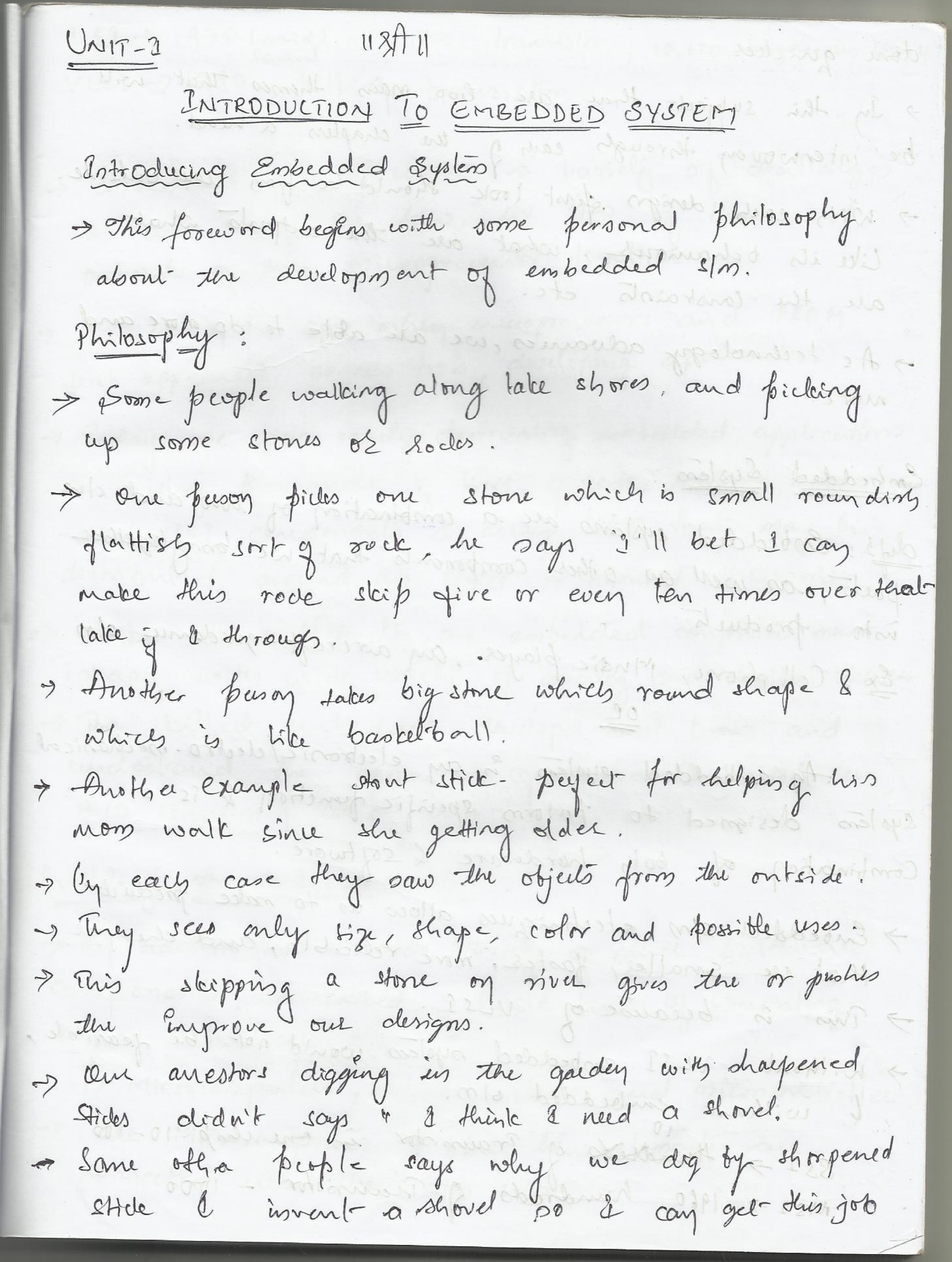


UNIT-1 ॥ श्री ॥

# INTRODUCTION TO EMBEDDED SYSTEM

## Introducing Embedded System

→ This foreword begins with some personal philosophy about the development of embedded s/m.

## Philosophy :

→ Some people walking along lake shores, and picking up some stones or rocks.

→ One person picks one stone which is small roundish flattish sort of rock, he says i'll bet I can make this rock skip five or even ten times over that lake if I throug.

→ Another person takes big stone which round shape & which is like basketball.

→ Another example stout stick- perfect for helping his mom walk since she getting older.

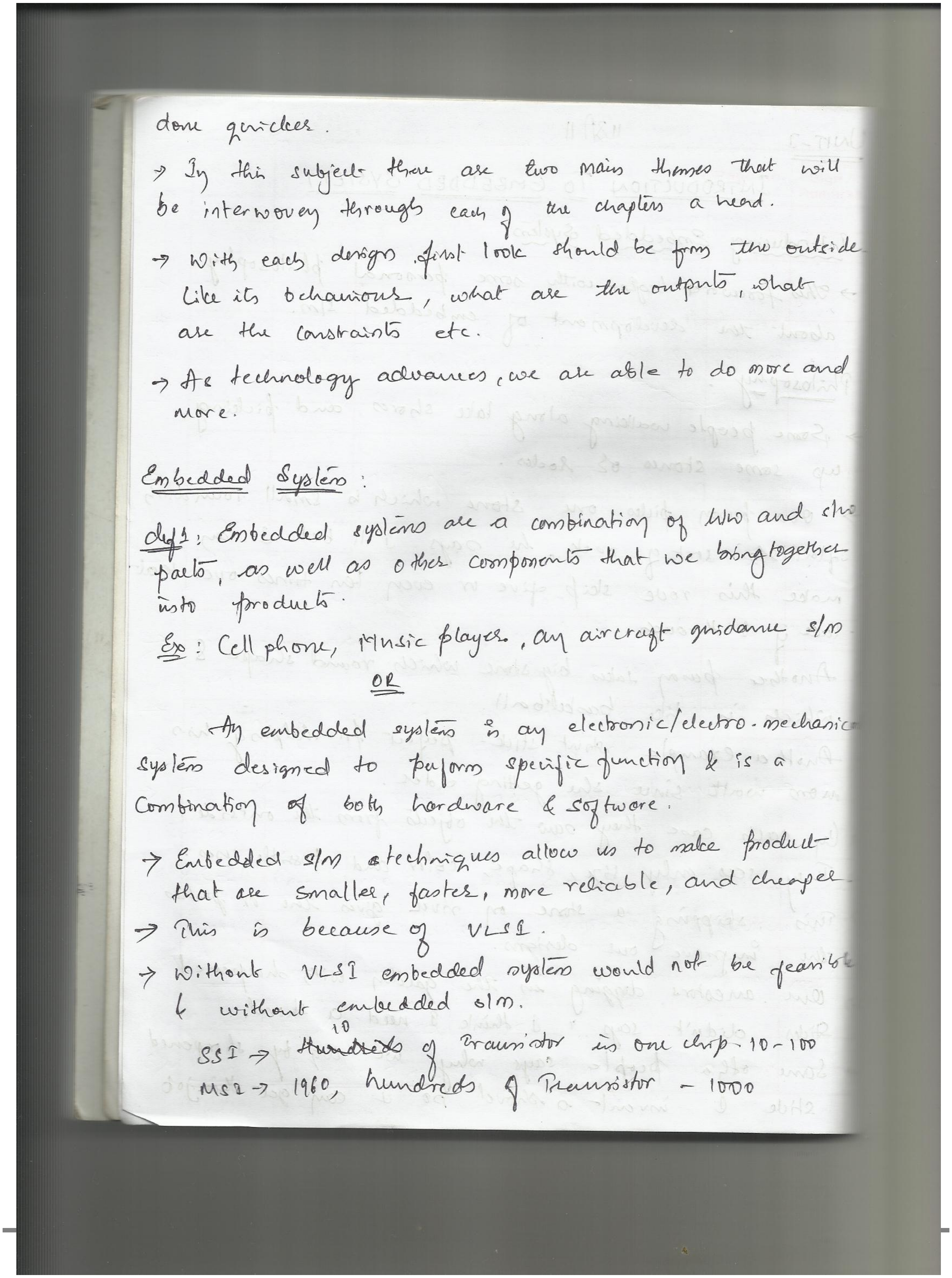
→ In each case they saw the objects from the outside.

→ They sees only size, shape, color and possible uses.

→ This skipping a stone on river gives the or pushes the improve our designs.

→ Our ancestors digging in the garden with sharpened sticks didn't says " I think I need a shovel.

→ Same other people says why we dig by sharpened stick & invent a shovel so I can get this job



done quicker.

→ In this subject there are two main themes that will be interwoven through each of the chapters a head.

→ With each design first look should be from the outside like its behaviours, what are the outputs, what are the constraints etc.

→ As technology advances, we are able to do more and more.

Embedded System:

def1: Embedded systems are a combination of h/w and s/w parts, as well as other components that we bring together into products.

Ex: Cell phone, Music player, any aircraft guidance s/m

OR

An embedded system is an electronic/electro-mechanical system designed to perform specific function & is a combination of both hardware & software.

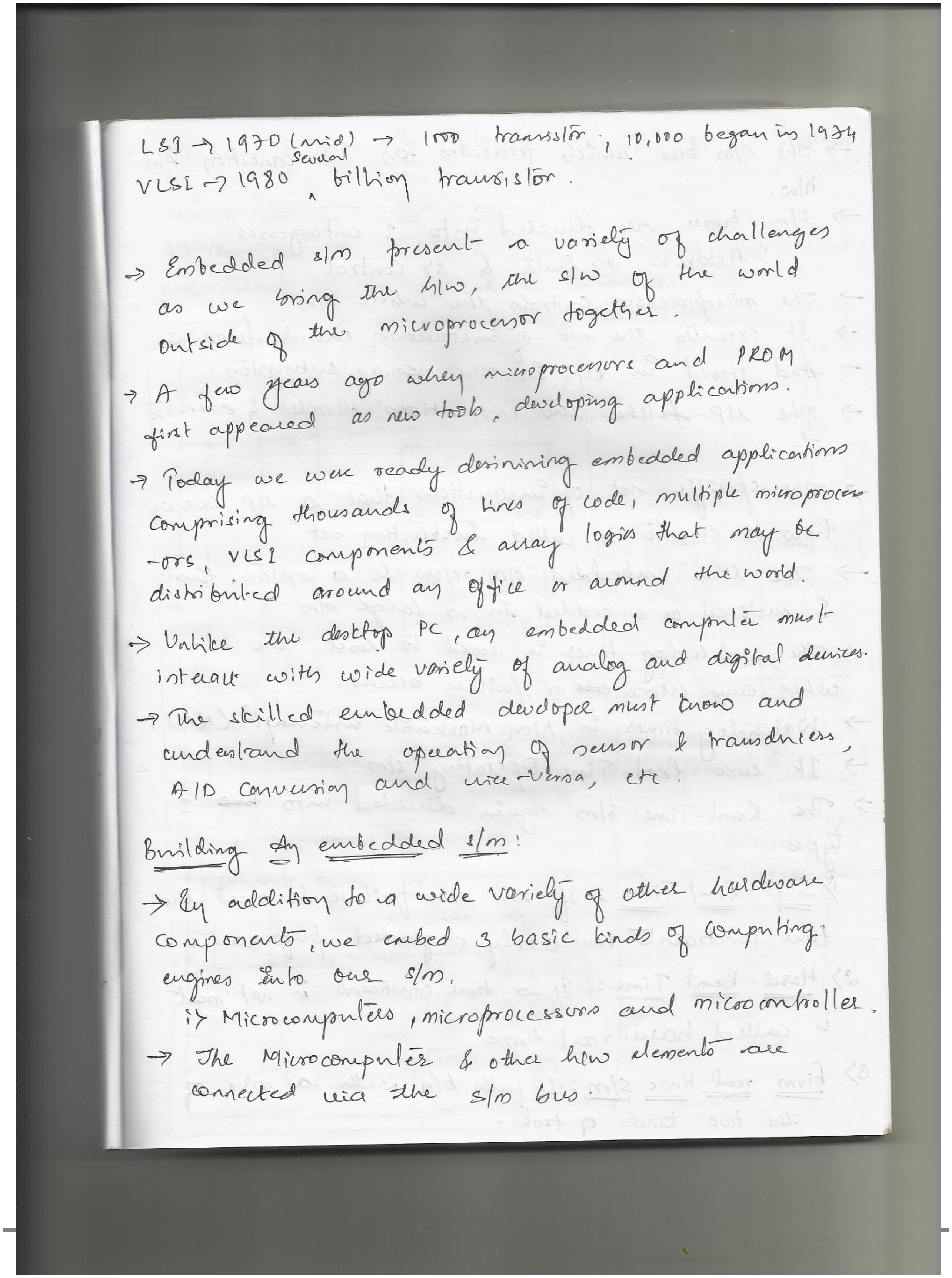
→ Embedded s/m techniques allow us to make product that are smaller, faster, more reliable, and cheaper

→ This is because of VLSI.

→ Without VLSI embedded system would not be feasible & without embedded s/m.

SSI → ~~Hundreds~~ 10 of Transistor in one chip. 10 - 100

MSI → 1960, hundreds of Transistor - 1000



LSI → 1970 (mid) → 1000 transistor, 10,000 began in 1974

VLSI → 1980 several billion transistor.

→ Embedded s/m present a variety of challenges as we bring the h/w, the s/w of the world outside of the microprocessor together.

→ A few years ago when microprocessors and PROM first appeared as new tools, developing applications.

→ Today we were ready desinining embedded applications comprising thousands of lines of code, multiple microprocessors, VLSI components & array logics that may be distributed around an office or around the world.

→ Unlike the desktop PC, an embedded computer must interact with wide variety of analog and digital devices.

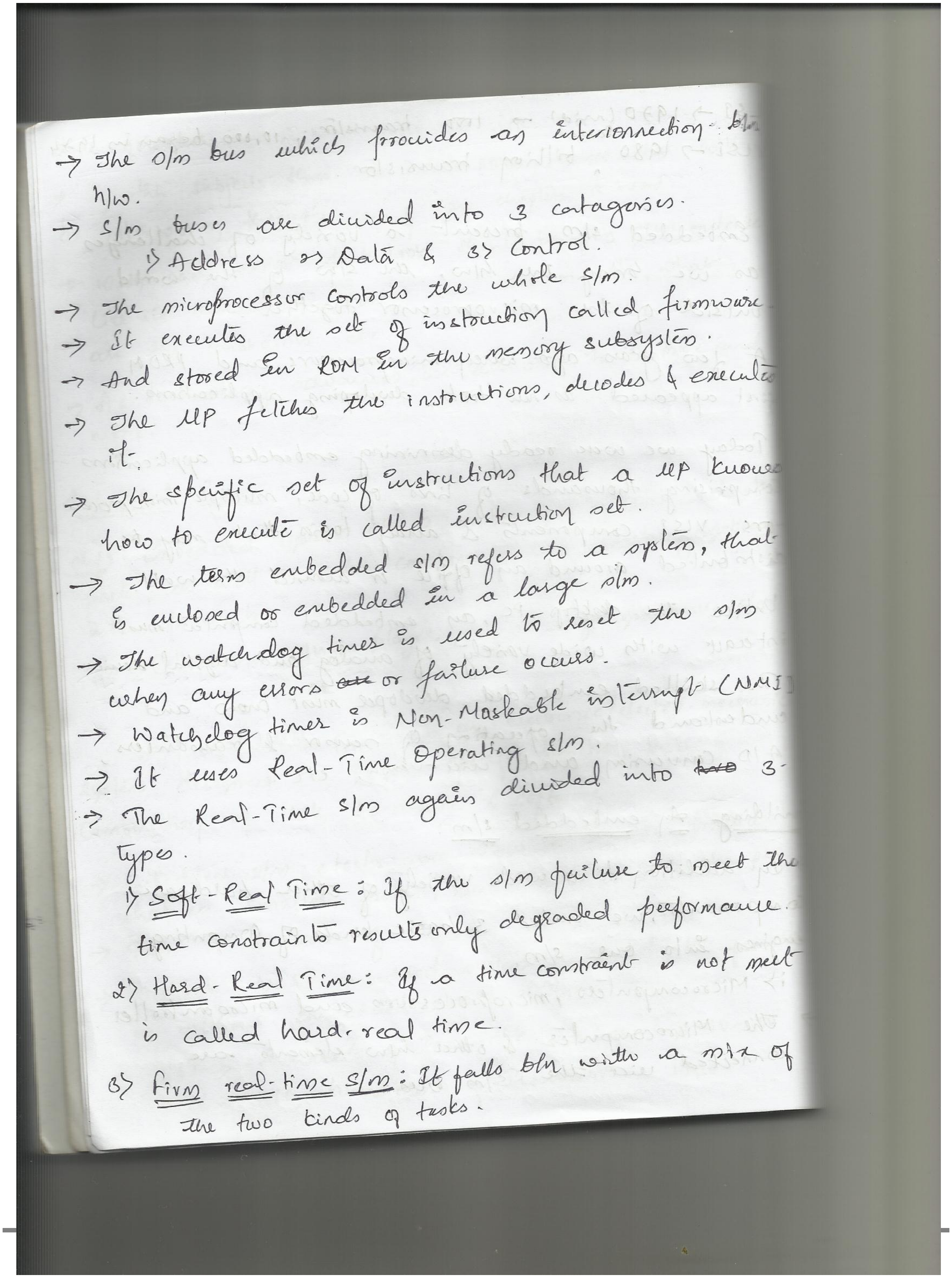
→ The skilled embedded developer must know and understand the operation of sensor & transducers, A/D conversion and vice-versa, etc.

## Building An embedded s/m:

→ In addition to a wide variety of other hardware components, we embed 3 basic kinds of computing engines into our s/m.

i) Microcomputers, microprocessors and microcontroller.

→ The Microcomputer & other h/w elements are connected via the s/m bus.



→ The s/m bus which provides an interconnection b/n h/w.

→ s/m buses are divided into 3 catagories.
1) Address 2) Data & 3) Control.

→ The microprocessor controls the whole s/m.

→ It executes the set of instruction called firmware

→ And stored in ROM in the memory subsystem.

→ The μP fetches the instructions, decodes & executes it.

→ The specific set of instructions that a μP knows how to execute is called instruction set.

→ The term embedded s/m refers to a system, that is enclosed or embedded in a large s/m.

→ The watchdog timer is used to reset the s/m when any errors ~~ete~~ or failure occurs.

→ Watchdog timer is Non-Maskable interrupt (NMI)

→ It uses Real-Time Operating s/m.

→ The Real-Time s/m again divided into ~~two~~ 3 types.

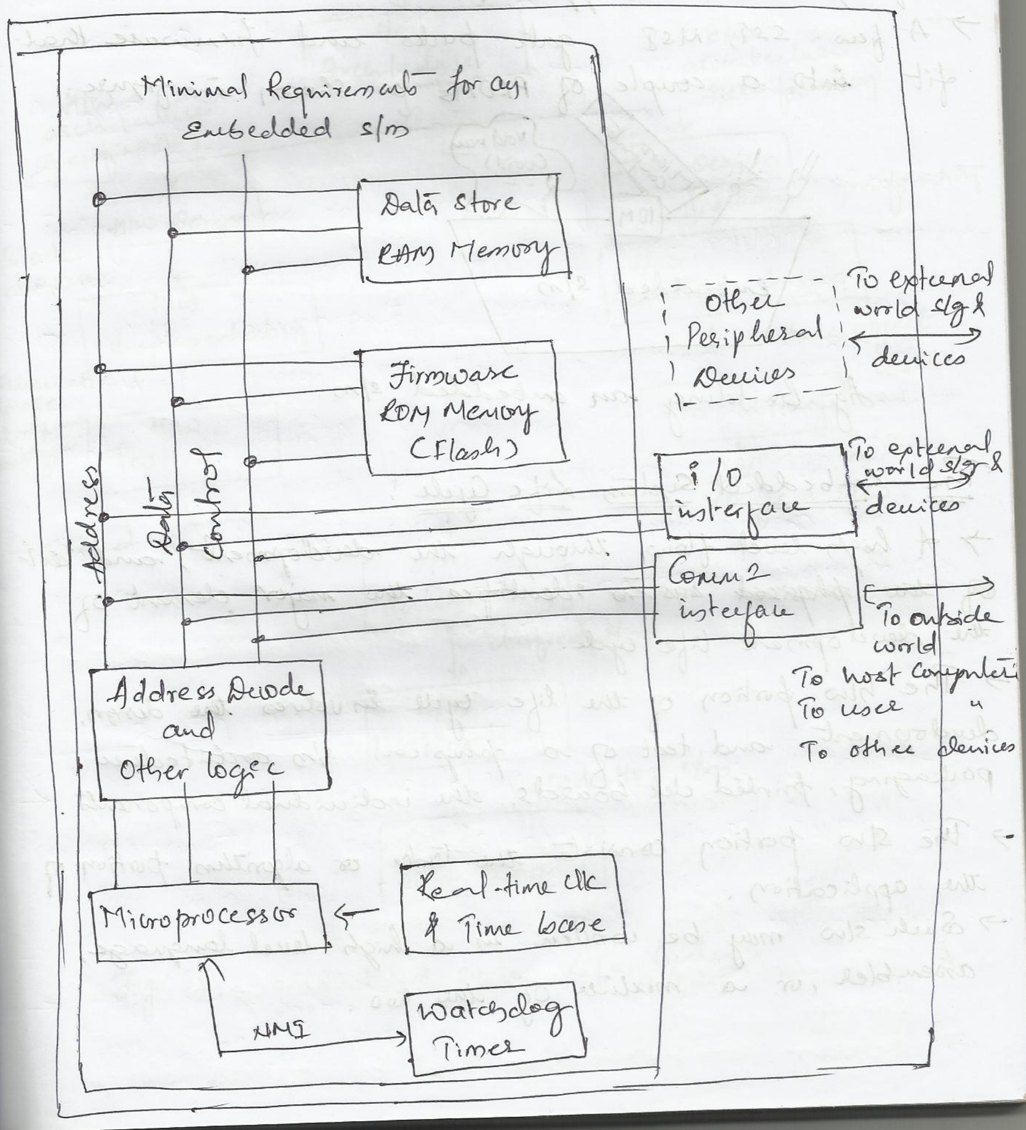
1) Soft-Real Time: If the s/m failure to meet the time constraints results only degraded performance.

2) Hard-Real Time: If a time constraint is not meet is called hard-real time.

3) Firm real-time s/m: It falls b/n with a mix of the two kinds of tasks.

→ An RTOS is specially designed and optimized to predictably handle the strict time constraints associated with events in real-time context.

→ All these are integrated to form an Microprocessor-based embedded system, as shown in figure.


Minimal Requirements for an Embedded s/m
Data store RAM Memory
Firmware ROM Memory (Flash)
Address
Data
Control
Other Peripheral Devices
To external world s/g & devices
i/o interface
To external world s/g & devices
Comm 2 interface
To outside world
To host computer
To user "
To other devices
Address Decode and Other logic
Microprocessor
Real-time clk & Time base
Watchdog Timer
NMI

## The Embedded Design and Development Process

→ The design of a new embedded appl$^{2}$ with some tha about the froblem. wrapped some regisstees, logic & buse aloand the microprocessor, wrote a few lines of assembly language code & debugged it.

→ A few SSI, MSI gate packs and firmware that fit into a couple of PROMs as shown in figure

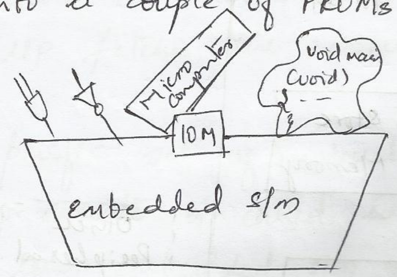


fig: Building an embedded s/m.

## The Embedded System Life Cycle!

→ A high-level flow through the development, and te of ~~the physical system~~ identifies the major element of the development life cycle.

→ The h/w portion of the life cycle involves the design, development, and test of a physical s/m architecture. packaging, printed ckt boards, the individual components.

→ The s/w portion consists the tasks or algorithm portion of the application.

→ Such s/w may be written in a high-level language, assembler, or a mixture of the two.

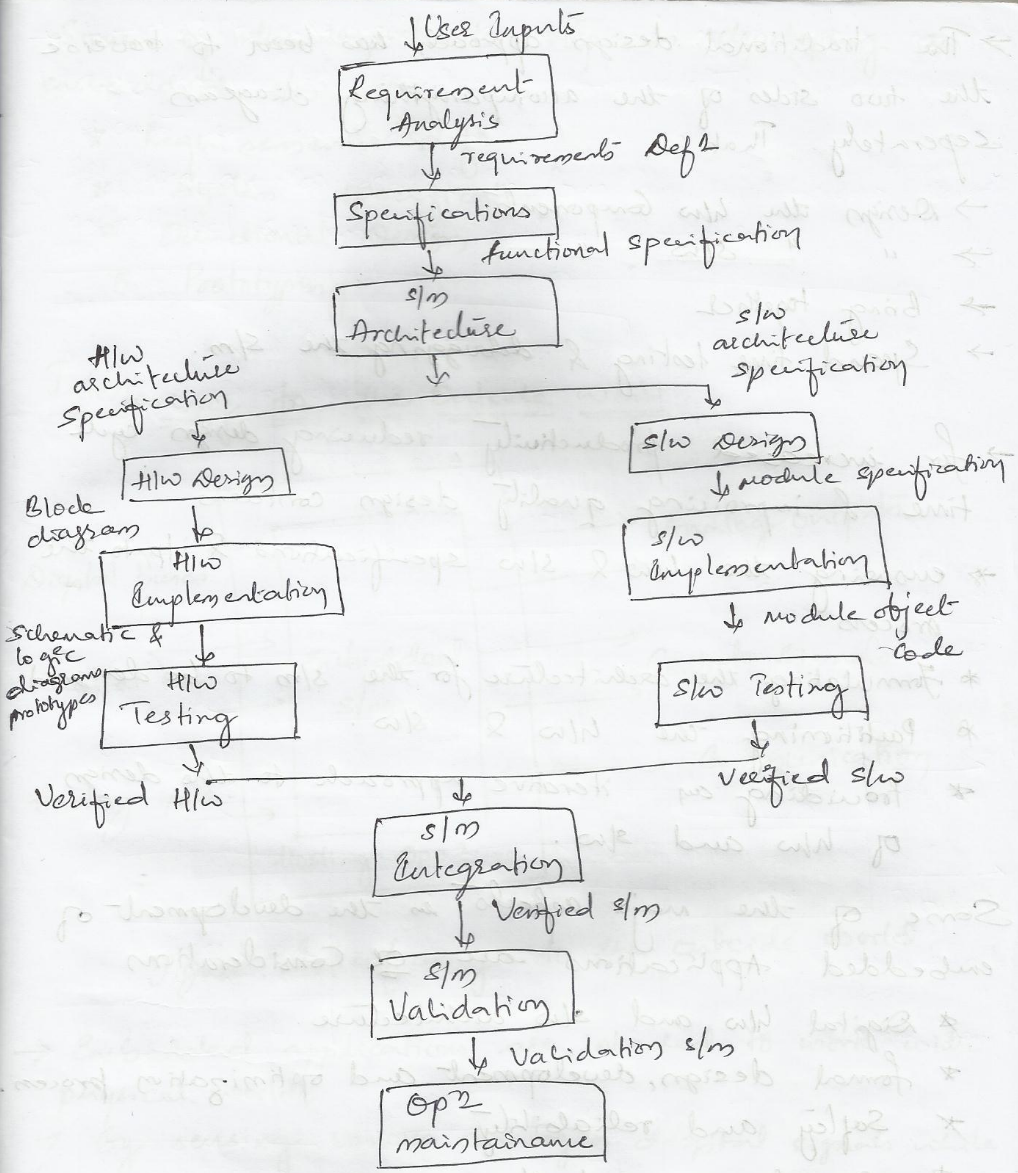
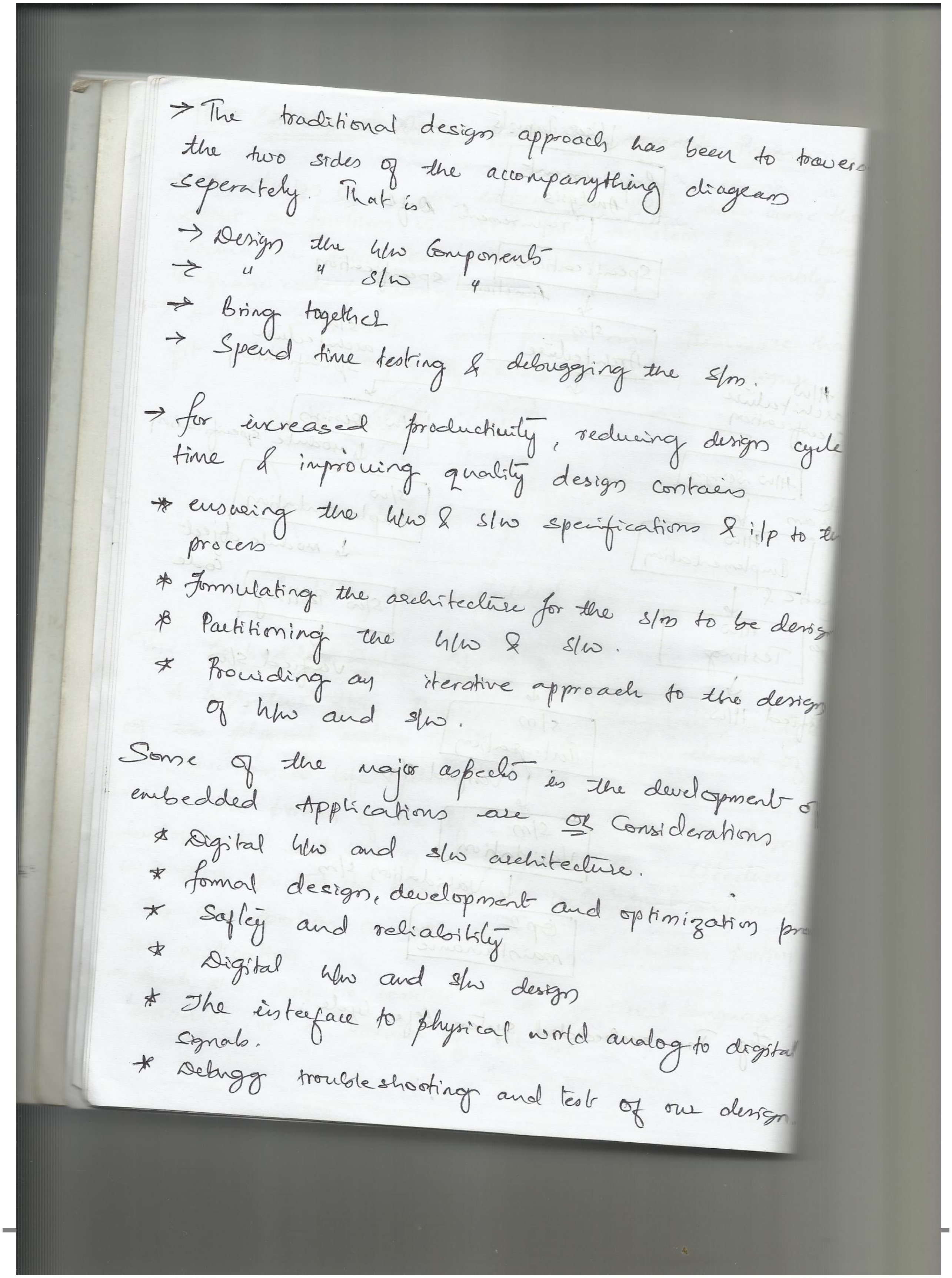


Fig: The embedded system Life Cycle.



→ The traditional design approach has been to travers the two sides of the accompanything diagram seperately. That is

→ Design the h/w Components

→ " " s/w "

→ Bring together

→ Spend time testing & debugging the s/m.

→ for increased productivity, reducing design cycle time & improving quality design contains

* ensuring the h/w & s/w specifications & i/p to th process !
* Formulating the architecture for the s/m to be desig
* Partitioning the h/w & s/w.
* Providing an iterative approach to the design of h/w and s/w.

Some of the major aspects in the development of embedded Applications are ~~of~~ Considerations

* Digital h/w and s/w architecture.
* formal design, development and optimization pro
* Safety and reliability
* Digital h/w and s/w design
* The interface to physical world analog to digital signals.
* debugg troubleshooting and test of our design.

Some of the important steps in developing an embedded s/m are

- ★ Requirements def$^{n}$
- ★ System specification
- ★ Functional Design
- ★ Prototyping

## Interfacing to the Outside World

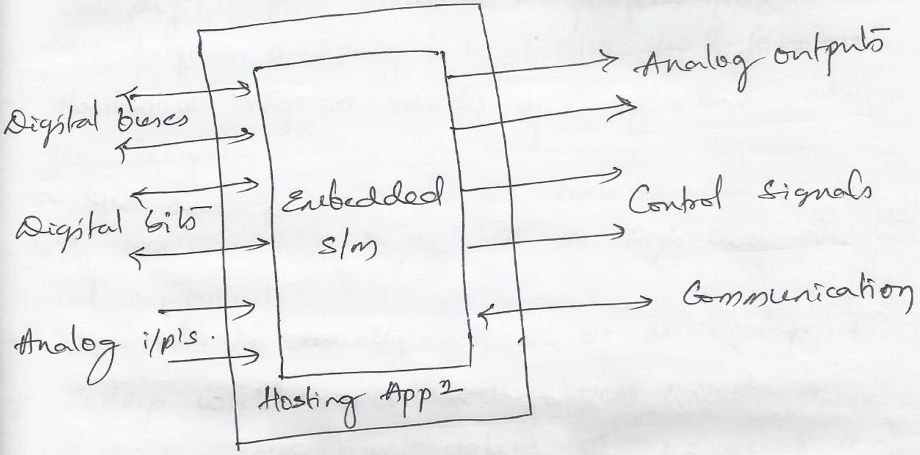
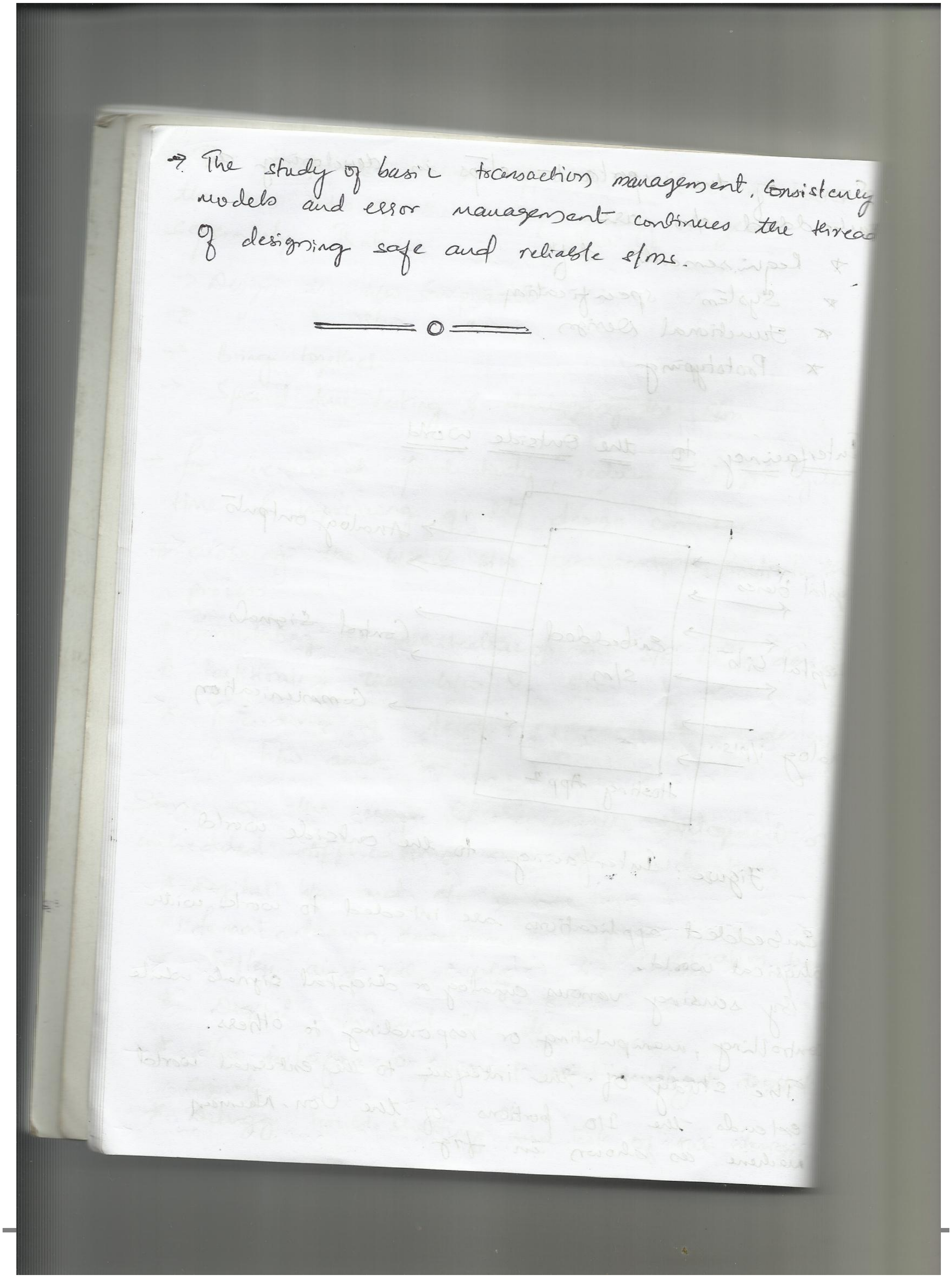


Figure: Interfacing to the outside world.

→ Embedded applications are inteded to work with physical world.

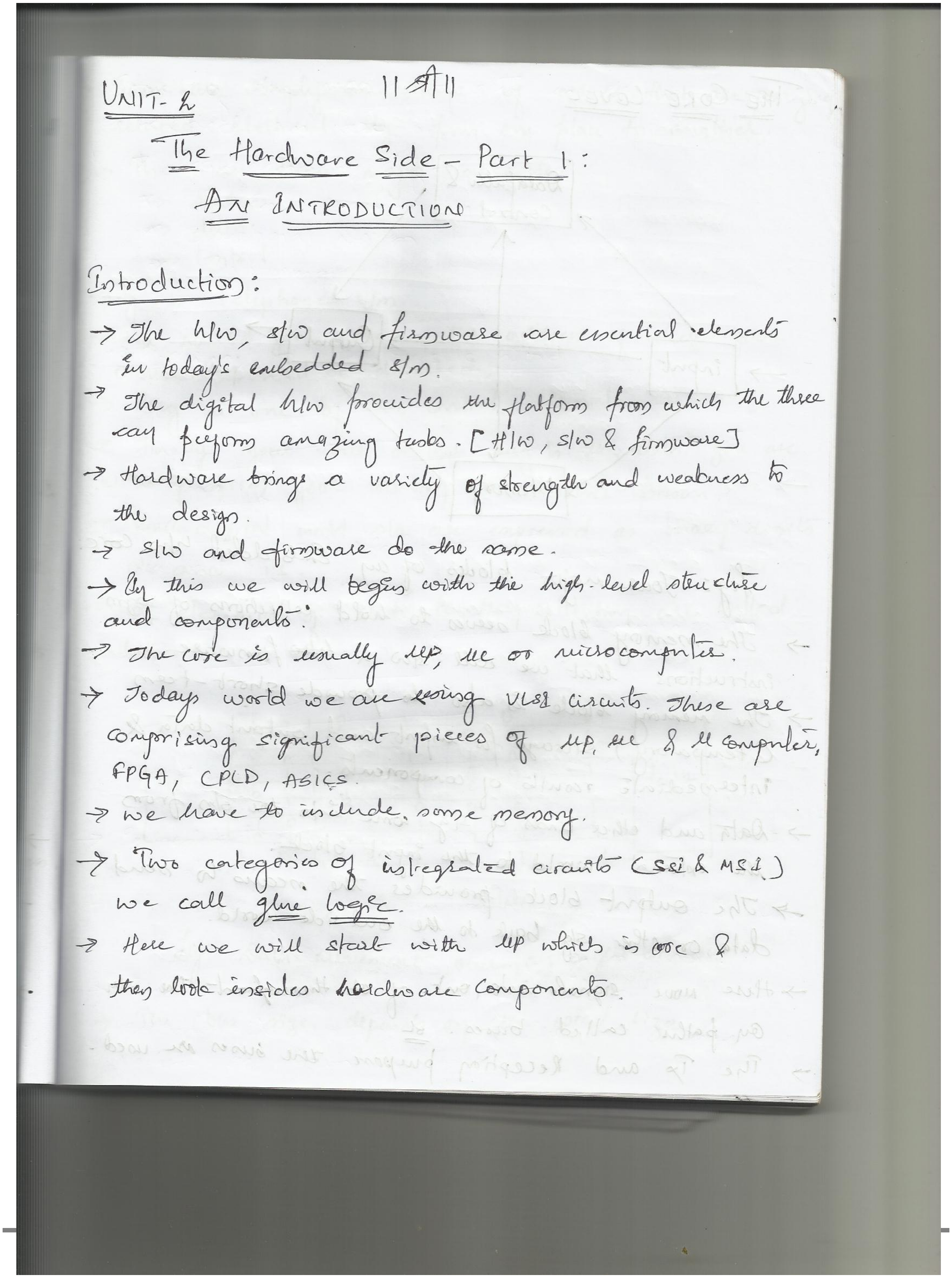
→ By sensing various analog or digital signals while controlling, manipulating or responding to others.

→ The study of the interface to the external world extends the I/O portions of the Von-Neuman Machene as shown in fig.



→ The study of basic transaction management, Consistency models and error management continues the thread of designing safe and reliable s/ms.

———○———



UNIT- 2 ||श्री||

## The Hardware Side – Part 1:

## AN INTRODUCTION

### Introduction:

→ The h/w, s/w and firmware are essential elements in today's embedded s/m.

→ The digital h/w provides the platform from which the three can perform amazing tasks. [H/w, s/w & firmware]

→ Hardware brings a variety of strength and weakness to the design.

→ s/w and firmware do the same.

→ By this we will begin with the high-level structure and components.

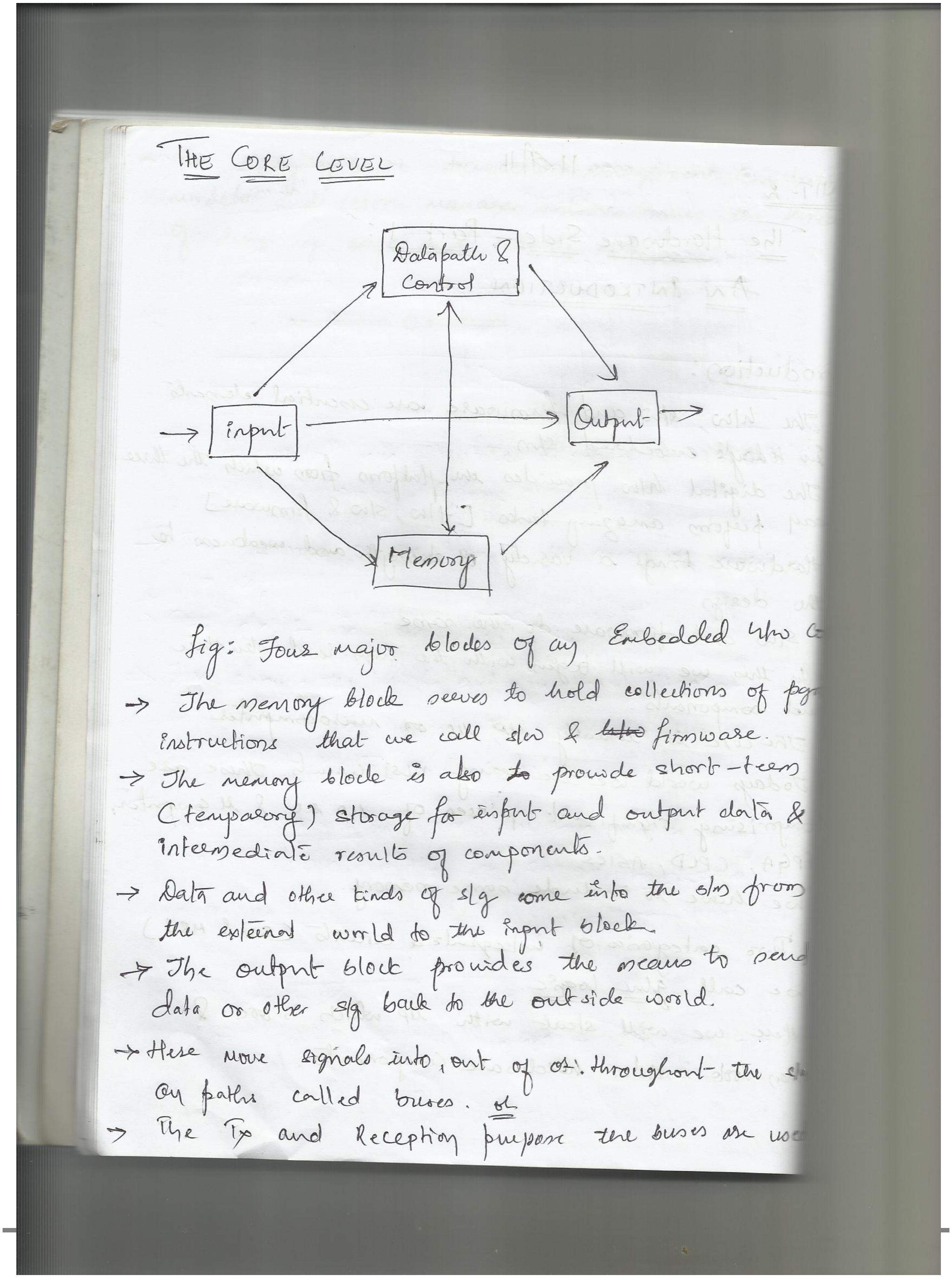
→ The core is usually μp, μc or microcomputer.

→ Todays world we are using VLSI circuits. These are comprising significant pieces of μp, μc & μ computer, FPGA, CPLD, ASICS.

→ we have to include some memory.

→ Two categories of integrated circuits (SSI & MSI) we call glue logic.

→ Here we will start with μp which is core & then look insides hardware components.



# THE CORE LEVEL

Datapath & Control
input
Output
Memory

fig: Four major blocks of an Embedded s/m

→ The memory block serves to hold collections of pg instructions that we call s/w & ~~h/w~~ firmware.

→ The memory block is also ~~to~~ provide short-term (temporary) storage for input and output data & intermediate results of components.

→ Data and other kinds of s/g come into the s/m from the external world to the input block.

→ The output block provides the means to send data or other s/g back to the outside world.

→ Here move signals into, out of or throughout the on paths called buses. or

→ The Tx and Reception purpose the buses are

→ Buses are simply connections of wires that are carrying related electrical slg from one place to another.

→ There are three major categories

- → Address
- → data
- → Control.

→ Eg: is telephone s/m.

→ The number you dial is the address of where your conversation will be directed.

→ Ring is one of the control signals.

→ Finally your voice or text msg is the data you are moving from your location to another person.

→ ~~Really~~ digital world slg are expressed as binary digits 0's and 1's.

→ The elements of such ~~location~~ collection are called bits.

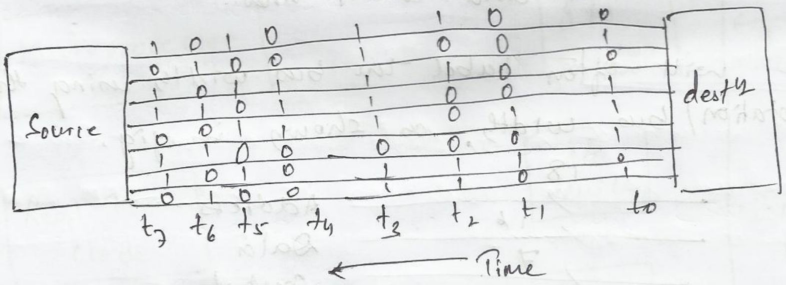


Fig: Data movement over an eight bit bus.

→ The bus size depends on the word size.

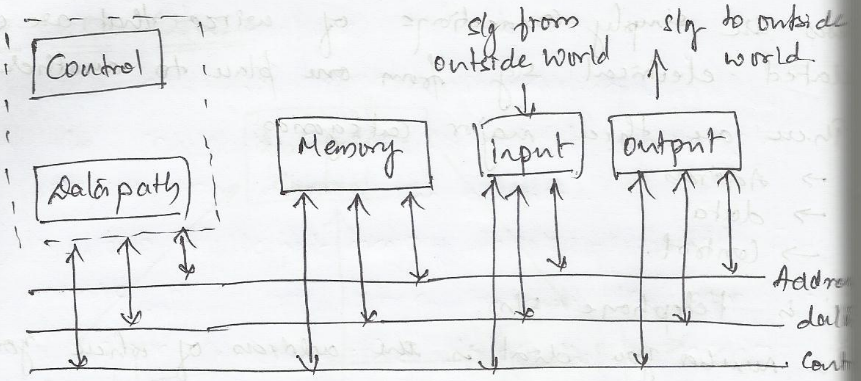


Fig: A typical bus structure comprising Address, Data & Control signals.

→ The source of the transfer the array of eight bit values.

→ The destn is our display.

→ Fig shows the high-level functional diagram, to illustrate a typical bus configiraration comprisi the address data and control lines.

→ We will often label the bus width using the annotation/ bus width, as shown in fig.

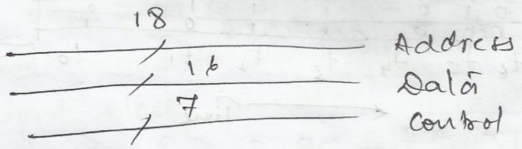


Fig: Identifying the number of slg in a bus.

## The Microprocessor :-

Defn: A μP is an integrated implementation of the central processing unit portion of the machine; it is often simply referred to as a CPU or datapath.

→ μP differs in complexity, power consumption and cost.

→ The Registers are small amounts of high-speed memory that are used to temporarily store frequently used values such as loop index or the index into a buffer.

→ To implement a complete computer s/m, we must still include the input/output subsystem & the external memory system.

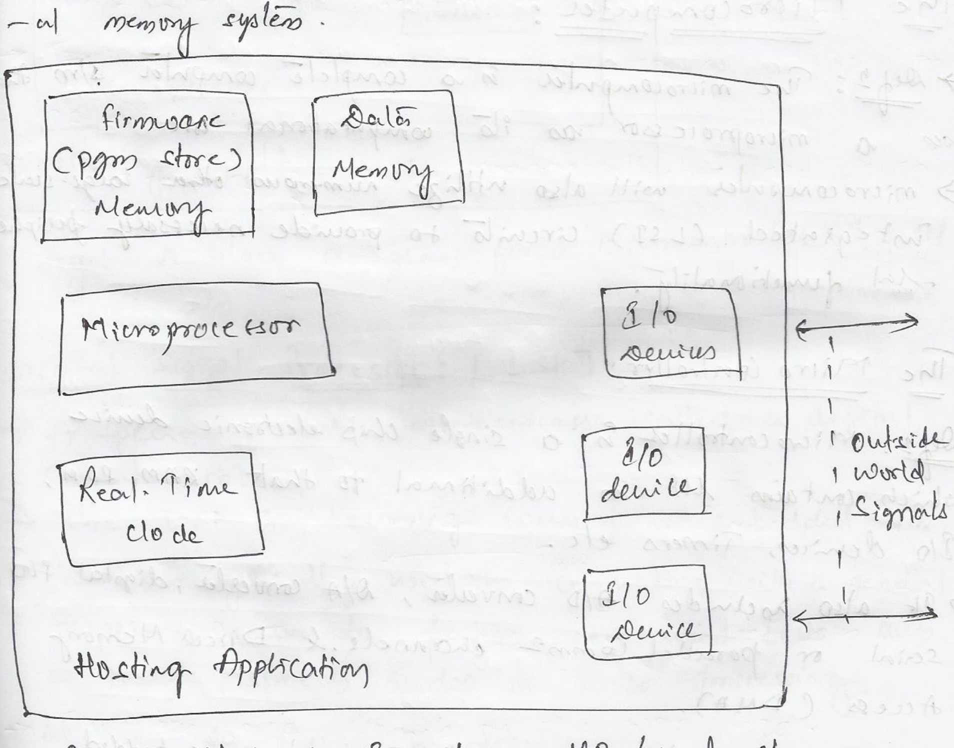
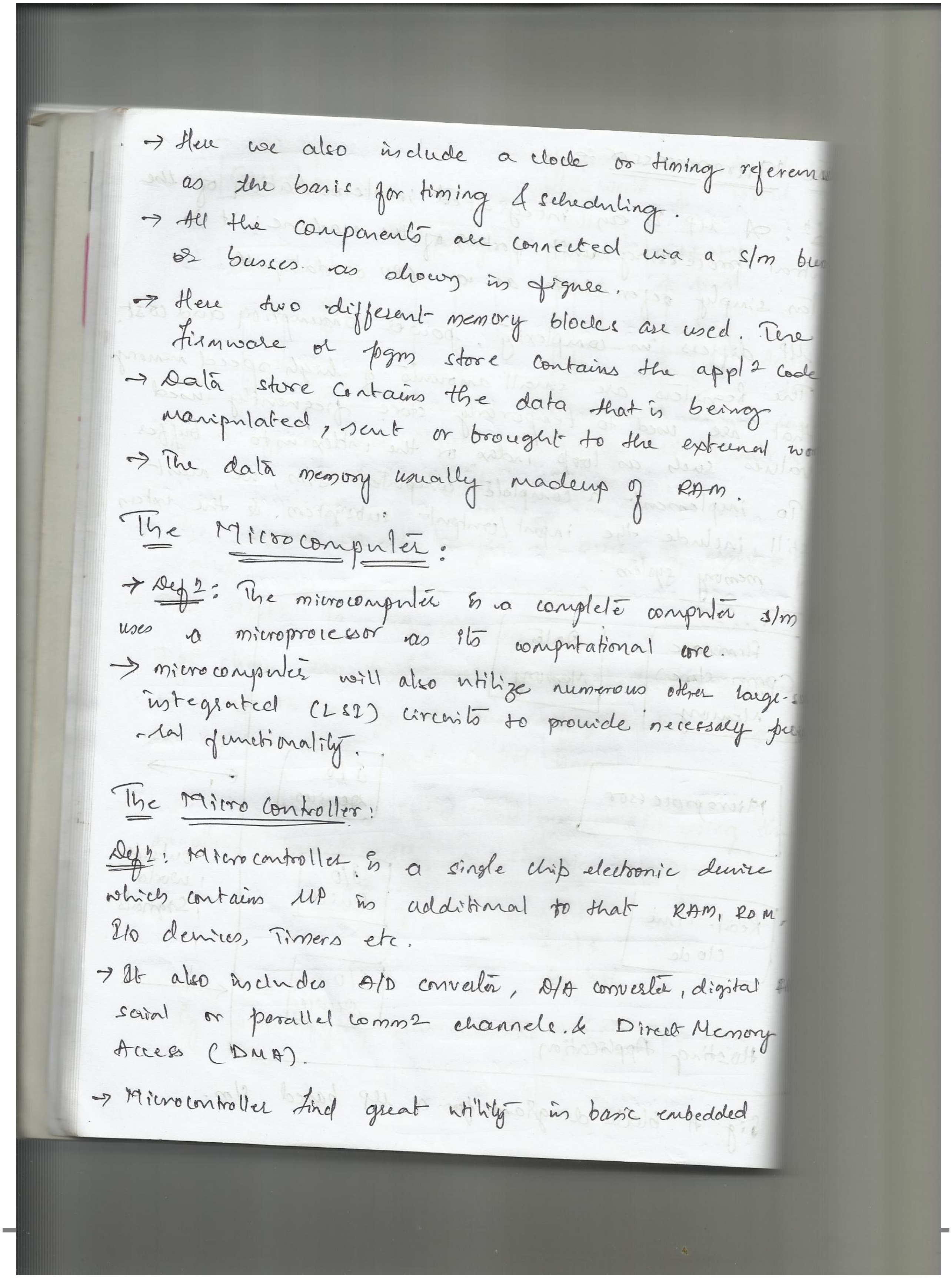


fig: A block diagram for a μp based s/m.



→ Here we also include a clock or timing referen as the basis for timing & scheduling.

→ All the components are connected via a s/m bu or busses as shown in figuee.

→ Here two different memory blocks are used. The firmware or pgm store contains the appl$^n$ code

→ Data store contains the data that is being manipulated, sent or brought to the external wo

→ The data memory usually madeup of RAM.

## The Microcomputer:

→ Def$^n$: The microcomputer is a complete computer s/m uses a microprocessor as its computational core.

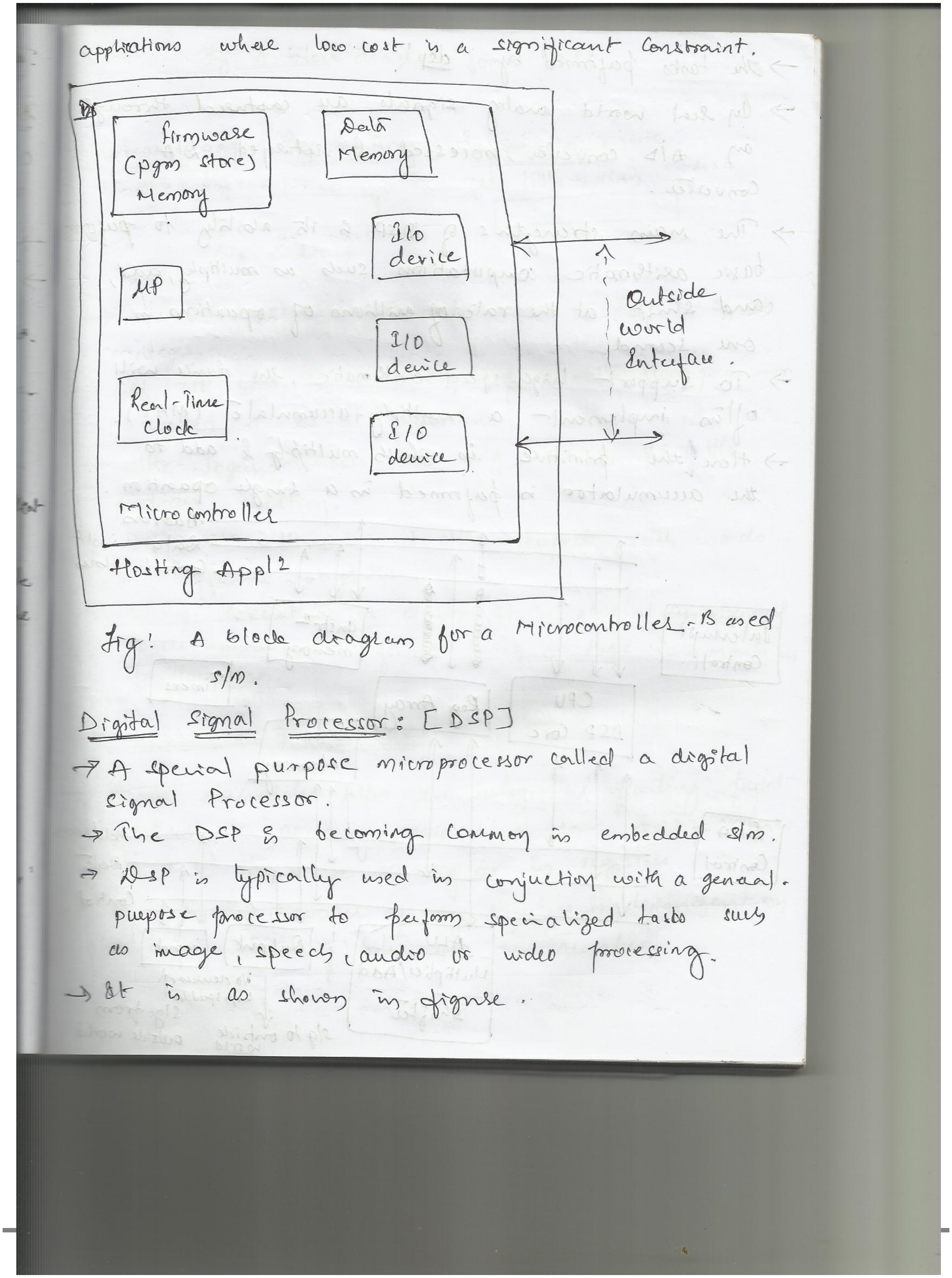
→ microcomputer will also utilize numerous other large-s integrated (LSI) circuits to provide necessary per -ral functionality.

## The Micro Controller:

Def$^n$: Microcontroller is a single chip electronic device which contains uP in additional to that RAM, ROM, I/O devices, Timers etc.

→ It also includes A/D converter, D/A converter, digital & serial or parallel comm$^n$ channels. & Direct Memory Access (DMA).

→ Microcontroller find great utility in basic embedded



applications where low cost is a significant constraint.

Firmware
(pgm store)
Memory
Data
Memory
I/O
device
µP
I/O
device
Real-Time
Clock
I/O
device
Microcontroller
Outside
world
Interface
Hosting Appl²

Fig: A block diagram for a Microcontroller-Based s/m.

## Digital Signal Processor: [DSP]

→ A special purpose microprocessor called a digital signal Processor.

→ The DSP is becoming common in embedded s/m.

→ DSP is typically used in conjuction with a general purpose processor to perform specialized tasks such as image, speech, audio or video processing.

→ It is as shown in figure.

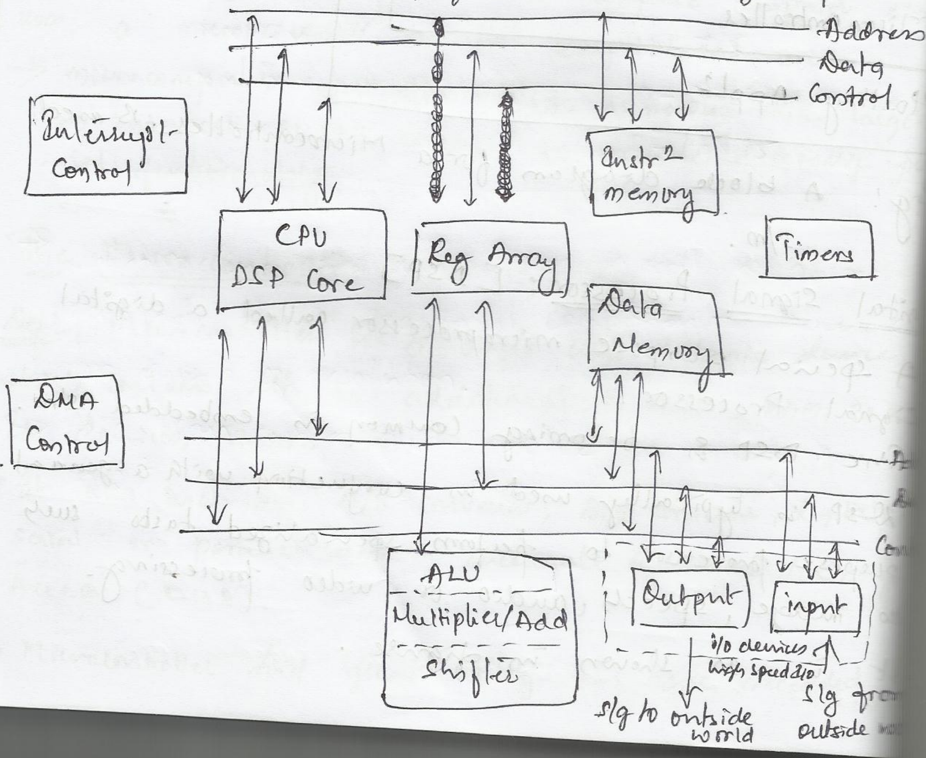
→ The tasks performed by "dsp'.

→ By real world analog signals are captured thro ay A/D converter, processed & returned D/A converter.

→ The main strengths of DSP is its ability to p basic arithmatic computations such as multiply, add and shift at the rate of millions of operations in one second.

→ To support high-speed arithmatic, the device wil often implement a multiply-accumulate (MAC).

→ Here the primitive in which multiply & add to the accumulator is performed in a single operation



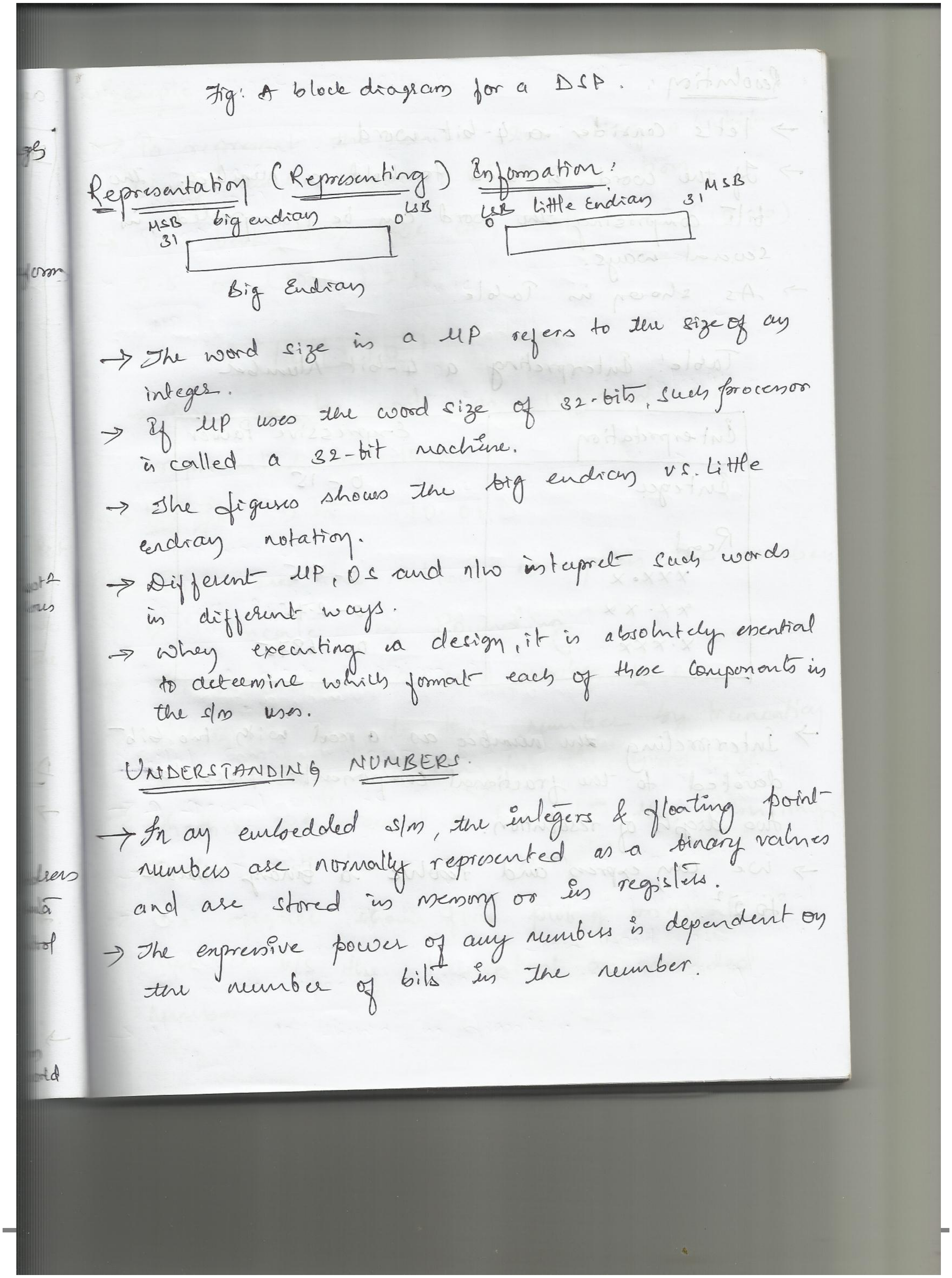


Fig: A block diagram for a DSP.

Representation (Representing) Information:

MSB 31 big endian 0 LSB

LSB 0 little Endian 31 MSB

Big Endian

→ The word size in a μP refers to the size of an integer.

→ If μP uses the word size of 32-bits, such processor is called a 32-bit machine.

→ The figure shows the big endian vs. little endian notation.

→ Different μP, OS and n/w intepret such words in different ways.

→ when executing a design, it is absolutely essential to determine which format each of those components in the s/m uses.

UNDERSTANDING NUMBERS.

→ In any embedded s/m, the integers & floating point numbers are normally represented as a binary values and are stored in memory or in registers.

→ The expressive power of any numbers is dependent on the number of bits in the number.

## Resolution :

→ let's consider a 4-bit word.

→ If the word is used to hold a number, bits comprising the word can be interpreted several ways.

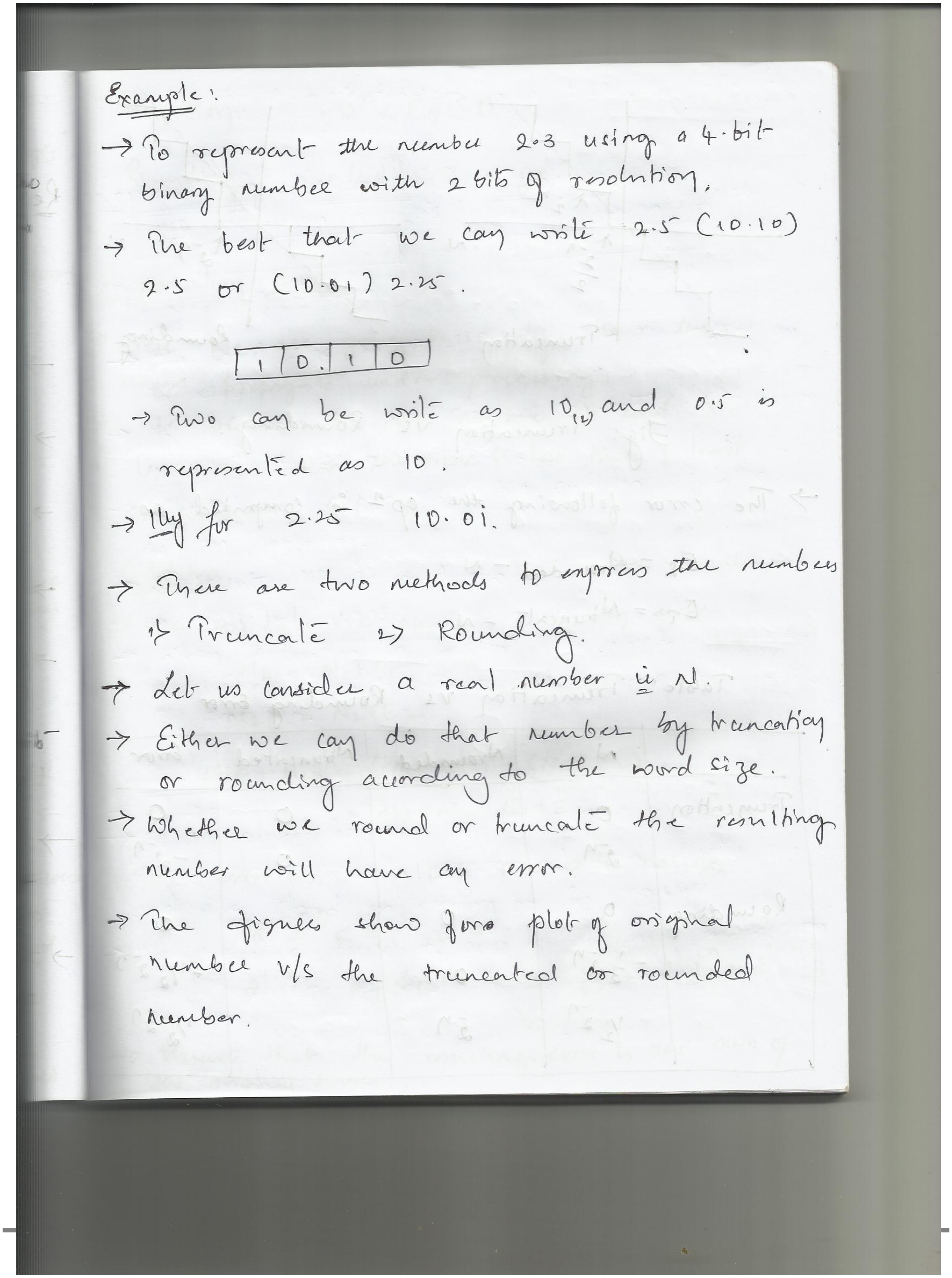
→ As shown in Table.

Table: Interpreting a 4-bit Number

| Interpretation | Expressive Power |
|---|---|
| Integer | 0 – 15 |
| Real | |
| xxx.x | 0 – 7.5 |
| xx.xx | 0 – 2.75 |
| x.xxx | 0 – 1.6875 |

→ Interpreting the number as a real with two devoted to the fractional component provides two digits of resolution.

→ We can express and resolve a binary number to $2^{-2}$.



Example :

→ To represent the number 2.3 using a 4-bit binary number with 2 bits of resolution.

→ The best that we can write 2.5 (10.10) 2.5 or (10.01) 2.25 .

| 1 | 0. | 1 | 0 |
|---|---|---|---|

→ Two can be write as $10_{(2)}$ and 0.5 is represented as 10 .

→ lly for 2.25 10.01.

→ There are two methods to express the numbers

1) Truncate 2) Rounding.

→ Let us consider a real number is N.

→ Either we can do that number by truncating or rounding according to the word size.

→ Whether we round or truncate the resulting number will have an error.

→ The figures show form plot of original number v/s the truncated or rounded number.

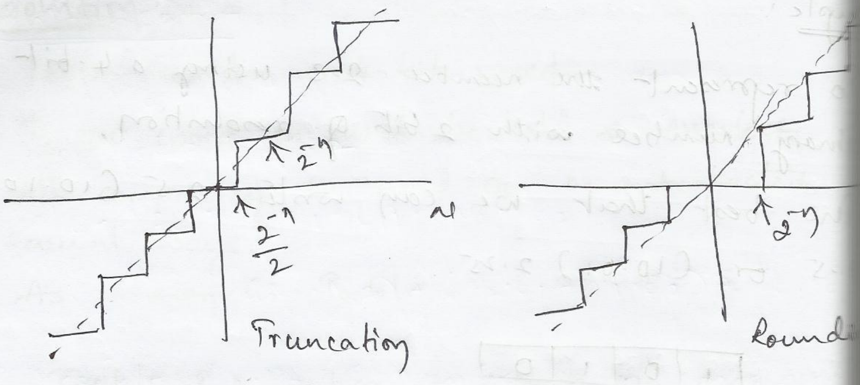


Fig: Truncation vs Rounding.

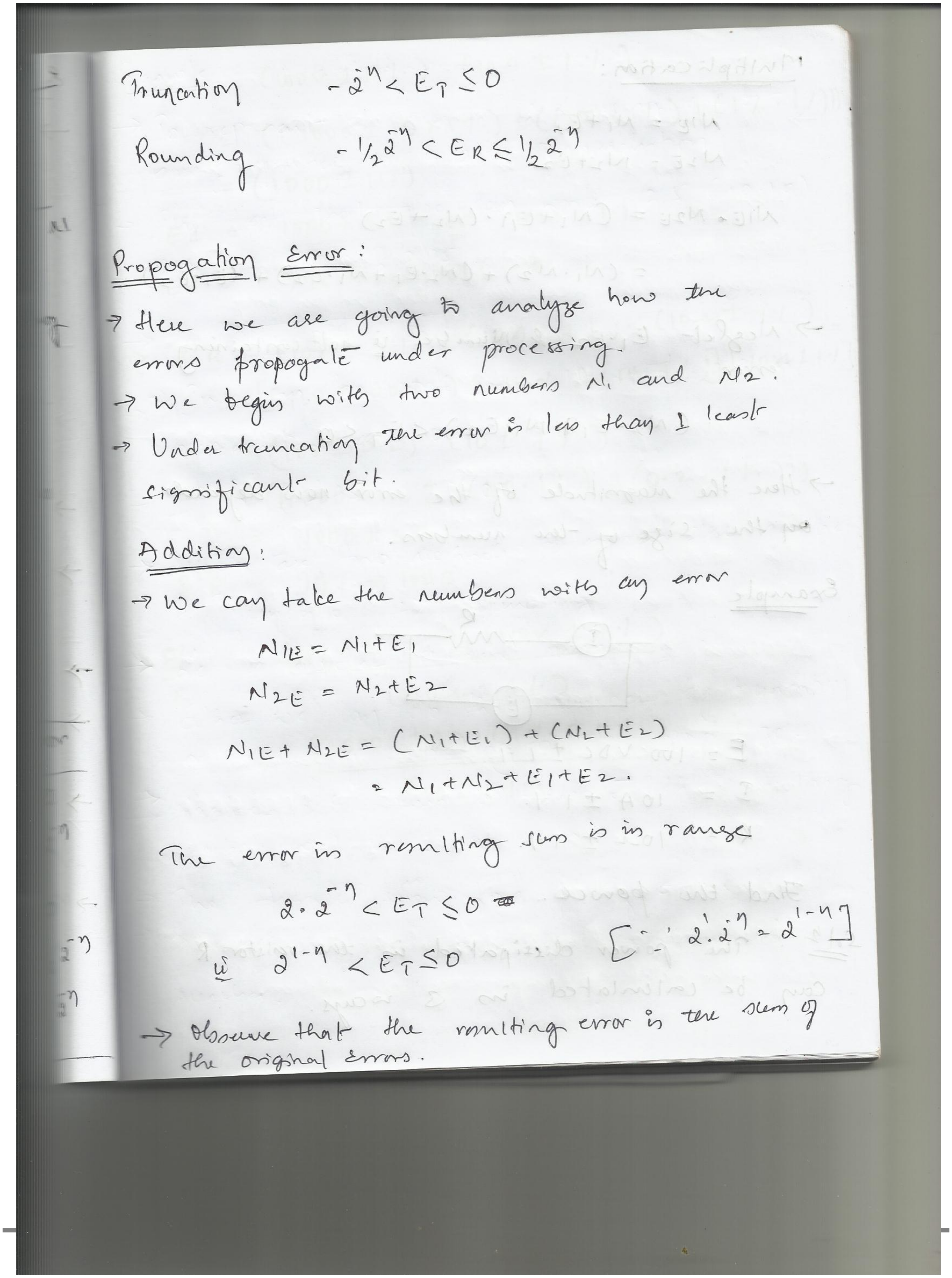
→ The error following the opⁿ is computed a

$E_R = N_{rounded} - N$

$E_T = N_{truncate} - N.$

Table: Truncation vs Rounding Error

| | N | Nrounded | Ntruncated | Err |
|---|---|---|---|---|
| Truncation | 0 | | 0 | 0 |
| | $2^{-n}$ | | 0 | $-2$ |
| Rounding | 0 | 0 | | 0 |
| | $-\frac{1}{2}2^{-n}$ | 0 | | $-\frac{1}{2}$ |
| | $\frac{1}{2}2^{-n}$ | $2^{-n}$ | | $\frac{1}{2}$ |



Truncation $-2^{-n} < E_T \le 0$

Rounding $-\frac{1}{2}2^{-n} < E_R \le \frac{1}{2}2^{-n}$

Propogation Error :

→ Here we are going to analyze how the errors propogate under processing.

→ We begin with two numbers $N_1$ and $N_2$.

→ Under truncation the error is less than 1 least significant bit.

Addition :

→ We can take the numbers with any error

$$N_{1E} = N_1 + E_1$$

$$N_{2E} = N_2 + E_2$$

$$N_{1E} + N_{2E} = (N_1 + E_1) + (N_2 + E_2)$$

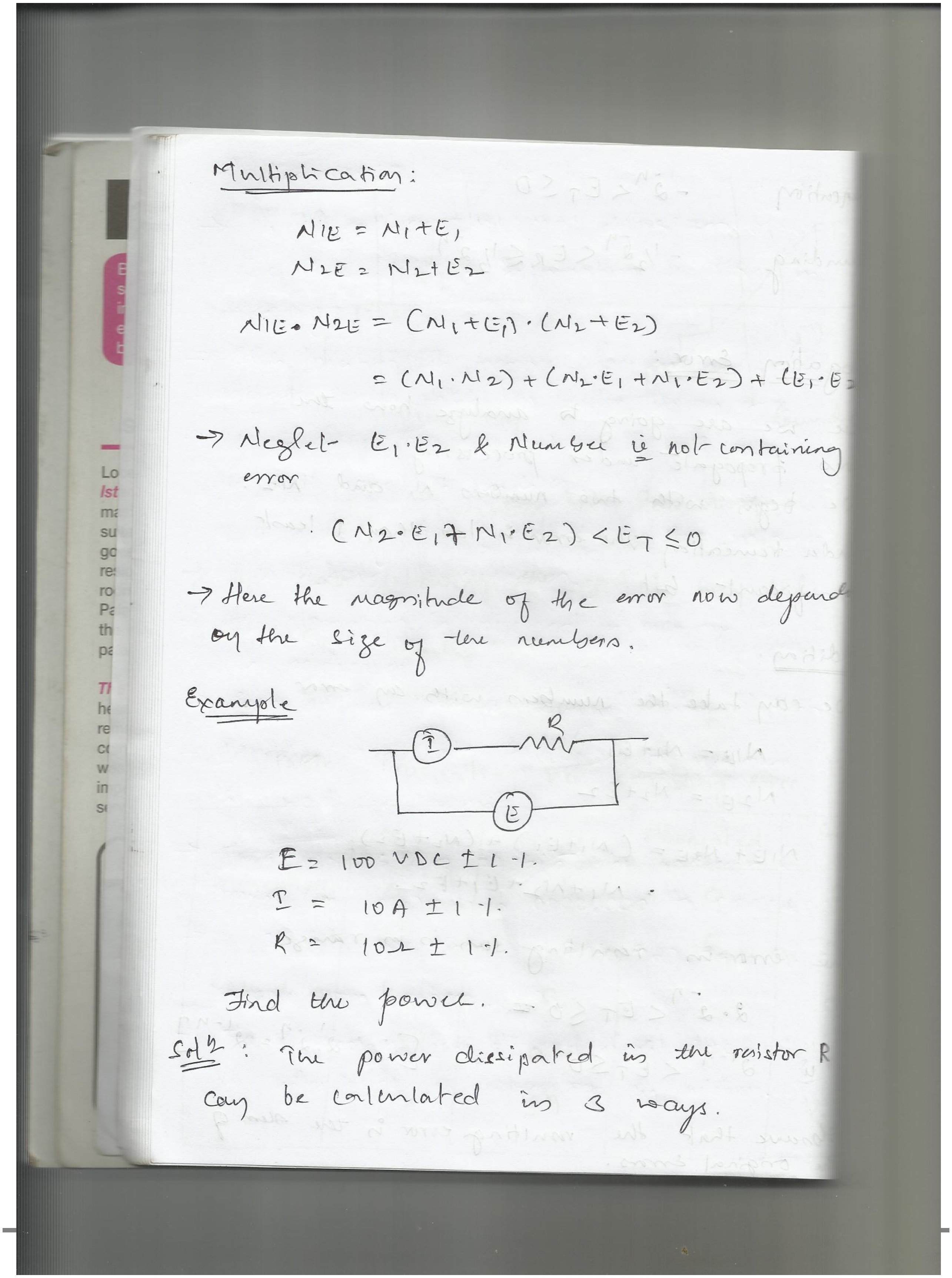
$$= N_1 + N_2 + E_1 + E_2.$$

The error in resulting sum is in range

$$2 \cdot 2^{-n} < E_T \le 0$$

ie $2^{1-n} < E_T \le 0$ $[\because 2^1 \cdot 2^{-n} = 2^{1-n}]$

→ Observe that the resulting error is the sum of the original errors.



Multiplication:

$$N_{1E} = N_1 + E_1$$

$$N_{2E} = N_2 + E_2$$

$$N_{1E} \cdot N_{2E} = (N_1 + E_1) \cdot (N_2 + E_2)$$

$$= (N_1 \cdot N_2) + (N_2 \cdot E_1 + N_1 \cdot E_2) + (E_1 \cdot E$$

→ Neglet $E_1 \cdot E_2$ & Number is not containing error

$$(N_2 \cdot E_1 + N_1 \cdot E_2) < E_T \leq 0$$

→ Here the magnitude of the error now depend on the size of the numbers.

Example

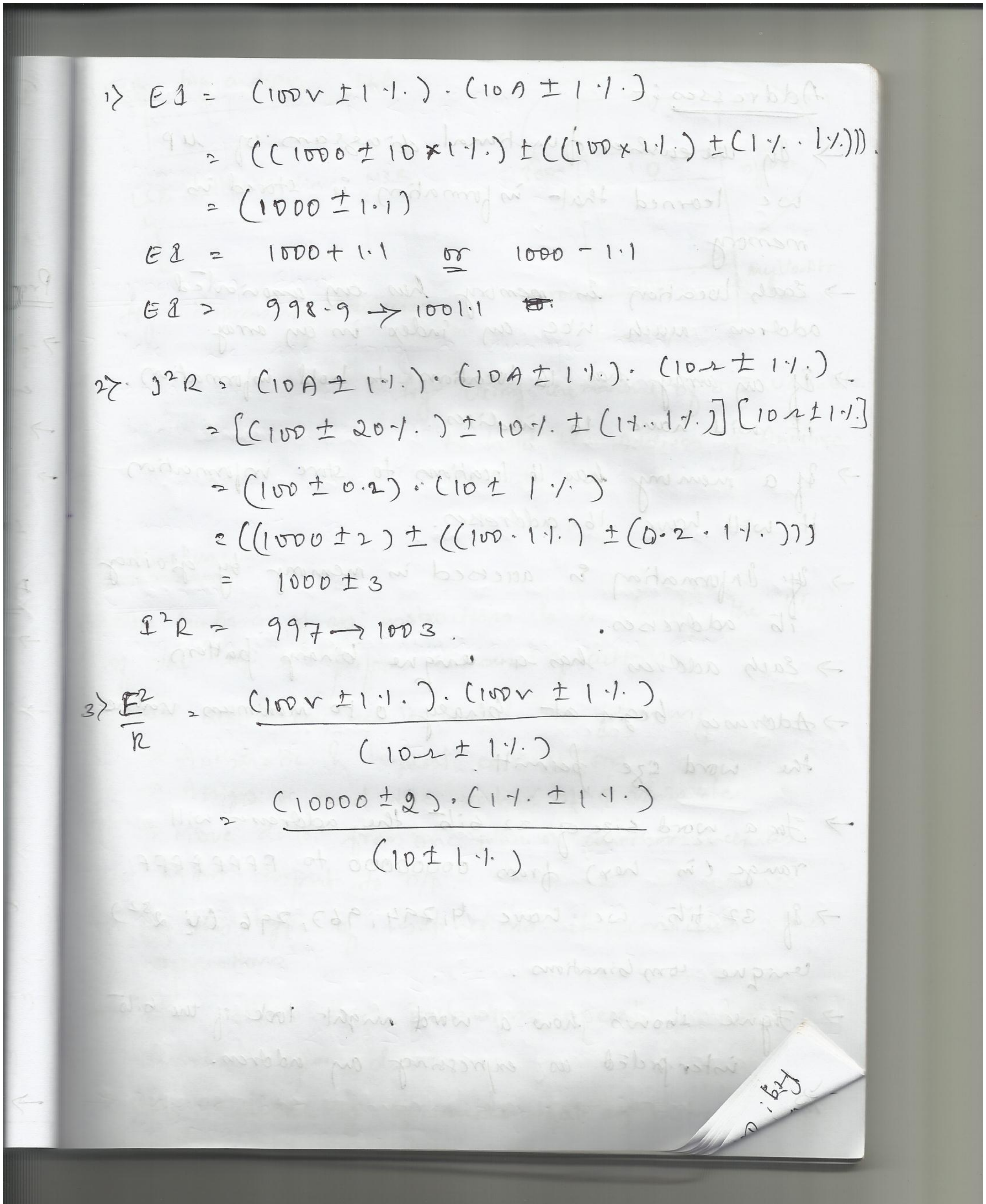
E = 100 VDC ± 1 %

I = 10 A ± 1 %

R = 10Ω ± 1 %

Find the power.

Sol$^n$: The power dissipated in the resistor R can be calculated in 3 ways.



1) $EI = (100V \pm 1\%) \cdot (10A \pm 1\%)$

$= ((1000 \pm 10 \times 1\%) \pm ((100 \times 1\%) \pm (1\% \cdot 1\%)))$.

$= (1000 \pm 1.1)$

$EI = 1000 + 1.1$ <u>or</u> $1000 - 1.1$

$EI \geq 998.9 \rightarrow 1001.1$

2) $I^2R = (10A \pm 1\%) \cdot (10A \pm 1\%) \cdot (10\Omega \pm 1\%)$.

$= [(100 \pm 20\%) \pm 10\% \pm (1\% \cdot 1\%)] [10\Omega \pm 1\%]$

$= (100 \pm 0.2) \cdot (10 \pm 1\%)$

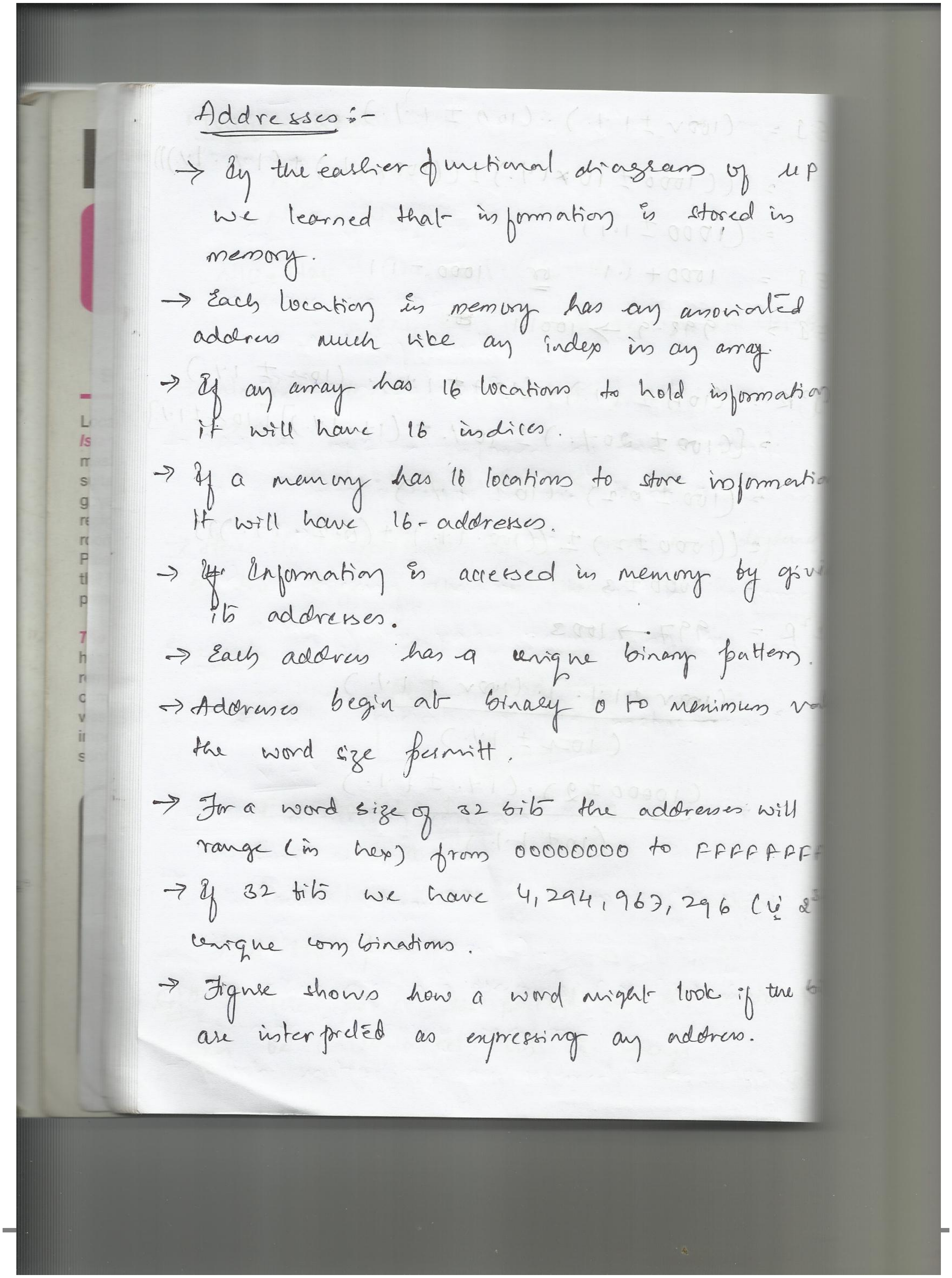
$= ((1000 \pm 2) \pm ((100 \cdot 1\%) \pm (0.2 \cdot 1\%)))$

$= 1000 \pm 3$

$I^2R = 997 \rightarrow 1003$.

3) $$\frac{E^2}{R} = \frac{(100V \pm 1\%) \cdot (100V \pm 1\%)}{(10\Omega \pm 1\%)}$$

$$= \frac{(10000 \pm 2) \cdot (1\% \pm 1\%)}{(10 \pm 1\%)}$$



Addresses :-

→ By the earlier functional diagram of uP we learned that information is stored in memory.

→ Each location in memory has an associated address much like an index in an array.

→ If an array has 16 locations to hold information it will have 16 indices.

→ If a memory has 16 locations to store information it will have 16-addresses.

→ It Information is accessed in memory by givi its addresses.

→ Each address has a unique binary pattern.

→ Addresses begin at binary 0 to maximum va the word size permitt.

→ For a word size of 32 bits the addresses will range (in hex) from 00000000 to FFFFFFFF

→ If 32 bits we have 4,294,967,296 (i.e $2^3$ unique combinations.

→ Figure shows how a word might look if the b are interpreted as expressing an address.

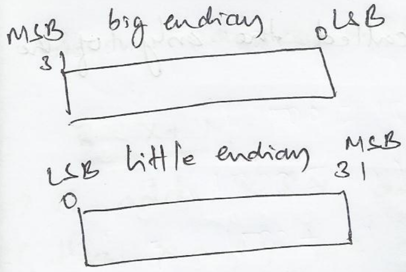


fig: Expressing Addresses.

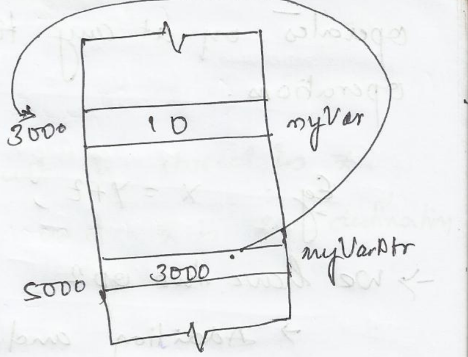


fig: Using the value of one variable to hold the Address of Another Variable.

Instructions :

→ The purpose of an instructions is to direct the h/w of the μp to perform a series of actions.

→ Such actions can be included the following

→ Arithmatic & logical calculations

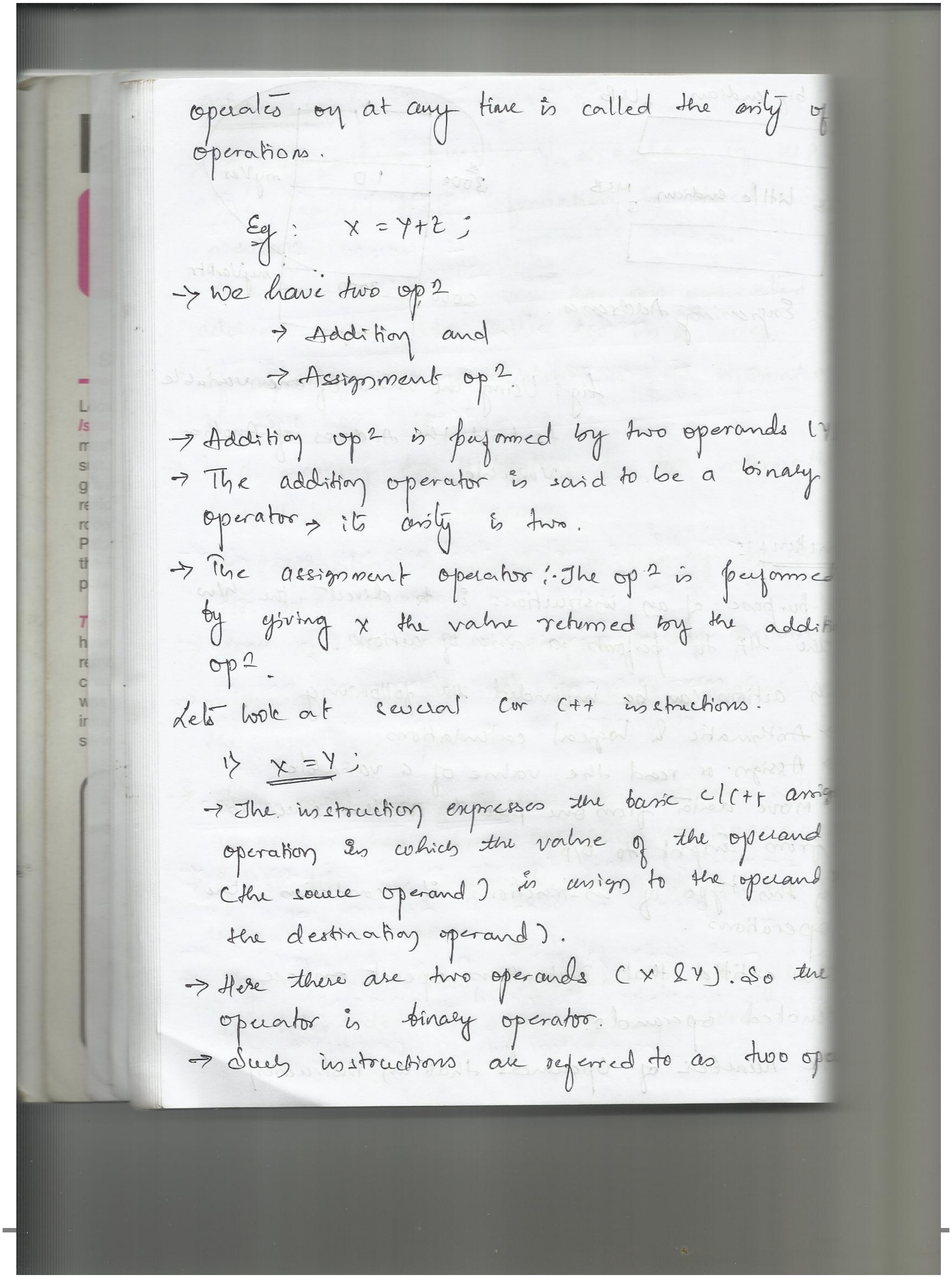
→ Assign· or read the value of a variable

→ Move data from one place to another such as from input to o/p.

→ By this type of instructions it contains the operations.

→ The entities that instructions operate on are denoted operands.

→ The number of operands that an instruction



operates on at any time is called the arity of operations.

Eg: $x = y + z;$

→ we have two op$^2$

→ Addition and

→ Assignment op$^2$.

→ Addition op$^2$ is performed by two operands (y

→ The addition operator is said to be a binary operator → its arity is two.

→ The assignment operator :- The op$^2$ is performed by giving x the value returned by the additi op$^2$.

Let's look at several C or C++ instructions.

1) $x = y;$

→ The instruction expresses the basic C/C++ assig operation in which the value of the operand (the source operand) is assign to the operand (the destination operand).

→ Here there are two operands (x & y). So the operator is binary operator.

→ Such instructions are referred to as two op of two address instructions.

2. $\underline{Z = X+Y}$.

→ It adds X & Y and result is stored in Z.

→ Here X and Y are sources and Z is the destination.

~~→ X+Z~~ .

→ Its having three operands.

3. $\underline{X = X+Y}$.

→ It has a two operands X & Y.

→ The result is assigned to X.

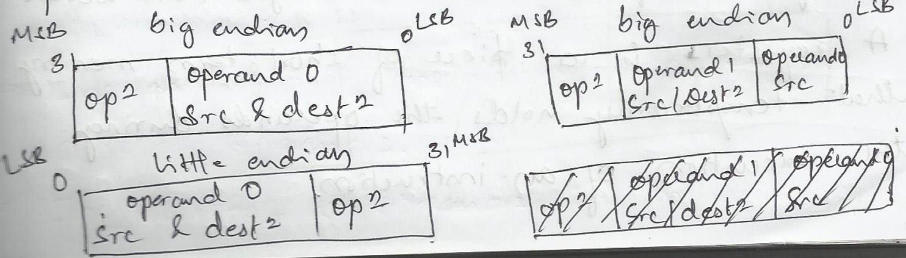
→ If we ignore intermediate result X & Y are source and 'X' again acts as destination.

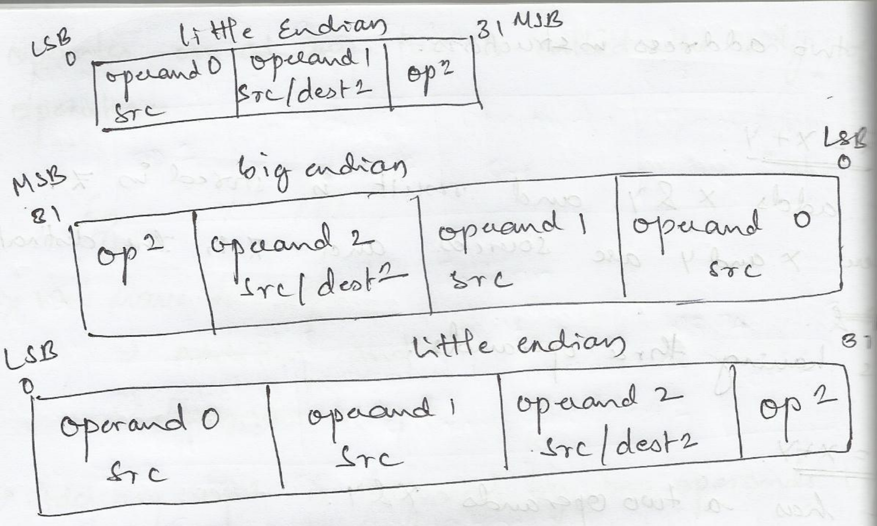
4. $\underline{++X}$ or $\underline{X++}$.

→ This operation is used to increament the value of the variable.

→ In this case X is both source and destination of the operation.

→ Such an instruction is designated as a one-operand or one-address instruction.





little Endian

LSB 0 — 31 MSB

| | | |
|---|---|---|
| operand 0<br>src | operand 1<br>src/dest 2 | op 2 |

big endian

MSB 31 — LSB 0

| | | | |
|---|---|---|---|
| op 2 | operand 2<br>src/dest 2 | operand 1<br>src | operand 0<br>src |

little endian

LSB 0 — 31

| | | | |
|---|---|---|---|
| operand 0<br>src | operand 1<br>src | operand 2<br>src/dest 2 | op 2 |

Fig: Expressing Instructions.

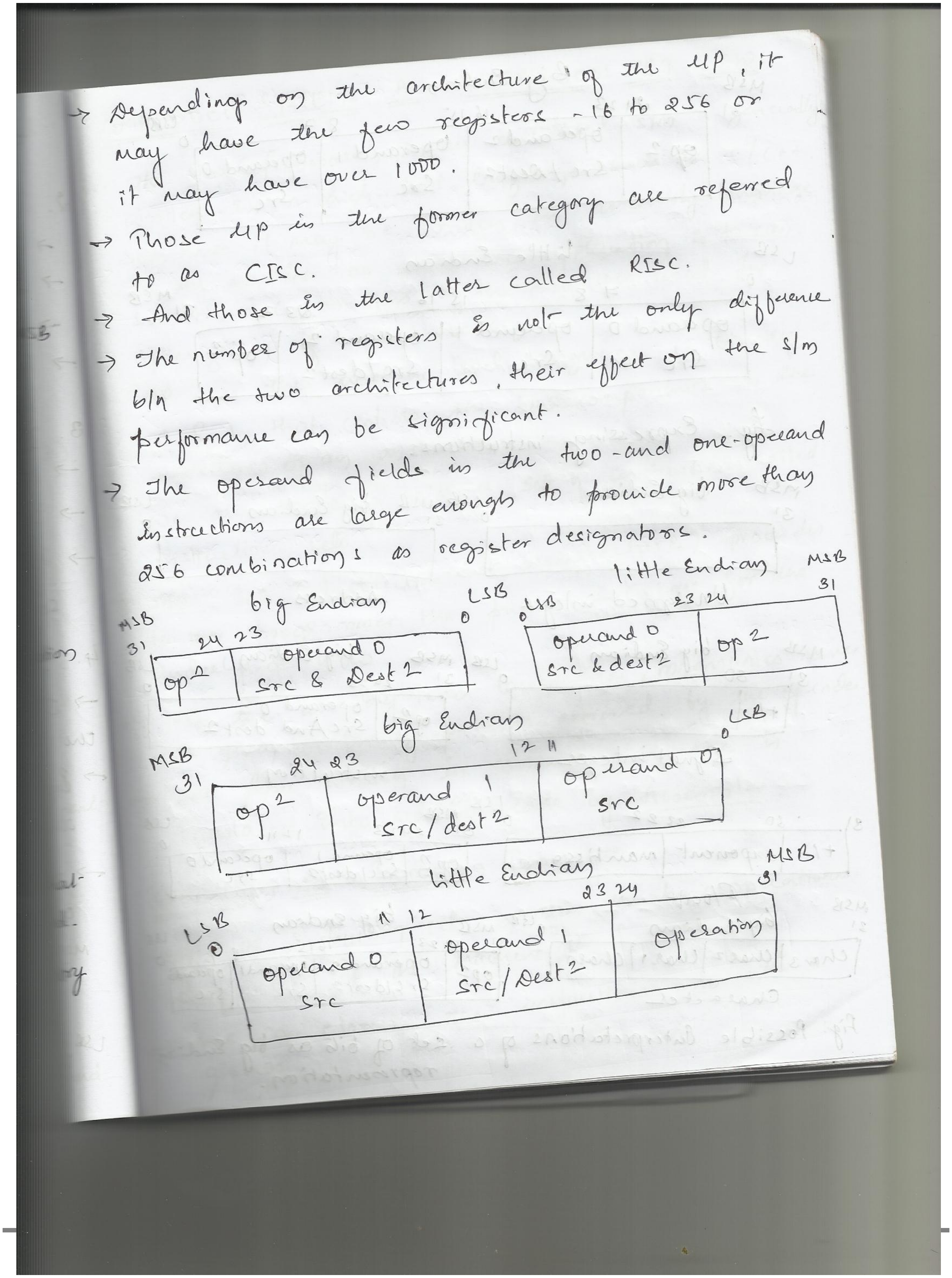
→ In fig we see that within 32-bit word, the bits are arranged into groups or fields.

→ Some of the fields are interpreted as the opera to be performed, and others are seen as the operands involved in the operation.

## Register - A first look.

→ A register is a special kind of memory is large enough to hold a single data wo

→ A registers is a piece of short-term mem that temporarily holds the operands during the execution of an instruction.



→ Depending on the architecture of the μP, it may have the few registers - 16 to 256 or it may have over 1000.

→ Those μP in the former category are refered to as CISC.

→ And those in the latter called RISC.

→ The number of registers is not the only difference b/n the two architectures, their effect on the s/m performance can be significant.

→ The operand fields in the two-and one-operand instructions are large enough to provide more than 256 combinations as register designators.

big Endian

| MSB 31 — 24 | 23 — LSB 0 |
|---|---|
| op 2 | operand 0<br>src & Dest 2 |

little Endian

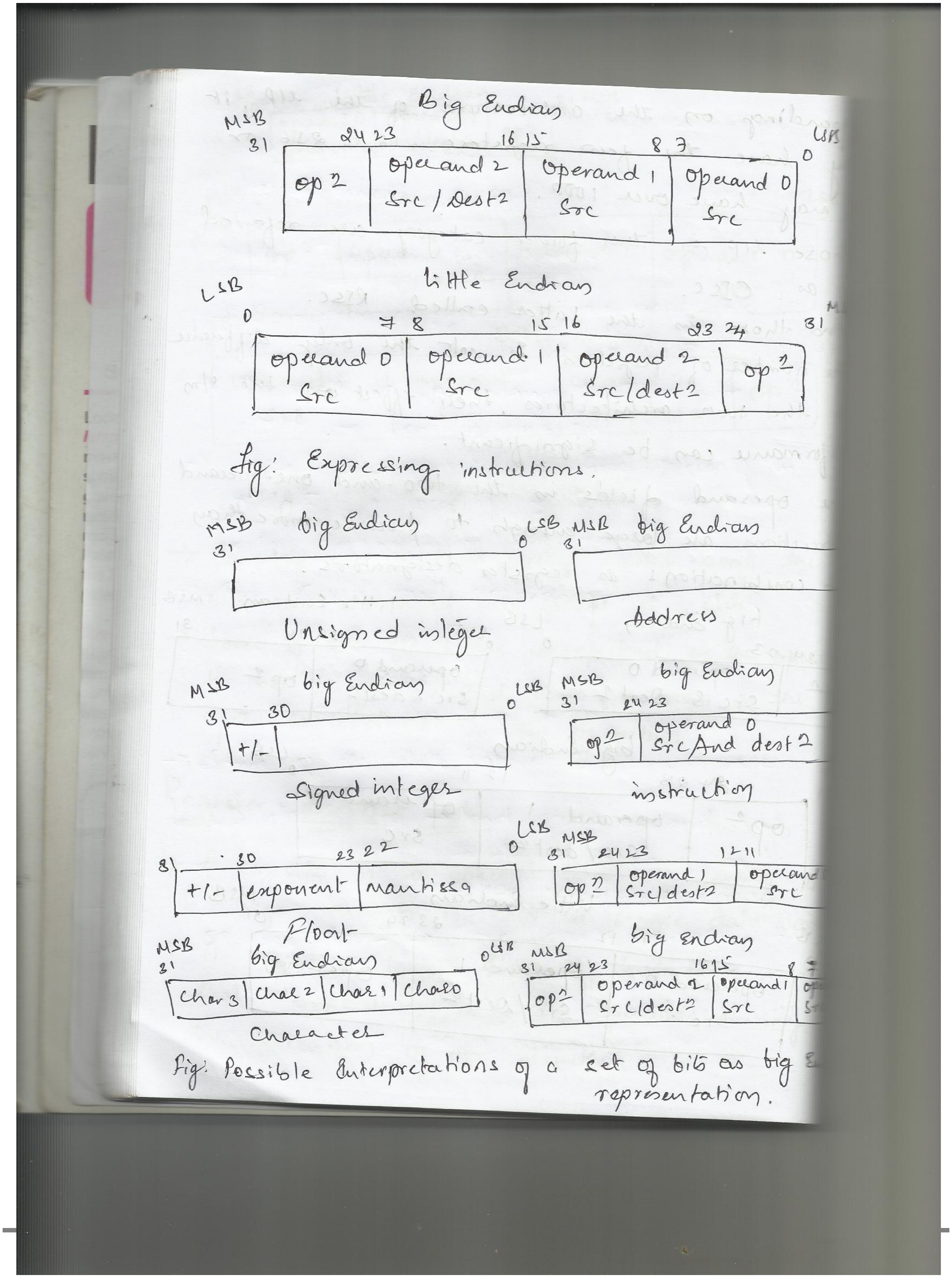
| LSB 0 — 23 | 24 — MSB 31 |
|---|---|
| operand 0<br>src & dest 2 | op 2 |

big Endian

| MSB 31 — 24 | 23 — 12 | 11 — LSB 0 |
|---|---|---|
| op 2 | operand 1<br>src / dest 2 | operand 0<br>src |

little Endian

| LSB 0 — 11 | 12 — 23 | 24 — MSB 31 |
|---|---|---|
| operand 0<br>src | operand 1<br>src / Dest 2 | operation |



Big Endian

MSB | | | LSB
---|---|---|---
31 24 | 23 16 | 15 8 | 7 0
op 2 | operand 2<br>Src / Dest2 | Operand 1<br>Src | Operand 0<br>Src

Little Endian

LSB | | | MSB
---|---|---|---
0 7 | 8 15 | 16 23 | 24 31
operand 0<br>Src | operand 1<br>Src | operand 2<br>Src/dest2 | op 2

fig: Expressing instructions.

MSB big Endian LSB

| 31 0 |
|---|
| |

Unsigned integer

MSB big Endians

| 31 |
|---|
| |

Address

MSB big Endian LSB

| 31 | 30 0 |
|---|---|
| +/- | |

Signed integer

MSB big Endian

| 31 24 | 23 |
|---|---|
| op 2 | operand 0<br>Src And dest 2 |

instruction

LSB

| 31 | 30 23 | 22 0 |
|---|---|---|
| +/- | exponent | mantissa |

Float

MSB

| 31 24 | 23 12 | 11 |
|---|---|---|
| op 2 | operand 1<br>Src/dest2 | operand<br>src |

big endian

MSB big Endian LSB

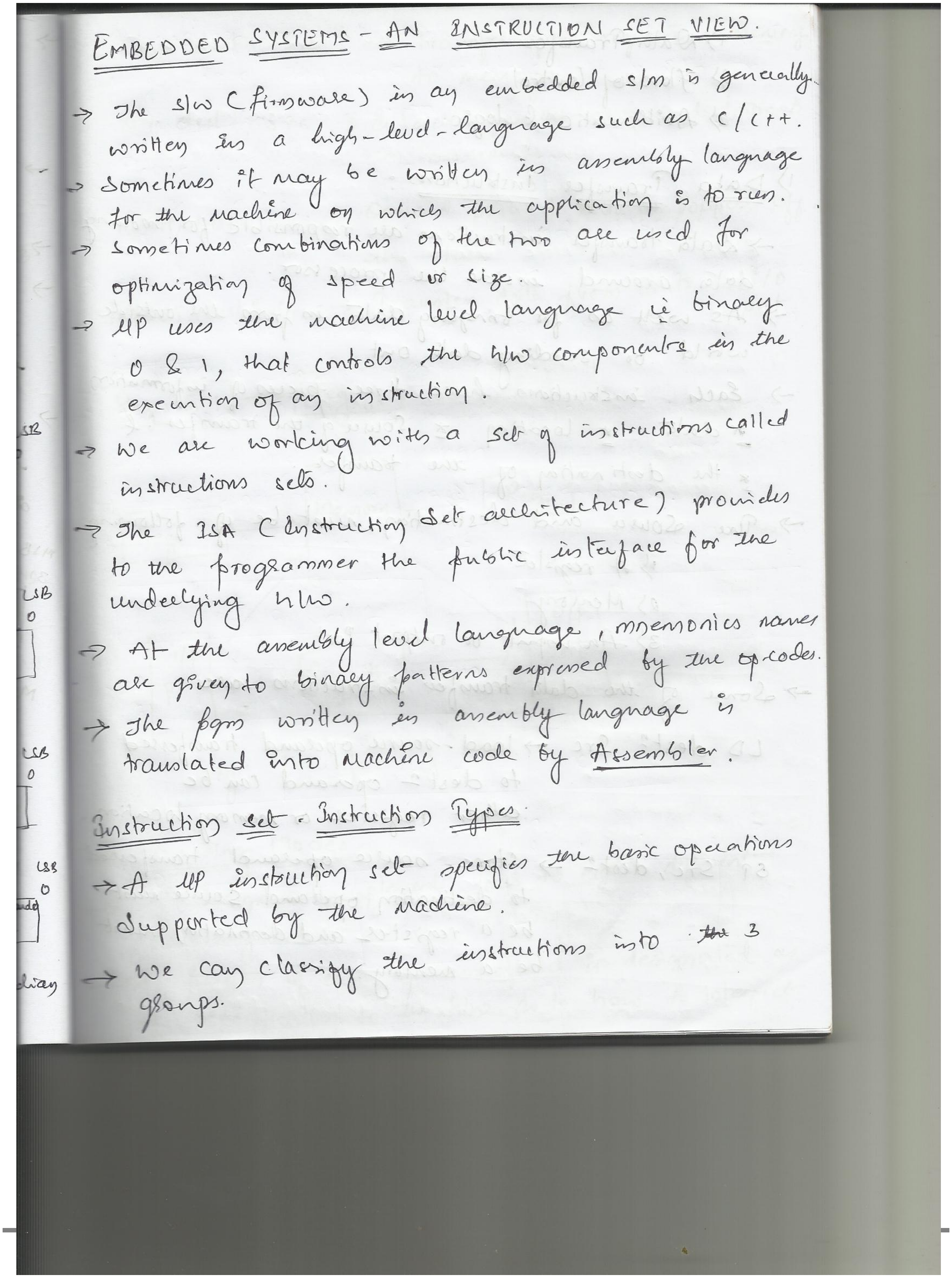
| 31 | | | 0 |
|---|---|---|---|
| Char 3 | Char 2 | Char 1 | Char 0 |

Character

MSB

| 31 24 | 23 16 | 15 8 | 7 |
|---|---|---|---|
| op 2 | operand 2<br>Src/dest2 | operand 1<br>Src | o<br>S |

fig: Possible interpretations of a set of bits as big E representation.



## EMBEDDED SYSTEMS - AN INSTRUCTION SET VIEW.

→ The s/w (firmware) in an embedded s/m is generally written in a high-level-language such as c/c++.

→ Sometimes it may be written in assembly language for the machine on which the application is to run.

→ Sometimes combinations of the two are used for optimization of speed or size.

→ µP uses the machine level language i.e binary 0 & 1, that controls the h/w components in the execution of an instruction.

→ We are working with a set of instructions called instructions sets.

→ The ISA (Instruction set architecture) provides to the programmer the public interface for the underlying h/w.

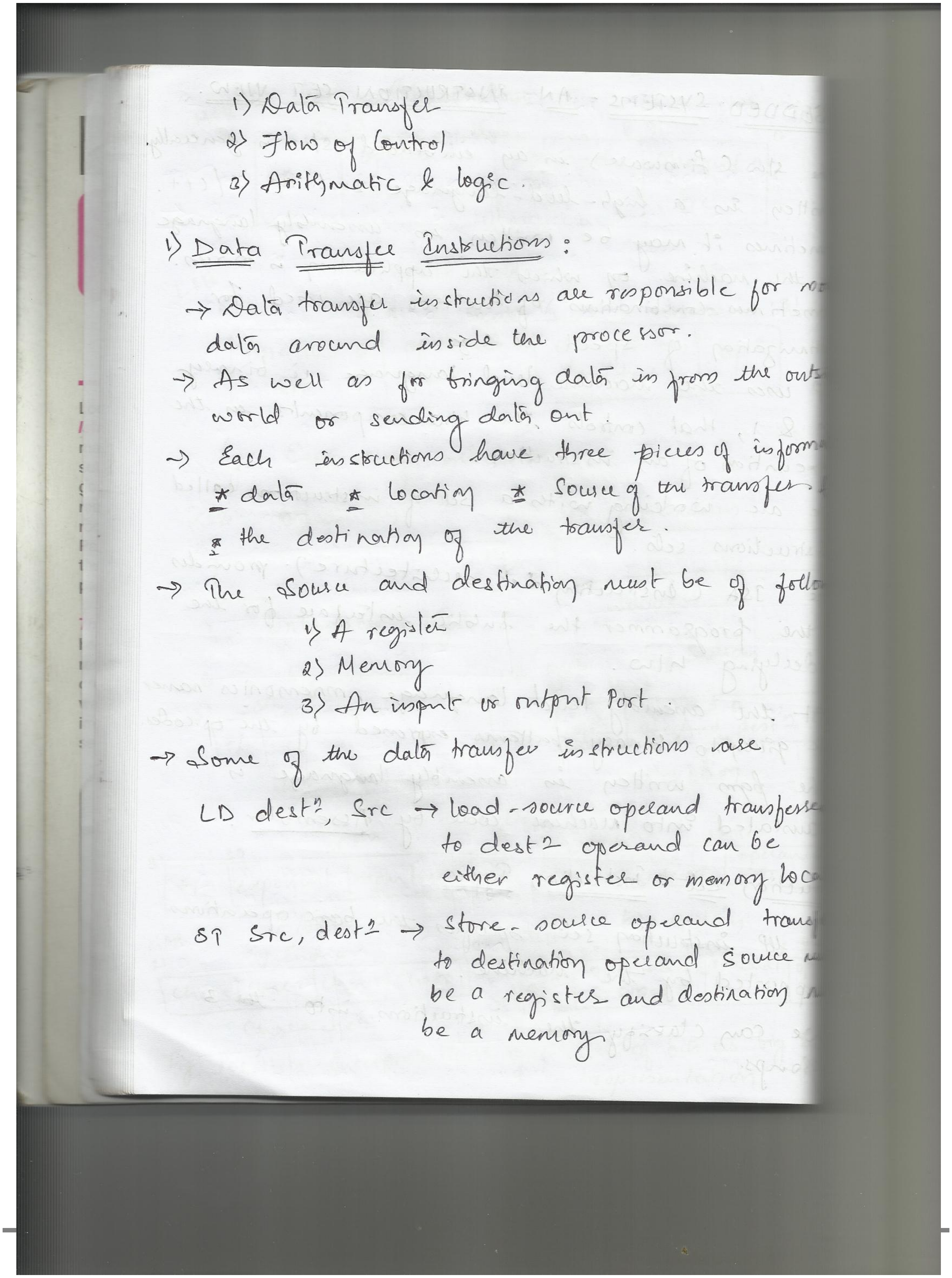
→ At the assembly level language, mnemonics names are given to binary patterns expressed by the op-codes.

→ The pgm written in assembly language is translated into machine code by Assembler.

## Instruction set - Instruction Types

→ A µP instruction set specifies the basic operations supported by the machine.

→ We can classify the instructions into ~~the~~ 3 groups.



1) Data Transfer
2) Flow of Control
3) Arithmatic & logic.

1) Data Transfer Instructions :

→ Data transfer instructions are responsible for mo data around inside the processor.

→ As well as for bringing data in from the outs world or sending data out.

→ Each instructions have three pieces of inform
* data * location * Source of the transfer
* the destination of the transfer.

→ The source and destination must be of follo
1) A register
2) Memory
3) An input or output Port.

→ Some of the data transfer instructions are

LD dest$^n$, Src → load - source opeand transferre to dest$^n$ operand can be either register or memory loca

ST Src, dest$^n$ → store - source opeand trans to destination opeand source m be a register and destination m be a memory

MOVE dest$^n$, Src → Transfer from register or memory to memory.

XCH dest$^n$, Src → Interchange the source and destination operands.

PUSH/POP → Operand pushed onto or popped off the stack.

IN/OUT dest$^n$, Src → Transfer data from or to an I/O port.

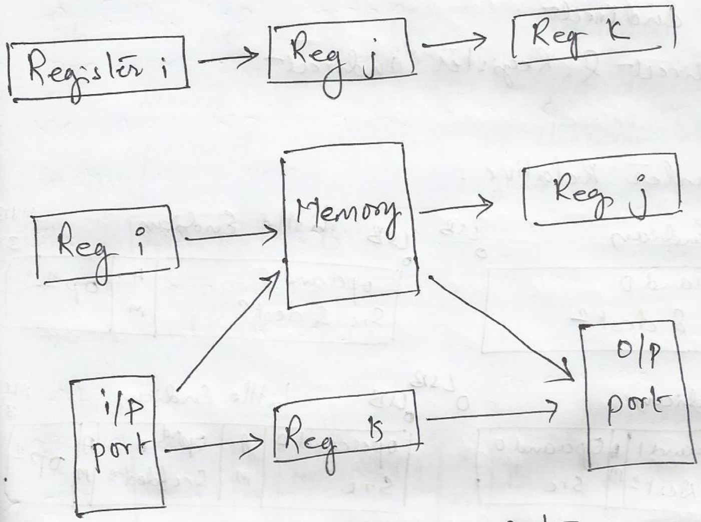
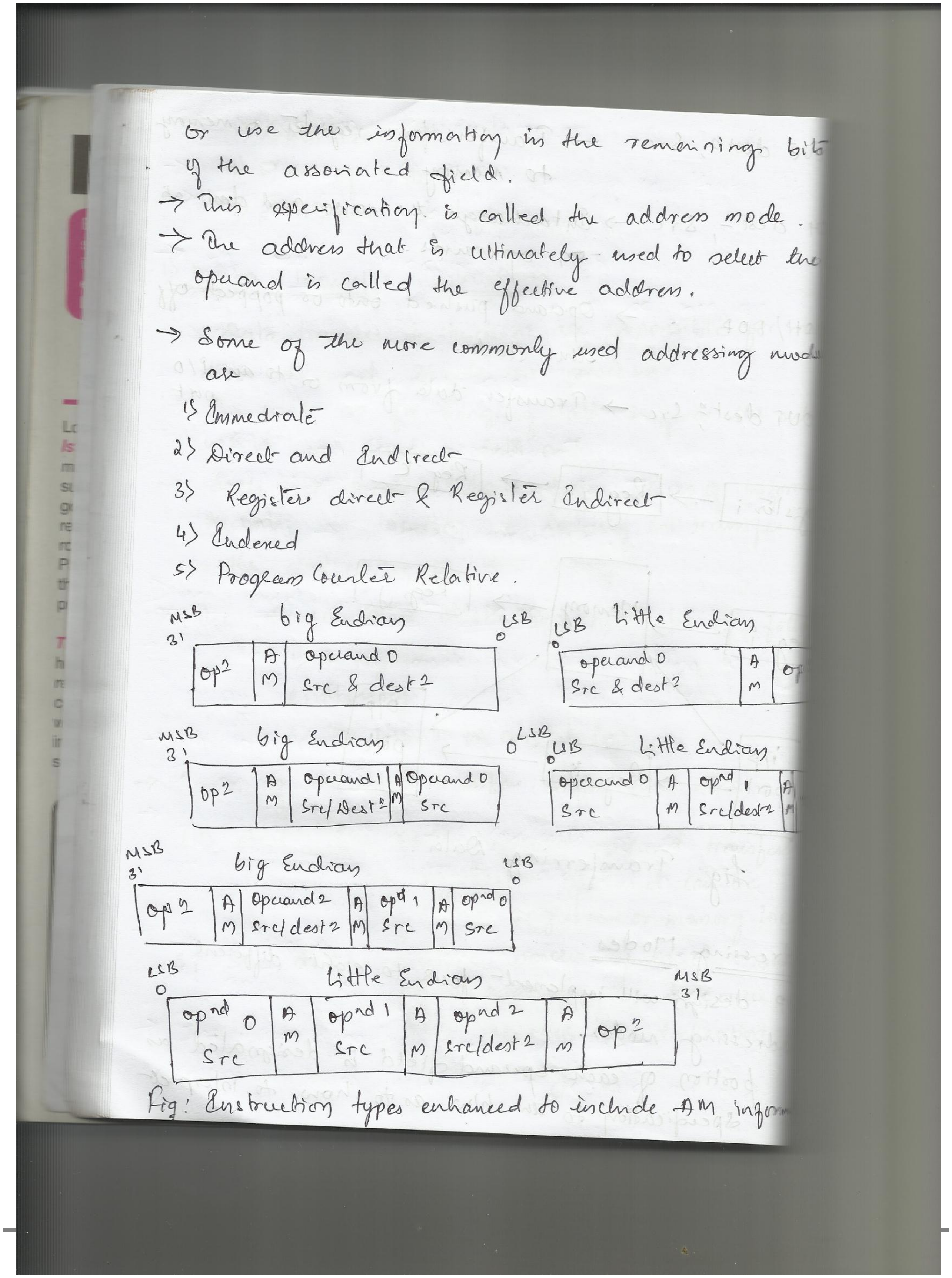


fig: Transferring Data.

## Addressing Modes.

→ uP design will implement four to eight different addressing modes.

→ A portion of each operand field is designated as a specification to the h/w as to how to interpret



or use the information in the remaining bits of the associated field.

→ This specification is called the address mode.

→ The address that is ultimately used to select the operand is called the effective address.

→ Some of the more commonly used addressing modes are

1> Immediate

2> Direct and Indirect

3> Register direct & Register Indirect

4> Indexed

5> Program Counter Relative.

MSB 31 big Endian LSB 0

| | | |
|---|---|---|
| op² | A<br>M | operand 0<br>src & dest 2 |

LSB 0 little Endian

| | | |
|---|---|---|
| operand 0<br>Src & dest 2 | A<br>M | op |

MSB 31 big Endian 0 LSB

| | | | | |
|---|---|---|---|---|
| op² | A<br>M | operand 1<br>Src/Dest 2 | A<br>M | Operand 0<br>Src |

LSB 0 little Endian

| | | | |
|---|---|---|---|
| operand 0<br>Src | A<br>M | opnd 1<br>Src/dest 2 | A<br>M |

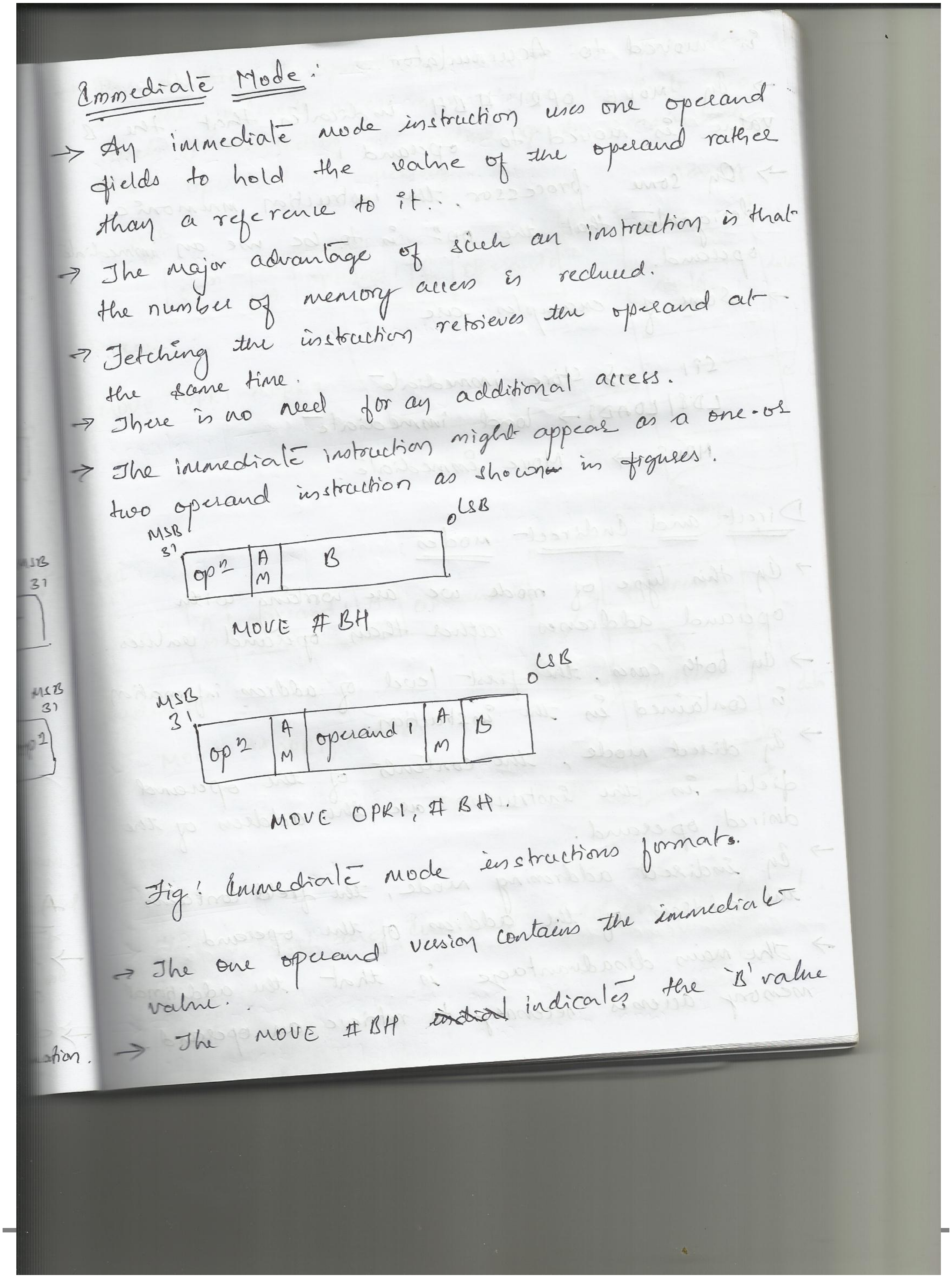
MSB 31 big Endian LSB 0

| | | | | | | |
|---|---|---|---|---|---|---|
| op² | A<br>M | Operand 2<br>src/dest 2 | A<br>M | opd 1<br>Src | A<br>M | opnd 0<br>Src |

LSB 0 little Endian MSB 31

| | | | | | | |
|---|---|---|---|---|---|---|
| opnd 0<br>Src | A<br>M | opnd 1<br>Src | A<br>M | opnd 2<br>src/dest 2 | A<br>M | op² |

Fig: Instruction types enhanced to include AM infor



## Immediate Mode :

→ An immediate mode instruction uses one operand fields to hold the value of the operand rather than a reference to it. .

→ The major advantage of such an instruction is that the number of memory acces is reduced.

→ Fetching the instruction retrieves the operand at the same time.

→ There is no need for any additional access.

→ The immediate instruction might appear as a one-or two operand instruction as shown in figures.

MSB 31 … 0 LSB

| op$^n$ | A M | B |
|---|---|---|

MOVE #BH

MSB 31 … 0 LSB

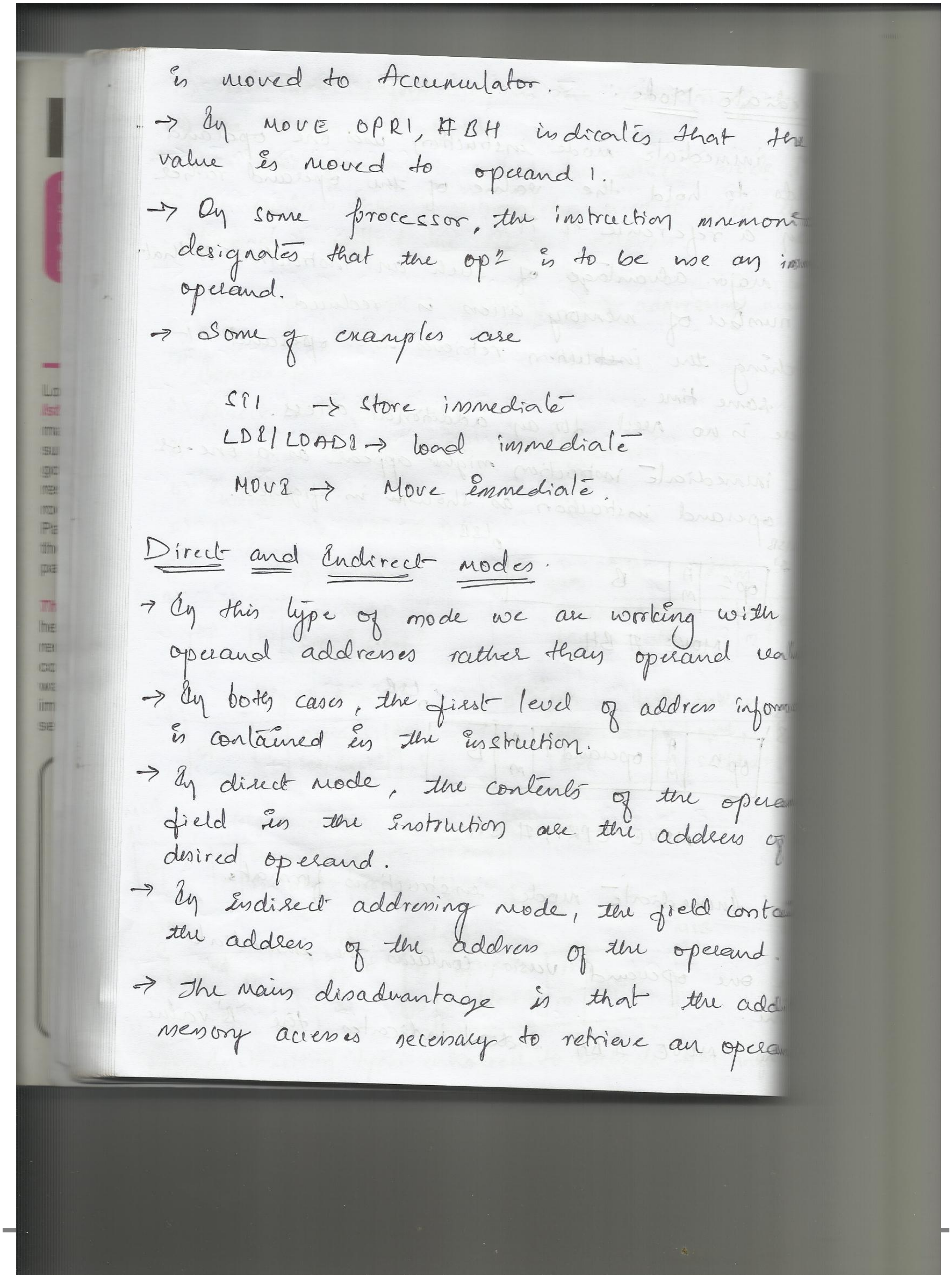
| op$^n$ | A M | operand 1 | A M | B |
|---|---|---|---|---|

MOVE OPR1, #BH.

Fig: Immediate mode instructions formats.

→ The one operand version contains the immediate value.

→ The MOVE #BH ~~indical~~ indicates the 'B' value



is moved to Accumulator.

→ In MOVE OPR1, #BH indicates that the value is moved to operand 1.

→ In some processor, the instruction mnemoni designates that the $op^n$ is to be use an im operand.

→ Some of examples are

STI → store immediate
LDI/LOADI → load immediate
MOVI → Move immediate.

<u>Direct and Indirect modes</u>.

→ In this type of mode we are working with operand addresses rather than operand val

→ In both cases, the first level of address inform is contained in the instruction.

→ In direct mode, the contents of the opera field in the instruction are the address of desired operand.

→ In indirect addressing mode, the field conta the address of the address of the operand.

→ The main disadvantage is that the add memory accesses necessary to retrieve an opera

→ The double ** symbols preceding the operands in the indirect access mode indicate that two levels of indirection are necessary to reach the final operand in memory.

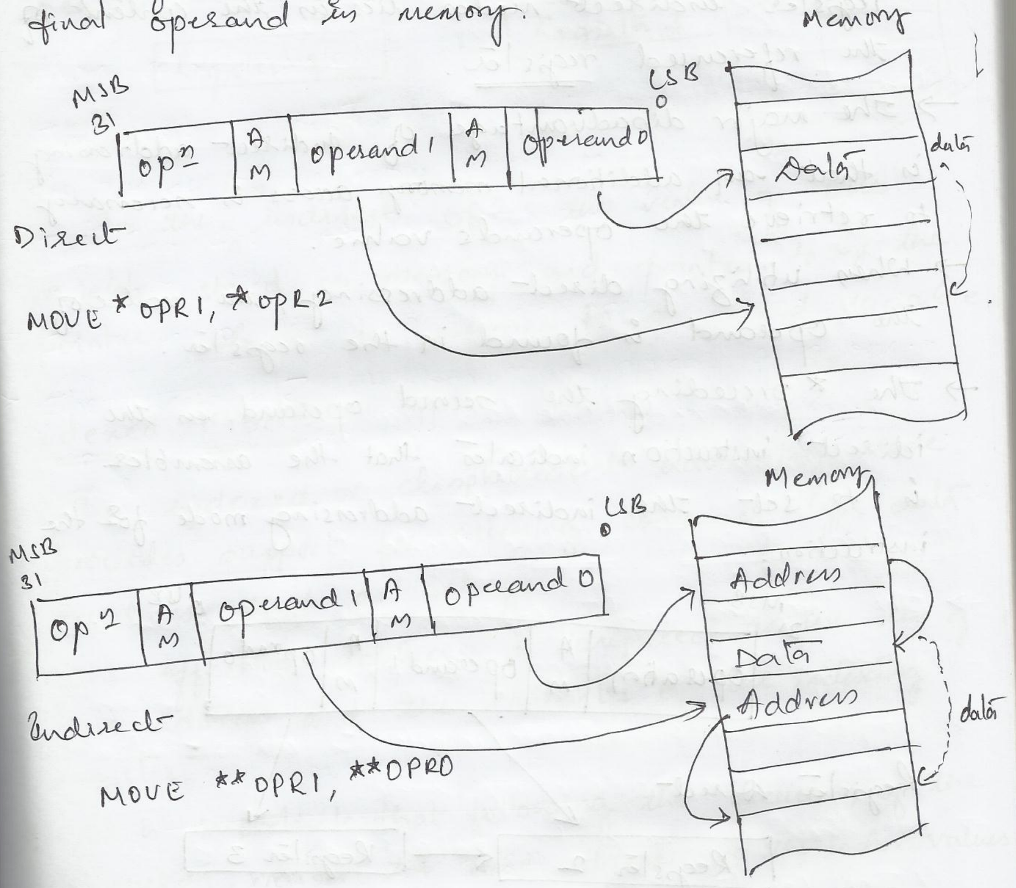
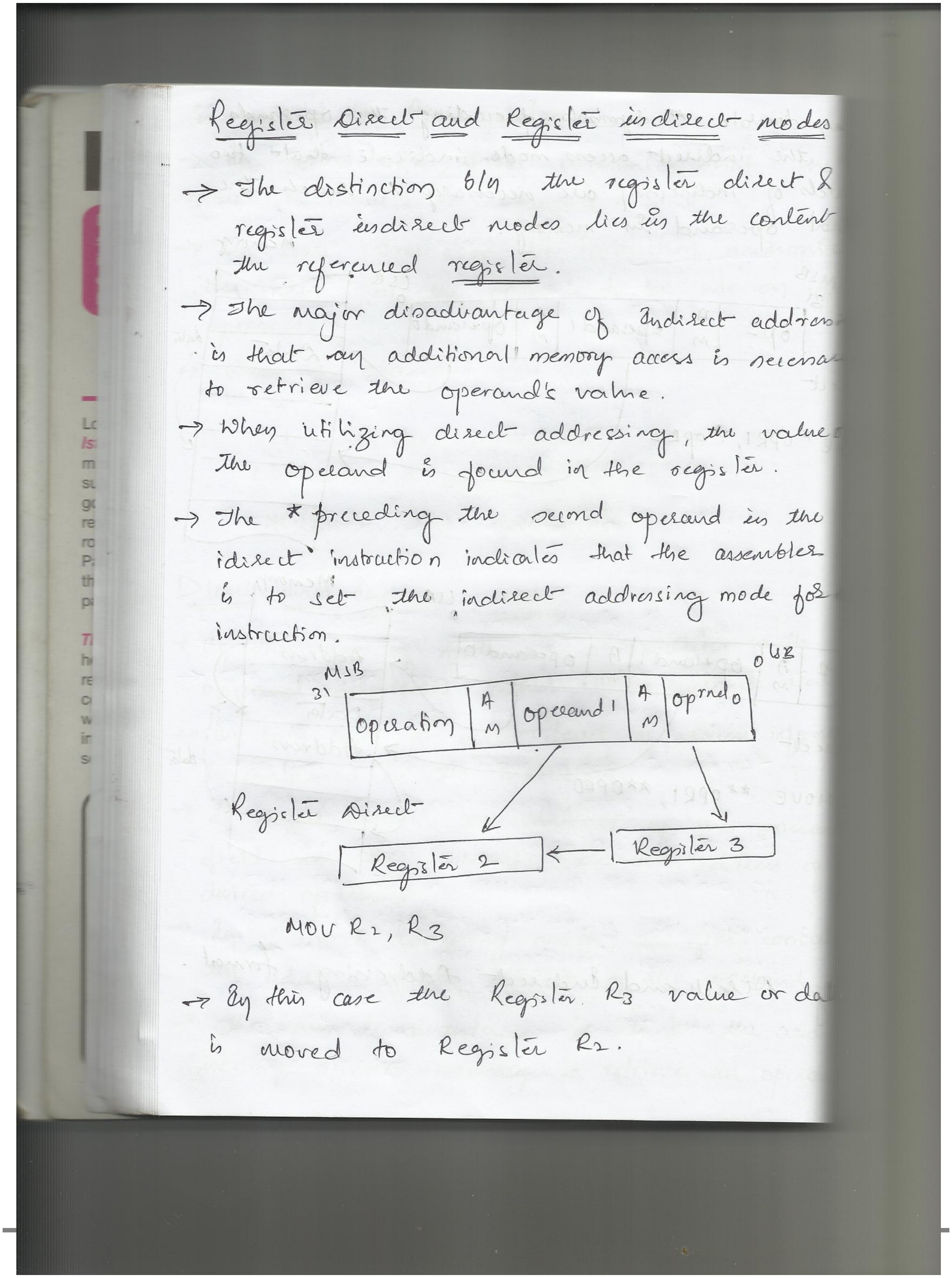


Fig: Direct and Indirect Addressing format.



## Register Direct and Register indirect modes

→ The distinction b/w the register direct & register indirect modes lies in the content the referenced register.

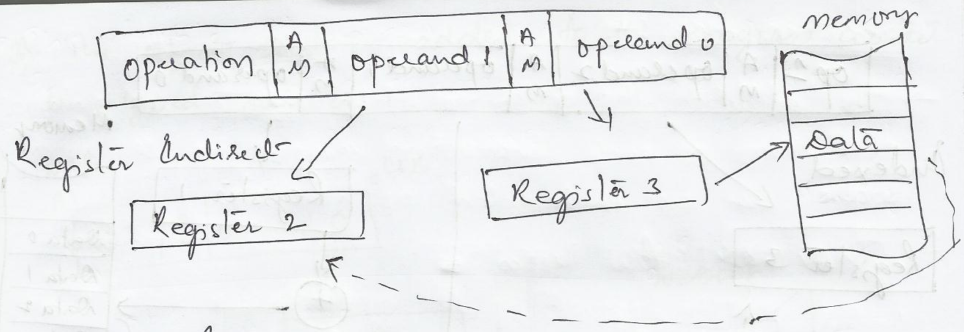
→ The major disadvantage of Indirect address is that an additional memory access is necessa to retrieve the operand's value.

→ When utilizing direct addressing, the value The operand is found in the register.

→ The * preceding the second operand in the idirect instruction indicates that the assembler is to set the indirect addressing mode for instruction.

MOV R2, R3

→ In this case the Register R3 value or da is moved to Register R2.



→ For the Reg indirect op2 the value of one variable, stored in memory and pointed to by the pointer, which is asigned to a second variable.

## Indexed Mode:

→ The indexed or displacement addressing mode provides support for accessing container-type data structures such as arrays.

→ The effective address is computed as the sum of the base address and the contents of the indexing register.

→ It is important that following the execution of the instruction, neither the base address nor the index values are changed.

→ The major disadvantage of indexed addressing is the time burdan associated with computing the address of the operand & then retrieving the value from memory.

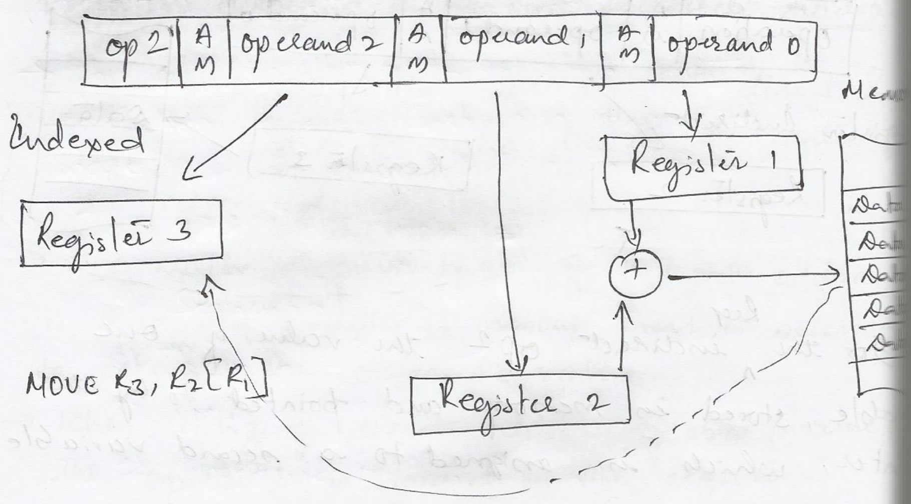


Fig: Indexed Mode Data Transfer Operations.

## Program Counter relative Mode:

→ The Program Counter contains the address in m
of the next instruction to be executed.

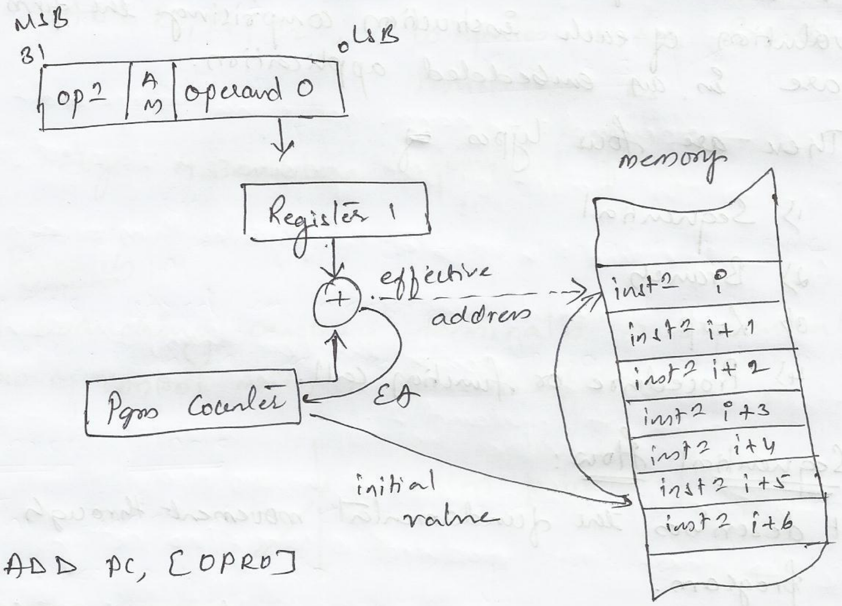
→ Program counter relative mode is almost iden
to the indexed addressing mode.

→ But there are several important differences.

1) The counter is assigned the value of the computed effective address, ie the contents of pgm counter are modified as a result of executing the instructions.

2) The value in the pgm counter serves as th
base address.

3) The offset that is added to the Program counter is signed number.



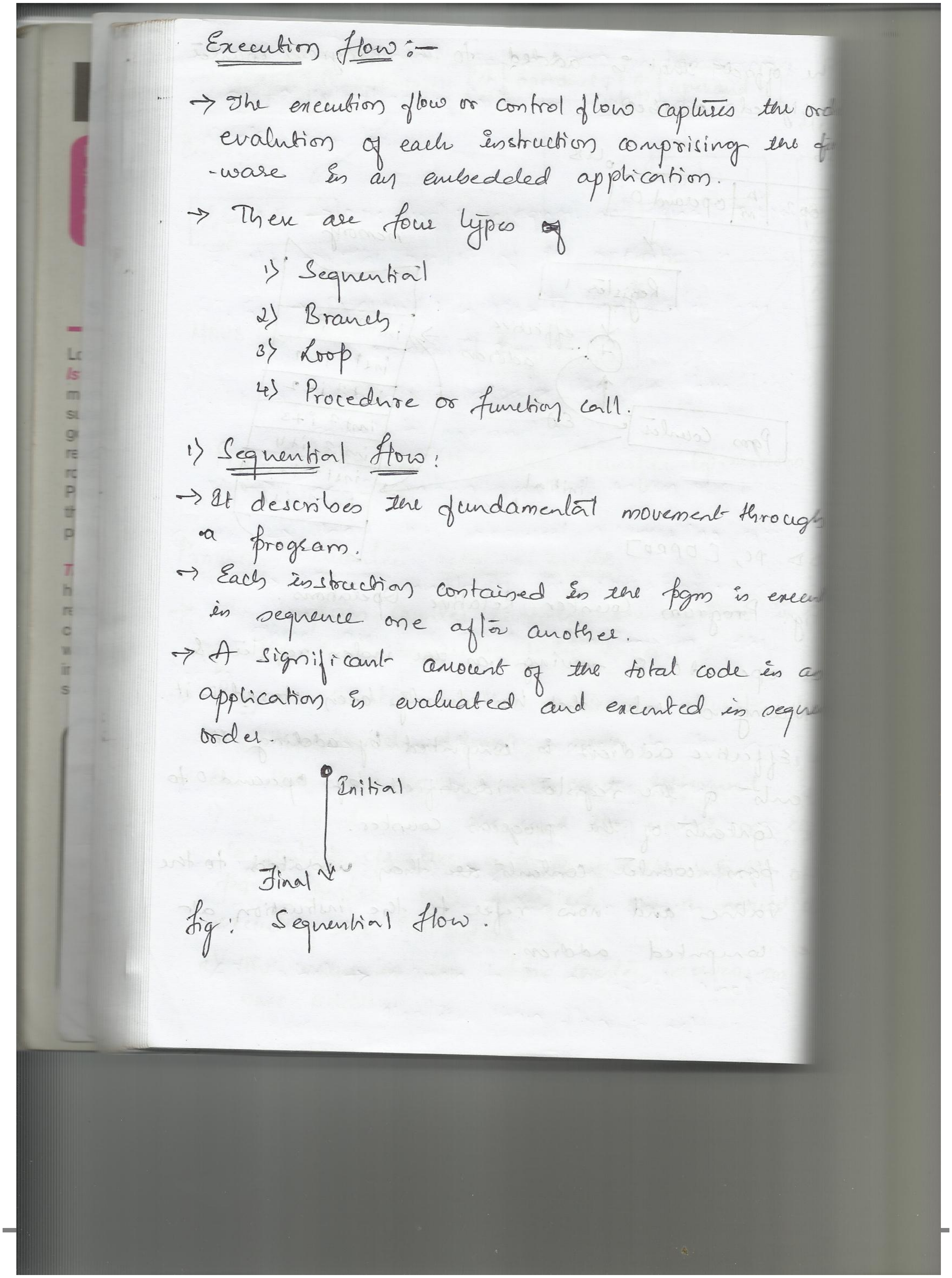
ADD PC, [OPR0]

Fig: Program Counter Relative Operations.

→ The operand 0 is serving as the index register & is holding a value that has already been stored in it.

→ The effective address is computed by adding the contents of the register identified by operand 0 to the contents of the program counter.

→ The pgm counter contents are then updated to the new value and now refer to the instruction at the computed address.



## Execution flow :-

→ The execution flow or control flow captures the ord evalution of each instruction comprising the f-ware in an embedded application.

→ There are four types of

1) Sequential
2) Branch
3) Loop
4) Procedure or function call.

## 1) Sequential flow:

→ It describes the fundamental movement through a program.

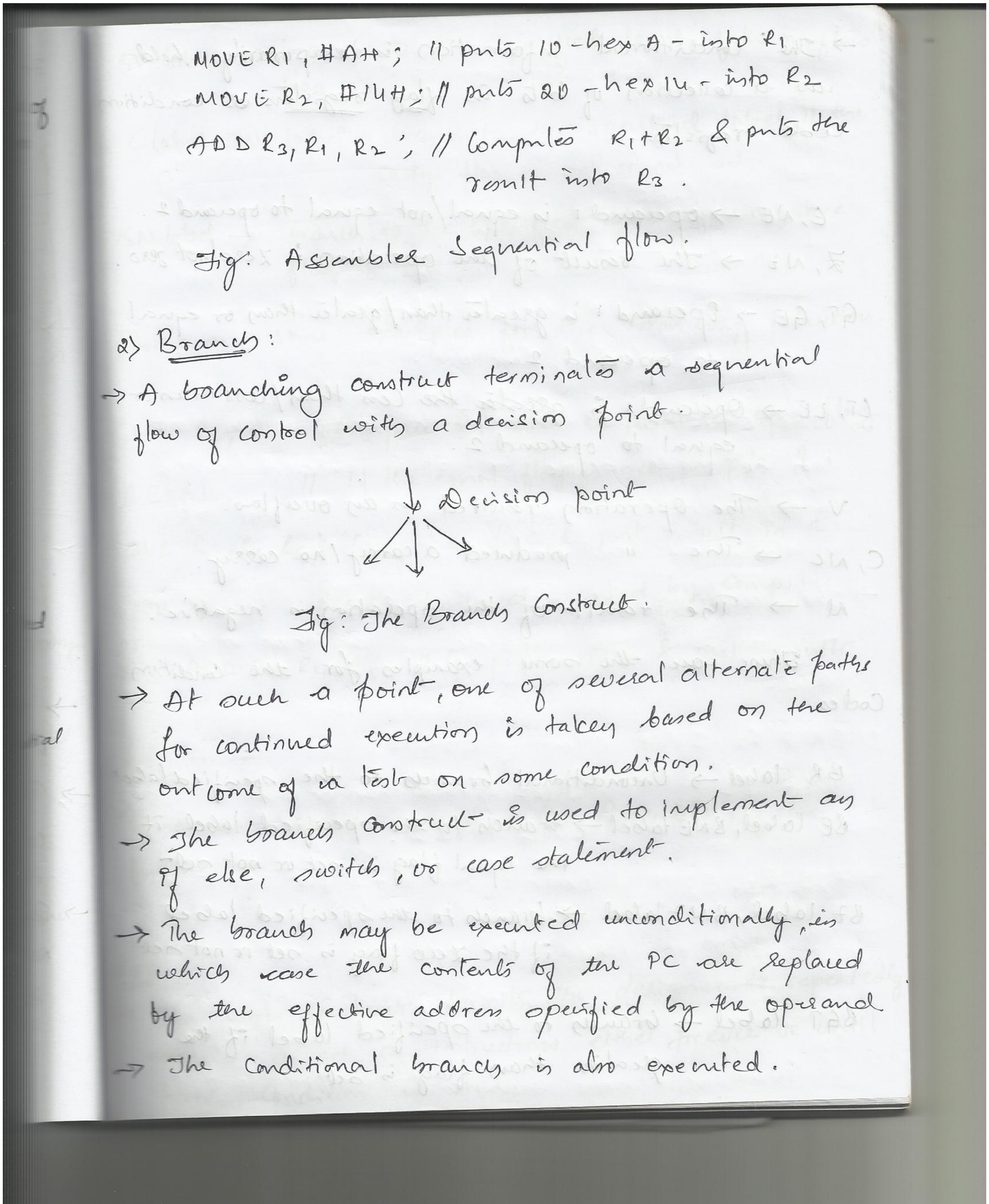
→ Each instruction contained in the pgm is execu in sequence one after another.

→ A significant amount of the total code in a application is evaluated and executed in sequ order.

Initial

Final

fig: Sequential flow.



MOVE R1, #AH; // puts 10-hex A- into R1
MOVE R2, #14H; // puts 20 -hex 14- into R2
ADD R3, R1, R2; // Computes R1+R2 & puts the result into R3.

Fig: Assembler Sequential flow.

2) Branch:

→ A branching construct terminates a sequential flow of control with a decision point.

Decision point

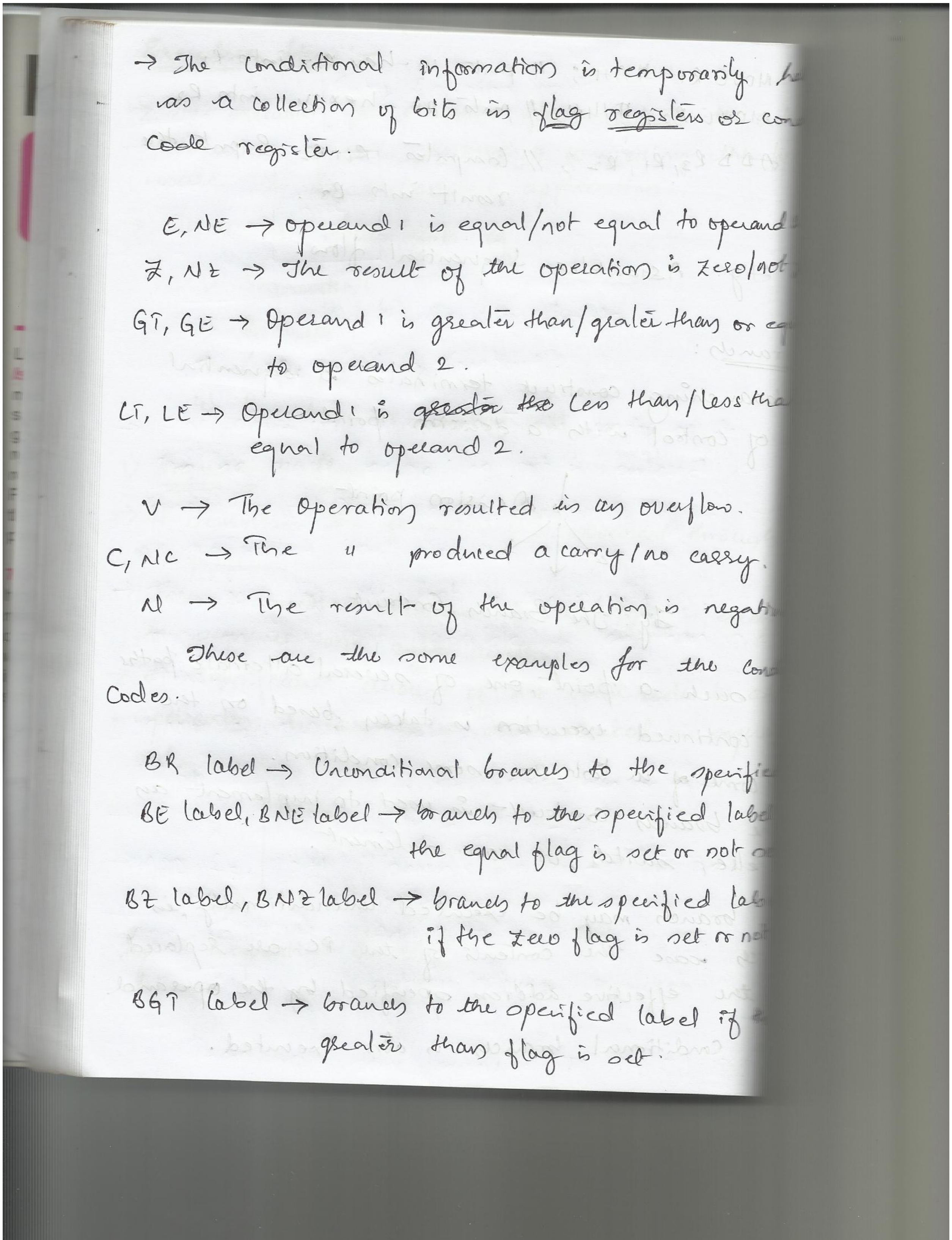
Fig: The Branch Construct.

→ At such a point, one of several alternate paths for continued execution is taken based on the outcome of a test on some condition.

→ The branch construct is used to implement an if else, switch, or case statement.

→ The branch may be executed unconditionally, in which case the contents of the PC are replaced by the effective address specified by the operand.

→ The conditional branch is also executed.



→ The conditional information is temporarily h
as a collection of bits in flag registers or con
Code register.

E, NE → operand 1 is equal/not equal to operand

Z, NZ → The result of the operation is zero/not

GT, GE → Operand 1 is greater than/greater than or eq
to operand 2.

LT, LE → Operand 1 is ~~greater the~~ less than/less tha
equal to operand 2.

V → The Operation resulted in an overflow.

C, NC → The " produced a carry/no carry.

N → The result of the operation is negat

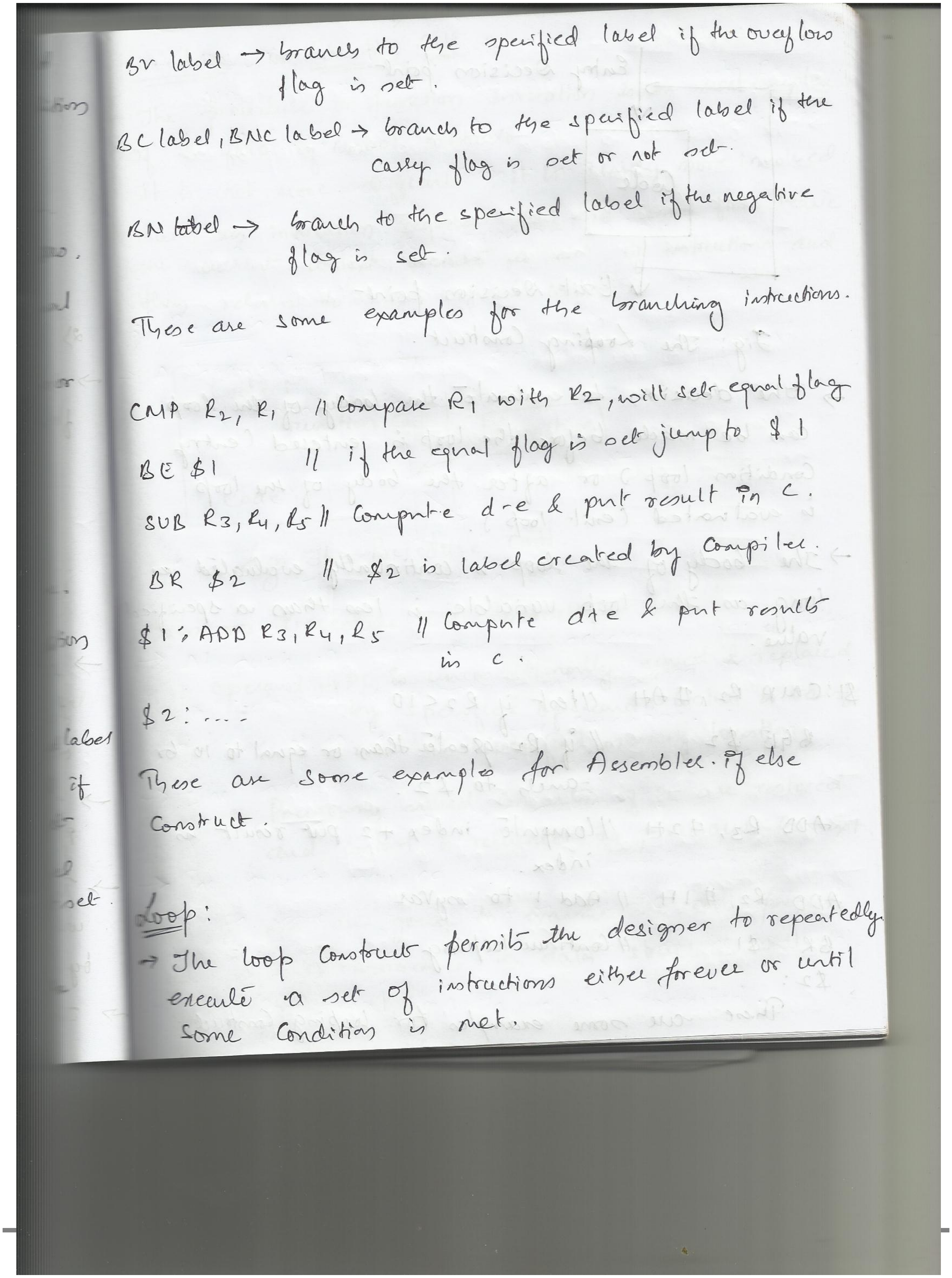
These are the some examples for the Con
Codes.

BR label → Unconditional branch to the specifie

BE label, BNE label → branch to the specified labe
the equal flag is set or not o

BZ label, BNZ label → branch to the specified lab
if the zero flag is set or no

BGT label → branch to the specified label if
greater than flag is set.



Bv label → branches to the specified label if the overflow flag is set.

BC label, BNC label → branch to the specified label if the carry flag is set or not set.

BN label → branch to the specified label if the negative flag is set.

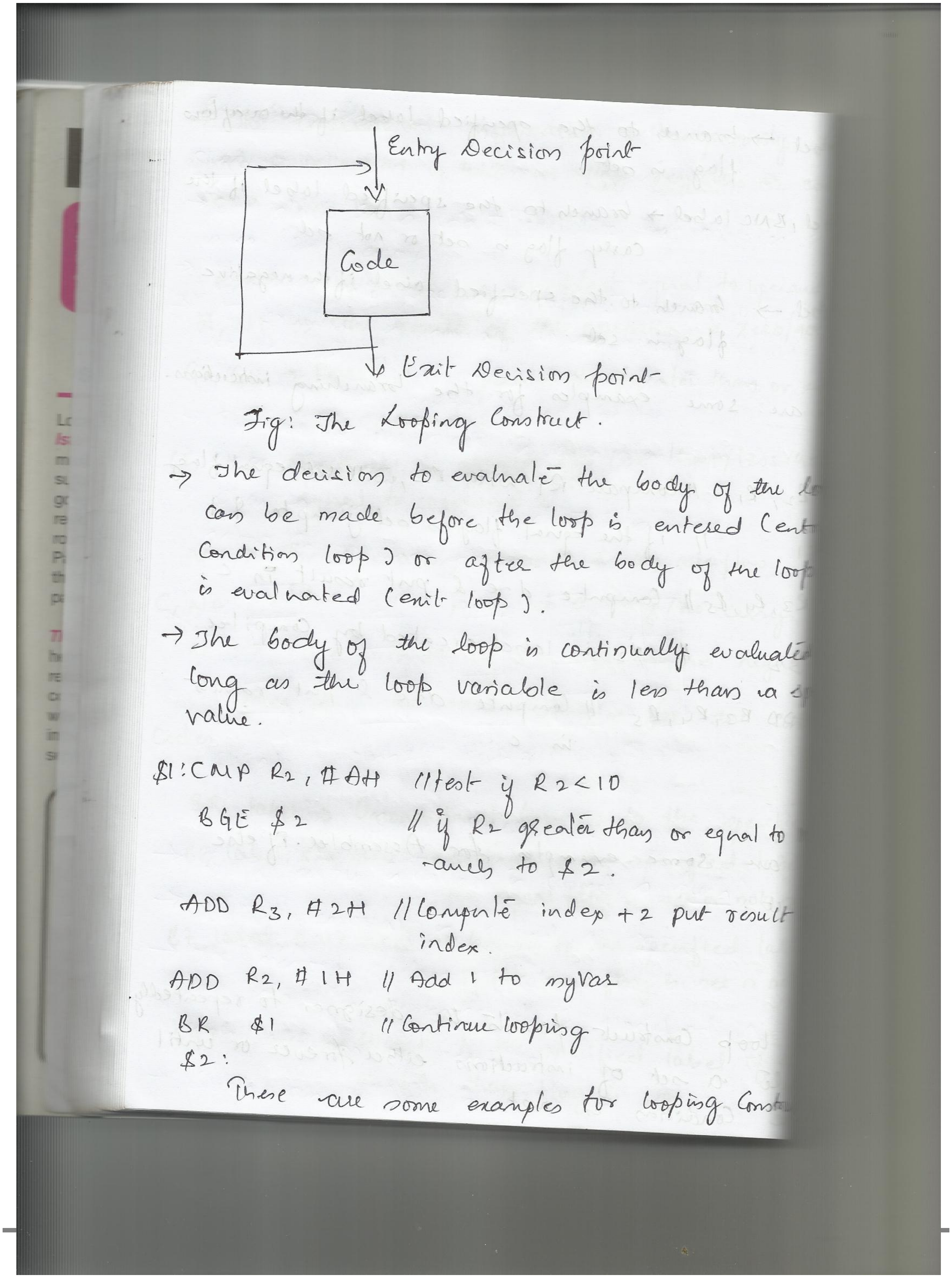
These are some examples for the branching instructions.

```
CMP R2, R1   // Compare R1 with R2, will set equal flag
BE $1        // if the equal flag is set jump to $1
SUB R3, R4, R5 // Compute d-e & put result in c.
BR $2        // $2 is label created by Compiler.
$1: ADD R3, R4, R5  // Compute d+e & put results
                       in c.
$2: ....
```

These are some examples for Assembler. if else construct.

**Loop:**

→ The loop construct permits the designer to repeatedly execute a set of instructions either forever or until some condition is met.



Entry Decision point

Code

Exit Decision point

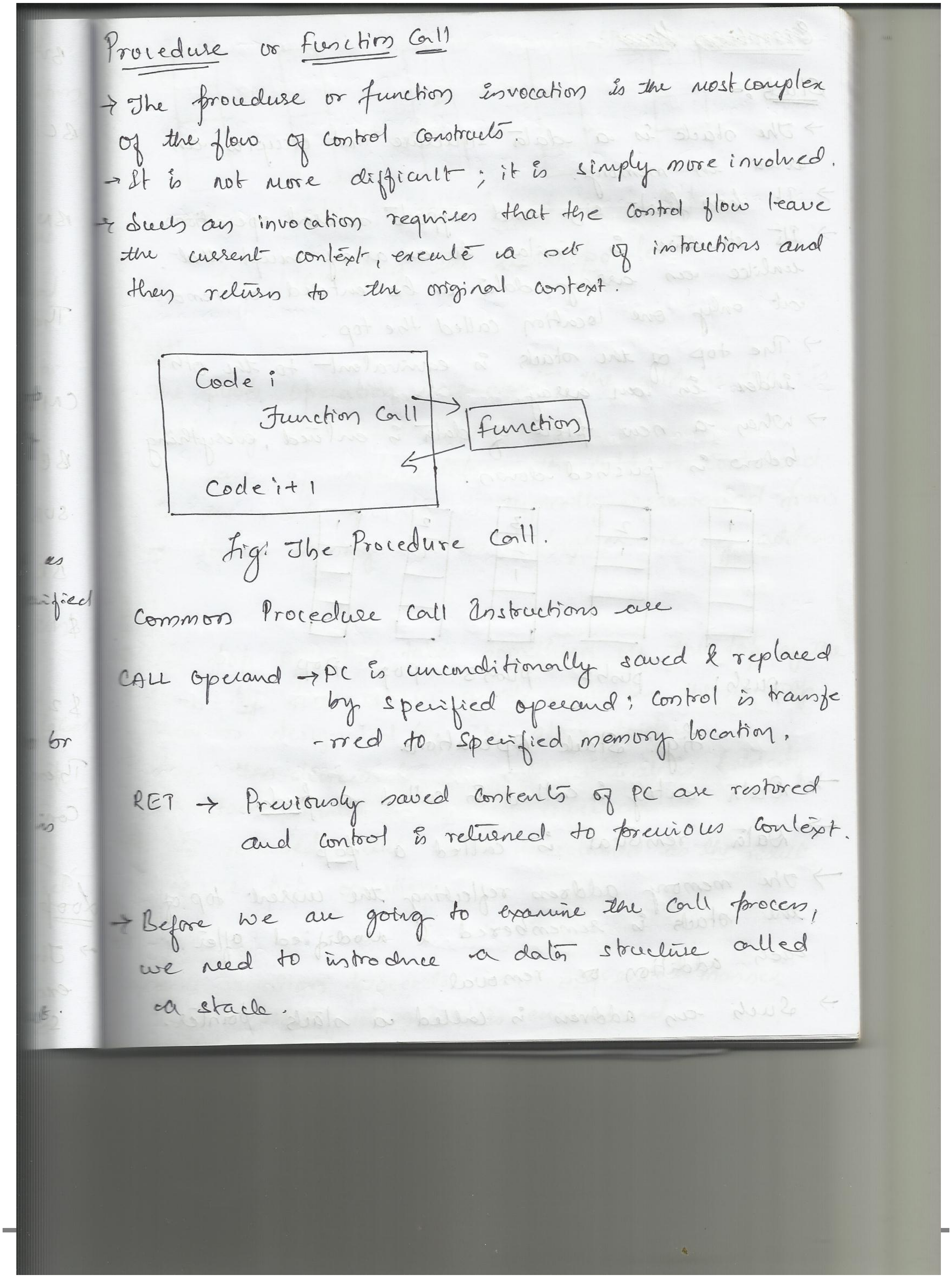
Fig: The Looping Construct.

→ The decision to evaluate the body of the lo
can be made before the loop is entered (ent
Condition loop) or after the body of the loop
is evaluated (exit loop).

→ The body of the loop is continually evaluate
long as the loop variable is less than a sp
value.

```
$1:CMP R2, #AH    //test if R2<10
   BGE $2          // if R2 greater than or equal to
                      -anch to $2.
  ADD R3, #2H     //Compute index +2 put result
                      index.
 ADD R2, #1H      // Add 1 to myVar
  BR  $1           // Continue looping
 $2:
```

These are some examples for looping Constr



## Procedure or Function Call

→ The procedure or function invocation is the most complex of the flow of control constructs.

→ It is not more difficult; it is simply more involved.

→ Such an invocation requires that the control flow leave the current context, execute a set of instructions and then return to the original context.

Code i
Function Call
function
Code i+1

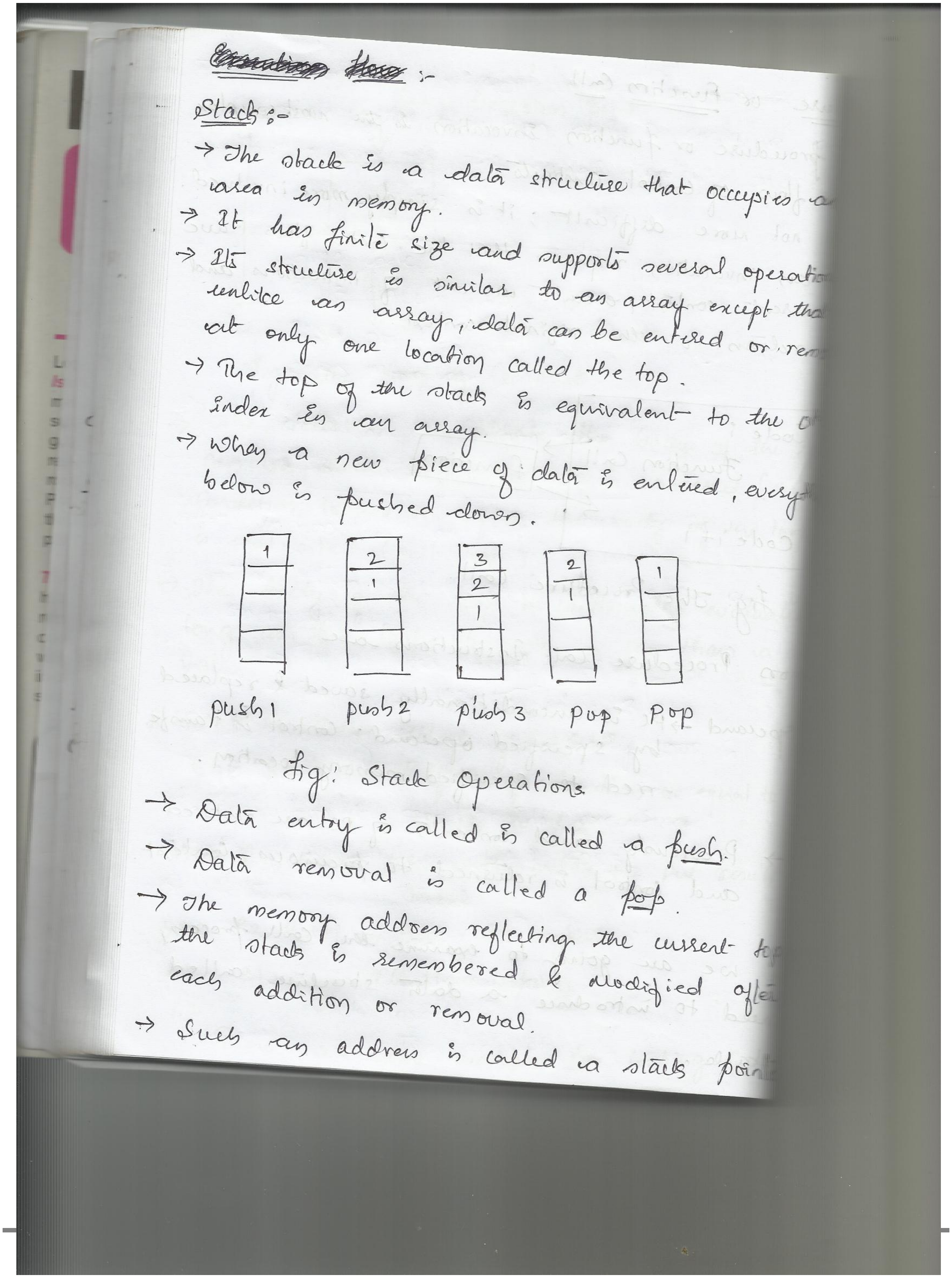
Fig: The Procedure Call.

Common Procedure call Instructions are

CALL operand → PC is unconditionally saved & replaced by specified operand; control is transferred to specified memory location.

RET → Previously saved contents of PC are restored and control is returned to previous context.

→ Before we are going to examine the call process, we need to introduce a data structure called a stack.



[illegible] :-

Stack :-

→ The stack is a data structure that occupies a area in memory.

→ It has finite size and supports several operation unlike an array, data can be entered or remo at only one location called the top.

→ The top of the stack is equivalent to the o index in an array.

→ When a new piece of data is entered, everyth below is pushed down.

1
2
1
3
2
1
2
1
1
push 1
push 2
push 3
pop
pop

fig: Stack operations

→ Data entry is called is called a push.

→ Data removal is called a pop.

→ The memory address reflecting the current to the stack is remembered & modified afte each addition or removal.

→ Such an address is called a stack poin

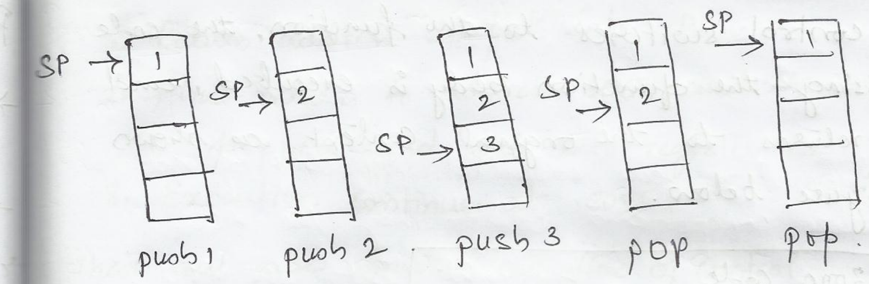


Fig: Managing the stack pointer.

PUSH :

→ The push operation increments the address that is held by the stack pointer.

→ For ease of implementation, the address contained in the stack pointer is typically incremented from a lower memory address to higher memory address.

POP :

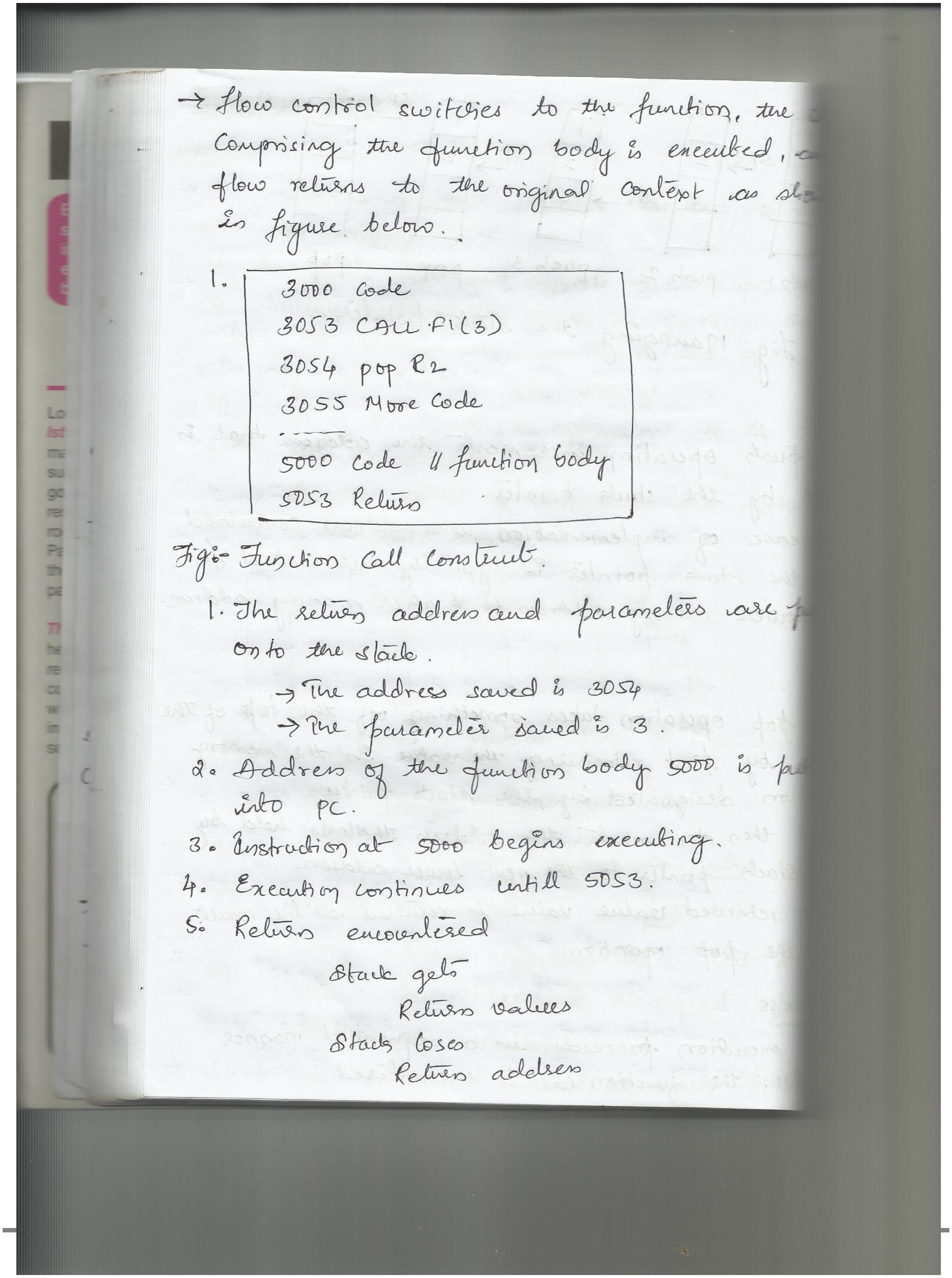
→ The pop operation takes something off the top of the stack by first retreiving the value in the memory location designated by the stack pointer.

→ And then decrements the address that is held by the stack pointer to the next lower address.

→ The retrieved value value is returned as the result of the pop operation.

→ Process :

→ code execution proceeds in a sequential manner untill the function call is encountered.



→ Flow control switches to the function, the Comprising the function body is executed, a flow returns to the original context as sho in figure below.

```
1.  3000 Code
    3053 CALL F1(3)
    3054 pop R2
    3055 More Code
    ....
    5000 Code // function body
    5053 Return
```

Fig:- Function Call Constuct.

1. The return address and parameters are p onto the stack.
   → The address saved is 3054
   → The parameter saved is 3.
2. Address of the function body 5000 is pu into PC.
3. Instruction at 5000 begins executing.
4. Execution continues untill 5053.
5. Return encountered
   Stack gets
   Return values
   Stack loses
   Return address

6. Return address is put into PC.

7. flow returns to address 3054, and the top of stack is popped and put into register R2.

8. Execution continues at 3055.

These all are function call flow of control.

→ An additional function call been encountered in function F1, an additional identical process would have occured.

→ The process can be repeated multiple times.

→ we must aware of stack. If it is overflow means we begins to lose information, particularly the return address.

## ARITHMETIC & LOGIC

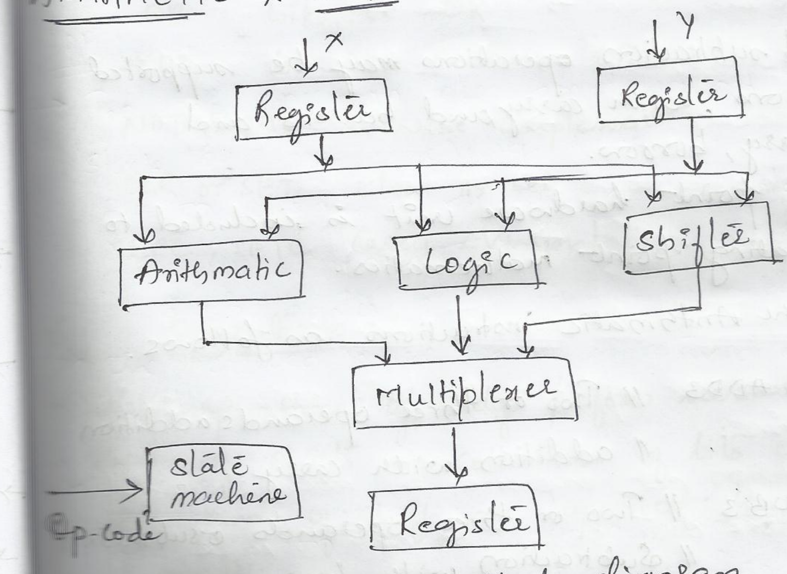
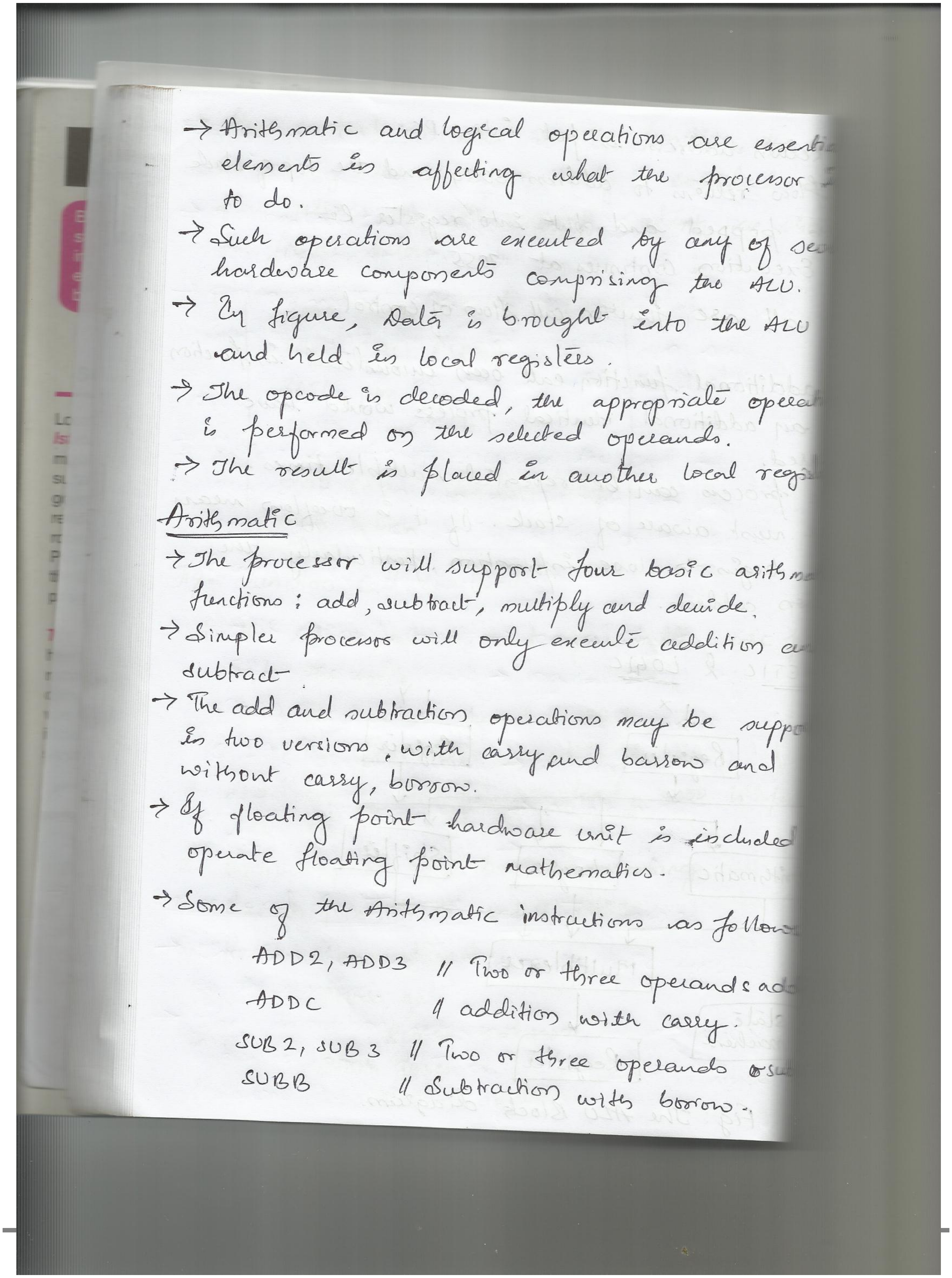


Fig: The ALU Block diagram.



→ Arithmatic and logical operations are essenti elements in affecting what the processor i to do.

→ Such operations are executed by any of sev hardware components comprising the ALU.

→ In figure, Data is brought into the ALU and held in local registers.

→ The opcode is decoded, the appropriate operat is performed on the selected operands.

→ The result is placed in another local regis

Arithmatic

→ The processor will support four basic arithm functions: add, subtract, multiply and devide.

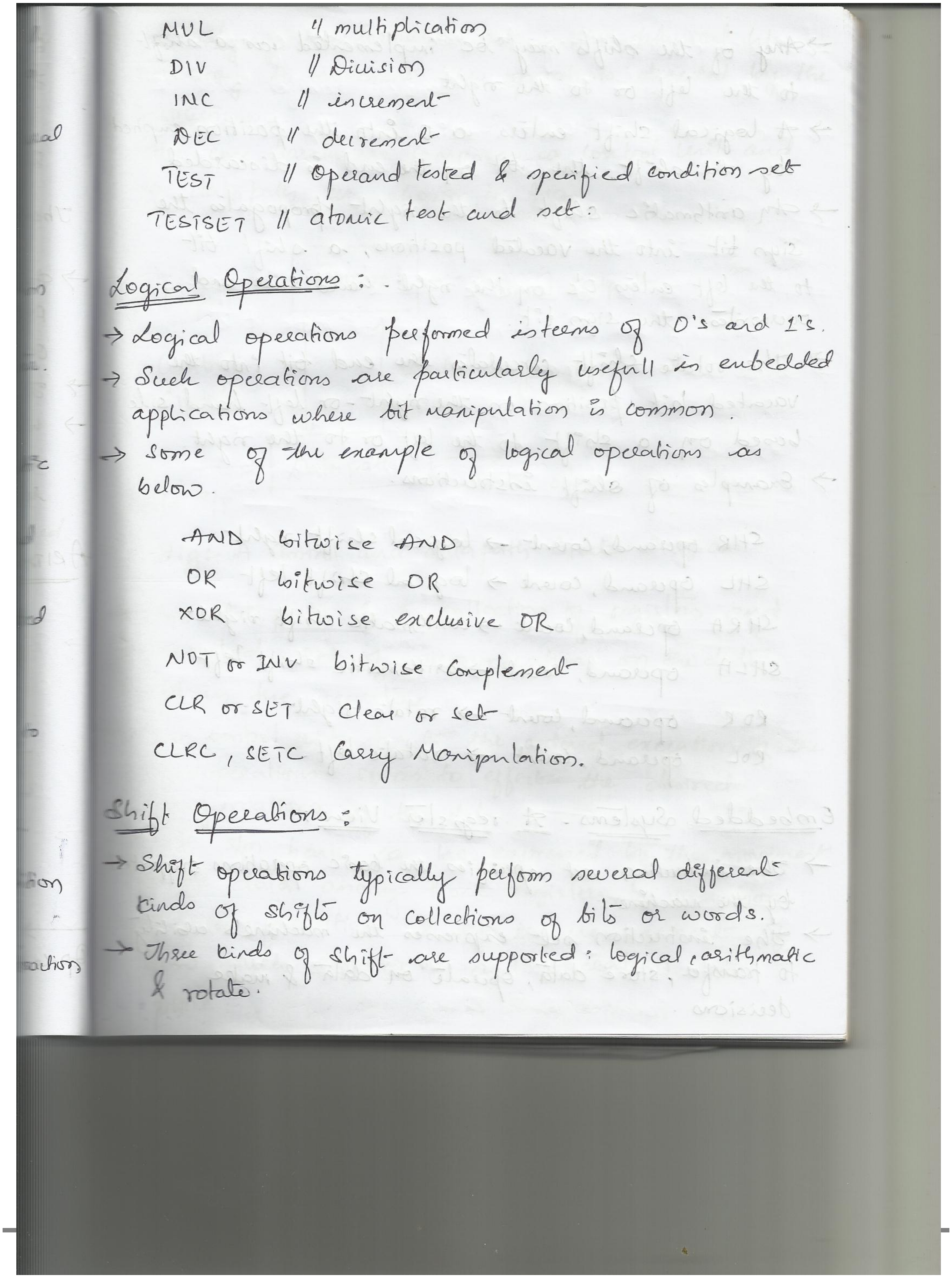
→ Simpler processors will only execute addition an subtract.

→ The add and subtraction operations may be suppo in two versions, with carry and borrow and without carry, borrow.

→ If floating point hardware unit is included operate floating point mathematics.

→ Some of the Arithmatic instructions as follow

```
ADD2, ADD3    // Two or three operands ad
ADDC          // addition with carry.
SUB2, SUB3    // Two or three operands o sub
SUBB          // Subtraction with borrow.
```



| | |
|---|---|
| MUL | // multiplication |
| DIV | // Division |
| INC | // increment |
| DEC | // decrement |
| TEST | // Operand tested & specified condition set |
| TESTSET | // atomic test and set. |

## Logical Operations :

→ Logical operations performed interms of 0's and 1's.

→ Such operations are particularly usefull in embedded applications where bit manipulation is common.

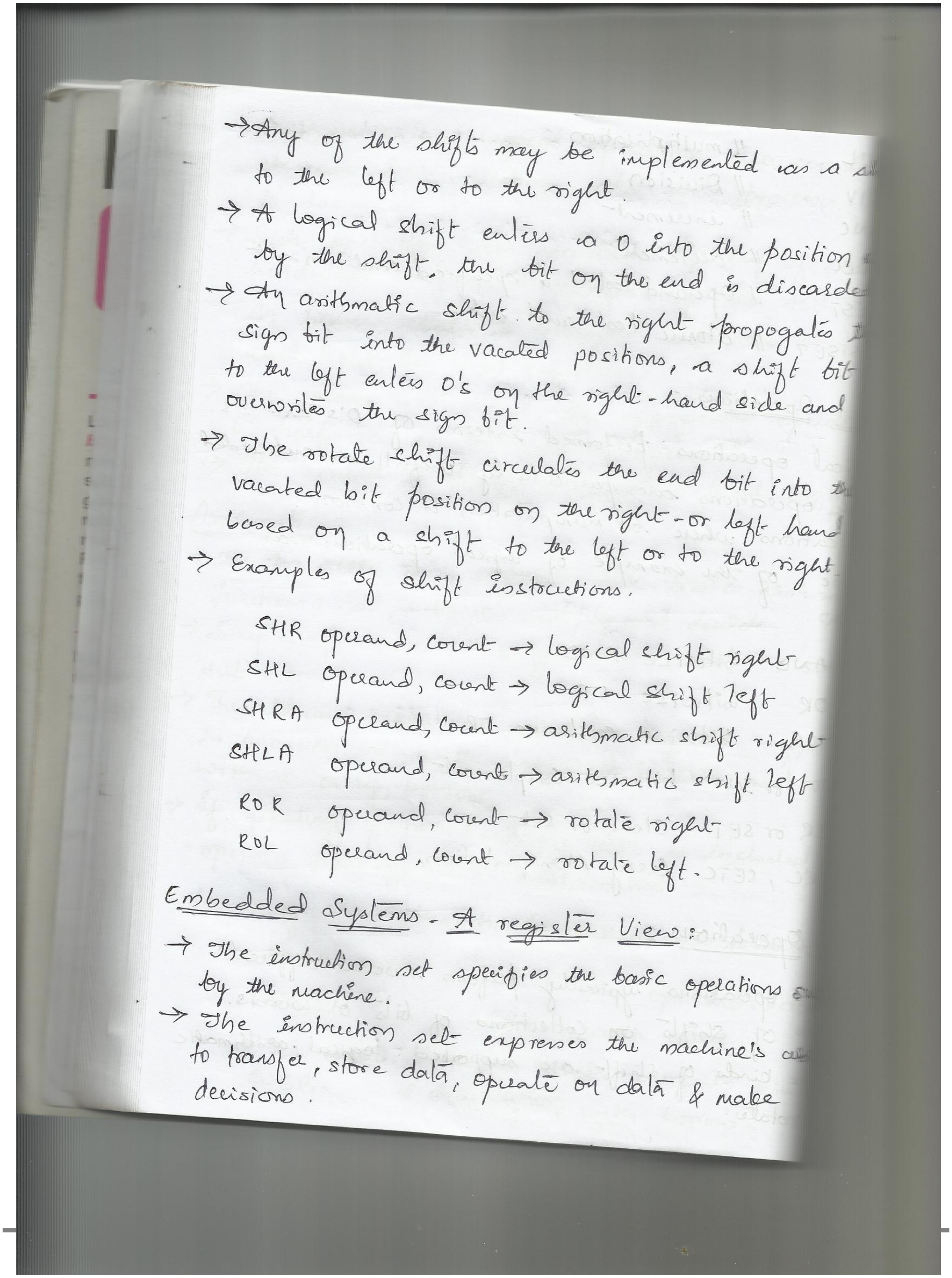
→ Some of the example of logical operations as below.

| | |
|---|---|
| AND | bitwise AND |
| OR | bitwise OR |
| XOR | bitwise exclusive OR |
| NOT or INV | bitwise Complement |
| CLR or SET | Clear or set |
| CLRC, SETC | Carry Manipulation. |

## Shift Operations :

→ Shift operations typically perform several different kinds of shifts on collections of bits or words.

→ Three kinds of shift are supported: logical, arithmatic & rotate.



→ Any of the shifts may be implemented as a
to the left or to the right.

→ A logical shift enters a 0 into the position
by the shift. the bit on the end is discarded

→ An arithmatic shift to the right propogates
sign bit into the vacated positions, a shift bit
to the left enters 0's on the right-hand side and
overwrites the sign bit.

→ The rotate shift circulates the end bit into
vacated bit position on the right-or left hand
based on a shift to the left or to the right

→ Examples of shift instructions.

SHR operand, count → logical shift right
SHL operand, count → logical shift left
SHRA operand, count → arithmatic shift right
SHLA operand, count → arithmatic shift left
ROR operand, count → rotate right
ROL operand, count → rotate left.

Embedded Systems - A register View:

→ The instruction set specifies the basic operations
by the machine.

→ The instruction set expresses the machine's
to transfer, store data, operate on data & make
decisions.

→ Underlying the instructions set is the physical h/w necessary to implement the operations directed by the instructions.

→ The core hardware comprises a control unit and a data path as shown in fig

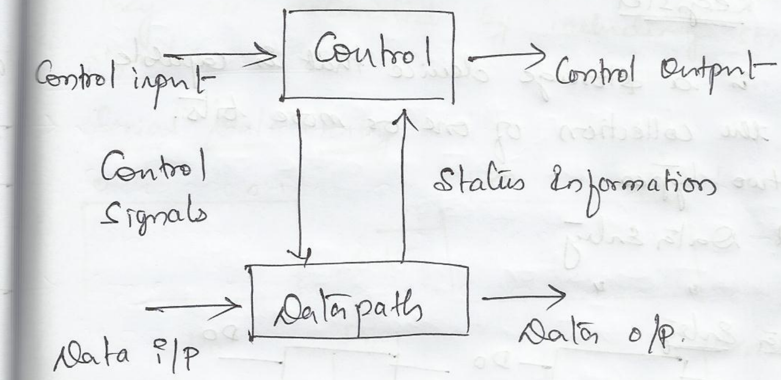


Fig:- A Control and datapath Block diagram.

→ The data path is a collection of registers and associated set of microoperations on the data held in the registers.

→ The Control unit directs the ordered execution of the microoperations so as to effect the desired transformations of the data.

→ The s/m hence can be expressed by the movement of the data among those registers, by operations and transformations & by the management of how such movements and operation take place.

→ Here we are going to use the set of operat
at the register level or Register transfer level

→ We will find that working initially at the
level is natural and convenient.

## The Basic Register

→ A register is a storage device that is capab
of holding the collection of one or more bits.

→ There are two types

1) Parallel Data Entry

2) Serial " "

1) Parallel Data Entry

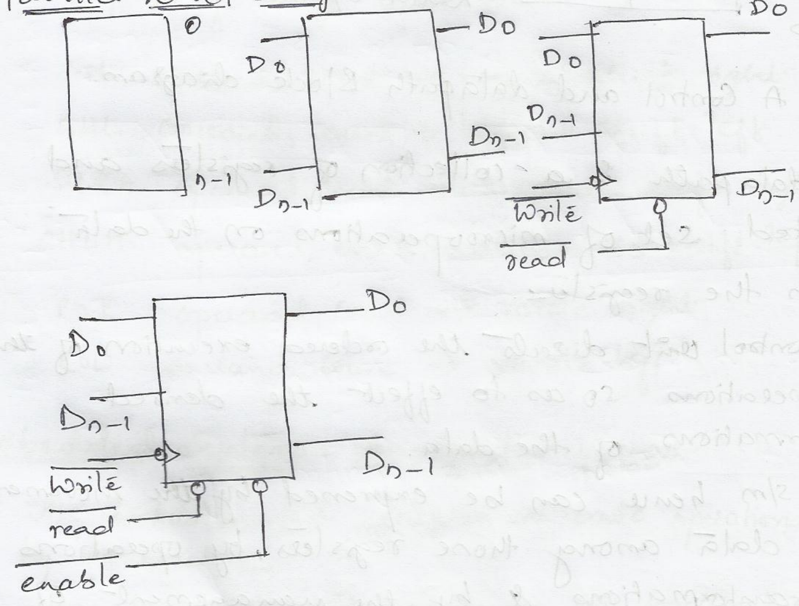


Fig: The registers at several levels of Abstracti
Parallel Data Entry.

→ By parallel data entry we are going to give input simultaneously.

→ And we will get output also simultaneously.

→ The figure shows a simple box with the bits numbered to reflect the size and outputs of the register.

→ They are elaborated by including some control signals.

2) Serial Data Entry

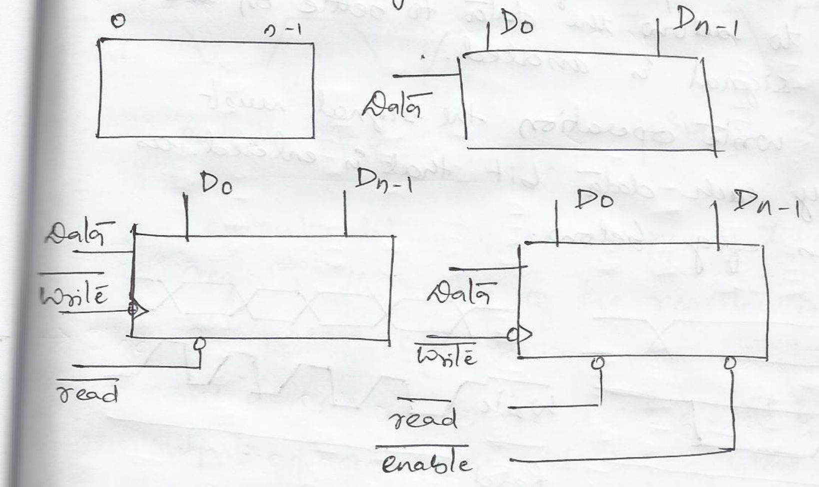


Fig: A register at several levels of Abstraction Serial Data Entry.

## Register Operations

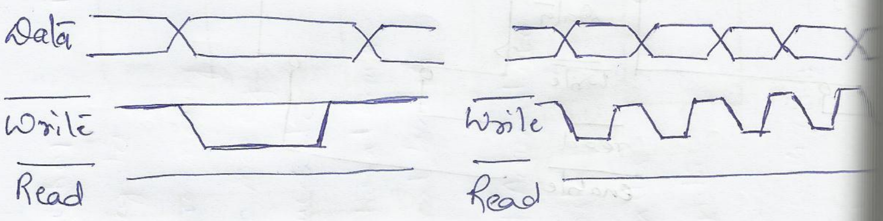
→ Registers support two basic operations.
- 1> Read
- 2> Write

### Write to a register

→ A parallel write operation begins when the data placed onto the inputs of the registers.

→ A delay to allow the data to settle on the the write signal is asserted.

→ A serial write operation the signal must accompany each data bit that is entered. shown in fig below.



parallel write

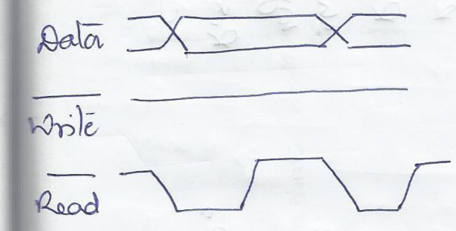
Serial write

Fig: Writing to a Registers

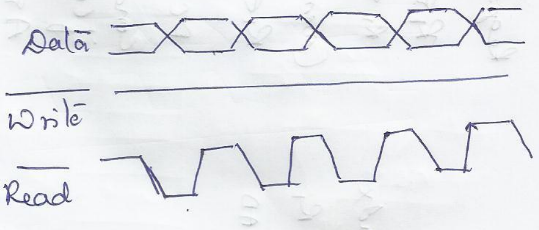
## Read from a register

→ The read operation is excuted as shown is fig below.

→ The read signal is issued; following some delay the data appears on the register output.



parallel Read



Serial Read

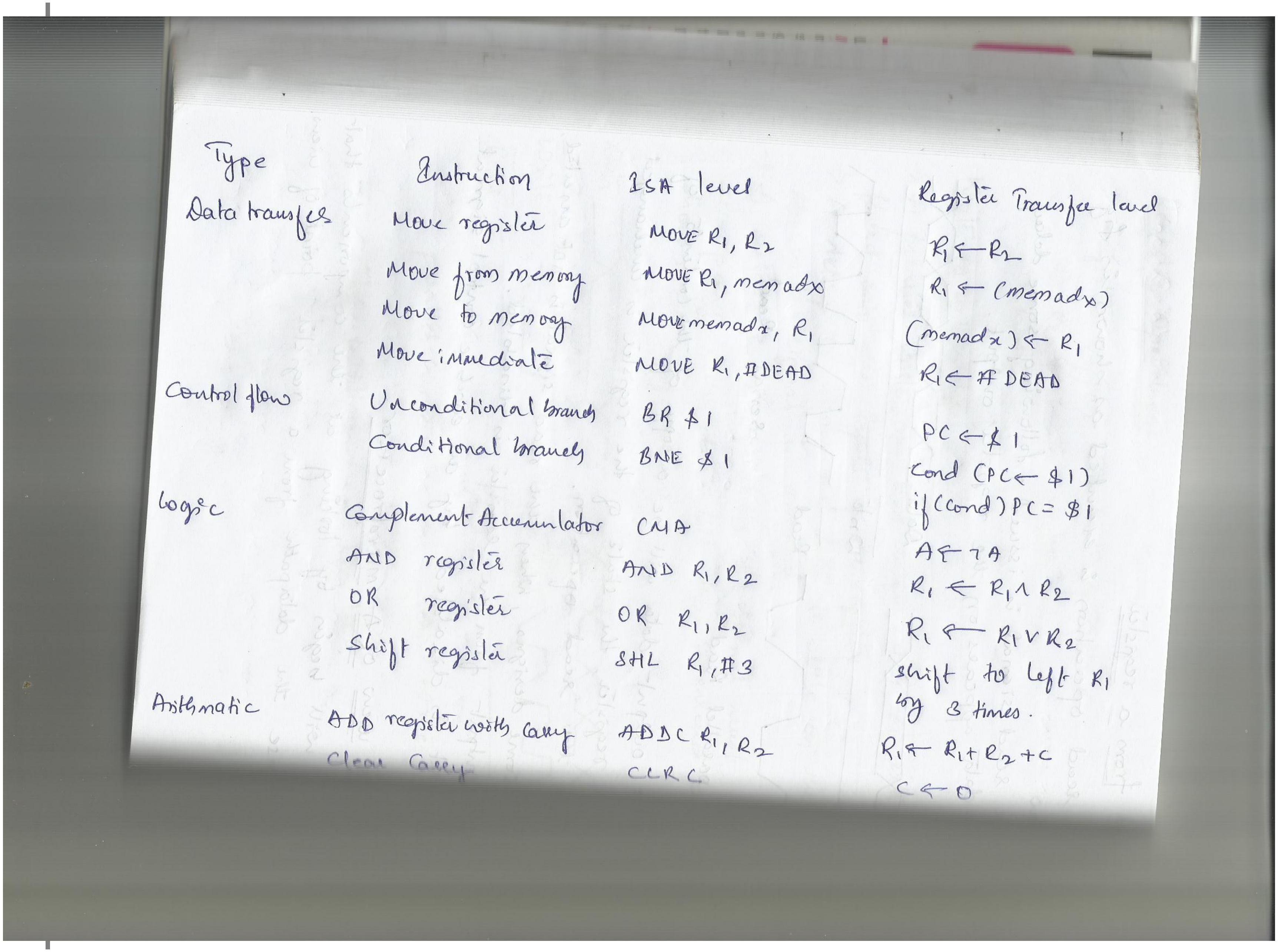
→ The output data will be a copy of the contents of the register, the state of the register is unchanged by the read operation.

→ In some designs, when the read signal is not asserted, the output from the register is disabled.

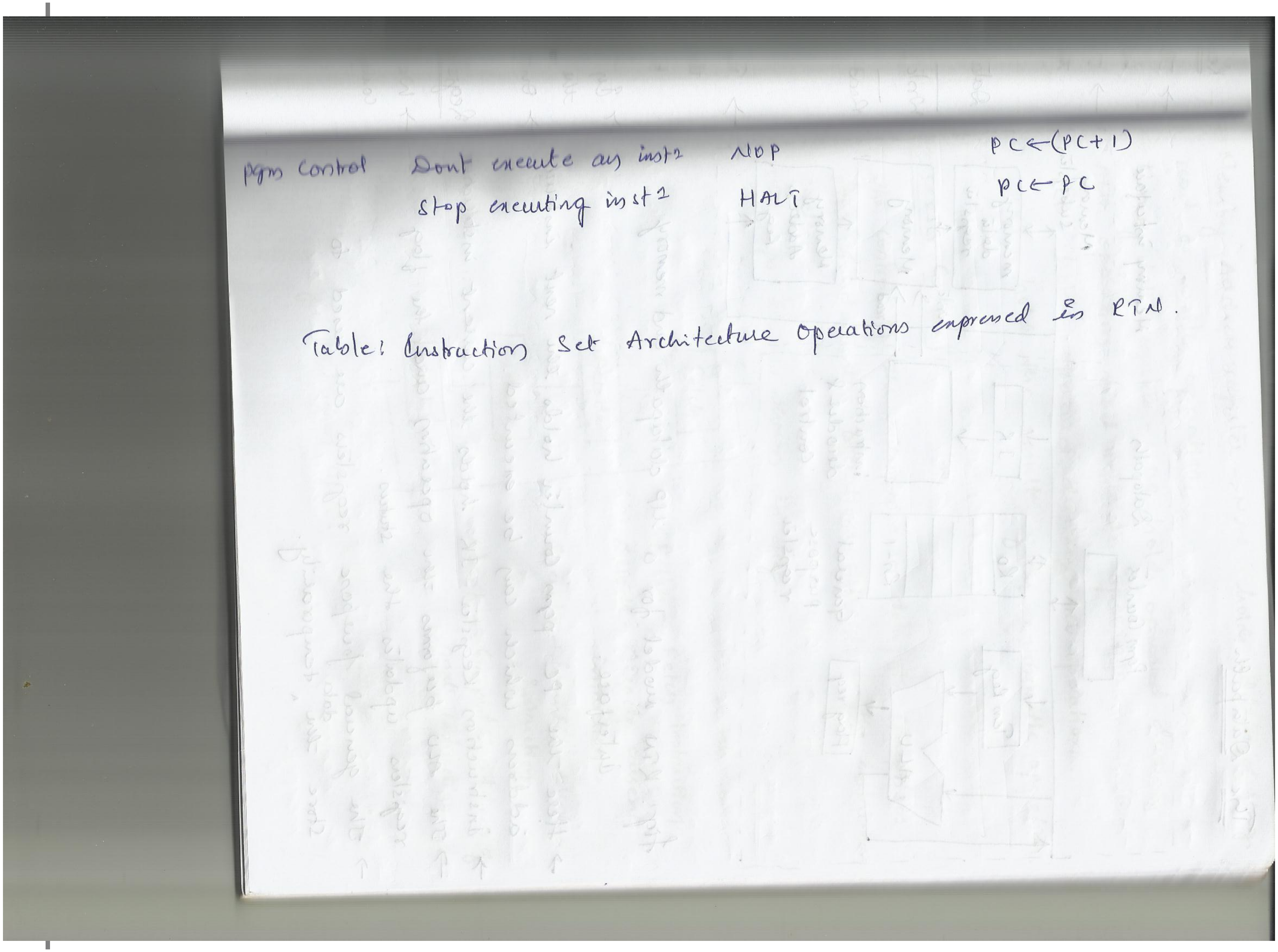
→ Output is disabled by an enable control signal.

## Register view of A microprocessor.

→ We will begin by looking at the components that comprise the datapath from a register point of view.

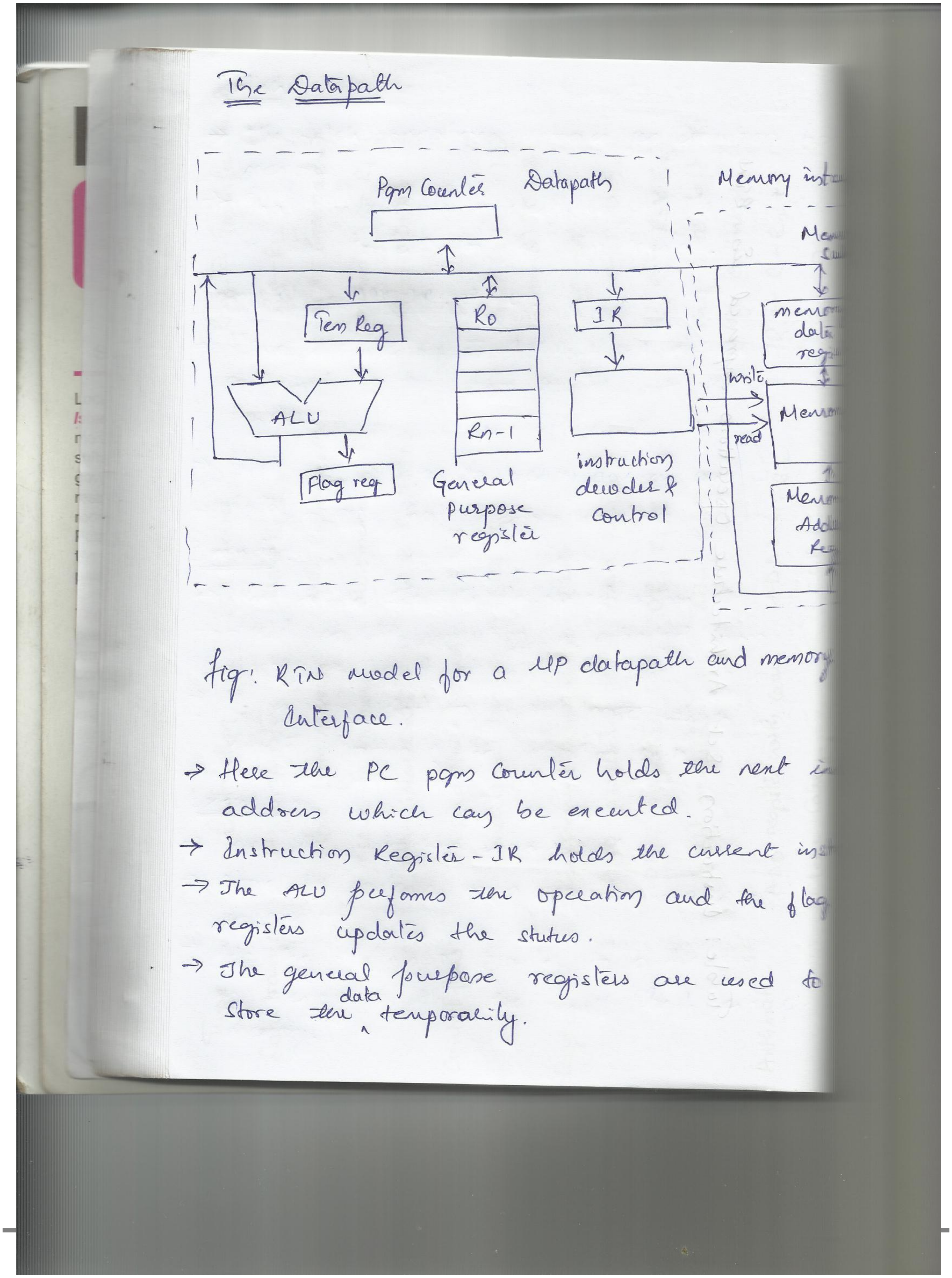


| Type | Instruction | ISA level | Register Transfer level |
|---|---|---|---|
| Data transfer | Move register | MOVE $R_1, R_2$ | $R_1 \leftarrow R_2$ |
| | Move from memory | MOVE $R_1$, memadx | $R_1 \leftarrow (memadx)$ |
| | Move to memory | MOVE memadx, $R_1$ | $(memadx) \leftarrow R_1$ |
| | Move immediate | MOVE $R_1$, #DEAD | $R_1 \leftarrow$ # DEAD |
| Control flow | Unconditional branch | BR $1 | PC ← $1 |
| | Conditional branch | BNE $1 | Cond (PC ← $1)<br>if(cond) PC = $1 |
| Logic | Complement Accumulator | CMA | $A \leftarrow \neg A$ |
| | AND register | AND $R_1, R_2$ | $R_1 \leftarrow R_1 \wedge R_2$ |
| | OR register | OR $R_1, R_2$ | $R_1 \leftarrow R_1 \vee R_2$ |
| | Shift register | SHL $R_1$, #3 | shift to left $R_1$ by 3 times. |
| Arithmatic | ADD register with carry | ADDC $R_1, R_2$ | $R_1 \leftarrow R_1 + R_2 + C$ |
| | Clear Carry | CLR C | $C \leftarrow 0$ |



| | | | |
|---|---|---|---|
| Pgm Control | Dont execute any instn | NOP | $PC \leftarrow (PC+1)$ |
| | stop executing instn | HALT | $PC \leftarrow PC$ |

Table: Instruction Set Architecture Operations expressed in RTN.



## The Datapath

Pgm Counter
Datapath
Memory int
Mem
s
Tem Reg
R0
IR
ALU
Rn-1
Flag reg
General purpose register
instruction decoder & Control
memory data reg
write
read
Memory
Memory Add Re

fig: RTN model for a μP datapath and memory interface.

→ Here the PC pgm Counter holds the next i address which can be executed.

→ Instruction Register - IR holds the current ins

→ The ALU performs the operation and the flag registers updates the status.

→ The general purpose registers are used to store the data temporarily.

→ Memory Address register → MAR holds address during read or write operation.

→ Memory data registers - MDR holds ~~operand~~ during ~~ALU operation~~. read or write operation.

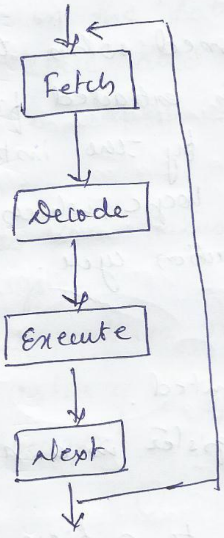
→ TRO holds operand during ALU operation.

→ TRI hold the result of an Arithmatic operation.

## Processor Control

→ The control of the microprocessor data path comprises four fundamental operations defined as the instruction cycle.

→ The steps are as shown in fig.



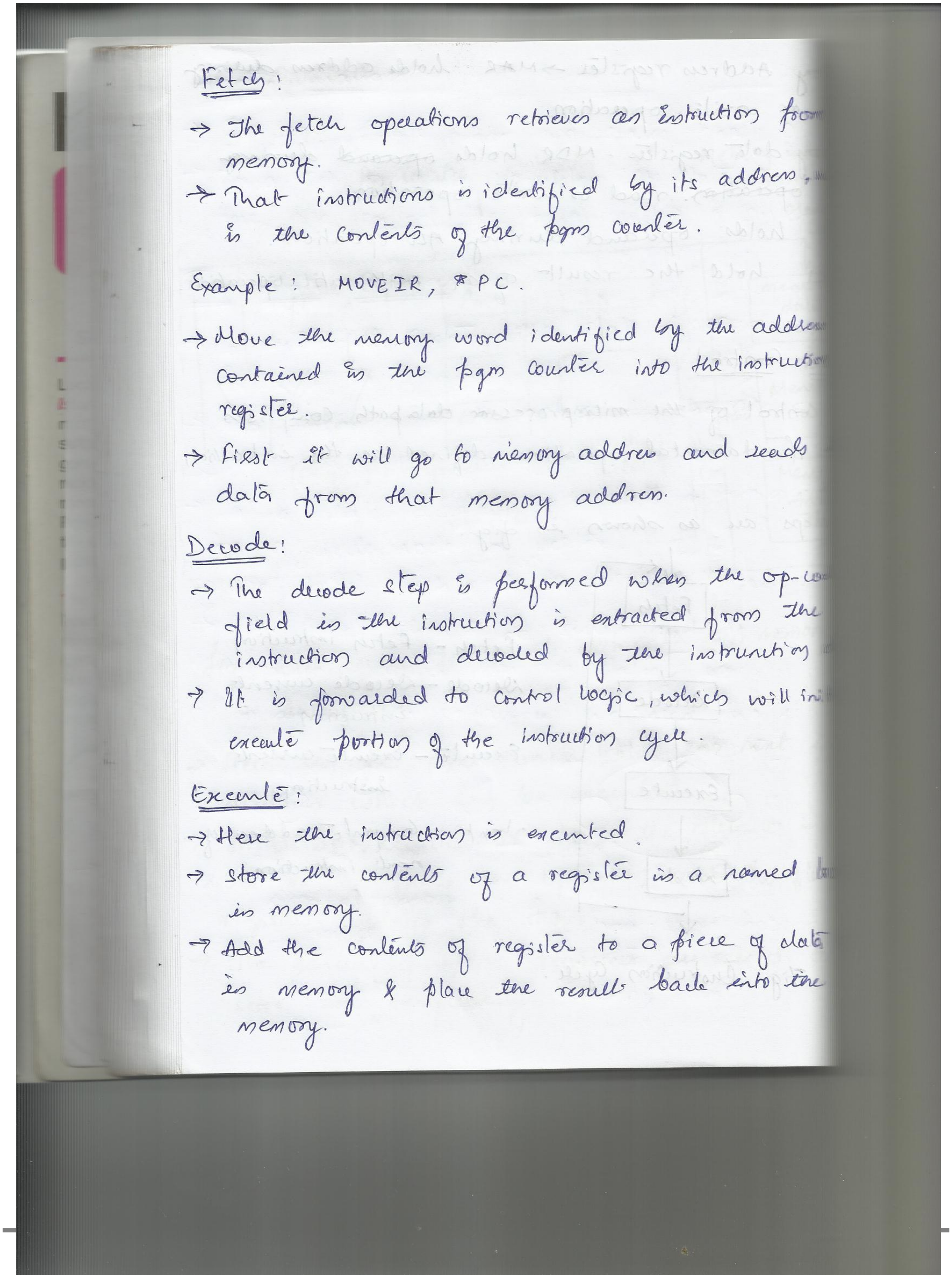
Fetch - Fetch instruction

Decode - Decode current instruction

Execute - Execute current instruction

Next - Compute address of next instruction.

Fig: Instruction Cycle.



Fetch:

→ The fetch operations retrieves an instruction from memory.

→ That instructions is identified by its address, w is the contents of the pgm counter.

Example: MOVE IR, *PC.

→ Move the memory word identified by the address contained in the pgm counter into the instructio register.

→ First it will go to memory address and reads data from that memory address.

Decode:

→ The decode step is performed when the op-co field in the instruction is extracted from the instruction and decoded by the instruction

→ It is forwarded to control logic, which will ini execute portion of the instruction cycle.

Execute:

→ Here the instruction is executed.

→ Store the contents of a register in a named lo in memory.

→ Add the contents of register to a piece of data in memory & place the result back into the memory.

Next:

→ It will fetch the next instruction after the execution of the previous instrunction.

## The concept of State and Time.

Time:

→ A combinational logic system has no notion of time or history.

→ Here neglecting the delays through the system, we find that the output is immediate and a direct function of the current input set.

→ The current output of a finite-state system depends both on the path the system took to reach the current state and potentially, the present values of the input set.

State:

→ State defines the status of an variables which present in the system.

→ Each set of values represents a unique state.

→ When the value of any variable changes, the state of s/m changes.

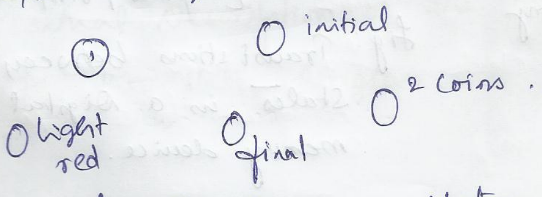
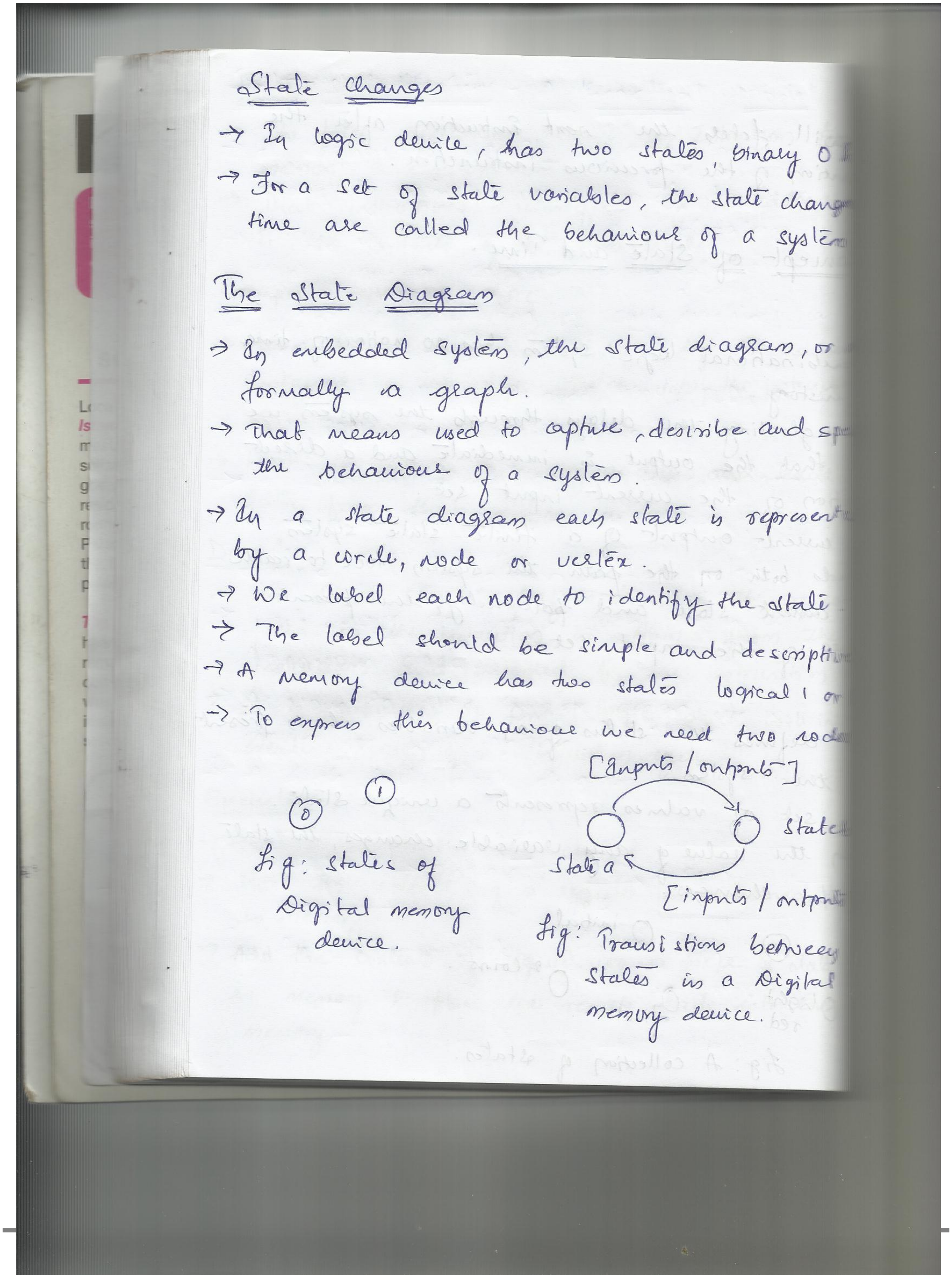


fig: A collection of states.



## State changes

→ In logic device, has two states, binary 0

→ For a set of state variables, the state chang time are called the behaviour of a syste

## The State Diagram

→ In embedded system, the state diagram, or formally a graph.

→ That means used to capture, describe and sp the behaviour of a system.

→ In a state diagram each state is represent by a circle, node or vertex.

→ We label each node to identify the state

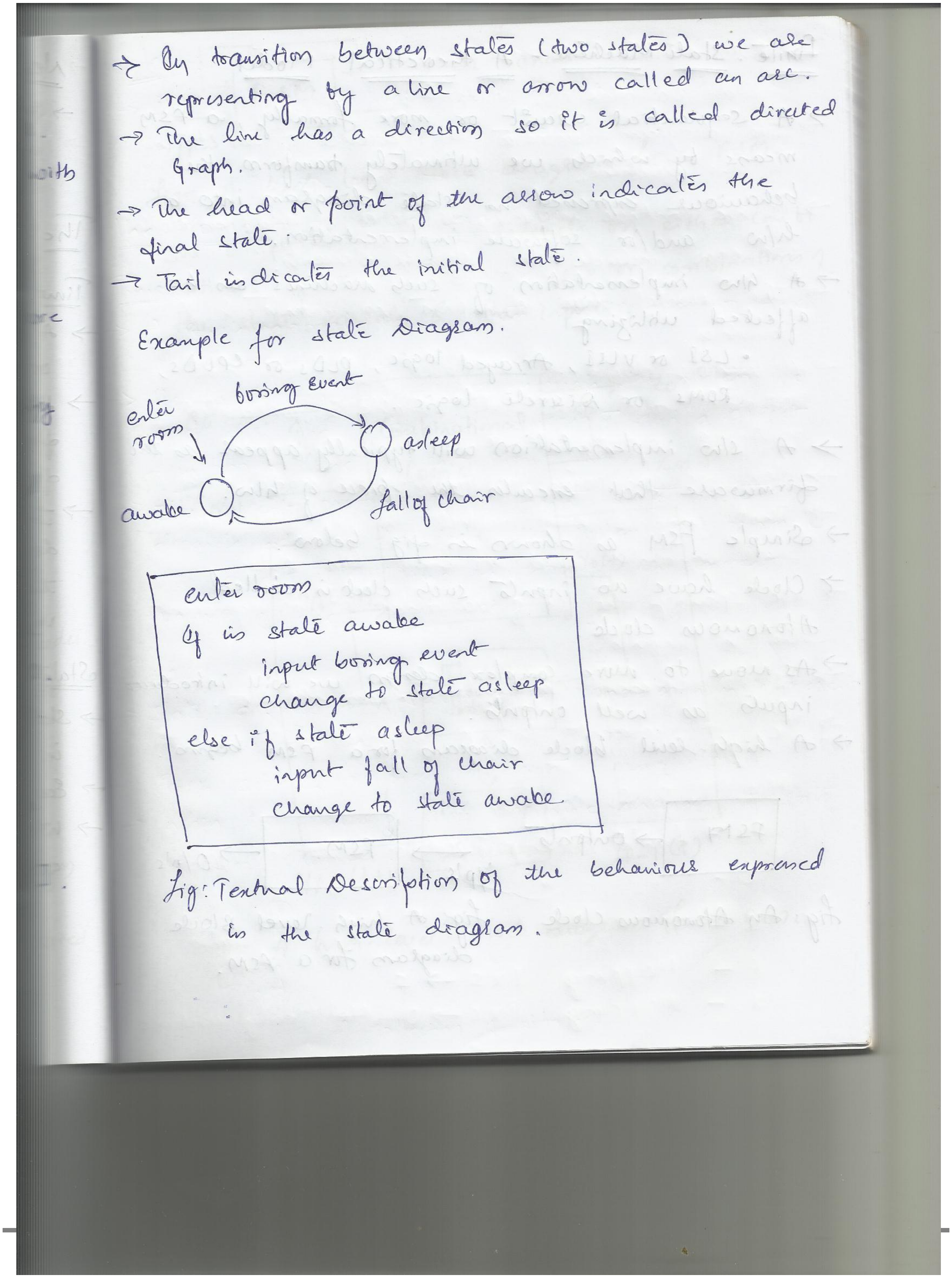
→ The label should be simple and descripti

→ A memory device has two states logical 1 or

→ To express this behaviour we need two nod

fig: states of Digital memory device.

fig: Transitions between states in a Digital memory device.



→ By transition between states (two states) we are representing by a line or arrow called an arc.

→ The line has a direction so it is called directed Graph.

→ The head or point of the arrow indicates the final state.

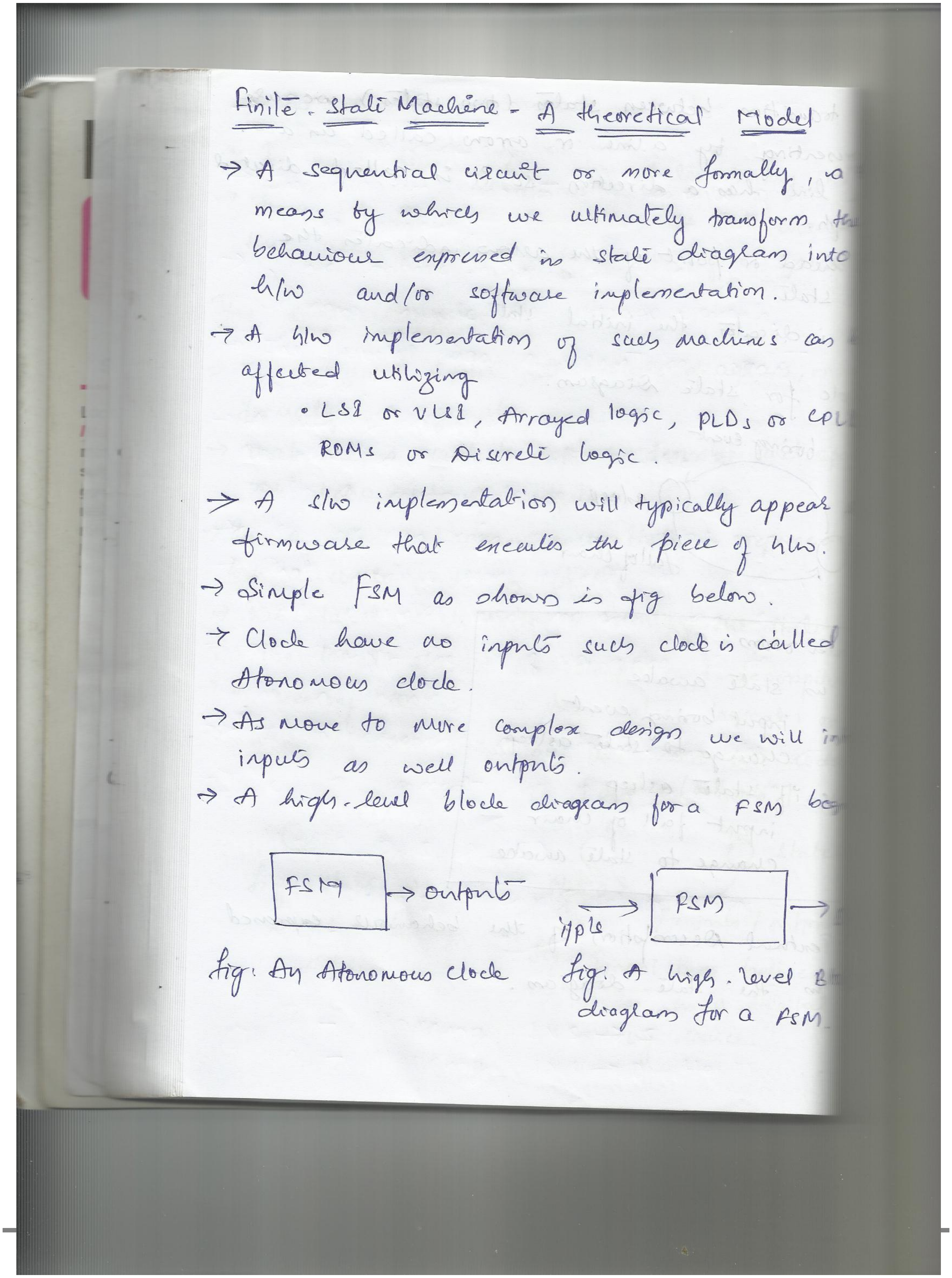
→ Tail indicates the initial state.

Example for state Diagram.

enter room
boring event
asleep
awake
fall of chair

```
enter room
If is state awake
        input boring event
        change to state asleep
else if state asleep
        input fall of chair
        change to state awake
```

fig: Textual Description of the behaviours expressed in the state diagram.



Finite . Stali Machine - A theoretical Model

→ A sequential circuit or more formally, a means by which we ultimately transform th behaviour expressed in stali diagram into h/w and/or software implementation.

→ A h/w implementation of such machines can affected utilizing

- LSI or VLSI, Arrayed logic, PLDs or CPL ROMs or Discrete logic.

→ A s/w implementation will typically appear firmware that executes the piece of h/w.

→ Simple FSM as shown in fig below.

→ Clock have no inputs such clock is called Atonomous clock.

→ As move to more complex design we will in inputs as well outputs.

→ A high-level block diagram for a FSM be

FSM → outputs

i/p's → FSM →

fig: An Atonomous clock

fig: A high-level B diagram for a FSM

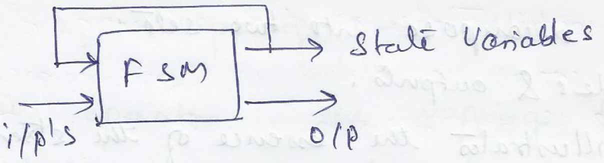


Fig: A high-level Block diagram for a FSM.

→ The outputs may be depends on either combi-nations of the state variables or combinations of the state varials & the inputs.

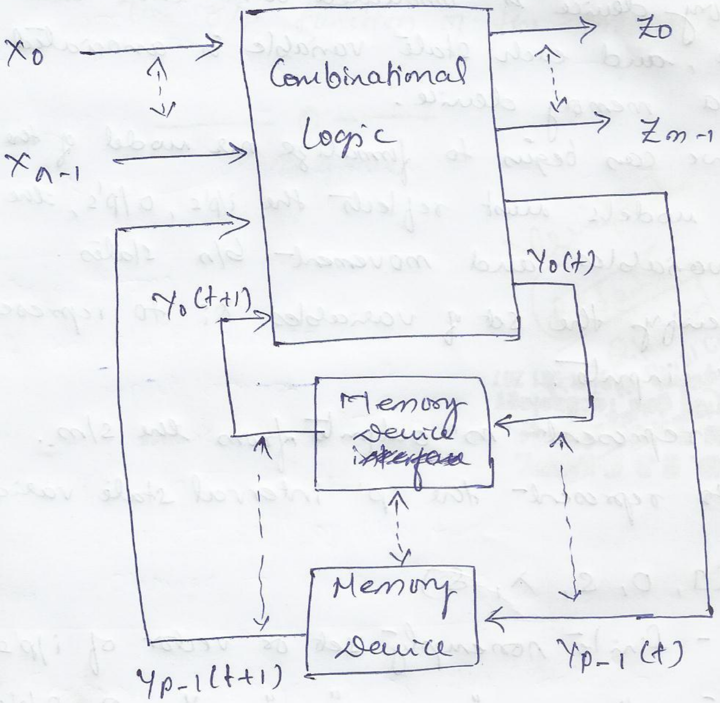
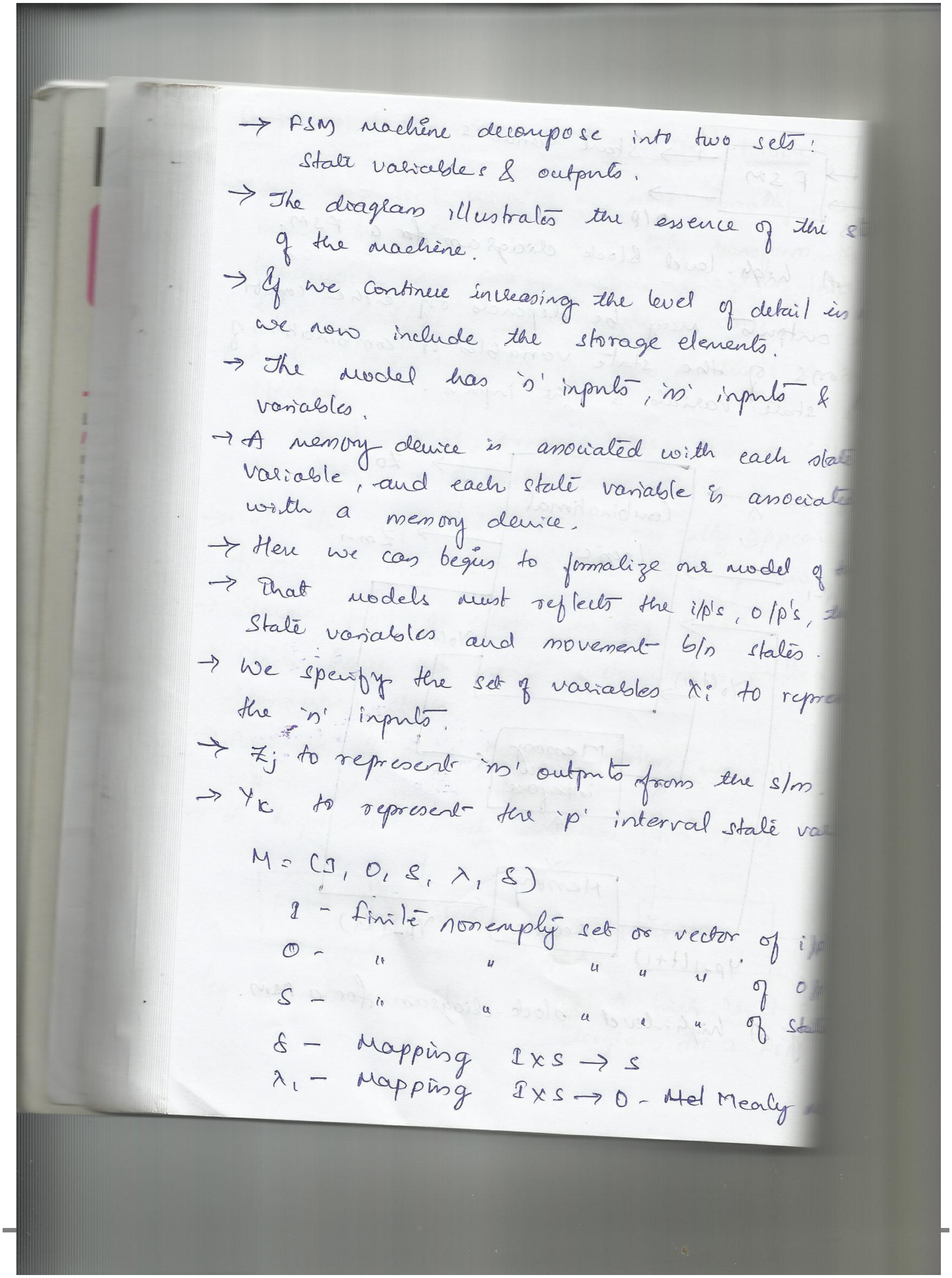


Fig. A high-level block diagram for a FSM.



→ FSM machine decompose into two sets:
State variables & outputs.

→ The diagram illustrates the essence of the s
of the machine.

→ If we continue increasing the level of detail in
we now include the storage elements.

→ The model has 'n' inputs, 'm' inputs &
variables.

→ A memory device is associated with each stat
Variable, and each state variable is associat
with a memory device.

→ Here we can begin to formalize our model of

→ That models must reflects the i/p's, o/p's,
State variables and movement b/n states.

→ We specify the set of variables $X_i$ to repr
the 'n' inputs.

→ $Z_j$ to represent 'm' outputs from the s/m

→ $Y_k$ to represent the 'p' interval state va

$M = (I, O, S, \lambda, \delta)$

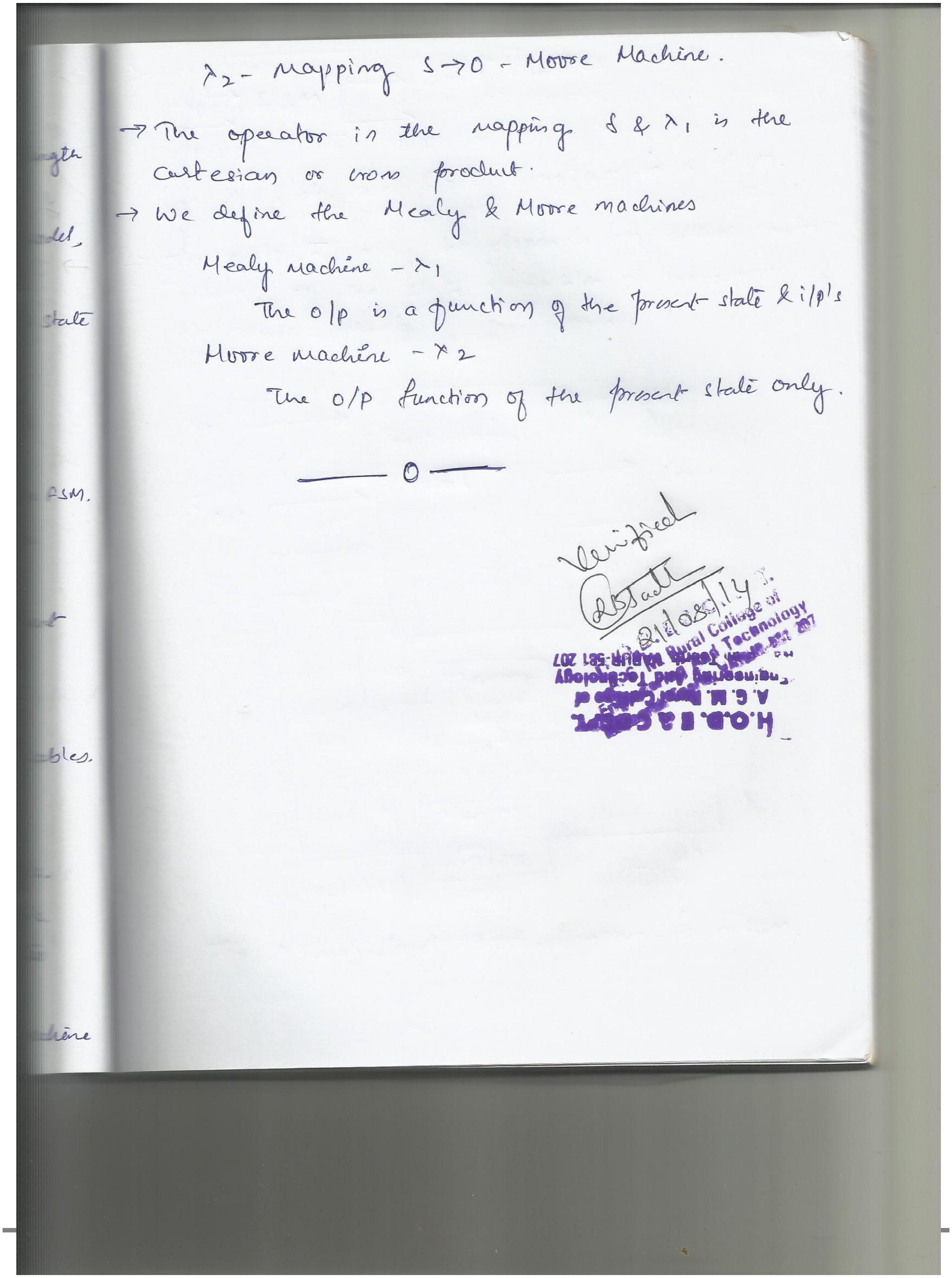
I - finite nonempty set or vector of i/

O - " " " " " of o

S - " " " " " of st

$\delta$ - Mapping $I \times S \rightarrow S$

$\lambda_1$ - Mapping $I \times S \rightarrow O$ - Mel Mealy



$\lambda_2$- mapping $S \to O$ - Moore Machine.

→ The operator in the mapping $\delta$ & $\lambda_1$ is the cartesian or cross product.

→ We define the Mealy & Moore machines

Mealy machine - $\lambda_1$

The o/p is a function of the present state & i/p's

Moore machine - $\lambda_2$

The o/p function of the present state only.

— 0 —

Verified

21/08/14

H.O.D. E & C Dept.
A.G.M. Rural College of
Engineering and Technology
Varur-581 207