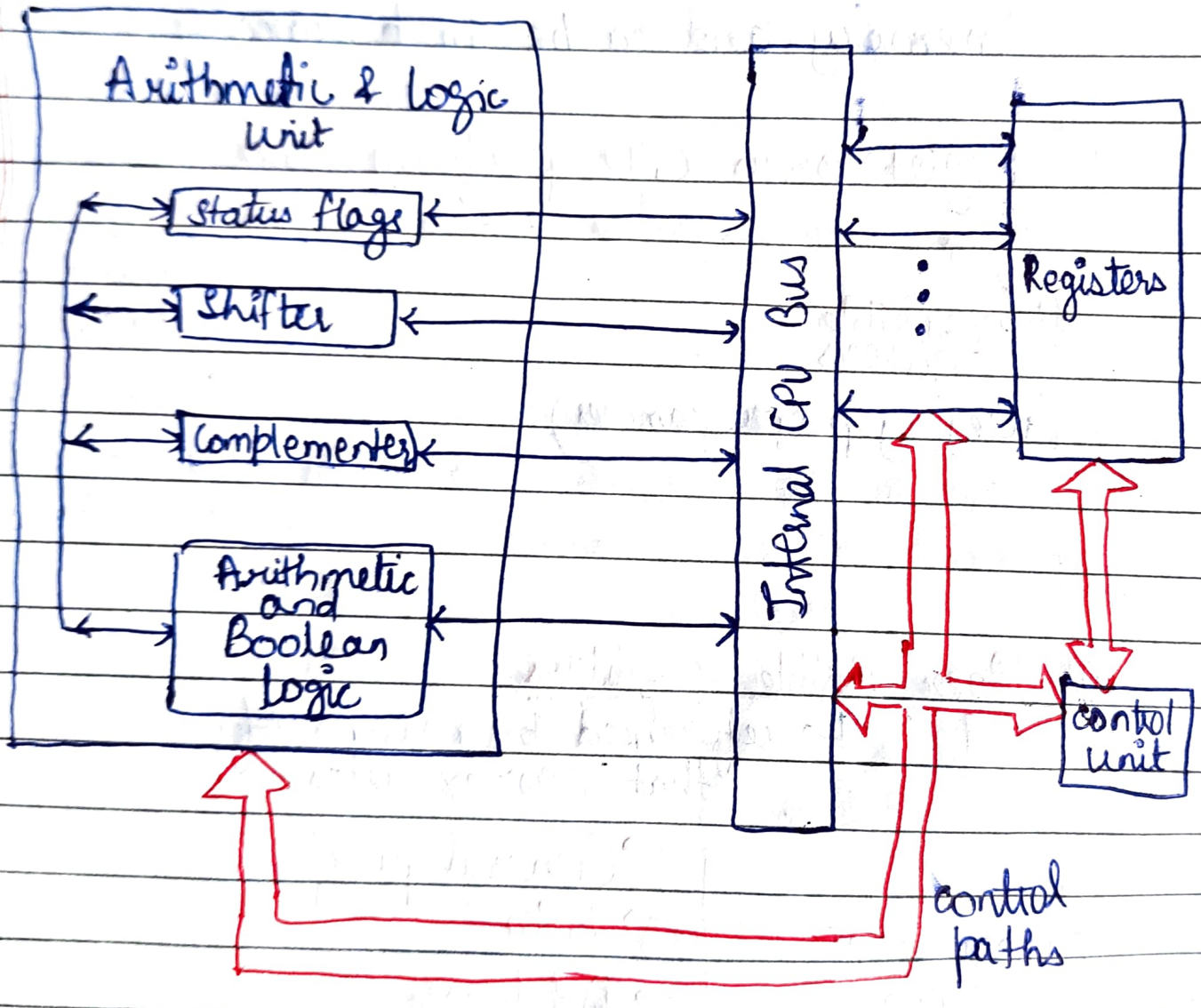
- → fetch instruction from memory
- → Interpret instruction (decode)
- → fetch data from memory or an I/O module
- → Process data using arithmetic or logical operation
- → Write data to memory or I/O module



Simplified view of CPU

Date / /
Page No.

→ ALU does actual computation or processing of data
→ control unit controls movement of data & instructions into and out of CPU & controls operation of ALU
→ Minimal internal memory, consisting of a set of storage locations, called registers



Detailed Internal structure of CPU

Register Organization

→ Computer System uses a memory hierarchy. At higher levels of hierarchy, memory is faster, smaller, and more expensive (per bit)

→ Within CPU, there is a set of registers that function as a level of memory above main memory and cache in hierarchy.

Registers in CPU perform two roles

- User visible registers (used by proprogrammer)
- Control and status registers (used by control unit to control operations of CPU)

1. User visible registers
   - ↳ referenced by means of machine language that CPU executes
   - categorized into
     - → General purpose
     - → Data
     - → Address
     - → Condition Codes

(i). General Purpose registers
- can be used to contain operand for any opcode
- can be used for addressing functions (e.g. register indirect, displacement)

(ii). Data registers used only to hold data and cannot be employed in calculation of an operand address.

(iii). Address Registers (somewhat general purpse, or they may be devoted to a particular addressing mode.

Examples include following
- Segment pointers: holds address of base of segment in machine with segmented addressing
- Index registers: used for indexed addressing and may be autoindexed.
- Stack pointer: If there is user-visible stack addressing, then typically stack is in memory and there is a dedicated register that points to top of stack.

This allows implicit addressing; i.e., push, pop and other stack instructions need not contain an explicit stack operand.

(iv). Condition codes (also referred to as flags)
↳ partially visible to user

→ bits set by CPU hardware as the result of operation.

eg. an arithmetic operation may produce a positive, negative, zero, or overflow result. In addition to the result itself being stored in a register or memory, a condition code is also set.

→ The code may be subsequently be tested as part of a conditional branch operation.

→ Condition code bits are collected into one or more registers. Usually they form part of a control register.

→ Generally, machine instructions allow these bits to be read by implicit reference, but programmer cannot alter them.

Date / /
Page No.

(2). Control and Status Registers

- used to control operation of CPU.
- Most of these are not visible to user
- Some of them may be visible to machine instructions executed in a control mode.

Four registers are essential to instruction execution:

(i). Program Counter (PC) contains address of an instruction to be fetched.

(ii). Instruction register (IR) contains instruction most recently fetched.

(iii). Memory address register (MAR) contains address of a location in memory.

(iv). Memory buffer register (MBR) contains a word of data to be written to memory or word most recently read.

→ Typically CPU updates PC after each instruction fetch so that PC always points to next instruction to be executed.

→ A branch or skip instruction will also modify contents of PC.

→ fetched instruction is loaded into an IR, where opcode and operand specifiers are analyzed.

→ Data are exchanged with memory using MAR and MBR

→ In bus-organized system, MAR connects directly to address bus, & MBR connects directly to data bus.

→ User-visible registers, in turn, exchange data with MBR.

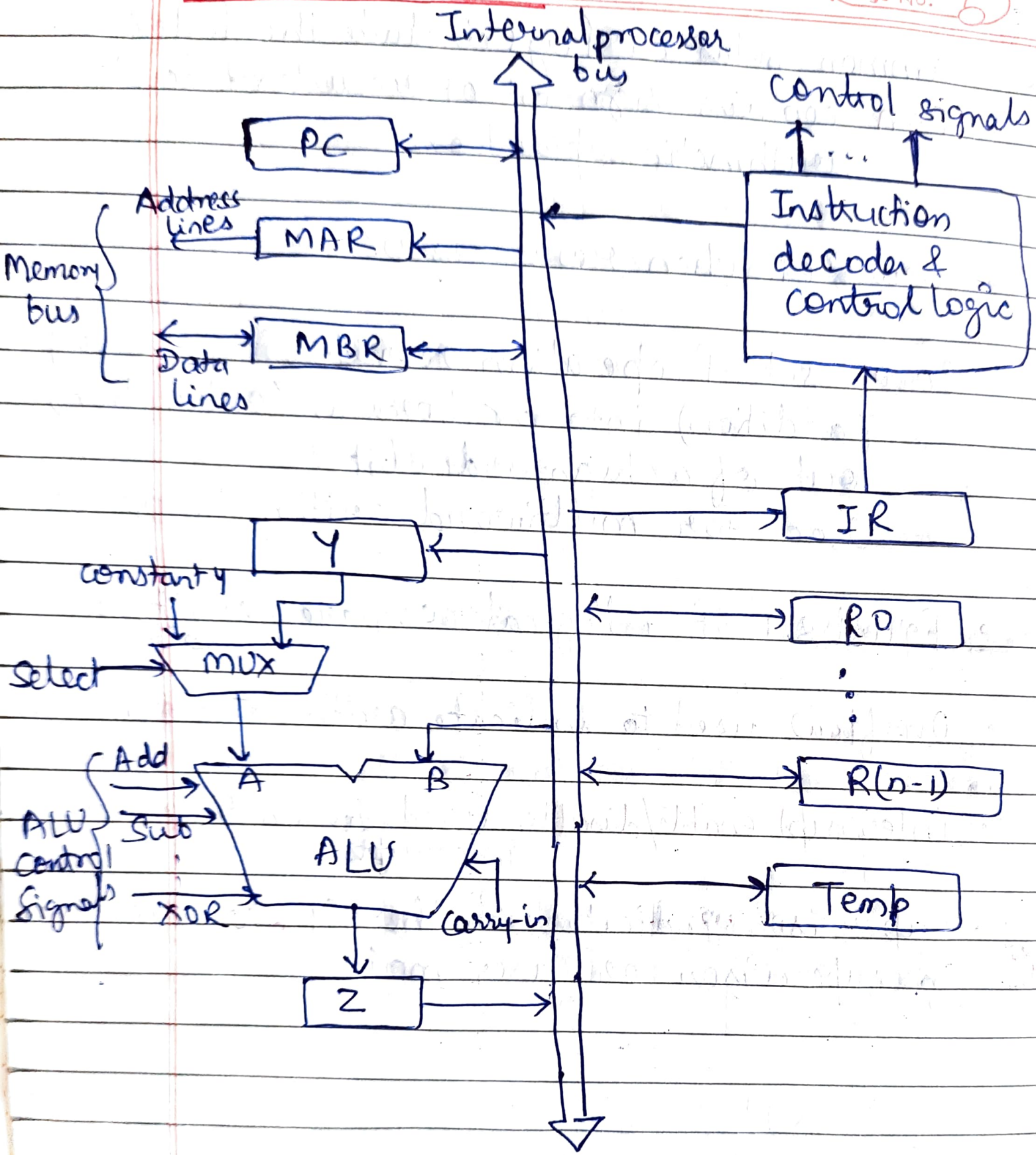
→ All CPU designs include a register or set of registers, often known as program status word (PSW), that contain status information

→ PSW typically contains condition codes plus other status information.

Common fields or flags include the following:

- Sign contains sign bit of result of last arithmetic operation
- Zero set when result is 0.
- Carry set if operation resulted in a carry (addition) into or borrow (subtraction) out of a high-order bit.
  Used for multiword arithmetic operation
- Equal set if a logical compare result is equality
- Overflow used to indicate arithmetic overflow
- Interrupt enable/disable used to enable or disabl interrupts.
- Supervisor indicates whether CPU is executing in supervisor or user mode.

Data Path in CPU



Single bus organization of datapath inside processor

Three registers Y, Z, and temp are transparent to programmer, i.e., programmer need not be concerned with them because they are never referenced explicitly by an instruction.

↳ they are used by processor for temporary storage during execution of some instructions.

constant 4 is used to increment contents of PC assuming memory is byte addressable

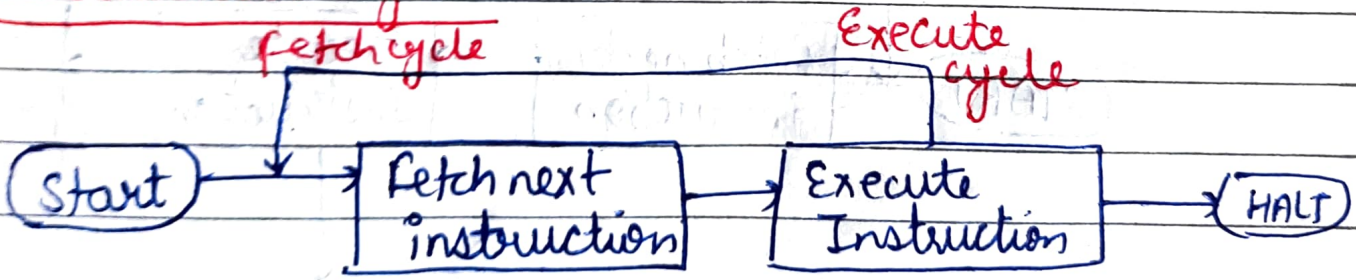
∴ PC increment its contents by 4 after execution of an instruction

$PC \leftarrow [PC] + 4$

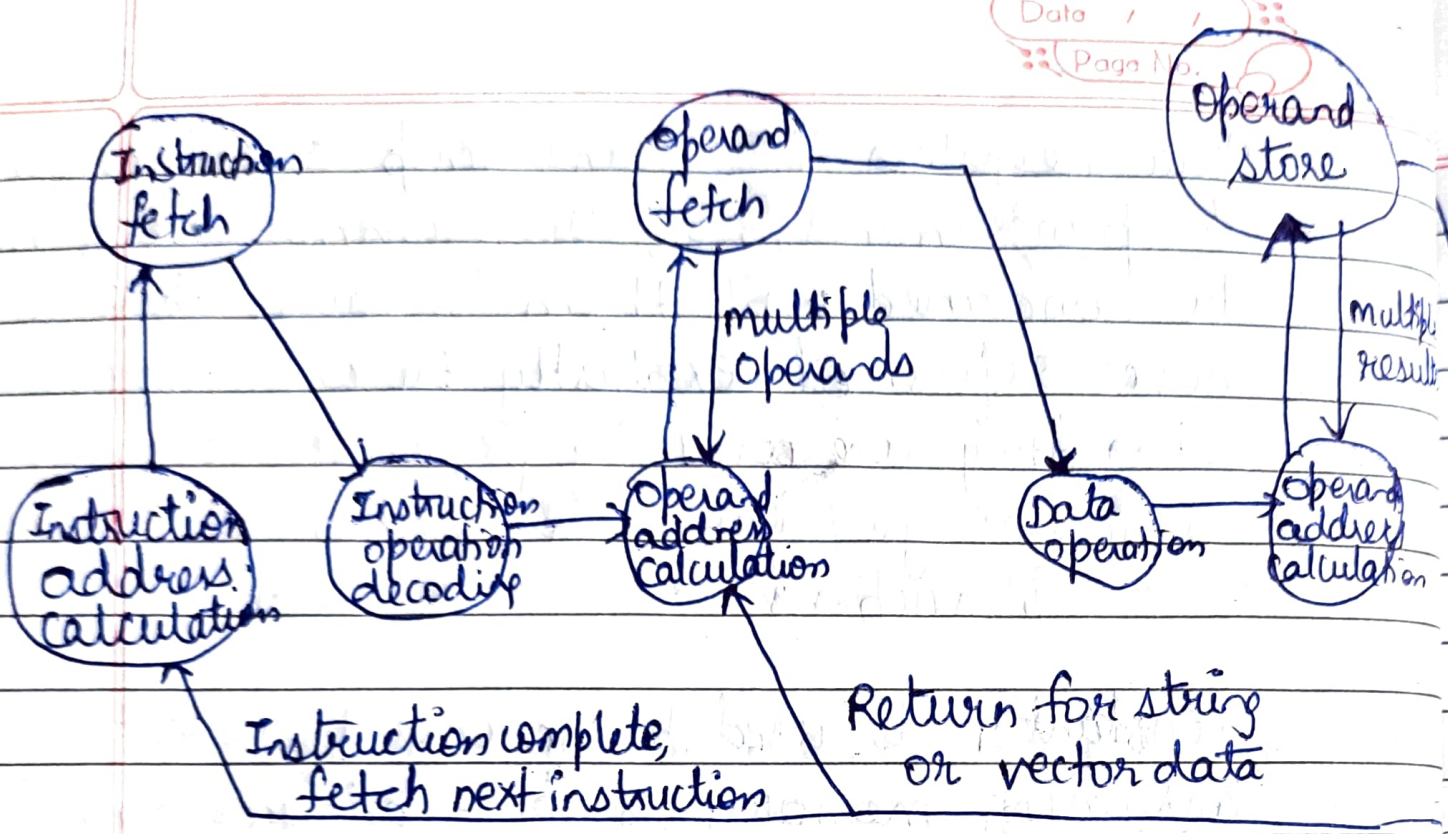
→ Registers, ALU and interconnecting bus are collectively refered to as datapath.

Instruction cycle

(I).

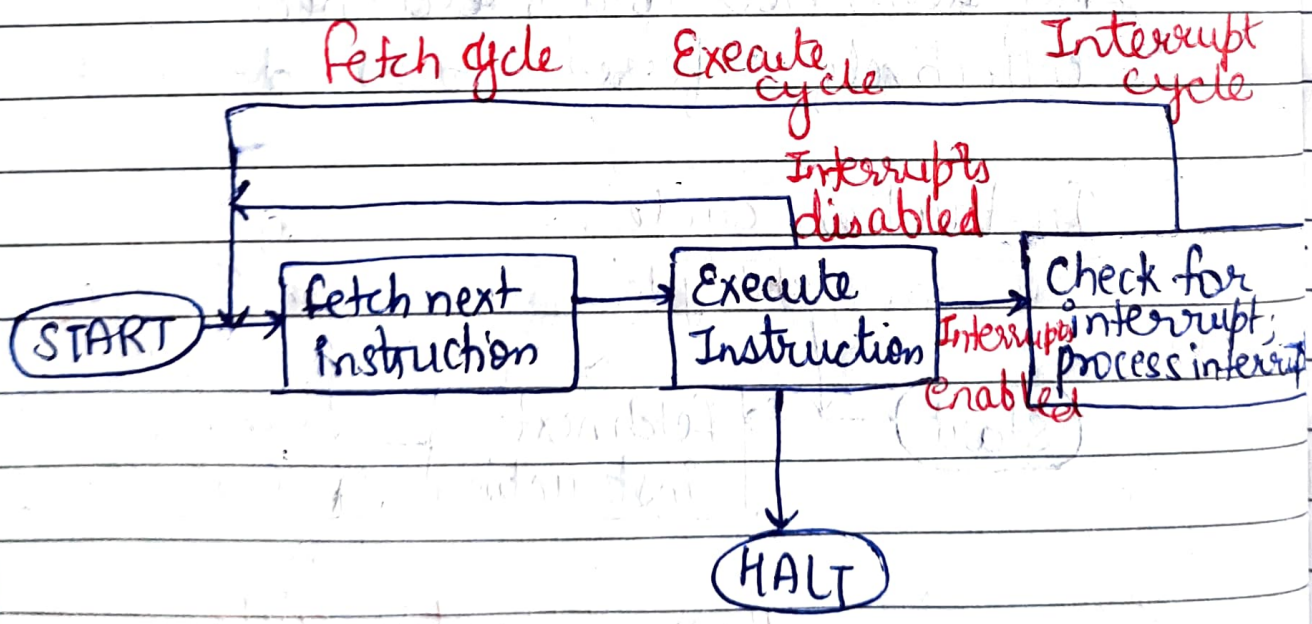


Basic Instruction cycle

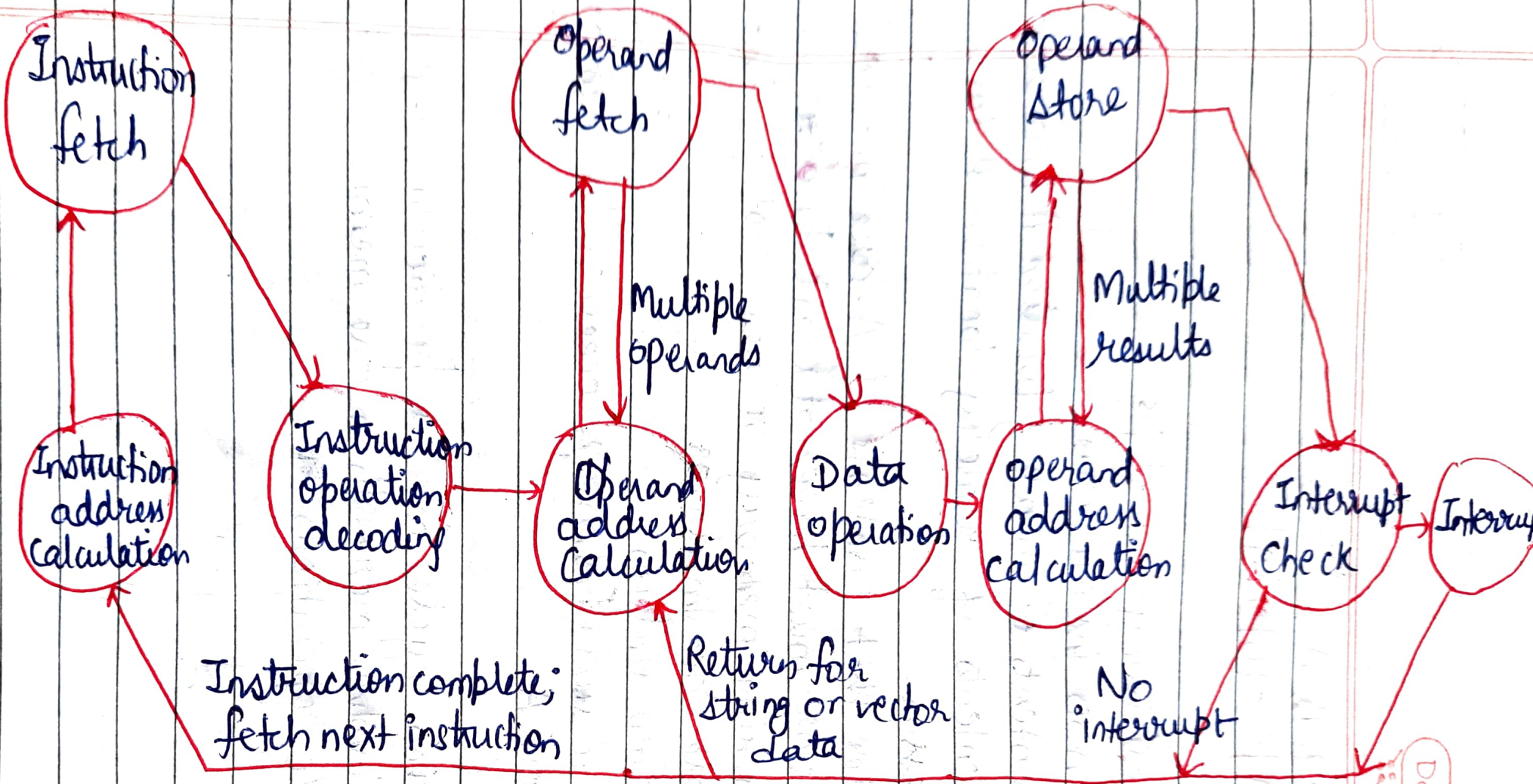


Instruction cycle state diagram

(II)



Instruction Cycle with Interrupts



Instruction Cycle State Diagram with Interrupts

Date / /
Page No.

e.g. ADD B, A

$[A] \leftarrow [B] + [A]$

Instruction cycle steps

1. Fetch ADD instruction
2. Read contents of memory location A into processor
3. Read contents of memory location B into processor.

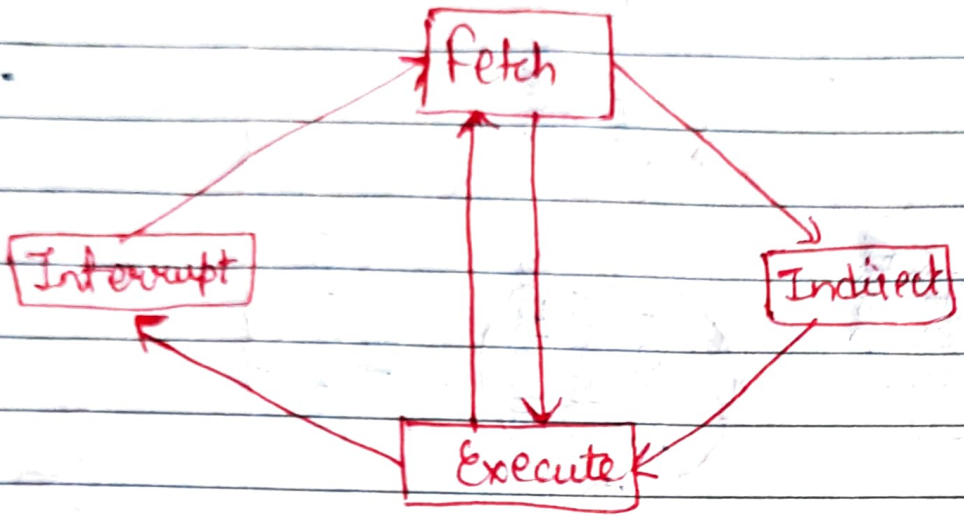
*In order that contents of A are not lost, processor must have at least two registers for storing memory values, rather than a single accumulator

4. Add two values
5. Write result from processor to memory location A.

Imp: Execution cycle for a particular instruction may involve more than one memory reference or I/O operations.

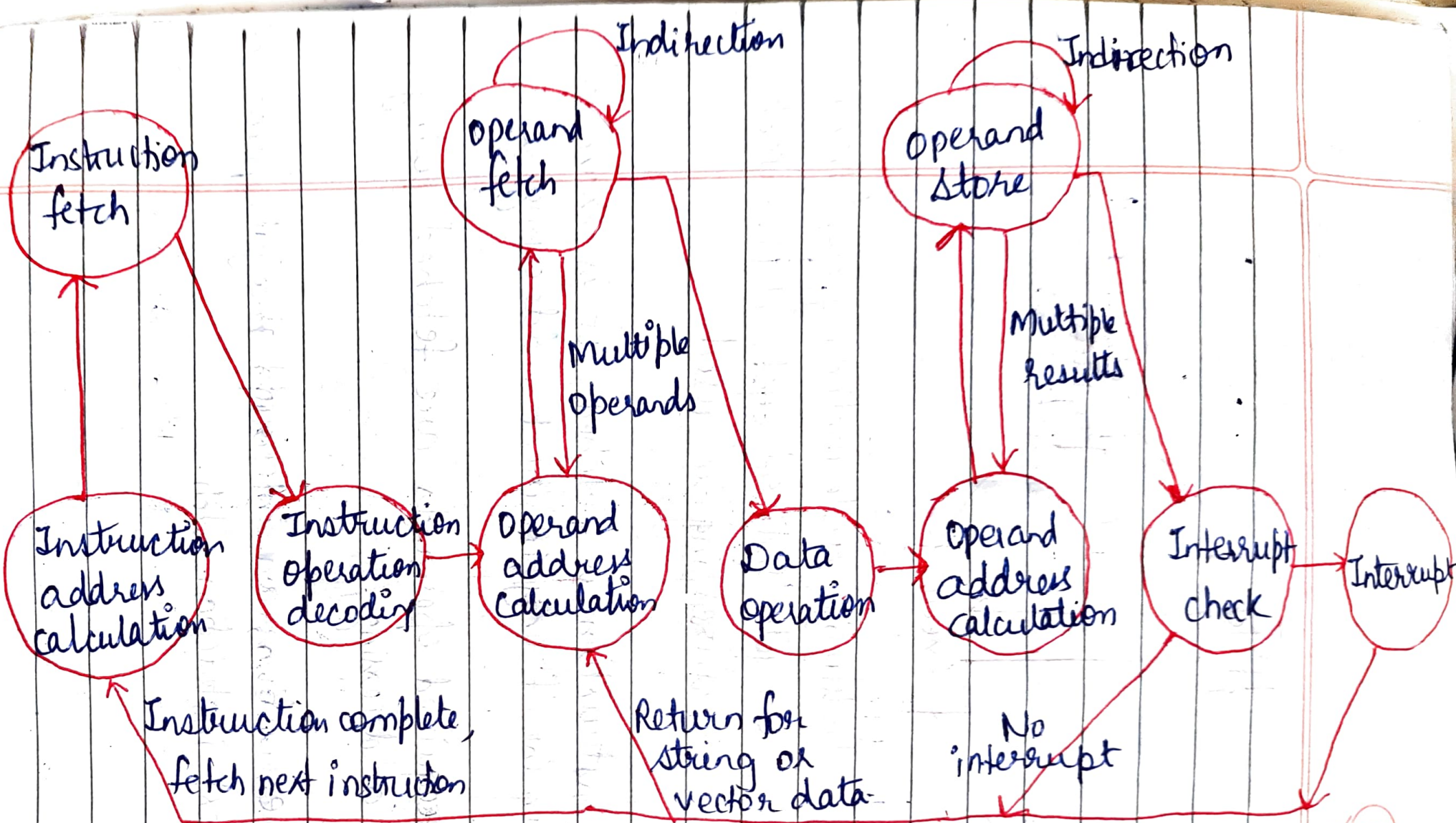
Date / /
Page No. 6

(III)



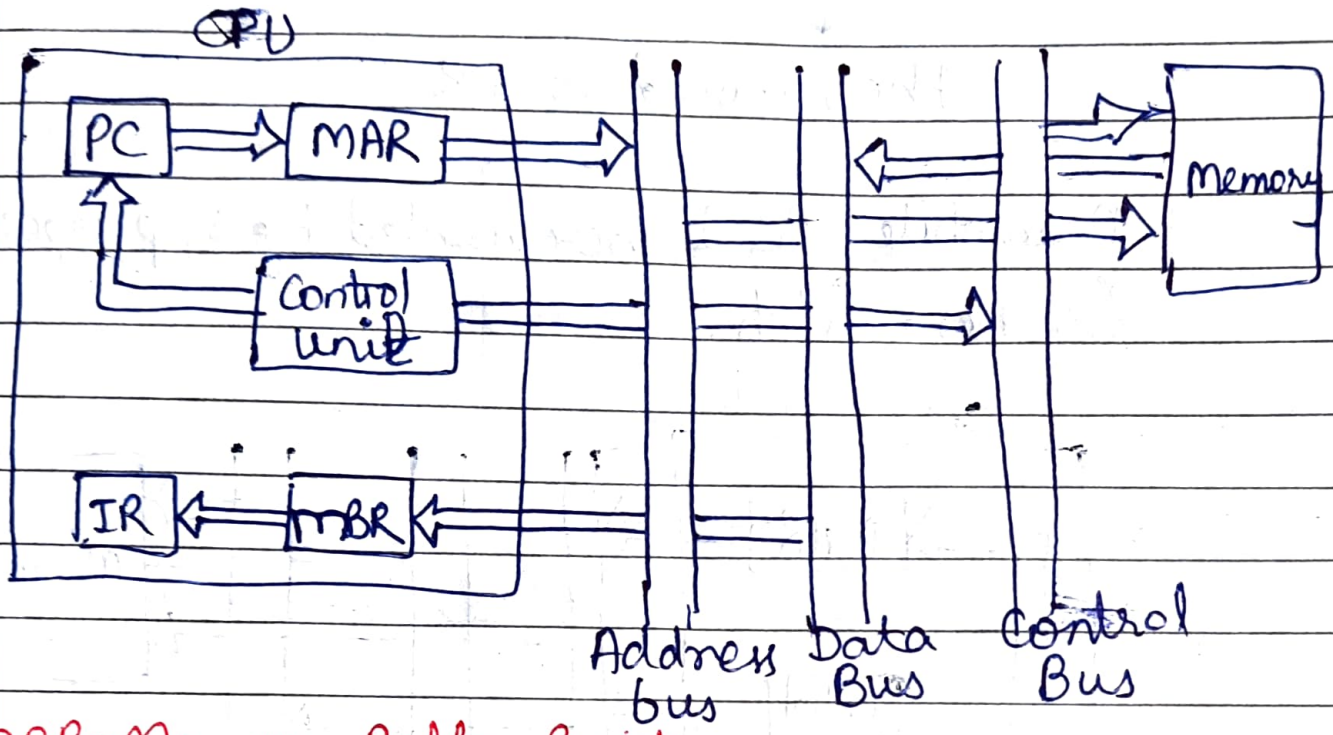
Instruction cycle including additional Subcycle Indirect cycle if indirect addressing is used

- After an instruction is fetched, it is examined to determine if any indirect addressing is involved.
- If so, required operands are fetched using indirect addressing.
- following execution, an interrupt may be processed before next instruction fetch.



Instruction cycle state diagram including interrupt & indirect cycle

Data flow



MBR = Memory Buffer Register
MAR = Memory Address Register
IR = Instruction Register
PC = Program counter

Data flow, fetch cycle

During fetch cycle, → instruction is read from memory.

PC contains address of next instruction to be fetched

↓

This address is moved to MAR & placed on address bus

↓

Control unit requests a memory read

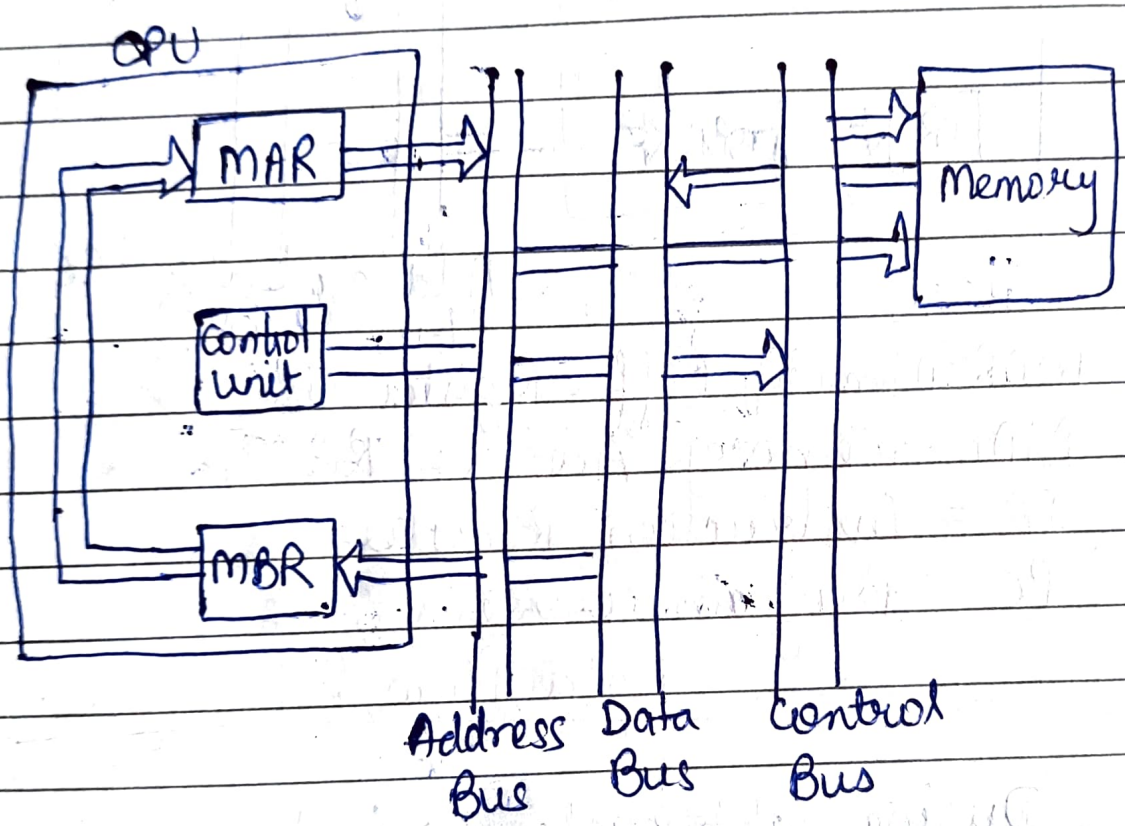
↓

Result is placed on data bus & copied into MBR

↓

then moved to IR

Meanwhile PC is incremented by 1, preparatory for next fetch.



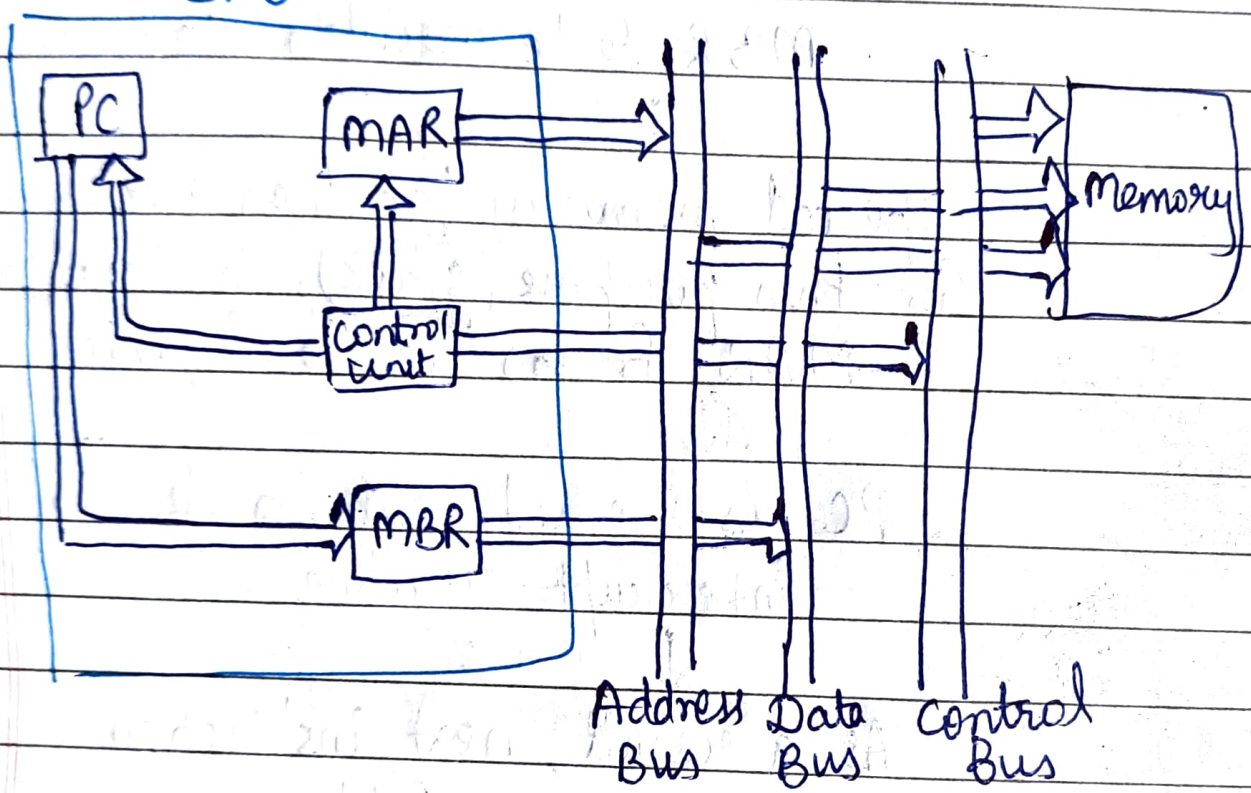
Data flow, Indirect Cycle

Once fetch cycle is over, control unit examines contents of IR to determine if it contains an operand specifier using indirect addressing. If so, indirect cycle is performed.

Rightmost N bits of MBR, which contains address reference are transferred to the MAR

↓

Then control unit requests a memory read to get desired address of operand into MBR



Data flow, Interrupt Cycle

Execute cycle takes many forms depending on various machine instructions. It may involve transferring data among registers, read or write from memory or I/O, and/or invocation of ALU

Date / /
Page No.

During interrupt cycle

Current contents of PC must be saved so that CPU can resume normal activity after interrupt

↓

∴ contents of PC are transferred to MBR to be written into memory

↓

Special memory location reserved for this purpose (Stack pointed by SP) is loaded into MAR from control unit

↓

PC is loaded with address of interrupt routine

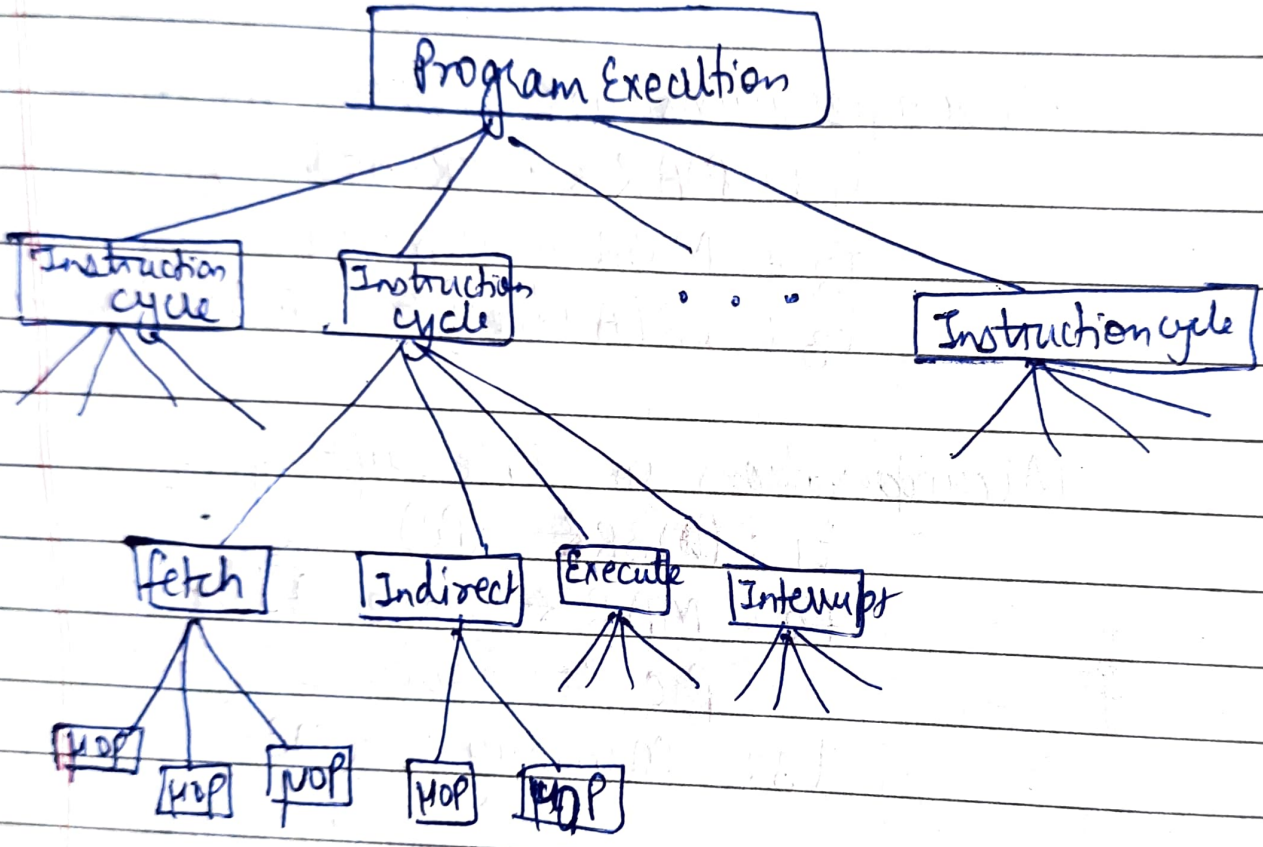
↓

As a result, next instruction cycle will begin by fetching appropriate instruction.

Date / /
Page No.

Micro-operations

- functional or atomic operations of a processor
- execution time sequence of instructions



Constituent Elements of a Program Execution

concept of micro-operations —serves as guide→ to design of control unit

Date / /
Page No.

Microoperation in fetch cycle

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

Microoperations in Indirect cycle

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

Microoperations in Interrupt Cycle

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

Microoperation in Execute cycle

Consider add instruction

Add R1, X
(R1) ← (R1) + (X)
X → memory loc.

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

Control Unit of processor

- **Functional Requirements**

Three step process leading to characterization of control unit are as follows:

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

Basic elements of processor:

- ALU
- Registers
- Internal data paths (used to move data b/w registers and b/w register & ALU)
- External data paths (link registers to memory & I/O modules by means of system bus)
- control unit
  - ↳ causes operations to happen within processor

Microoperations fall into one of following categories

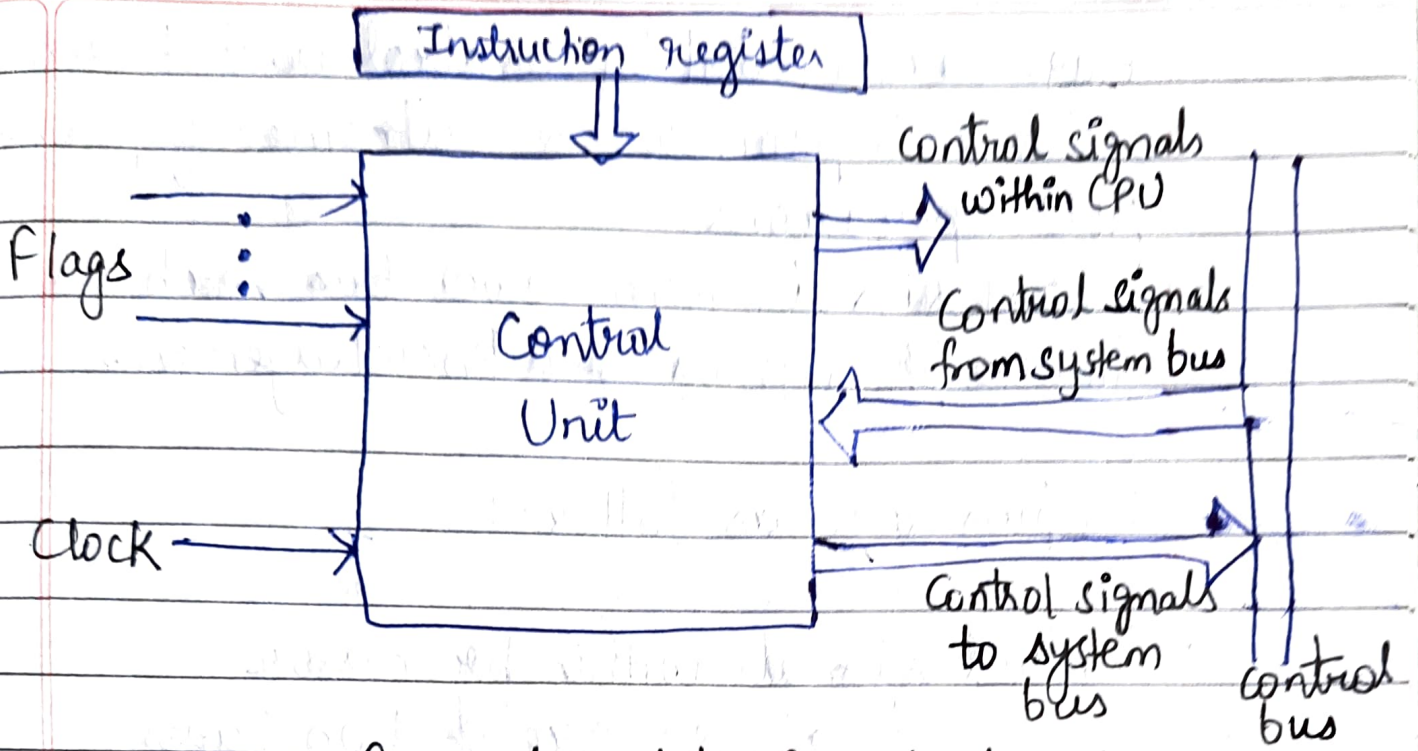
→ Transfer data from one register to another

→ Transfer data from a register to an external interface (eg. system bus)

→ Transfer data from an external interface to a register
→ Perform an arithmetic or logic operation, using registers for input & output.

∴ Control unit perform two basic tasks:
- Sequencing: Control unit causes processor to step through a series of microoperations in proper sequence, based on program being executed.
- Execution: Control unit causes each micro-operation to be performed.

The key to how control unit operates is use of control signals
↓
- control unit must have i/ps that allow it to determine state of system and o/ps that allow it to control behaviour of system
  ↳ these are external specifications of control unit.
- Internally control unit must have the logic required to perform its sequencing & execution functions.



General model of control unit

Inputs are as follows

- Clock : This is how control unit "keeps time". Control unit causes one micro-operation (or a set of simultaneous micro-operations) to be performed for each clock pulse.
  ↳ known as processor cycle time or clock cycle time

- Instruction register (IR) → opcode of current instruction is used to determine which micro-operations to perform during execute cycle.

- Flags: are needed by control unit to determine status of processor & outcome of previous ALU operations.
- Control signals from control bus such as interrupt signals & acknowledgements.

Outputs are as follows:

- Control signals within processor
  - these are of two types
    - those that cause data to be moved from one register to another
    - those that activate specific ALU functions
- Control Signals to control bus
  - two types
    - Control signals to memory
    - Control signals to I/O modules

Three types of control signals:
1. those that activate ALU function
2. those that activate datapath
3. those that are signals on external system bus or other external interface.

All of these signals are ultimately applied directly as binary inputs to individual logic gates.

Two techniques for implementation of control unit
- Hardwired Implementation
- Microprogrammed Implementation

1. Hardwired Control Unit
   - control unit is essentially a combinational circuit
   - → Its i/p logic signals are transformed into a set of output logic signals, which are control signals.

   Key i/ps → IR, Clock, flags & control bus signals

→ In case of flags & control bus signals, each individual bit typically has some meaning.

→ Other two i/ps, however, are not directly useful to control unit

Consider IR
↳ control unit makes use of opcode & perform different actions (issue a different combination of control signals) for different instructions

To simplify control logic, there should be a unique logic i/p for each opcode.
↳ performed by a decoder
(takes encoded i/p & produces a single o/p)
↳ converts n binary i/ps to 2^n binary o/ps

- Each of 2^n different i/p patterns will activate a single unique o/p

Consider clock portion
↳ issues a repetitive sequence of pulses.

→ useful f

→ Essential enough data pat

→ However signals instructio

∴ a co with ad T_1, T_2

→ At the unit mu at T_1

→ for eac for tha of inp

→ This p control

→ useful for measuring duration of micro-operations

→ Essentially period of clock pulses must be long enough to allow propagation of signals along data paths & through processor circuitry.

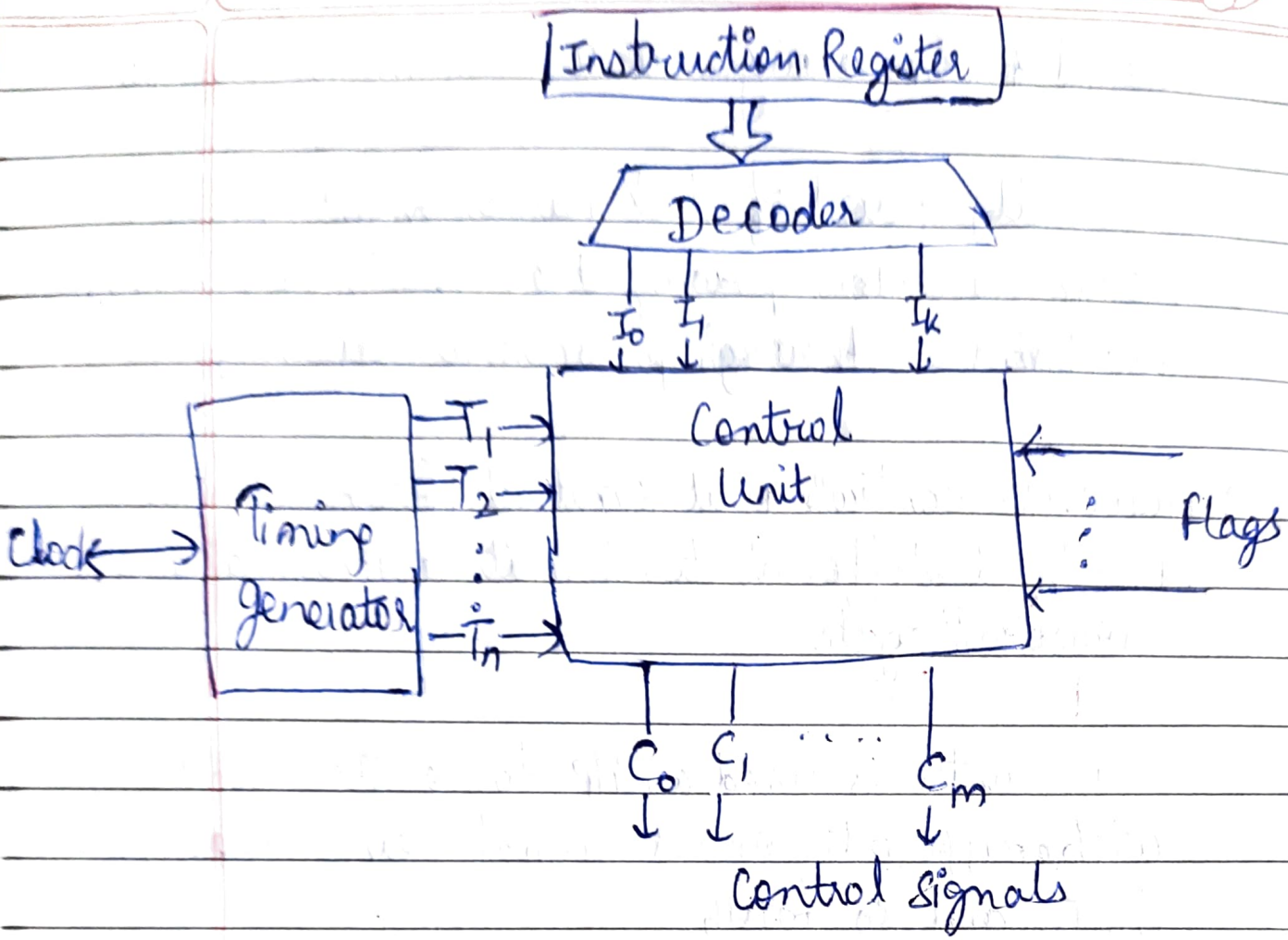
→ However, control unit emits different control signals at different time units within a single instruction cycle.

∴ a counter is used as i/p to control unit with a different control signal being used for T_1, T_2 and so forth.

→ At the end of an instruction cycle, control unit must feedback to counter to reinitialize it at T_1.

→ for each control signal, a Boolean expression for that signal is derived as a function of inputs.

→ This process could be repeated for every control signal generated by processor



Hardwired Control unit with Decoded I/ps

→ The result would be a set of Boolean equations that define behavior of control unit and hence of processor.

→ However in modern complex processor, no. of Boolean equations needed to define control unit is very large.
∴ Task of implementing a combinational circuit

that satisfies all of these equations becomes extremely difficult.

∴ Simpler approach known as microprogramming is usually used.

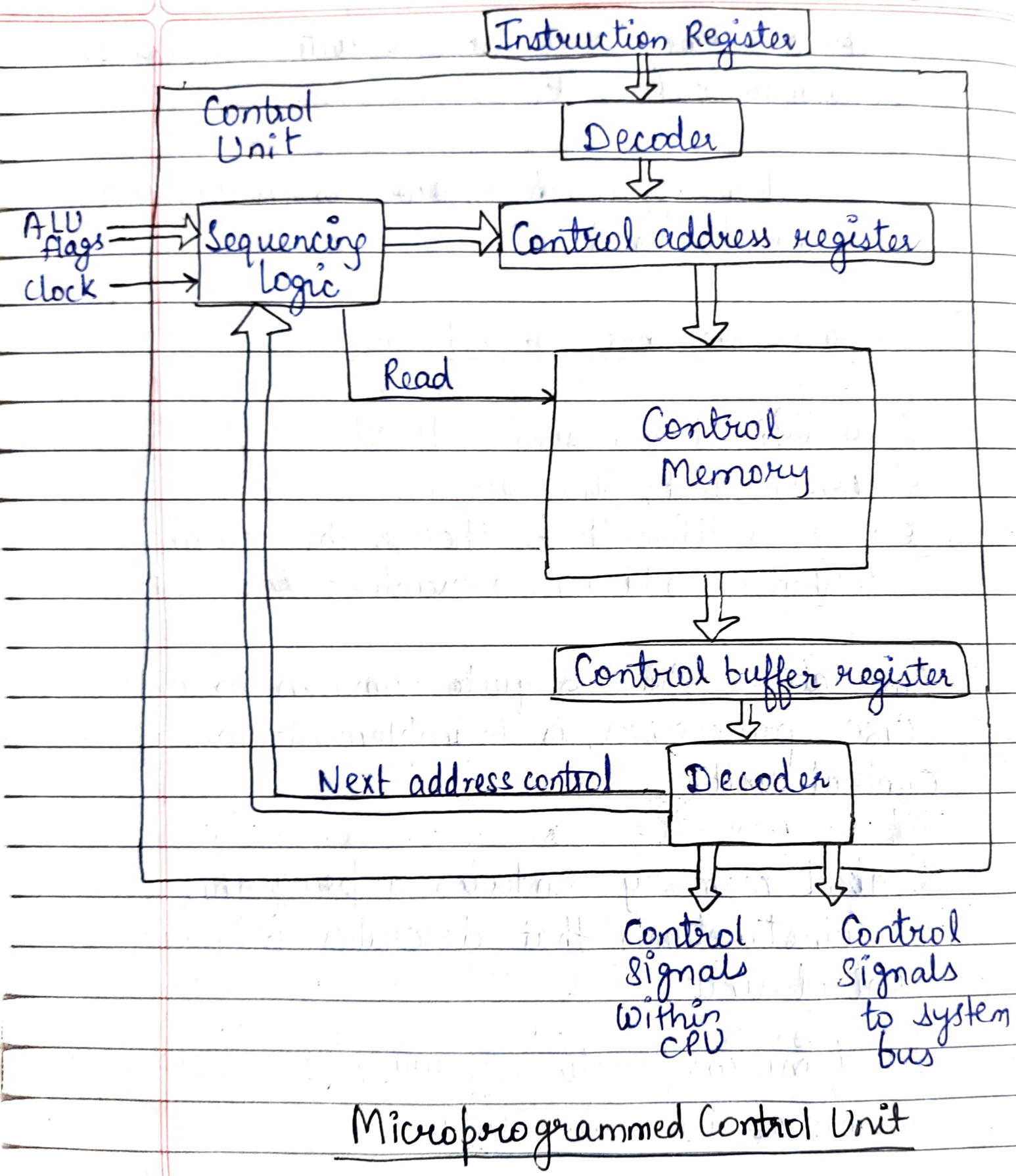
② Microprogrammed Control Unit

It is difficult to design and test complex hardware of control unit. Also design is relatively inflexible.
e.g. It is difficult to change the design if one wishes to add a new machine instruction.

Alternative, which is quite common in contemporary CISC processors, is to implement microprogrammed Control unit.

- Control memory contains a program (set of microinstructions) that describes behavior of control unit.

* Set of micro-operations occuring at one time is known as a microinstruction



Microprogrammed Control Unit

A sequence of instructions is known as a microprogram, or firmware.

- Control address register contains the address of next microinstruction to be read.
- When a microinstruction is read from control memory, it is transferred to a control buffer register.

→ Reading a microinstruction from control memory is same as executing that microinstruction

- Sequencing unit loads control address register and issues read command.

Microprogrammed control unit functions as follows:

1. To execute an instruction, the sequencing logic unit issues a READ command to control memory.
2. The word whose address is specified in the control address register is read into control buffer register.
3. The content of control buffer register generates control signals & next address information for sequencing logic unit.

4. The sequencing logic unit loads a new address into control address register based on the next-address information from the control buffer register and ALU flags.

All this happens during one-clock pulse.

→ At the conclusion of each microinstruction, the sequencing logic unit loads a new address into control address register. Depending on value of ALU flags & control buffer register, one of three decisions is made:

- Get Next instruction: Add 1 to control address register
- Jump to a new routine based on a jump microinstruction: Load address field of control buffer register into control address register.
- Jump to a machine instruction routine: Load control address register based on the opcode in IR.

→ Upper decoder translates opcode of IR into a control memory address.

→ lower decoder translates, code used for each action to be performed into individual control signals.

Disadvantage of microprogrammed control unit is that it will be slower than a hardwired unit of comparable technology.

TCS-404 Computer Organization

NOTES

Date .....

Unit II

Performance

- most important measure of the performance of a computer
  - ↳ how quickly it can execute program
    - ↳ depends on design of its hardware & its machine language instructions

* Performance also affected by compiler translates high level language into machine prog.

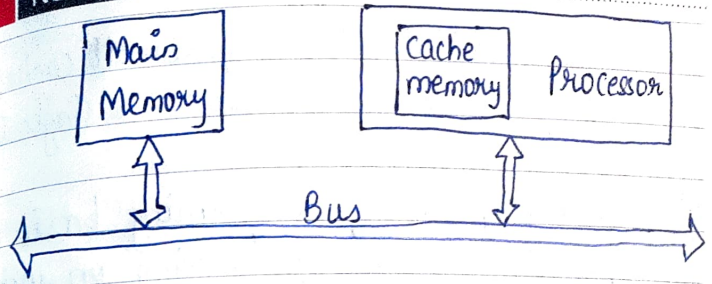
Processor Performance	→	is terms of processor time needed to execute program

- processor time depends on hardware involved in the execution of individual machine instructions.

This hardware involves — processor, memory } usually connected by a bus

NOTES

Date .....



Flow of program instructions and data b/w memory and processor

At start of execution, all program instructions & data required stored in main memory

↓

As execution proceeds, instructions are fetched one by one over bus into processor and copy placed in cache

↓

Required data for execution of instruction fetched from main memory & a copy placed in cache

↓

Later if same instruction or data item is needed a second time, it is read directly from cache

NOTES

Date .....

- Processor and a relatively small cache memory can be fabricated on a single IC.

Imp Internal speed of performing basic steps of instruction processing on such chips is very high and is considerably faster than speed at which instructions and data can be fetched from main memory.

A program will be executed faster if movement of instructions and data b/w main memory and processor is minimized.
↳ that is achieved by using cache.

* Processor Clock
↳ a timing signal that controls processor circuits

regular time intervals after which clock waveform repeats itself → clock cycles

NOTES

Date .....

- To execute a machine instruction, processor divides actions to be performed into a sequence of basic steps, such that each step can be completed in one clock cycle.

Clock rate, $R = 1/P$
↳ measured in cycles per second

where P = length of one clock cycle
↳ imp. parameter that affects processor performance

Currently, processors have clock rates ranging from few hundred million to over a billion cycles per second.
Hz

million → Mega (M)
billion → Giga (G)

e.g. 500 millions cycles per second means
$R = 500\ MHz$

clock period $P = \frac{1}{R} = \frac{1}{500 \times 10^6} = 2\ ns$

NOTES

Date

* Basic Performance Equation

Let T = processor time required to execute a program prepared in some high level language

- compiler generates a machine language object program that corresponds to source program

Assume N machine language instructions are required to be executed to completely execute program.

Imp N → actual no. of instructions executions
↳ not necessarily equal to number of machine instructions in object program.

Some instructions may be executed more than once (in case of instructions inside program loop)

NOTES

Date

others may not be executed at all, depending on input data used.

Suppose, S = average no. of basic steps needed to execute one machine instruction

→ each basic step is completed in one clock cycle.

clock rate = R cycles per second

Program execution time

$$T = \frac{N \times S}{R}$$

↳ Basic performance equation

∴ To achieve high performance, computer designer must seek ways to reduce T

↓

means ↓ N and S and ↑ R

N↓ → if source program is compiled into fewer machine instructions

NOTES

Date

S↓ → if instructions have a smaller no. of basic steps to perform or if execution of instructions is overlapped.

R↑ → by using high frequency clock that means time required to complete a basic execution step is reduced.

Imp N, S and R are not independent parameters; changing one may affect another.

* Pipelining and Superscalar Operation

Program execution time $T = \frac{N \times S}{R}$

S↓ → if execution of instruction is overlapped

- A substantial improvement in performance can be achieved by overlapping execution

NOTES

Date

of successive instructions, using a technique called pipelining

e.g. Add R1, R2, R3 // [R3] ← [R1] + [R2]

contents of R1 and R2 are first transferred to i/p^s of ALU. After add operation is performed, sum is transferred to R3

Processor can read next instruction from memory while addition is being performed by ALU.

Then if that instruction also uses ALU, its operands can be transferred to ALU i/p^s at same time that result of Add instruction is being transferred to R3.

Imp In ideal case, if all instructions are overlapped to maximum degree possible, execution proceeds at the rate of one instruction completed in each clock cycle.

NOTES Date

Individual instructions still require several clock cycles to complete.

* But for purpose of computing T, effective value of $S = 1$
  → difficult to achieve

pipelining ↑ rate of executing instructions significantly and causes effective value of S to approach 1.

Higher degree of concurrency —achieved→ if multiple instruction pipelines implemented in processor

↓

Multiple functional units are used creating parallel paths through which different instructions can be executed in parallel.

NOTES Date

In such arrangement, it is possible to start execution of several instructions in every clock cycle

→ This mode of operation is called superscalar execution

↓ if sustained for a long time during program execution

effective value of S < 1

* Clock Rate

Two ways to ↑ clock Rate (R)

- improving IC technology thus making faster logic circuits, ↓ time needed to complete a basic step. P ↓ thus R ↑
- ↓ amount of processing done in one basic step thus ↓ P
  (disadv no. of basic steps needed may ↑ if actions to be performed by instruction are

Date

In presence of cache, % of accesses to main memory is small. ∴ performance gain ↑ from use of faster technology.

* **Instruction Set** : CISC and RISC

RISC	CISC
• Simple instructions → require small no. of basic steps to execute	• Complex instructions → larger no. of steps
• large value of N and small value of S	• Lower value of N and larger value of S

Complex instructions combined with pipelining would achieve better performance

pipelining → S ≈ 1

However, it is much easier to implement efficient pipelining in processors with simple instruction sets.

NOTES

Date

* **Compilers**

To reduce N, → need of suitable machine instruction set and a compiler that makes good use of it

An optimizing compiler takes advantage of various features of target processor to reduce product $N \times S$

→ total no. of clock cycles needed to execute a program

The compiler may rearrange program instruction to achieve better performance.

* **Performance Measurement**

Comp