Chapter 1
Introduction

1.1 Embedded System

Embedded system is an optimized computing system consisting of the mechanical or electrical system, often with real time computing constraints designed for performing some dedicated functions.

The common characteristics of embedded system are as follows:-

1) Single Functioned

An embedded system is designed to perform specific tasks repeatedly. Newer program version update can be done in some rare cases.

2) Tightly Constrained

An embedded system have tightly constrained design metrics such as low cost, portable size, high perfomance, limited power consumption and so on.

3) Reactive and real time

An embedded system should continuously react to changes in the system's environment and should provides output in a real time within a specified time frame. Delayed computation may result in system failure.

For eg: digital camera.

→ It is designed to capture image or videos repeatedly.
- It is cheap, single chip system, small in size and low power system.
- It is reactive to some extent i.e. auto flash mode, smile mode, etc.

2.2 Classification based on generation

1) First Generation Embedded System

First generation ES are designed with 8 bit microprocessor or 4 bit microcontroller. They have very simple hardware circuits. The firmwares are designed in assembly language. For eg: digital telephone keypad, stepper motor control, etc.

2) Second Generation Embedded System

Second generation ES are designed with 16 bit μp or 8 bit μc. They have complex hardware configuration than first generation. They may contain embedded OS. Eg: DAS.

3) Third Generation Embedded System

Third generation ES are designed with 32 bit μp or 16 bit μc. They consist of application specific IC. The concept of DSP emerges. Eg: Robotics.

4) Fourth Generation Embedded System

Fourth generation ES are designed with 64 bit μp or 32 bit μc. They consist of system on chip and multi core processors with tight integration and miniaturization. Eg: smart phones.

1.3 Purpose of Embedded System

1) Data collection, storage and representation
2) Data communication
3) Data processing
4) Monitoring
5) Control
6) Application specific UI

1.4 Applications of ES

1) Consumer Electronics (Camera, radio)
2) Home Appliances (TV, DVD player, washing machine)
3) Home Automation and security (CCTV, burglar alarm, fire alarm)
4) Industry Automation (Engine control)
5) Telecommunication (cell phones)
6) Networking (Routers, switch)
7) Computer peripherals (printer, fax)
8) Health science (ECG machine, disease identification system)
9) Biometrics (Finger print recognition, face detection)
10) Banking (ATM, currency counter)

1.5 Embedded System Design Metrics

1) Unit Cost (monetary cost of manufacturing each copy of system)
2) NRE Cost (one time monetary cost of designing system)
3) Size (Physical space in bytes or gates required)
4) Performance (Execution time of system)
5) Power (Amount of power consumed by the system)

6) Flexibility (ability to change functionality without incurring NRE cost)
7) Time to prototype (time needed to build working version)
8) Time to market (time required to develop a system upto release)
9) Maintainability (ability to modify system after initial release)
10) Correctness (correct functioning by the system)
11) Safety (Probability that system will not cause harm)

→ Design metrics are the measures of an implementational features of a system

1.6 Embedded system Vs. General Purpose Computing System

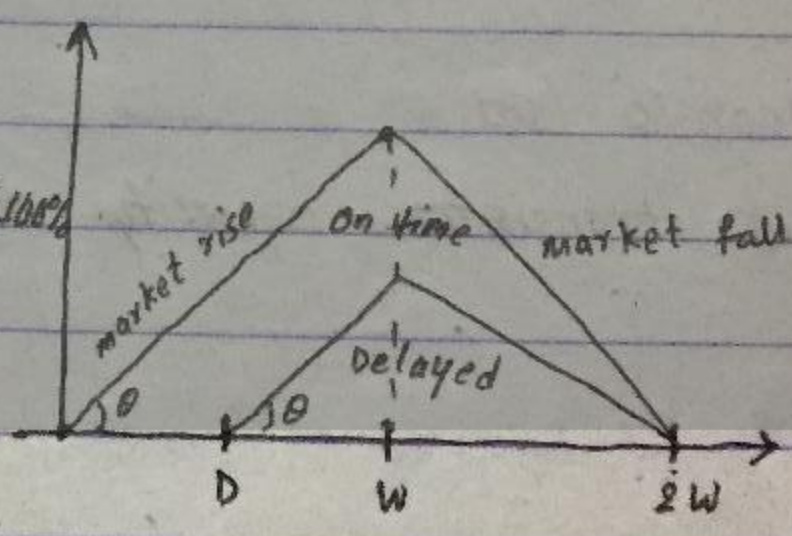
1) ES consists of specific hardware, embedded OS.
   GPCS consists of generic hardware, GPOS.
2) ES is designed to perform specific set of functions.
   GPCS is designed to perform various functions.
3) OS in ES is mandatory but in GPCS it is compulsory.
4) Performance is key factor of GPCS.
5) Maximum power saving requirements for ES.
6) ES is time critical.
7) ES is deterministic.
8) Programs in GPCS are alterable by end users.

Q) How can we meet best optimization of ES?

There are many design metrics that impacts the embedded system design. The key challenge is that improving one metric leads to worsening of another. So, to meet best optimization, the designer must be comfortable with various implementation technologies and must be able to migrate from one technology to another. This ensure best implemetatation for a given system.

Q) Simplified Revenue Model

→ It is used to investigate loss of revenue occurs due to delayed entry of a product in the market.
→ It assumes peak of market occurs at halfway point (W) of a product life even for delayed entry.
→ The revenue for an on-time market entry is area of Δ on time.
→ The revenue for delayed entry is area of Δ delayed.



∴ Revenue loss = A(On time) − A(delayed)

∴ Percentage revenue loss = {A(on time) − A(delayed)} / on time * 100%

Let market rise angle be θ.

So,

$A(\text{on time}) = 2(\frac{1}{2} b * h) = 2(\frac{1}{2} \times W \times W \tan\theta) = W^2 \tan\theta$

$A(\text{delayed}) = \frac{1}{2} \times (2W - D) \times ((W - D) \tan\theta)$

Now,

$\%\text{Percentage revenue loss} = [W^2 \tan\theta - W^2 \tan\theta + \frac{3}{2} WD \tan\theta - \frac{1}{2} D^2 \tan\theta] / W^2 \tan\theta * 100\%$

$= \left[\frac{1}{2W^2} (3WD - D^2)\right] \times 100\%$

a) NRE and Unit Cost

→ total cost = NRE cost + unit cost * no. of units

→ Pre product cost = $\frac{\text{Total cost}}{\text{no. of units}}$

a) The design of disk drive has NRE cost $100,000 and a unit cost $20. How much will we have to add to cost of each product to cover NRE cost assuming sell of 100 units?

Ans ⇒

Added cost = Amortized NRE cost = $\frac{\text{NRE cost}}{\text{no. of units}}$

$= \frac{100\,000}{100}$

= $1000

Ans.

8) Moore's law

IC transistor capacity doubles every 18 months.

Chapter 2

Hardware Design Issue

2.1 Processor

Processor is a digital circuit that performs computational tasks. The minimum requirement to be a processor is the presence of controller and datapath.

The different processor technologies are as follows:-

1) General Purpose Processor (GPP):

It is a programmable circuit that can perform varieties of tasks. It consists of program memory and general data path. The datapath has large register array and one or more general purpose ALU.

2) Single Purpose Processor (SPP):

It is a digital circuit designed to perform exactly one program. Digital camera is a SPP. It only consists of data memory but not program memory

3) Application Specific Processor (ASP)

It is a programmable circuit that is optimized for a particular class of applications. It consists of program memory, data memory, custom designed ALU and optimized datapath.

2.2 Custom Single Purpose Processor

→ Faster performance

→ Size is smaller

→ Lower power requirement

→ High NRE cost

→ Longer time-to-market

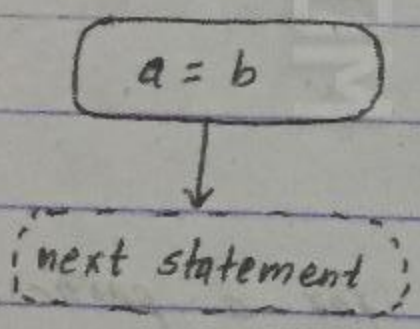
→ Less flexible

2.3 Designing custom single purpose processor

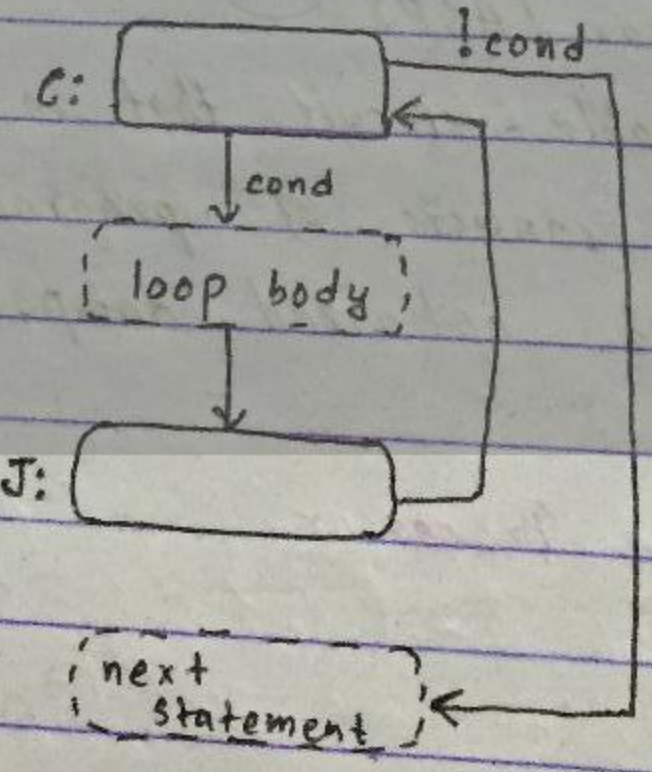
1) Develop an algorithm or function that computes the desired output.
2) Convert algorithm into a complex state diagram or FSMD (Finite State Machine with Data)
3) Divide functionality into datapath part and controller part. Datapath consists of interconnection of combinational and sequential components. Controller consists of pure FSM.
4) Complete controller design by implementing FSM with combinational logic.

#FSMD Templates

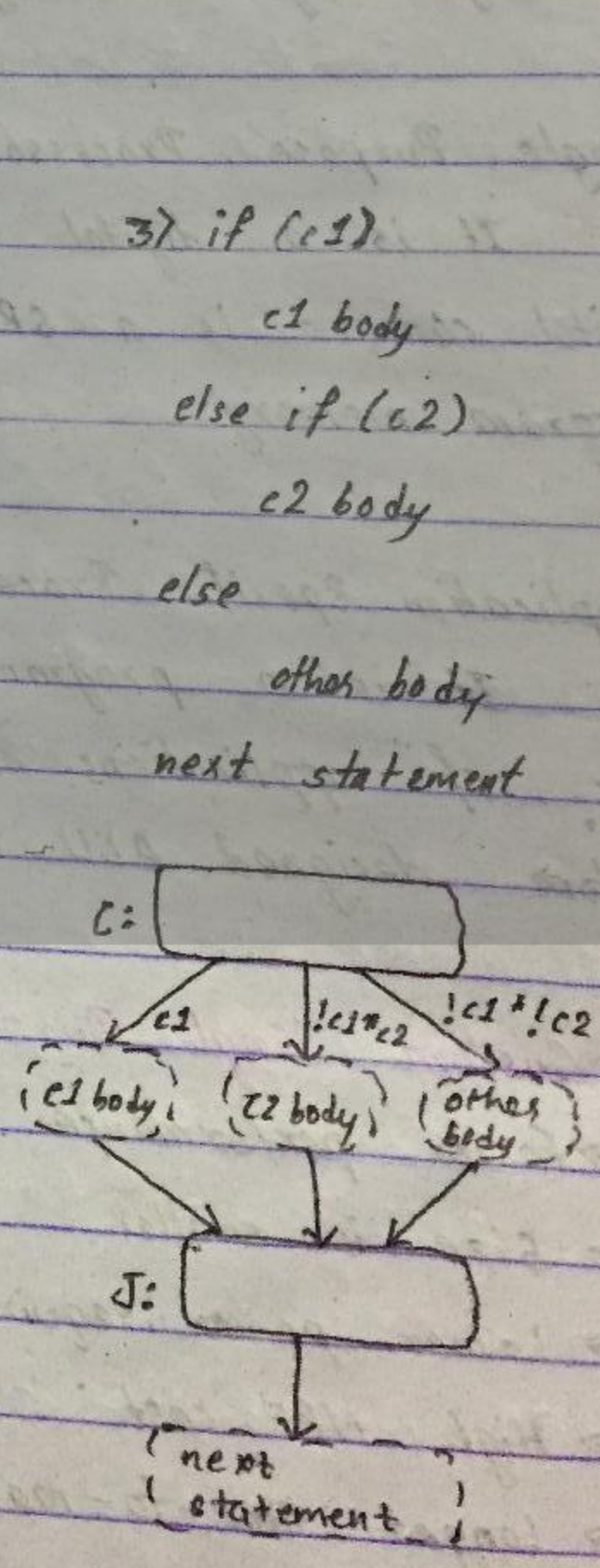
1) a = b
   next statement



2) while (cond) {
   loop body
   }
   next statement

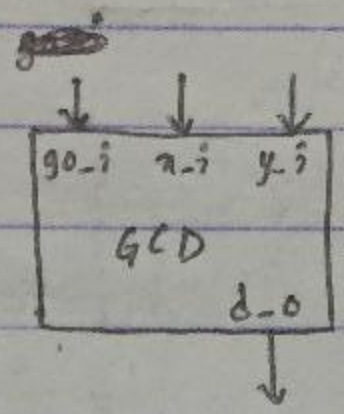


3) if (c1)
   c1 body
   else if (c2)
   c2 body
   else
   other body
   next statement



A) Greatest Common Divisor (GCD)

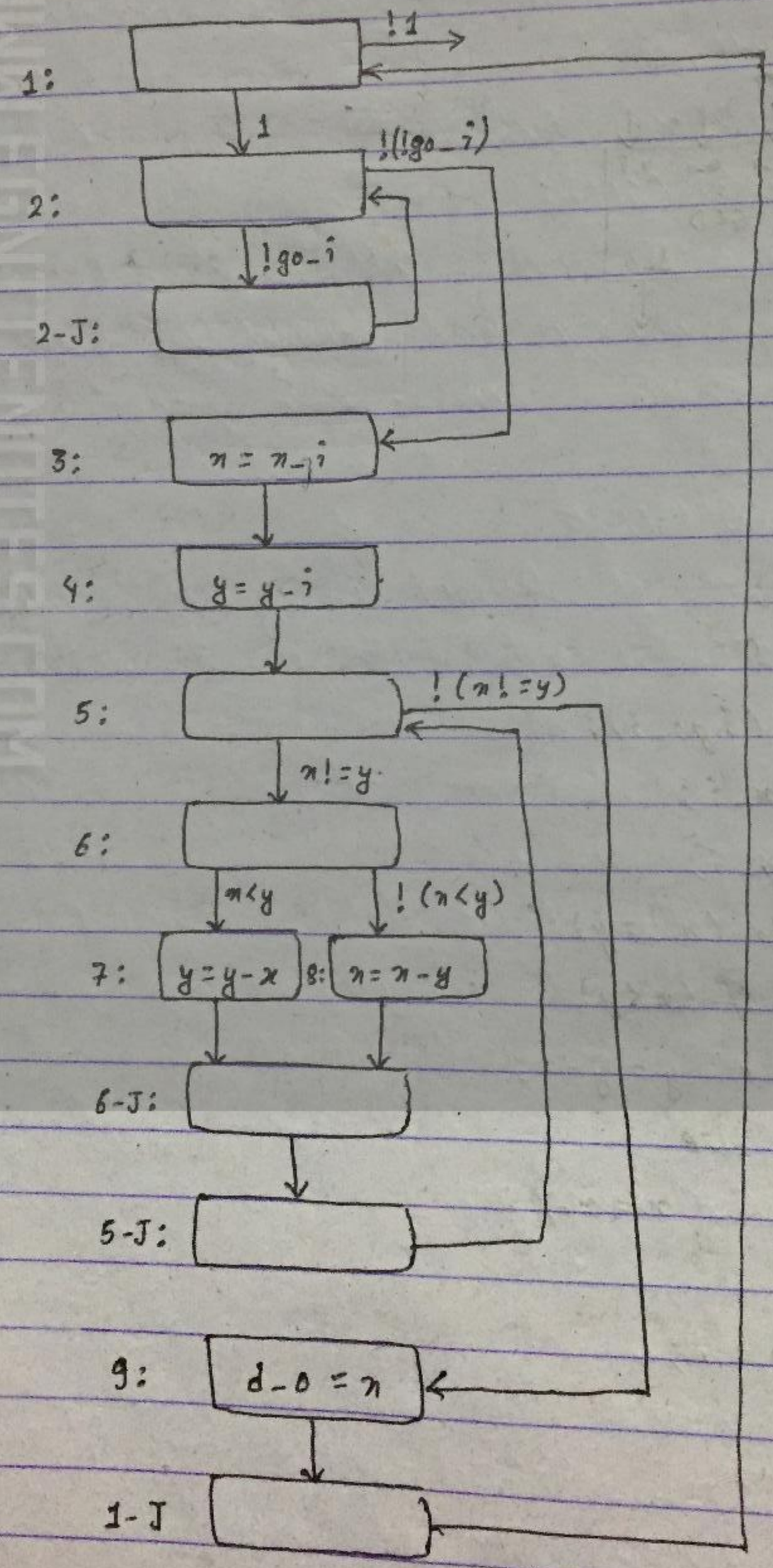
1) Black box diagram



2) Algorithm

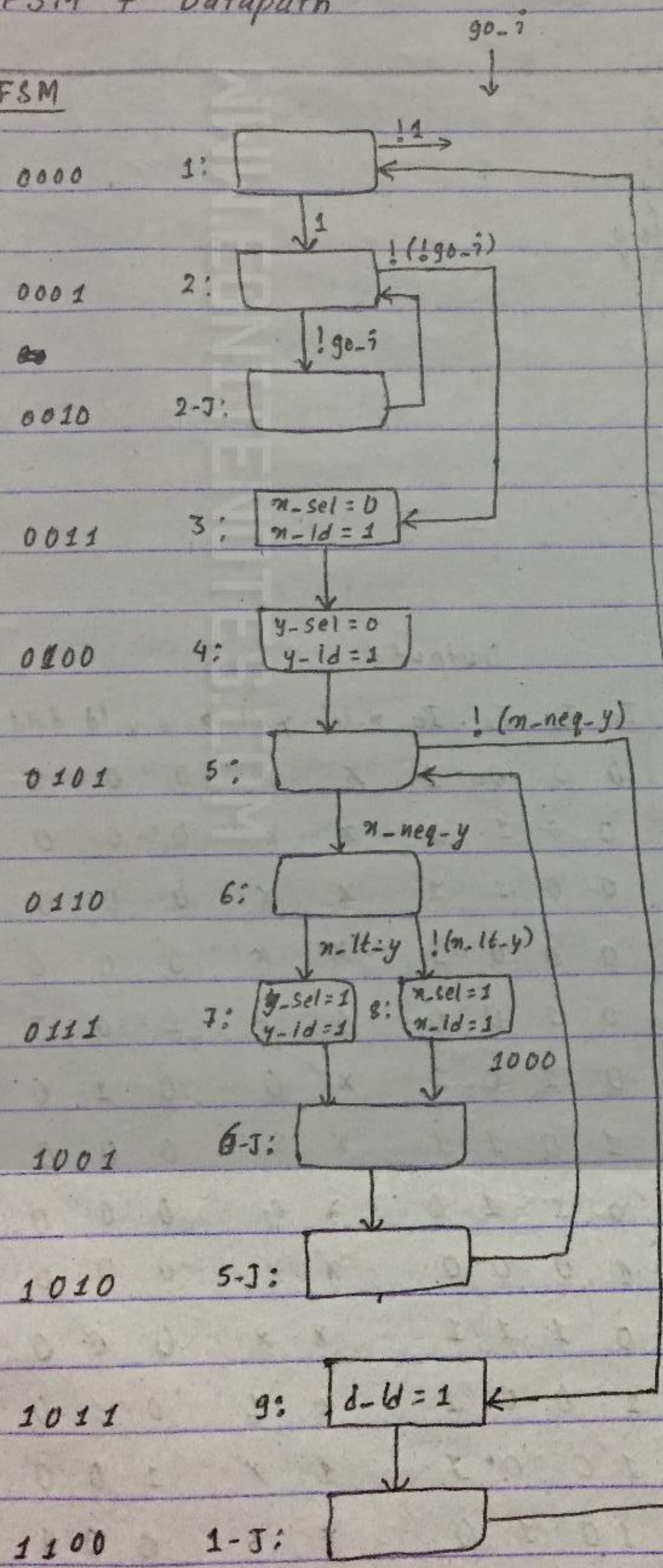
```
0: int x, y;
1: while (1) {
2:     while (!go_i);
3:     x = x_i;
4:     y = y_i;
5:     while (x != y) {
6:         if (x < y)
7:             y = y - x;
           else
8:             x = x - y;
       }
9:     d_o = x;
   }
```

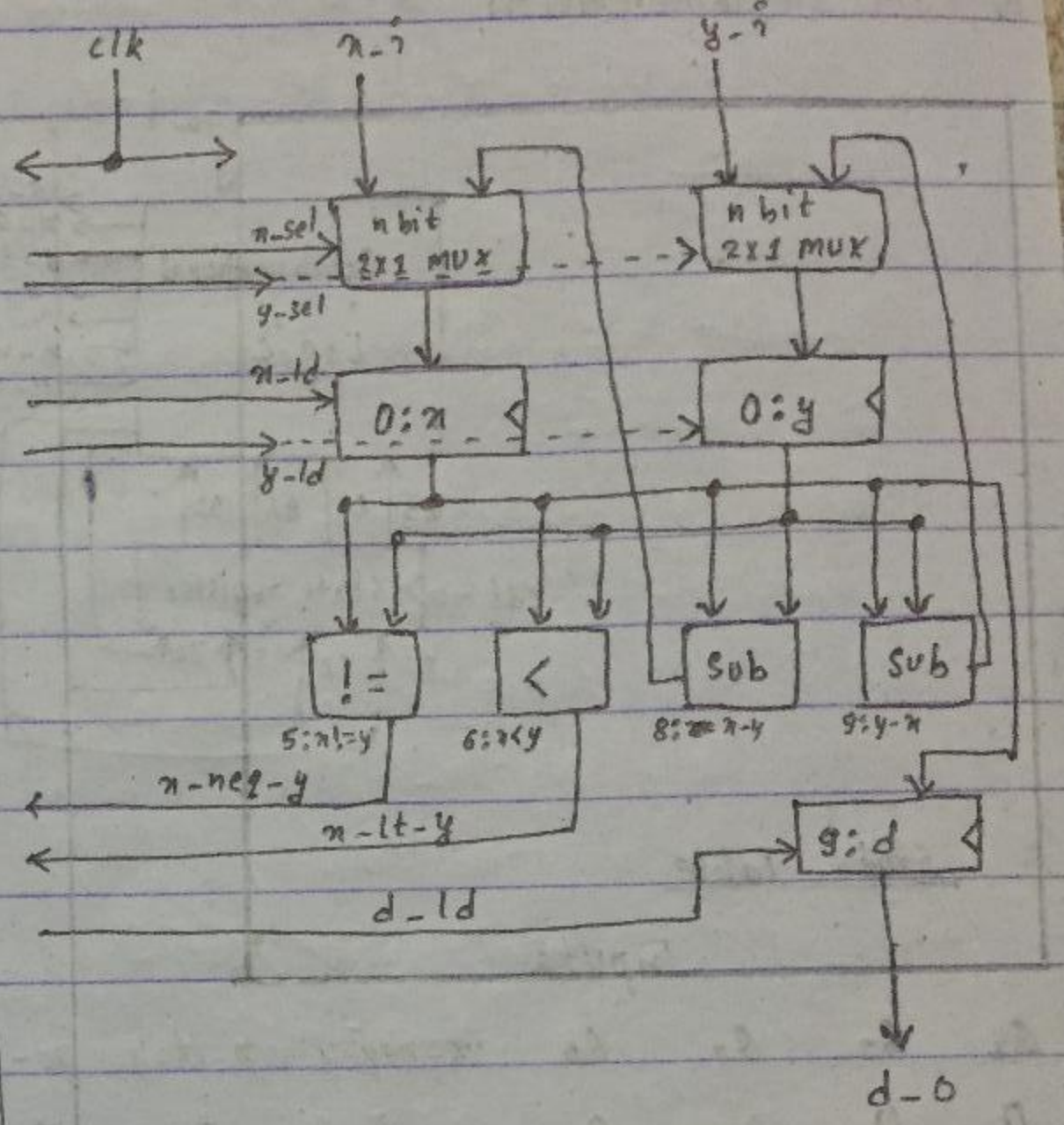
3) State Diagram or FSMD



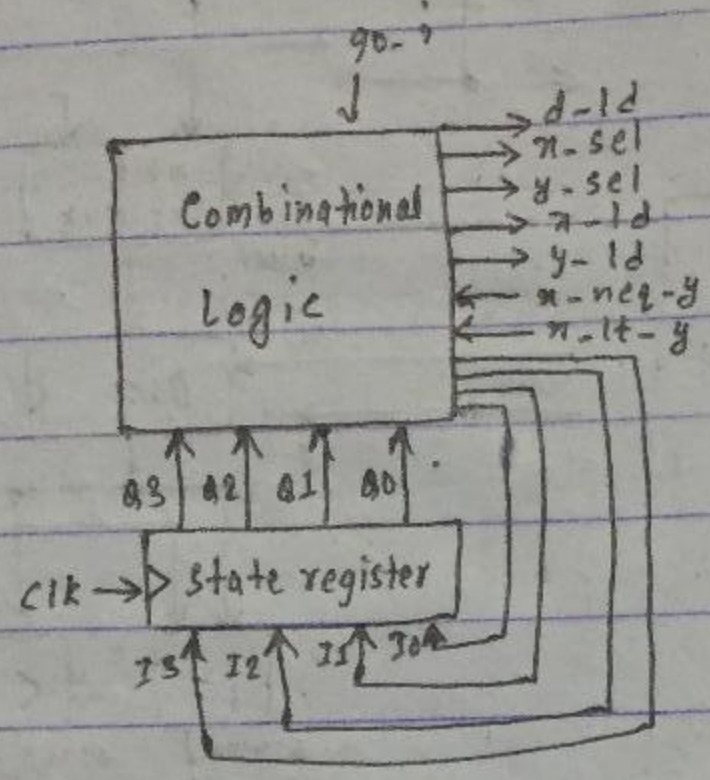
4) FSM + Datapath

FSM





5) FSM Implementation



State Table

Inputs							Outputs								
Q_3	Q_2	Q_1	Q_0	x-neq-y	x-lt-y	go-i	I_3	I_2	I_1	I_0	x-sel	y-sel	x-ld	y-ld	d-ld
0	0	0	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	0	1	*	*	0	0	0	1	0	X	X	0	0	0
						1	0	0	1	1	X	X	0	0	0
0	0	1	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	1	1	*	*	*	0	1	0	0	0	X	1	0	0
0	1	0	0	*	*	*	0	1	0	1	X	0	0	1	0
0	1	0	1	0	*	*	1	0	1	1	X	X	0	0	0
				1											
0	1	1	0	*	0	X	0	1	1	0	X	X	0	0	0
					1		1	0	0	0	X	X	0	0	0
0	1	1	1	*	*	*	0	1	1	1	X	X	0	0	0
1	0	0	0	*	*	*	1	0	0	1	X	1	0	1	0
1	0	0	1	*	*	*	1	0	0	1	1	X	1	0	0
1	0	1	0	*	*	*	1	0	1	0	X	X	0	0	0
1	0	1	1	*	*	*	0	1	0	1	X	X	0	0	0
1	1	X	X	*	*	*	1	1	0	0	X	X	0	0	0
							0	0	0	0	X	X	0	0	1
													0	0	0

a) Lowest Common Multiple (LCM)

→ LCM = ~~GCD~~ ~~a*b~~ $\frac{a*b}{GCD}$

Algorithm

```
0:  int x, y, z;
1:  while (1){
2:      while (!go-i);
3:      x = x-i;
4:      y = y-i;
5:      z = x*y;
6:      while (x!=y){
7:          if (x<y)
8:              y = y-x;
            else
9:              x = x-y;
        }
10:     d_o = z/x;
    }
```

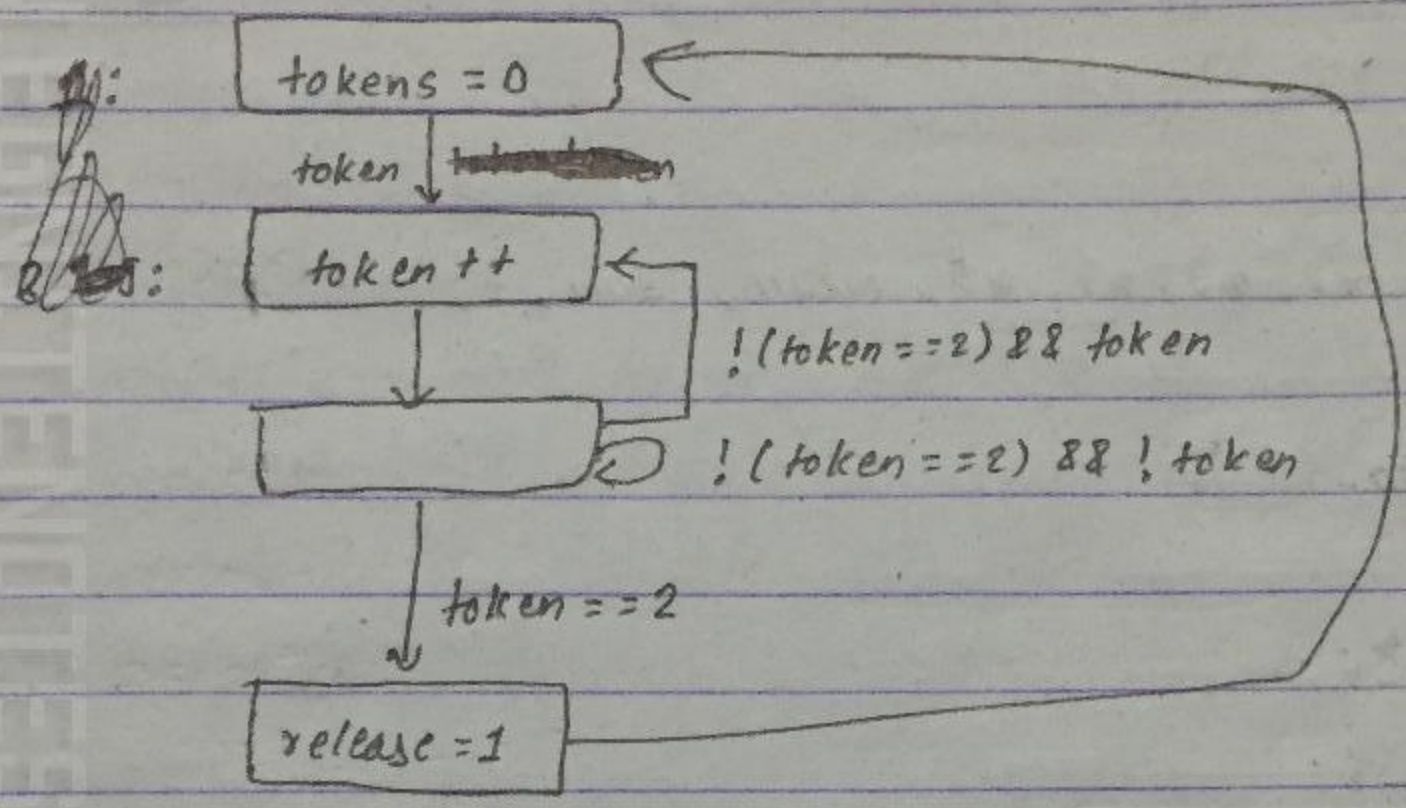
Q) Fibonacci number up to 'n' places

1, 1, 2, 3, 5, 8, 13, 21, ..., n^{th} place

Algorithm

```
int n1, n2, temp, count, n;
while (1){
    while (!go-i);
n1 = 1;
n2 = 1;
n = n_i;
count = 0;
while (count <n) {
  if (count !=0 && count !=1) {
   temp = n1;
    n1 = n2;
    n2 = n1 + temp;
} }
fib_0 = n2;
count ++;
}
}
```

Q) A subway has ES controlling turnstile, which releases when two tokens are deposited. Draw ESMD.



Q) Factorial and check if prime

```
int n, f, temp, i
while (1) {
    while (!go_i);
    i=2;
    n = n_i;
    temp = n
    f = 1;
    while (temp != 0) {
        f = f * temp;
        temp--;
    }
    f_0 = f;
    while (i < n) {
        if (n % i == 0)
            p_0 = 0;
        else
            p_0 = 1;
    }
}
```

a) Median and variance of 5 numbers entered by user

$$\text{Median} = \text{Value of } \left(\frac{n+1}{2}\right)^{th} \text{ item}$$

$$\text{Variance} = \frac{\Sigma (x-\mu)^2}{N}$$

```
int n, x1, x2, x3, x4, x5, mean, sum, i;
while (1){
while (!go_i);
n=5;
x1= x1_i;
x2 = x2_i;
x3 = x3_i;
x4 = x4_i;
x5 = x5_i;
mean = (x1 + x2 + x3 + x4 + x5) / 5;
median_o = x[ (n+1)/2 ];
sum=0;
i=0;
while (i<5){
sum = sum + (x[i] - mean) * (x[i]-mean);
i++;
}
variance_o = sum/n;
}
```

2.4 Optimization

Optimization is the technique of improving the design metrics so as to get the best possible values of various design metrics.

The optimization opportunities in custom SPP are as follows :-

1) Optimizing original program:

The algorithms are analyzed in terms of time complexity and space complexity, and hence we try to develop more efficient alternative algorithms. It involves decreasing of no. of computations and size of variables if possible.

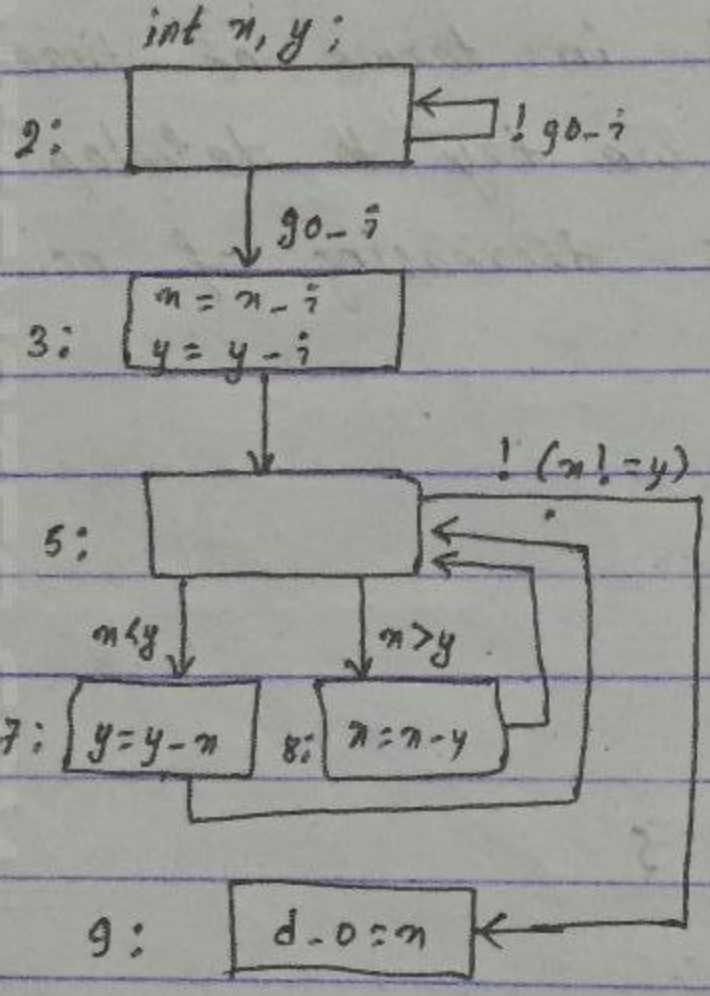
Eg: Optimized GCD program

```
int x, y, r ;
while (1) {
    while (!go_i);
    if (x_i >= y_i) {
        x = x - i;
        y = y - i;
    }
    else
    {
        x = y - i;
        y = x - i;
    }
    while (y != 0) {
        r = x % y;
        x = y;
        y = r;
    }
    d_o = x;
}
```

2) Optimizing FSMD

The states that can be merged are merged to reduce the number of states. The design must be aware of whether o/p timing may or may not be modified.

Eg: Optimized GCD FSMD



3) Optimizing Datapath

Many functional operations can share a single functional unit if those operations occur in different stages.

Eg: In GCD datapath, single subtractor can be used and selection can be done using multiplexor.

4) Optimizing FSM

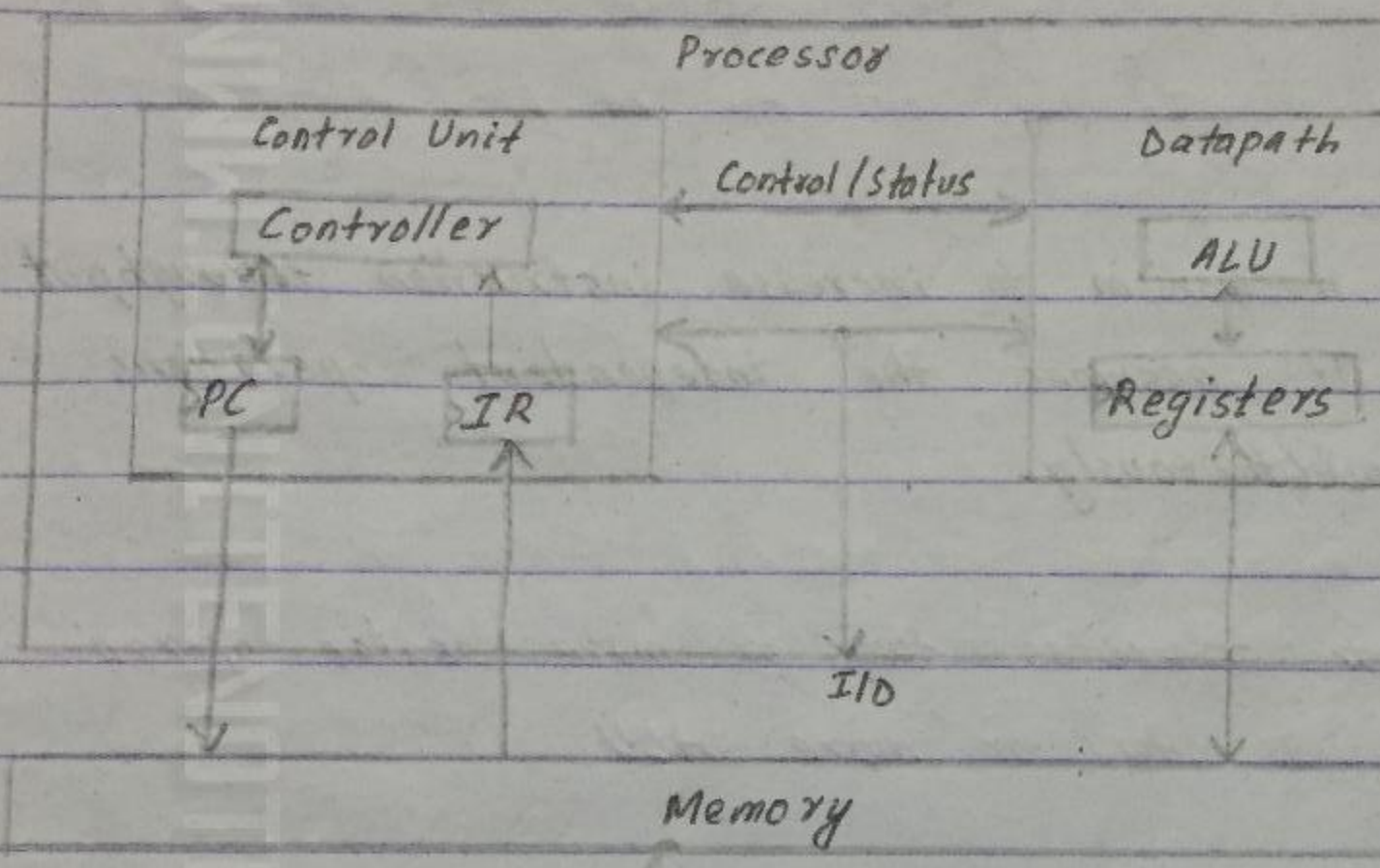
FSM can be optimized using state encoding and state minimization. State encoding is the task of assigning a unique bit pattern to each state in an FSM. We can use more than $\log_2(n)$ bits to encode n states.

State minimization is the task of merging equivalent states into a single state.

Chapter 3

General Purpose Processor

3.1 Basic Architecture



→ It consists of general datapath.
→ Control unit does not store algorithm.
→ Algorithm is programmed into memory.

3.2 Operations

1) Fetch Instruction: It gets the next instruction as indicated by location in memory pointed by PC and stores into IR.

2) Decode Instruction: It determines the actual operations performed by the instruction in IR.

3) Fetch Operands: It gets the operand data needed for instruction from memory to appropriate registers of datapath.

4) Execute: The actual arithmetic or logical operations is performed by moving data through ALU.

5) Store Results: It writes the data from datapath register into the memory.

Pipelining

Pipelining is the mechanism to increase instruction throughput of a microprocessor. It assumes the independent operations to be performed simultaneously.

→ Superscalar microprocessor executes two or more scalar operations in parallel and requires two or more ALU.

→ VLIW (Very long Instruction Word) architecture is a static superscalar microprocessor that encodes several operations in a single machine instruction.

3.3 Programmer's View

Programmer does not need detailed understanding of architecture. They just need to know which instructions can be executed. Generally, there are two levels of instructions: assembly level and structured languages.

→ Instruction set is the legal set of instructions that can be processed by a processor.

→ Addressing modes indicates how the data for any operation is referenced in the instruction.

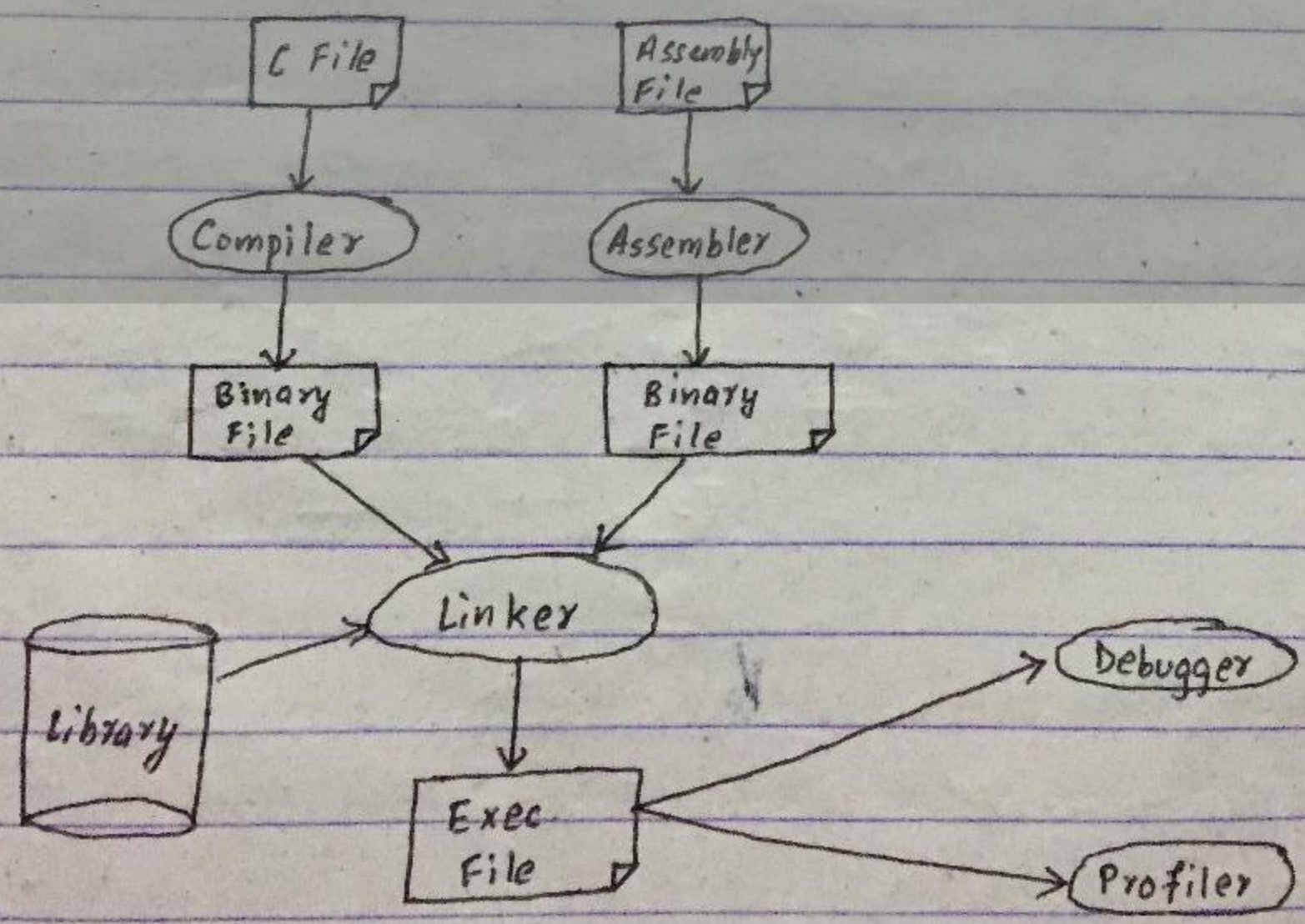
3.4 Programmer's Considerations

1) Program and data memory space
2) Registers
3) I/O
4) Interrupts
5) Operating System

3.5 Development Environment

→ Development processor : Processor on which programs are written.
→ Target processor : Processor that will run the program.
→ If development and target processor are different, the code can be run by downloading to target processor or by simulation.
→ Simulation can be done using HDL and Instruction Set Simulator(ISS)

3.6 Software Development Process



3.7 Application Specific Instruction - Set Processor (ASIP)

ASIP is the processor targeted for a particular domain. The architecture is domain specific but can be programmed.

Eg: microcontroller, Digital Signal Processor

3.8 Selection of microprocessor

1) Speed (Clock speed or instructions per second)
2) Power requirements
3) Size (Program and data memory space)
4) Cost
5) Development environment
6) Technical support
7) Experience

Chapter 4

Memory

4.1 Memory basics

→ m x n memory stores m words of n bits each.

→ $m = 2^k$, where k = no. of address input signals.

→ n indicates no. of data signals.

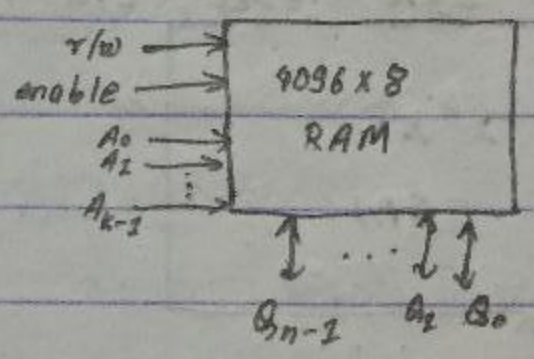
→ r/w selects read or write

→ enable : read or write only when asserted.

Consider a memory with 4096 x 8.

→ k = 12

→ n = 8



4.2 Memory write ability and Storage permanence

→ Write Ability

Write ability is the manner and speed that a particular memory can be written. The ranges of write ability are:-

1) High end (processor writes to memory quickly → RAM)

2) Middle range (processor writes to memory slowly → FLASH, EEPROM)

3) Lower range (special equipment used to write to memory → EPROM, OTP ROM)

4) Low end (bits stored only during fabrication → Masked ROM)

→ Storage Permanence

Storage permanence is the ability of memory to hold its stored bits after they have been written. The ranges are:-

1) High end (never loss bits) → Masked ROM)

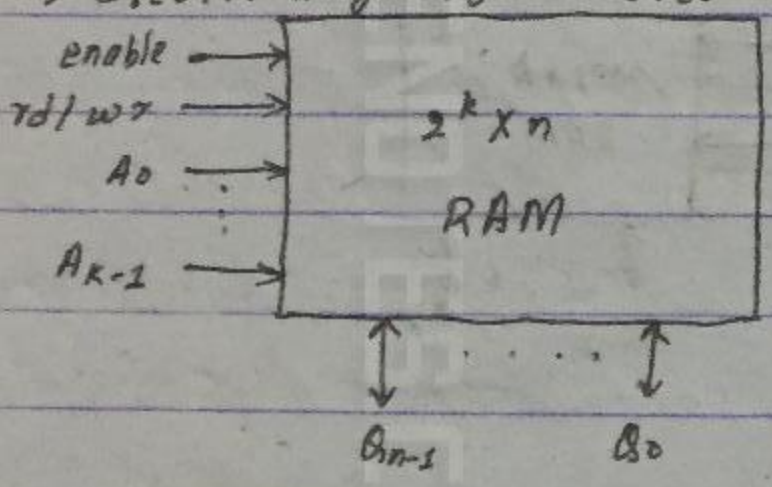
2) Middle range (holds bit for days or years after power cut off → NVRAM)

3) Lower range (holds bits as long as power is supplied → SRAM)
4) Low end (lose bits immediately after written → DRAM)

4.3 Types of Memory

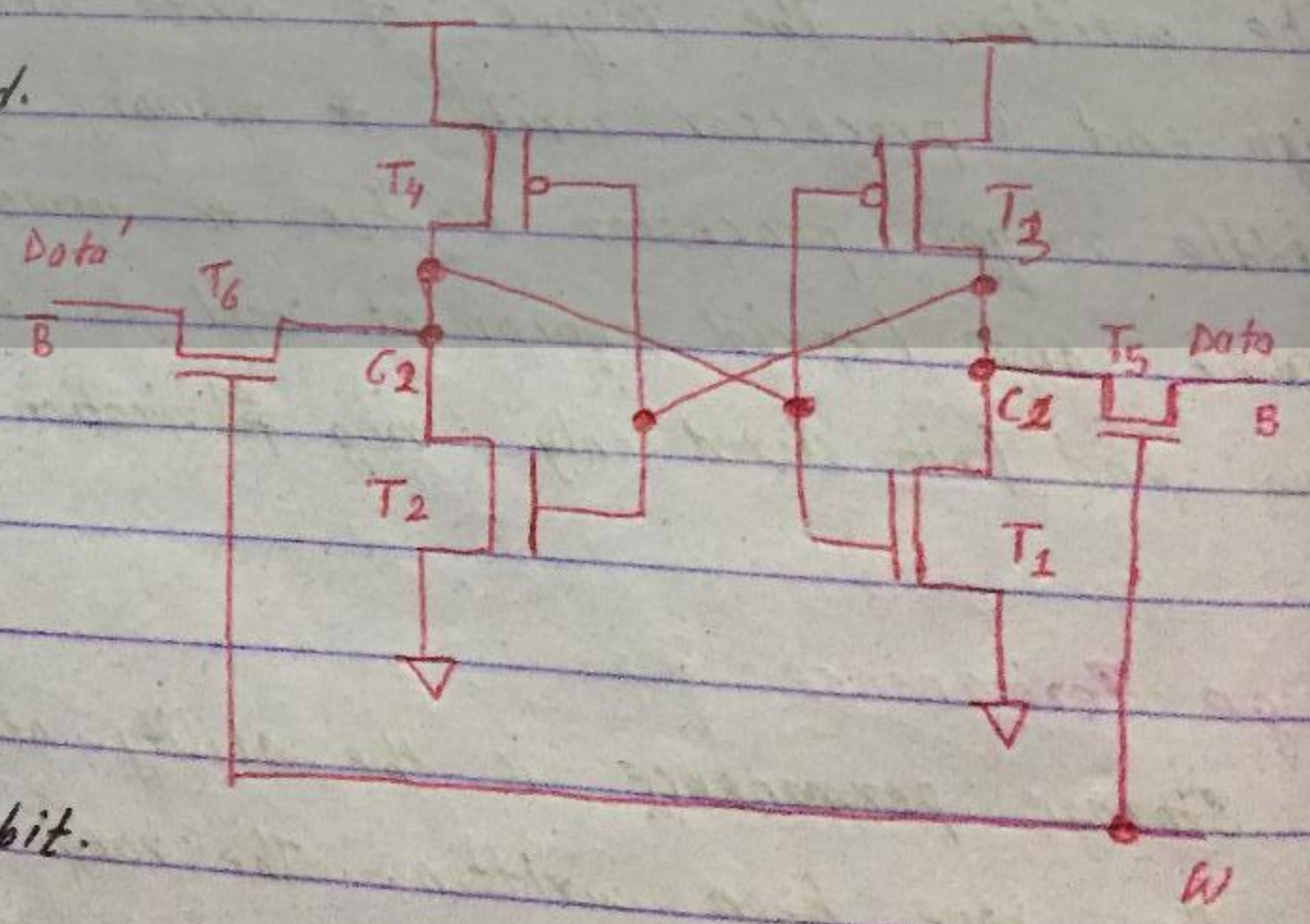
1) Random Access Memory (RAM)

→ It can be read and written easily.
→ It is volatile memory.
→ Electrically byte level erasing → Electrically write



→ SRAM

→ Bits are stored as on/off switches.
→ Charges are not leaked.
→ Refreshing is not needed.
→ No refresh circuits
→ Larger per bit
→ Complex construction
→ Expensive
→ Faster
→ Cache
→ Uses flip flops to store bit.



→ In state 1;

C_1 = high, C_2 = low

T_2 T_3 on

→ In state 0;

C_1 = low, C_2 = high

T_1 T_4 on

→ Write : apply value to B and $\overline{B}$

→ Read : value is on B

→ DRAM

→ Bits are stored as charge in capacitor

→ Charges leak

→ Need refreshing

→ Have refresh circuits

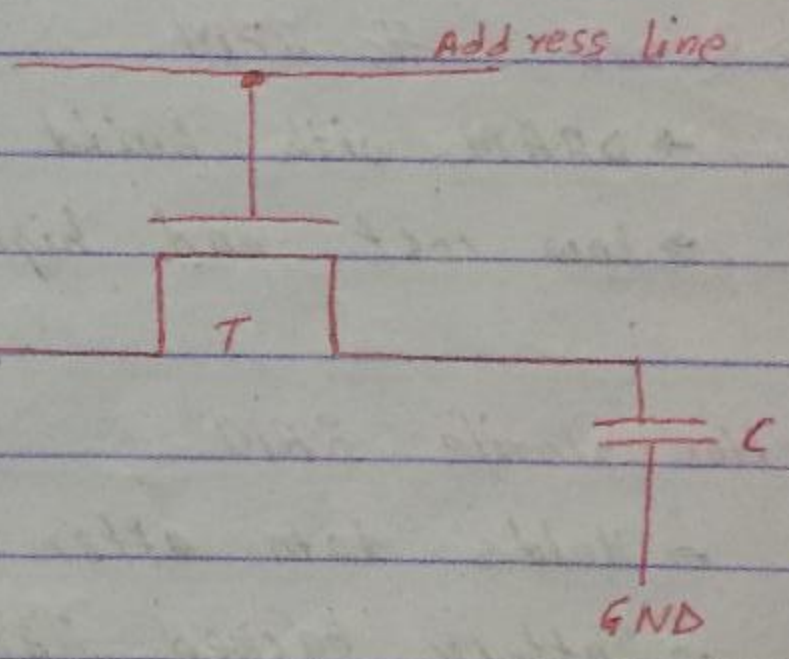
→ Cheaper

→ Slower

→ Smaller per bit

→ Main memory

→ Level of charge determines value



→ When address line is active, transistor switch closes, so bit is read or written

→ During write operation, transfers charge to capacitor.

→ During read operation, address line selected and charge from capacitor fed via B to sense amplifier.

→ Enhanced DRAM

→ Enhanced DRAM are built around conventional DRAM core.
→ It is a DRAM that includes small amount of static RAM so as to access memory faster.
→ First, data is checked in SRAM
→ If miss, data is accessed from DRAM.
→ Fast page mode DRAM : access row contents [RAS, CAS, CAS, CAS]
→ Extended data out DRAM : FPM with closely spaced CAS.
→ Synchronous DRAM : Driven with rising clock edge
→ Double data rate SDRAM : SDRAM with both clock edges.

→ Pseudo static RAM

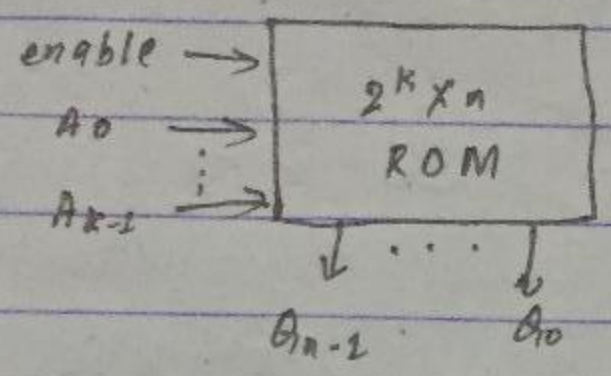
→ DRAM with build in memory refresh controller
→ Low cost and high density

→ Non volatile RAM

→ Holds data after external power removed
→ Battery backed RAM
→ SRAM with EEPROM

2) Read Only Memory (ROM)

→ Non-volatile memory



→ Masked ROM

→ Programmed during fabrication by creating masks.
→ Low write ability
→ High storage permanence

→ OTP ROM

→ External equipment is needed to write.
→ Electrically write mechanism

→ EPROM

→ Electrically write
→ Erase using UV light (chip level)

→ EEPROM

→ Byte level electrically erasing
→ Electrical write

→ Flash memory

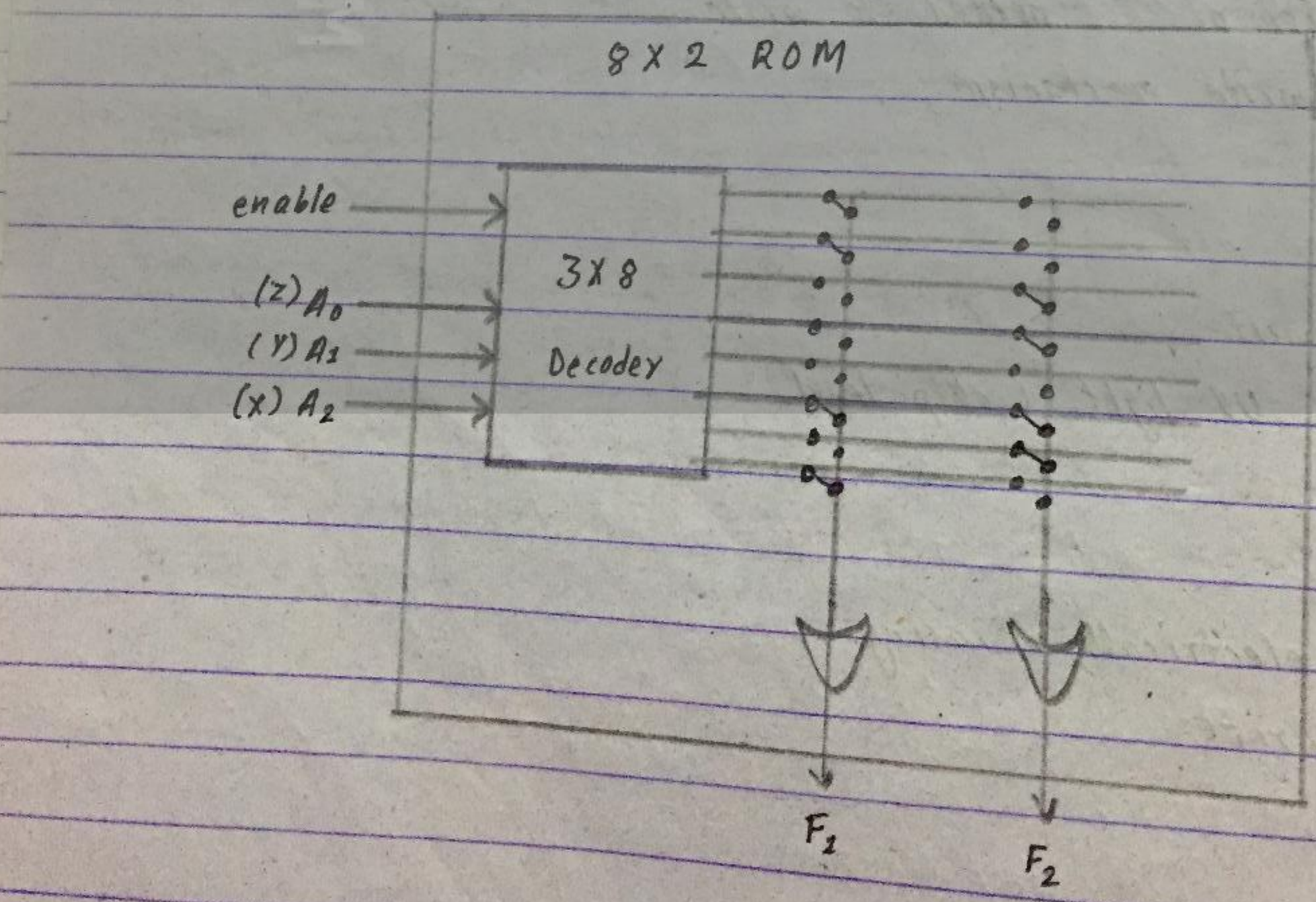
→ Block level electrically erasing

4.4 Composing memory

→ Design 8 X 2 ROM with:

X	Y	Z	F_1	F_2
0	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	1
1	0	1	1	1
1	1	0	0	1
1	1	1	1	0

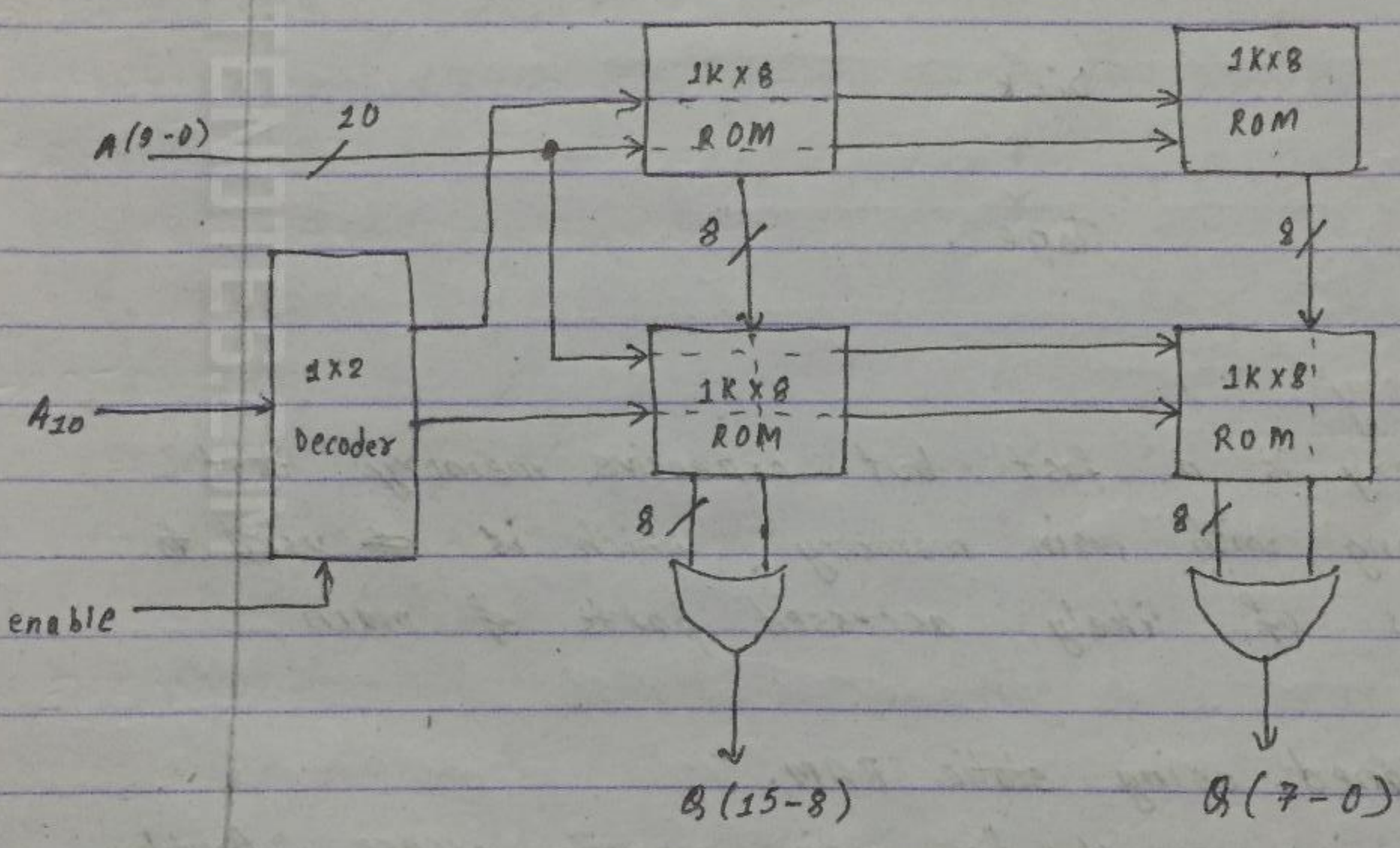
Ans →



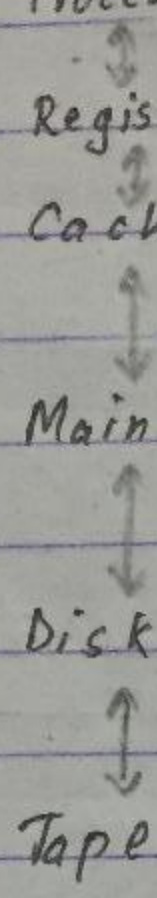
→ Compose 1K x 8 ROMs into 2K X16 ROM

Ans →

1K x 8 ROM	2K x 16 ROM
k = 10	k = 11
n = 8	n = 16



4.5 Memory Hierarchy



4.6 Cache Memory

→ Cache memory is a fast but expensive memory that is used along with main memory which is used to store copies of likely accessed parts of main memory.

→ It is designed using static RAM.

→ When there is request for main memory access, firstly copy in cache is checked. If cache hit, memory access becomes quick. If cache miss, read from the main memory.

→ Cache Mapping

1) Direct mapping
2) Fully associative mapping
3) Set associative mapping

→ Cache Replacement

→ Choosing which cache block to replace.
→ Random replacement
→ Least recently used (LRU) policy
→ First in first out (FIFO) policy

→ Cache Write Techniques

→ Updating the main memory
→ Write-through (Write to main memory whenever cache is written)
→ Write-back (Write to main memory only when cache block is being replaced)

→ Cache Impacts

	2K cache	4K cache	8K cache
miss rate	15%	6.5%	5.565%
hit cost	2 cycle	3 cycle	4 cycle
miss cost	20 cycle	20 cycle	20 cycle

av. cost of memory access = (hit rate * hit cost) + (miss rate * miss cost) cycles

Chapter 5

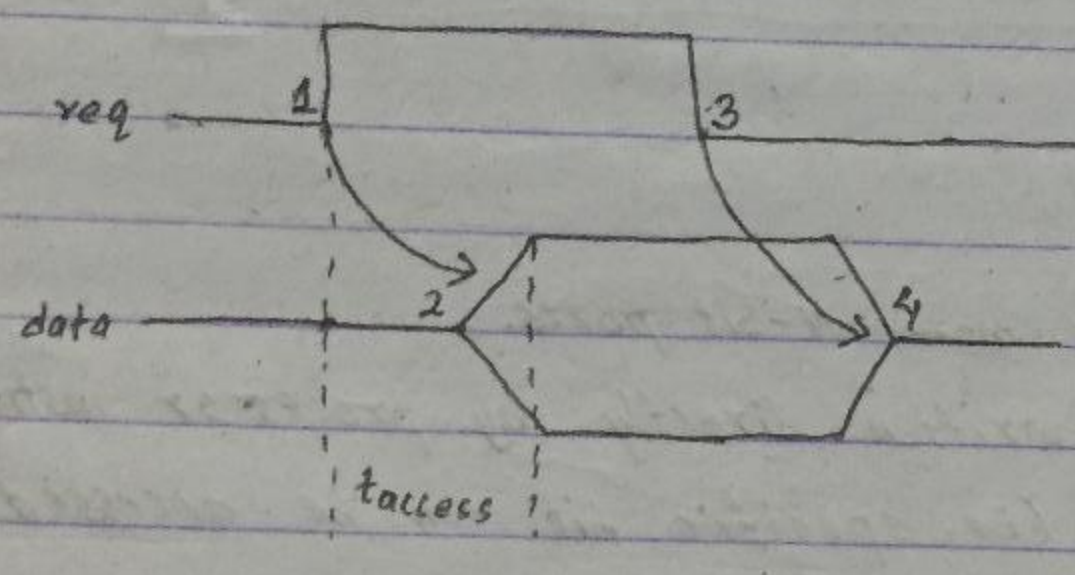
Interfacing

5.1 Communication Basics

→ Bus is a set of wires with a single functions within a communication.
→ Protocol describes the rules for communicating within a bus.
→ Each sub protocol is called transaction or bus cycle.
→ Actor is a processor or memory involved in data transfer.
→ Data direction indicates direction of transferred data.
→ Address indicate where data should go to or come from.
→ Control methods are schemes for initiating and ending the transfer.

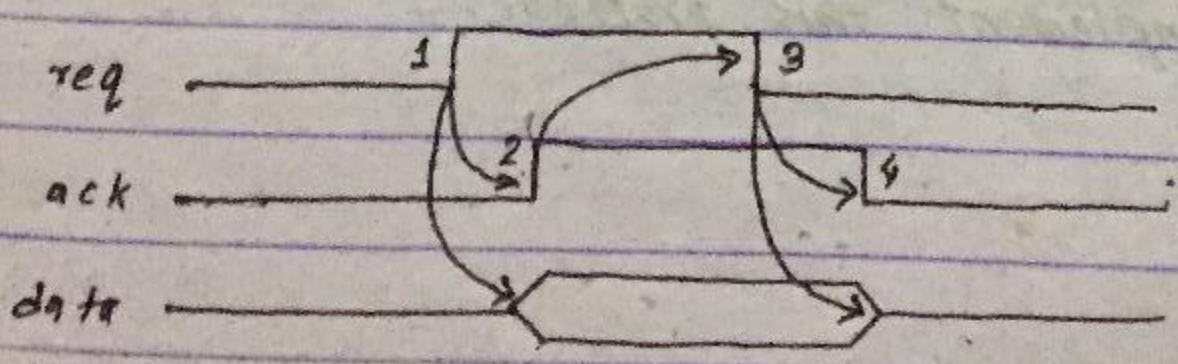
→ Strobe Protocol

→ The master uses one control line (request line) to initiate data transfer.
→ The transfer is considered to complete after some fixed time.

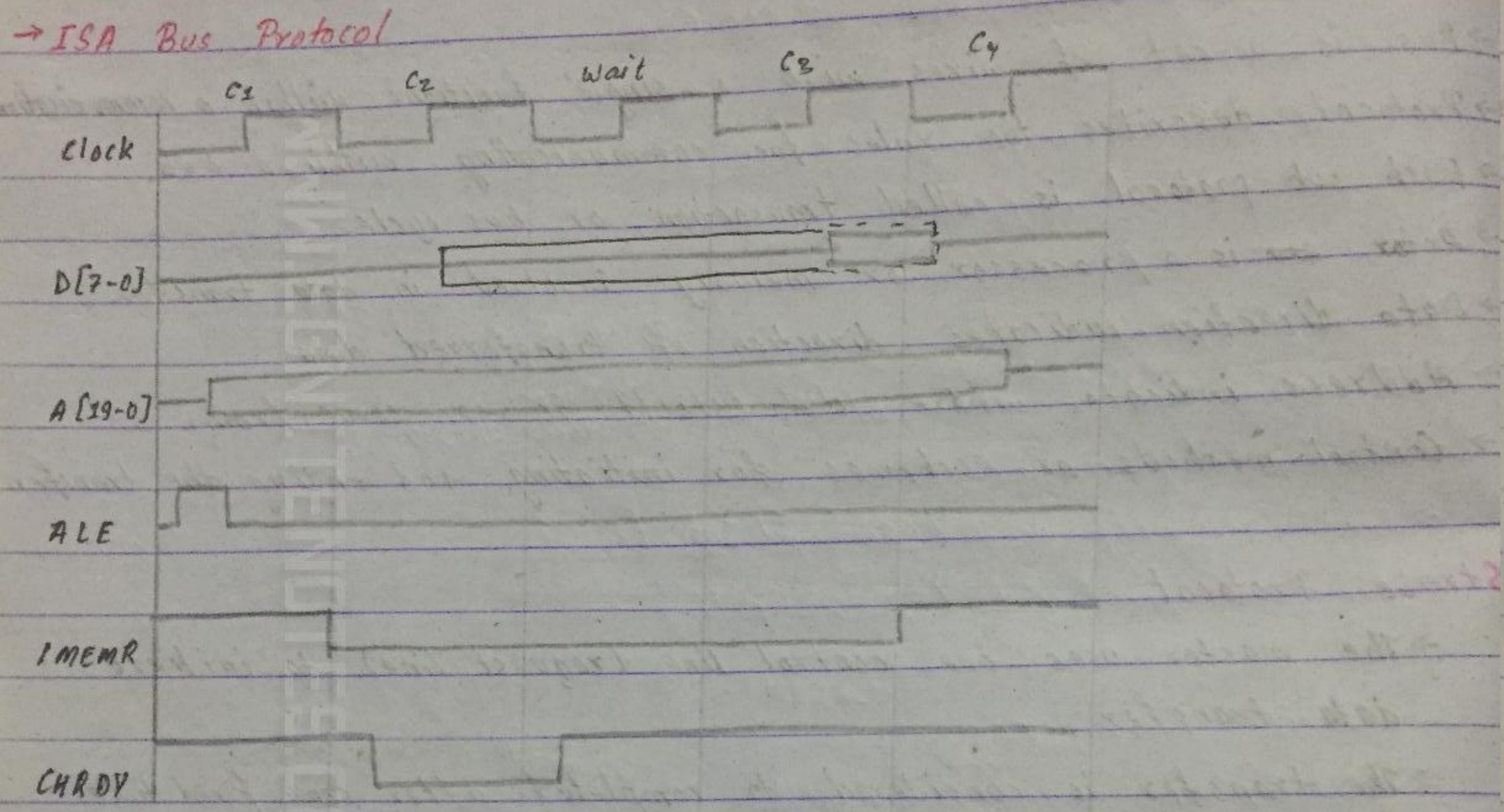


→ Handshake Protocol

→ Master uses request line to initiate transfer.
→ The servant uses acknowledge line to inform master when data is ready.



→ ISA Bus Protocol



5.2 Port and Bus Based I/O

→ Port based I/O

→ Processor contains one or more N-bit ports.
→ A port can be read or written directly by processor instruction.
→ Port are bit addressable (i.e. specific bit can be accessed)

→ Bus based I/O

→ Memory is accessed using bus lines.
→ Bus protocol is built within microprocessor
→ A software does not implement this protocol.

⇒ Memory mapped I/O and Standard I/O

→ Peripherals occupy specific addresses in existing address space. In standard I/O, bus includes M/$\overline{IO}$ to indicate whether the access is to memory or to a peripheral.

→ In memory mapped, no special instructions are needed to communicate with peripherals.

→ In standard I/O, memory address is not lost.

5.3 Interrupts

⇒ Using fixed ISR location

1) μp is executing the main program.

2) Peripheral (P1) receives i/p data and asserts Int to request service.

3) After completing current instruction, μp sees int. It saves PC and set PC to ISR fixed location.

4) The ISR reads data and perform necessary functions. After P1 is read, P1 deasserts Int.

5) The ISR returns and then PC is restored.

⇒ Using vectored interrupt

→ When μp sees int, it saves PC and asserts Inta

→ P1 detects Inta and puts interrupt address vector on data bus.

→ μp jumps to address on the bus.

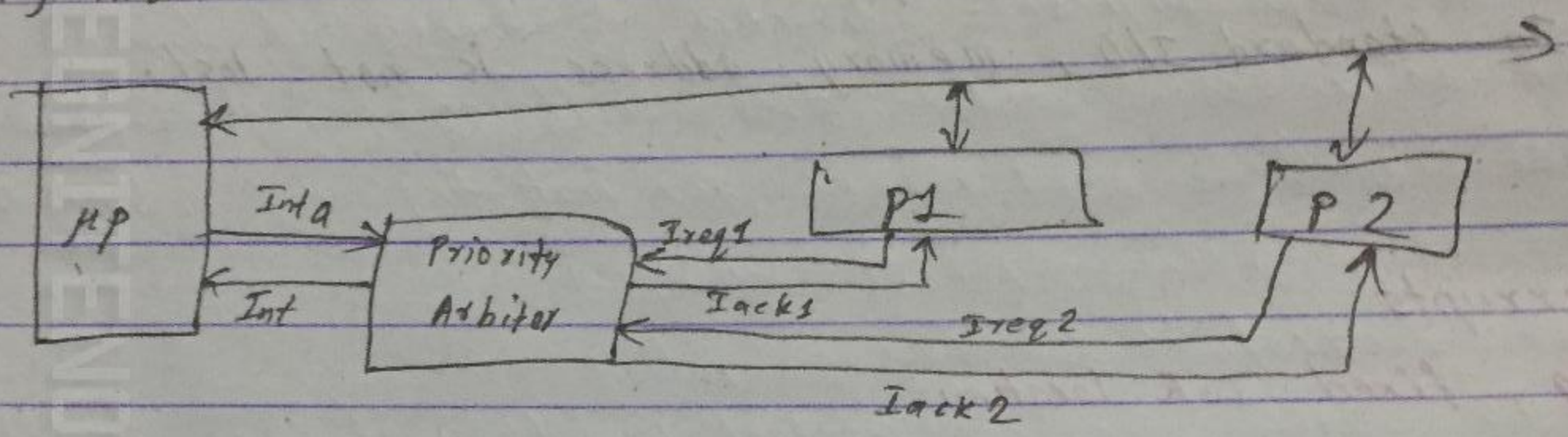
5.4 Direct Memory Access (DMA)

→ µp gives control of system bus to DMA controller

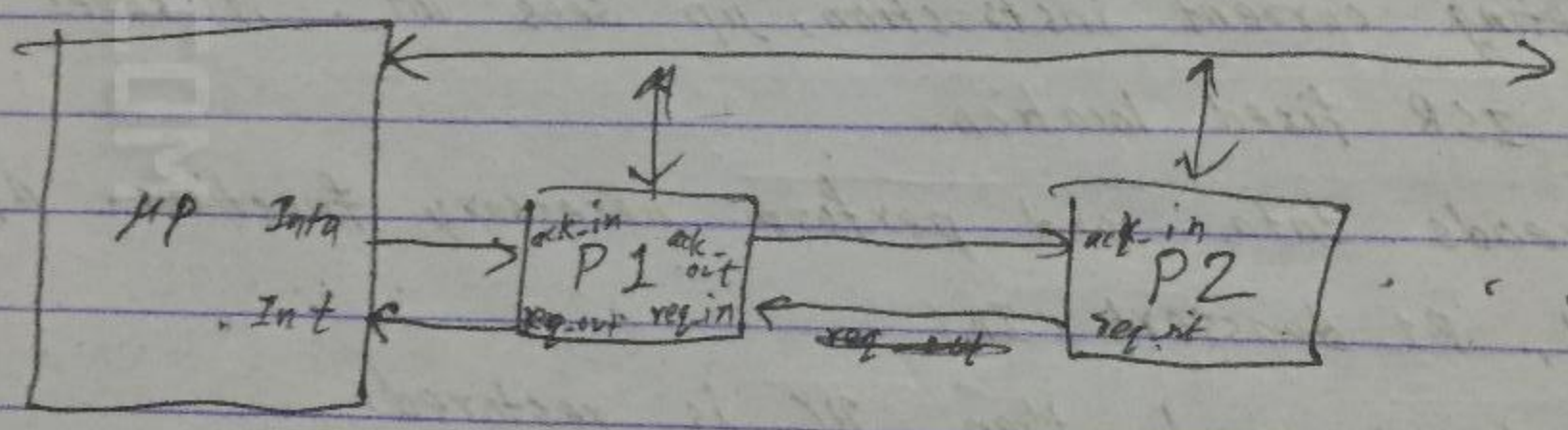
→ µp can execute its regular program.

5.5 Arbitration

→ Priority Arbiter



→ Daisy-chain arbitration



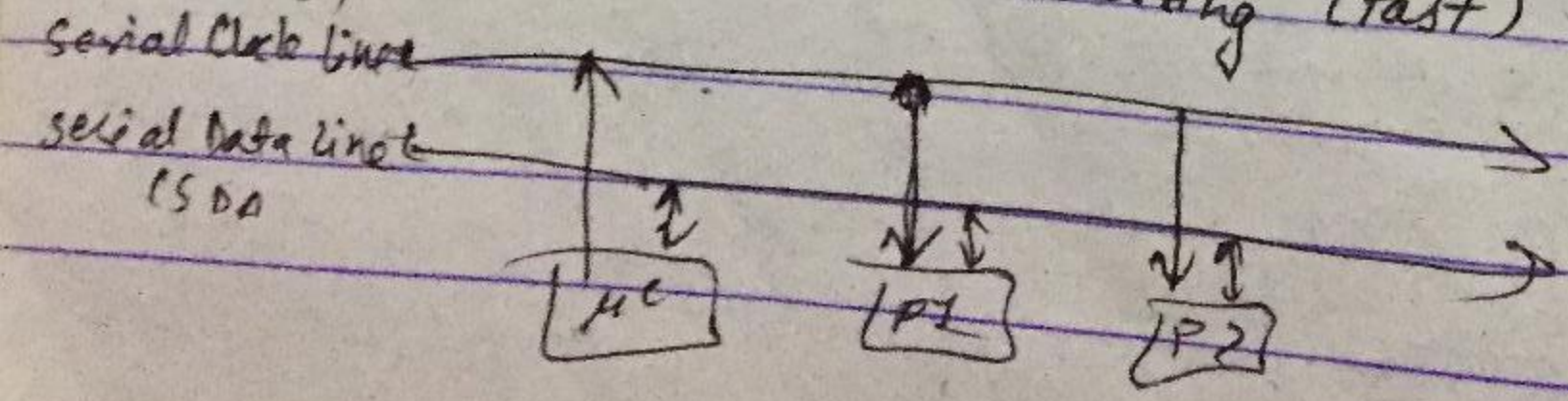
→ I²C Protocol

→ Inter-IC protocol

→ Enable peripheral ICs to communicate.

→ Data transfer rate upto 100 Kbits/s & 7-bit addressing (Normal)

→ 3.4 Mbits/s & 10 bit addressing (Fast)

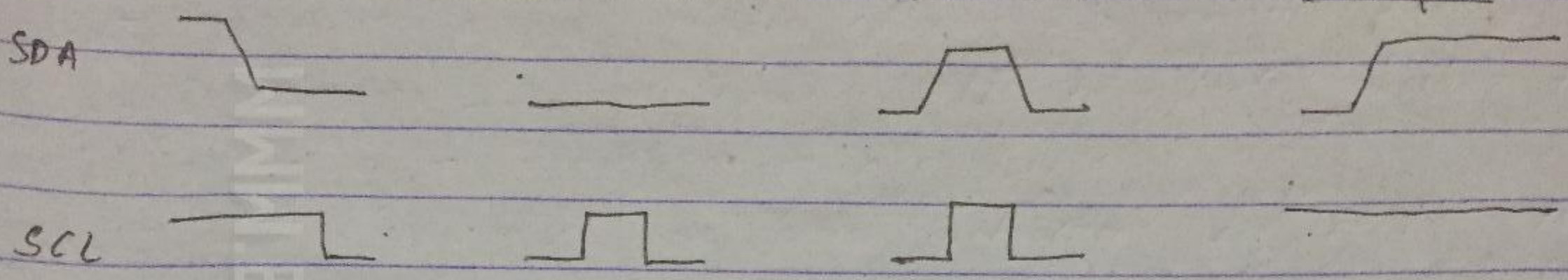


Start

Send 0

Send 1

Stop



Chapter 7
Control System

7.1 Open-loop and Close-loop control system

7.2 Control system and PID Controller

7.3 Software coding of a PID Controller

7.4 PID Tuning

7.1 Control System

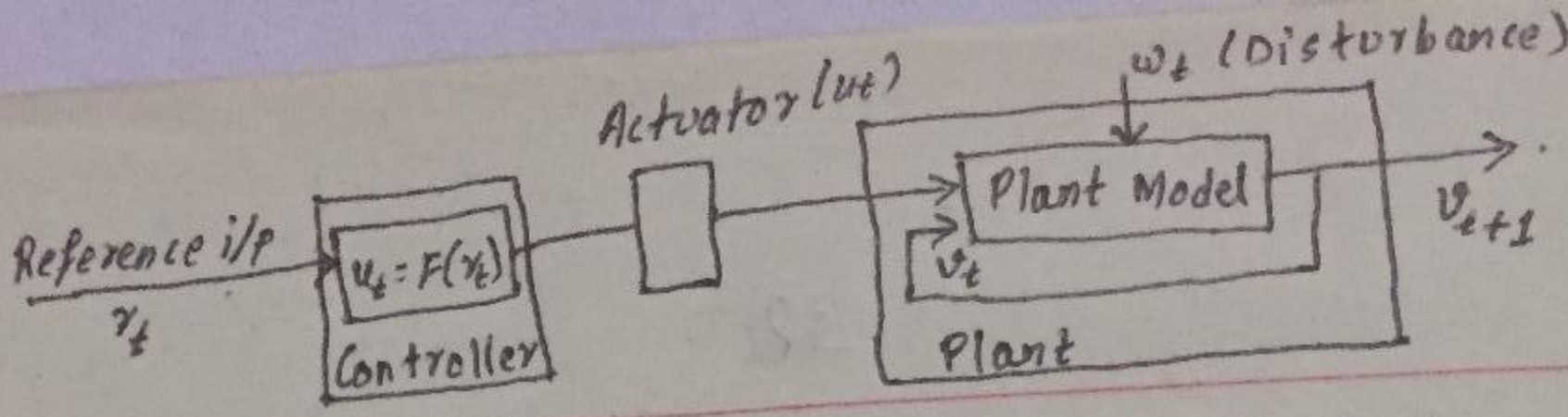
Control system is a system designed to control physical system's output to a desired limit by setting desired reference input.

→ Automobile Cruise Controller

This controller seeks to make a car's speed track a desired speed by setting the car's throttle and brake inputs.

→ Thermostat Controller

It forces a building's temp to a desired temp by turning on the heater or AC and adjusting the fan speed.



7.2 Open Loop Control System

→ It is a feed-forward control system

→ It consists of following parts:

→ Plant / Process

It is the physical system to be controlled.

→ Output

It is the particular aspect of physical system that we intend to control.

→ Reference input

It is the desired value that we want to see for the output.

→ Actuator

It is the device used to control i/p to the plant.

→ Controller

It is the system used to compute the i/p to the plant so as to achieve desired o/p.

→ Disturbance

It is an additional undesirable i/p imposed by environment that cause o/p to differ from our expectation.

7.3 Designing Open Loop Control System

7.3.1 Model of plant

Model of the plant is developed to make design phase easier and to derive the controller.

For eg: Throttle that goes from 0 to 45 degree

→ On flat surface at 50mph, open throttle to 40 degree

→ Wait 1 "time unit"

→ Measure the speed, let's say 55 mph

→ Then, the equation that holds for all scenario gives model of the plant.

i.e. $v_{t+1} = 0.7 * v_t + 0.5 * u_t$

$55 = 0.7 * 50 + 0.5 * 40$

7.3.2 Controller design

Controller provides a specific rule to achieve the goal of the control system.

For eg: Assume we use simple linear function.

$u_t = F(r_t) = p^* r_t$

where, p = constant

r_t = desired speed

Now,

$$v_{t+1} = 0.7 * v_t + 0.5 * u_t = 0.7 * v_t + 0.5 * p * r_t$$

Let $v_{t+1} = v_t$ at steady state $= v_{ss}$;

$$v_{ss} = 0.7 * v_{ss} + 0.5 * p * r_t$$

At steady state, we want $v_{ss} = r_t$

$$1 = 0.7 + 0.5 * p$$

$$\therefore p = 0.6$$

Hence, the control rule is defined as:

$$u_t = p * r_t = 0.6 * r_t$$

7.3.3 Analyzing Controller

Suppose $v_0 = 20$ mph and $r_0 = 50$ mph

Then, $v_{t+1} = 0.7 * v_t + 0.5 * 0.6 * r_t$

$= 0.7 * v_t + 0.3 * 50$

$= 0.7 * v_t + 15$

$\therefore$ Throttle position $= 0.6 * 50 = 30$ degree

7.3.4 Considering Disturbance

Assume road grade affect speed from -5mph to +5 mph.

Time (t)	v_t	v_t ($w = +5$)	v_t ($w = -5$)
0	20	20	20
1	29	24	34
2	35.30	26.80	43.80
3	39.71	28.76	50.66
4	42.80	30.13	55.46
5	44.96	31.09	58.82
6	46.47	31.76	61.18
7	47.53	32.24	62.82
8	48.27	32.56	63.98
9	48.79	32.80	64.78
10	49.15	32.96	65.35
11	49.41	33.07	65.74
12	49.58	33.15	66.02

3.3.5 Determining Performance

$$v_{t+1} = 0.7 v_t + 0.5 \rho r_0 - w_0$$

$$\therefore v_1 = 0.7 v_0 + 0.5 \rho r_0 - w_0$$

$$\therefore v_2 = 0.7 (0.7 v_0 + 0.5 \rho r_0 - w_0) + 0.5 \rho r_0 - w_0$$

$$= 0.7^2 v_0 + (0.7 + 1)(0.5 \rho r_0 - w_0)$$

$$\therefore v_t = 0.7^t v_0 + (0.7^{t-1} + 0.7^{t-2} + \dots + 0.7 + 1)(0.5 \rho r_0 - w_0)$$

→ Coefficient of v_t determines rate of decay of v_0

→ If > 1 or < -1, v_t will grow without bound

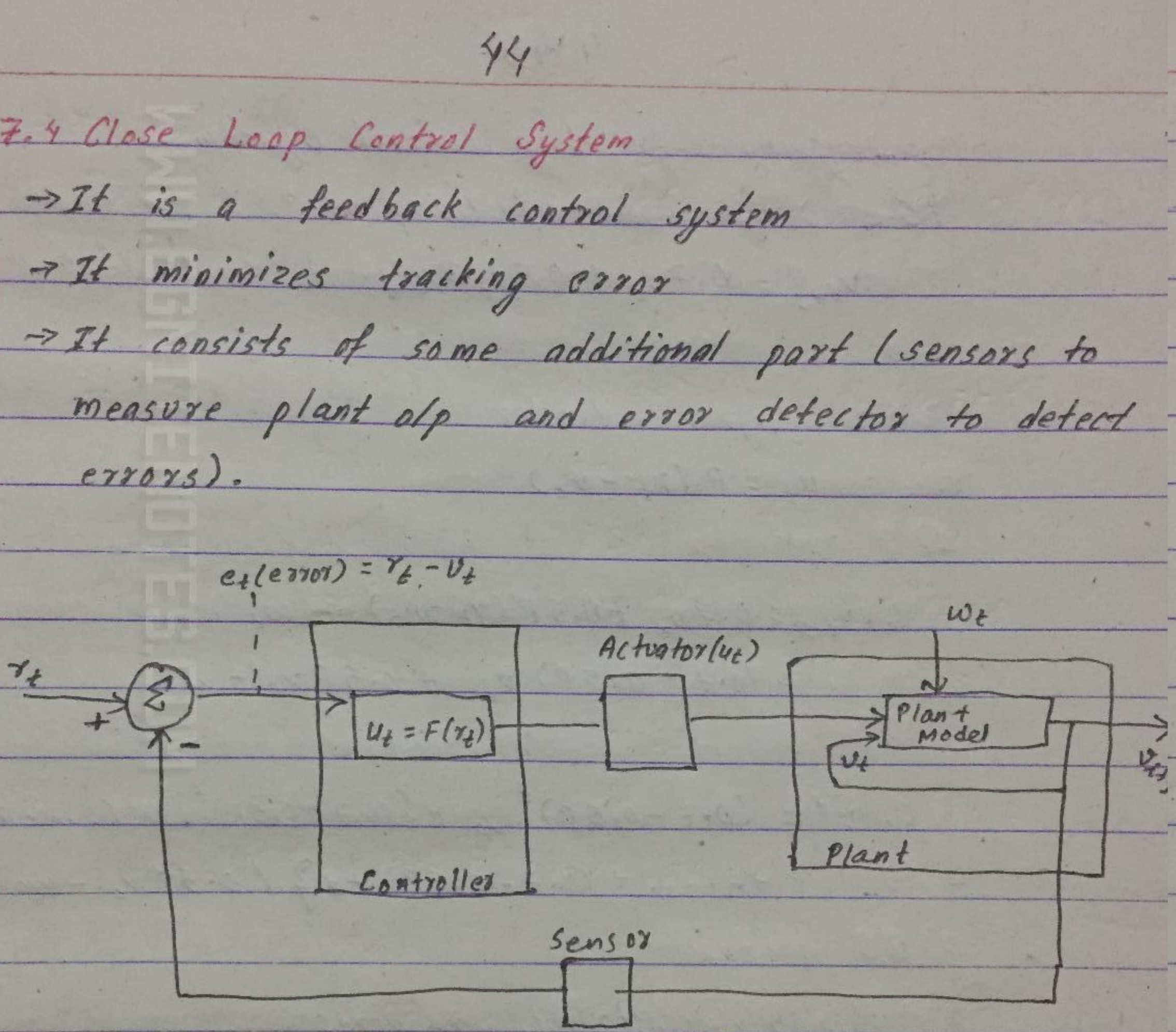
→ Otherwise, v_t will oscillate.

7.4 Close Loop Control System

→ It is a feedback control system

→ It minimizes tracking error

→ It consists of some additional part (sensors to measure plant o/p and error detector to detect errors).



7.5 Designing Close Loop Control System

7.5.1 Design model of the plant

$$v_{t+1} = 0.7 v_t + 0.5 u_t - w_t$$

7.5.2 Design controller

$$u_t = P(r_t - v_t)$$

$$\therefore v_{t+1} = 0.7 v_t + 0.5 P(r_t - v_t) - w_t$$

$$= (0.7 - 0.5P) v_t + 0.5 P r_t - w_t$$

$$\therefore v^t = (0.7 - 0.5P)^t v_0 + \{(0.7 - 0.5P)^{t-1} + (0.7 - 0.5P)^{t-2} + \ldots + (0.7 - 0.5P) + 1\} (0.5 P r_0 - w_0)$$

So,

Stability constraint requires :

$$|0.7 - 0.5P| < 1$$

or, $-1 < 0.7 - 0.5P < 1$

$\therefore \quad -0.6 < P < 3.4$

7.5.3 Reducing effect of v_0

To reduce effect of initial condition (i.e. v_0)

a) $0.7 - 0.5P$ must be as small as possible.

i.e. $0.7 - 0.5P = 0$

$\therefore P = 1.4$

7.5.4 Avoid Oscillation

To avoid oscillation, $(0.7 - 0.5P)$ must be greater than or equal to 0.

i.e. $0.7 - 0.5P >= 0$

$\therefore P <= 1.4$

7.5.5 Perfect Tracking

Assume at steady state v_{ss}, $v_{t+1} = v_t$;

$$v_{ss} = (0.7 - 0.5P)\, v_{ss} + 0.5P r_0 - w_0$$

or, $(1 - 0.7 + 0.5P)\, v_{ss} = 0.5P r_0 - w_0$

$$\therefore v_{ss} = \frac{0.5P r_0 - w_0}{0.3 + 0.5P} = \frac{0.5}{0.3+0.5P} r_0 - \frac{1}{0.3+0.5P} w_0$$

→ To make v_{ss} close to r_0, P should be as large as possible

7.5.6 Analyse Controller

→ Set P = 3.3, saturate at 0, 45. Oscillation

→ Set P = 1 to avoid oscillation, Terrible SS performance.

7.6 General Control System

1) P controller

→ It multiplies the tracking error by a constant.

→ $u_t = P(r_t - v_t)$

→ P affects transient response (stability), steady state tracking error and disturbance rejection.

2) PD controller

→ It considers the size of error over time.

→ It keep track of error derivative.

→ $u_t = P * e_t + D * (e_t - e_{t-1})$

→ P set for best tracking and disturbance control.

→ D set to control oscillation and convergence rate.

3) PI controller

→ It sum up error over time

→ $u_t = P * e_t + I * (e_0 + e_1 + \dots + e_t)$

→ P controls disturbance

→ I ensures steady state convergence.

4) PID controller

→ Combines proportional, integral and derivative control.

→ $u_t = P * e_t + D * (e_t - e_{t-1}) + I * (e_0 + e_1 + \dots + e_t)$

→ It achieves desired stable transient behavior.

7.7 Software Coding for PID controller

```
void main()
{
    double sen_val, act_val, err_current;
    PID_DATA pid_data;
    Pid Initialize (& pid_data);
    while (1)
    {
```

```
sen_val = SensorGetValue();
ref_value = ReferenceGetValue();
act_value = PidUpdate(&pid_data, sen_val, ref_val);
ActuatorSetValue(act_value);
}

}

typedef struct PID_DATA {
    double Pgain, Dgain, I.gain;
    double sen_val_previous; // find derivative
    double error_sum;  // cumulative error
}

double PidUpdate(PID_DATA *pid_data, double sen_val,
                 double ref_val)
{
    double Pterm, Iterm, Dterm;
    double err, diff;
    err = ref_val - sen_val;
    Pterm = pid_data -> Pgain * err;
```

```
pid_data->error_sum += error;
Iterm = pid_data*->Igain * pid_data->error_sum;
diff = pid_data->sen_val_previous - sen_val;
pid_data->sens.val_previous = sen_val;
Dterm = pid_data->Dgain * diff;
return (Pterm + I term + Dterm);
}
```

7.8 PID Tuning

→ PID tuning is the process of obtaining P, I and D by ad-hoc methods.

→ It is necessary because:-

a) With analytics, the plant model may be complex

b) We may not even have a plant model.

→ Algorithm

1) Start with a small P, I = D = 0

2) Increase D, until seeing oscillation.

→ Decrease D a bit.

3) Increase P, until seeing oscillation

→ Decrease P. a bit

4) Increase I, until seeing oscillation

5) Repeat from 2 until performance can not be further improved.

7.9 Issues with Computer Based Control

1) Quantization and Overflow

→ Quantization occurs when a signal must be altered to fit memory constraints.

→ Arithmetic operations cause quantization.

→ Eg: Assume an operation results 0.36 but it can not be stored as 4-bit fractional number. So, 0.25 is choosed.

→ Overflow occurs when operation outputs a number with magnitude too large to be represented by a processor.

→ Use fix point representation carefully (time consuming)

→ Use floating-point co-processor (expensive)

2) Aliasing

→ Aliasing is due to the discrete nature of sampling.

→ Eg: Sampling at 2.5 Hz, period of 0.4;

$y(t) = 1.0 * \sin(6\pi t)$, $f = 3$ Hz

$y(t) = 1.0 * \sin(\pi t)$, $f = 0.5$ Hz

They are indistinguishable.

3) Computational Delay

→ The computation causes delay to occur that causes actuation to delay.

→ The implementation delay must be negligible.

→ Software delay is hard to predict.

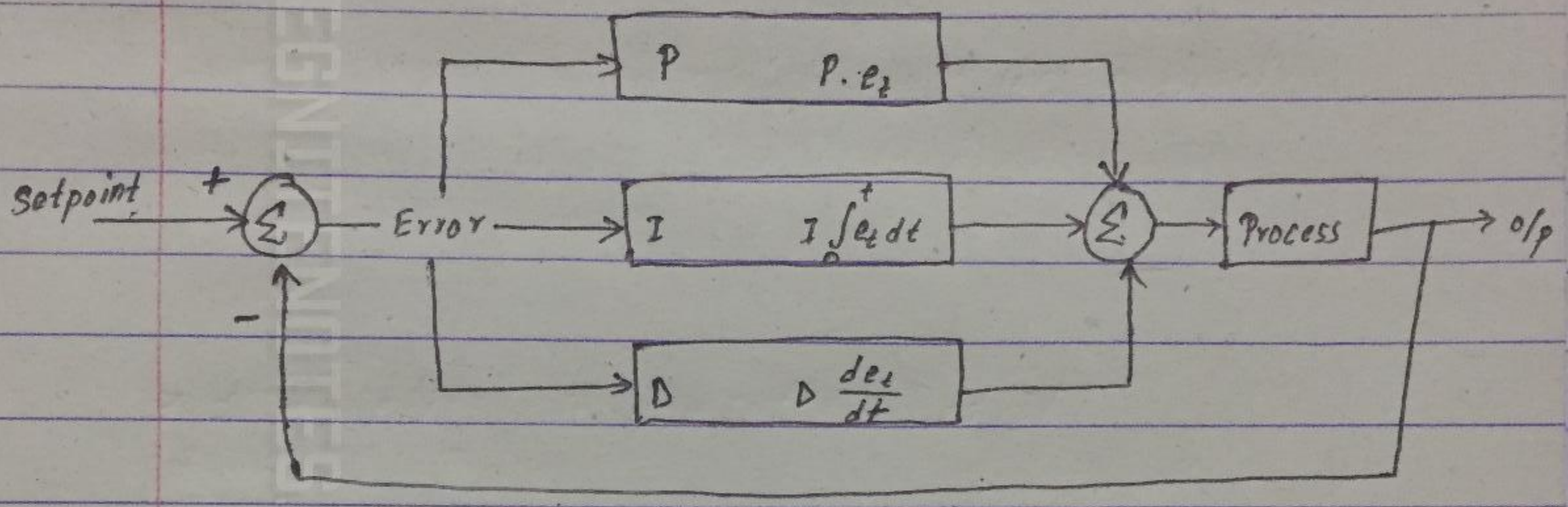
7.10 Benefits of Computer Control

1) Cost

2) Programmability

3) Reproducible

Q) Block Diagram of PID.



Chapter 8

IC Technology

8.1 Full Custom (VLSI) IC Technology

8.2 Semi Custom (ASIC) IC Technology

8.3 Programming Logic Device (PLD) IC Technology

8.1 IC Manufacturing Steps

IC is a semiconductor wafer in which various active and passive components along with external connections are fabricated in extremely tiny silicon chip

The various steps for IC manufacturing are as follows :-

1) Wafer Production

2) Photolithography

3) Doping

4) Metallization

5) Assembly and packaging

1) Wafer Production

Wafer is a round slice of semiconductor material such as silicon. In IC, Silicon wafer is used.

→ Purified polycrystalline Si is created from sand.
→ It is heated to produce molten liquid.
→ A small piece of solid Si is dipped into molten liquid
→ Solid silicon is slowly pulled from the melt.
→ The liquid cools to form single crystal.
→ A thin round wafer is cut using wafer slicer.
→ The wafer surface is smothened by polishing.
→ It is cleaned and dried using high purity low particle chemicals.

2) Photolithography

Photolithography is the process of printing a pattern of mask into the silicon wafer. It is carried out using light sensitive photoresist and controlled exposure to light.

Positive photolithography prints a pattern that is same as the pattern on the mask.

Negative photolithography prints a pattern that is opposite of the pattern as that on the mask.

3> Doping

To alter electrical character of silicon, atom with one less electron and atom with one more electron are introduced in to the area. The impurity atoms in semiconductor material are moved at high temperature.

4> Metallization

It is used to create contact with Si and to make interconnections on chip. A thin layer of aluminum is deposited over the wafer.

5> Assembly and packaging

Each wafer consists of many chips. These chips are separated and packaged by scribing and cleaving.

A diamond saw is used to cut wafer into chips. The tested and verified chip is mounted into a package. It is encapsulated for protection.

Design Manufacture

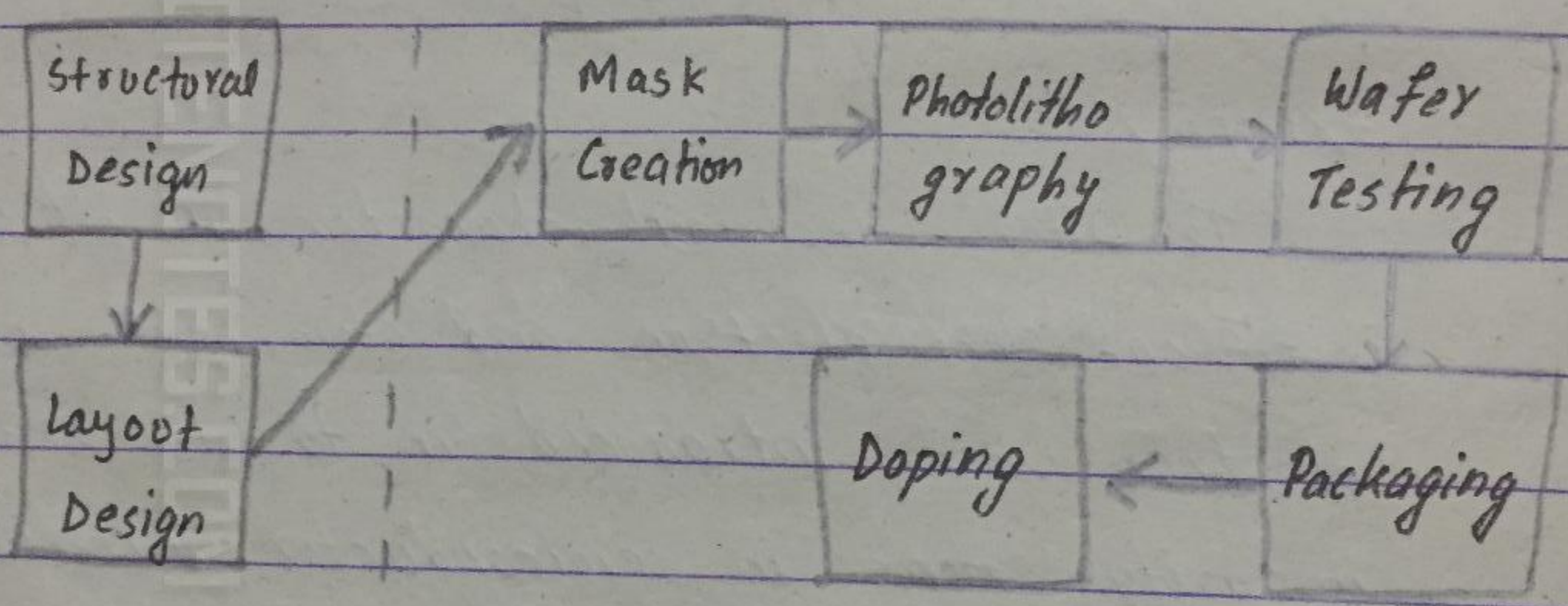


Fig. IC Manufacturing Steps

→ Wafer condition before

1) Surface includes film composition, bare surface and reflectivity.

2) It affects:

a) photoresist-to-wafer adhesion

b) alignment accuracy

c) linewidth resolution

d) exposure settings

e) bake time

→ Wafer condition after

a) resist coated wafer

b) patterned resist layer

c) withstand etching

d) withstand ion implanting

→ Steps of Photolithography

1) Surface Preparation

→ It increases adhesion of photoresist material to the substrate.

→ Dehydration bake removes water from substrate by baking at temp. of 200°C to 400°C for 30 to 60 minutes

→ The substrate is then cooled and coated.

→ Adhesion promoters like Hexamethyl disilizane (HMDS) are used to react chemically with surface and replace -OH with organic function group.

2) Photoresist Coating

→ A thin uniform coating of photoresist is accomplished by spin coating.

→ The photoresist in liquid form is poured onto the wafer and then spun at a high speed to produce desired film.

→ Thickness is affected by spin speed, time and volume of resist.

3) Soft Bake

→ The film contain about 20-40% by weight solvent.
→ It involves drying or removing of excess solvent.
→ It helps to stabilize the resist film.
→ It improves adhesion and uniformity
→ It is baked in oven at 95°C for 30 mins.

4) Alignment and Exposure

→ A photo mask with pattern on one side is aligned over the wafer.
→ A UV light (12mW) is exposed to the surface.
→ It transfers the mask image to the resist coated wafer.
→ The exposure time affects critical dimension. With increase in expose time, CD increase.

5) Develop

→ Soluble areas of photoresist are dissolved by developer chemical.
→ A visible patterns appear on wafer.

6) Hard bake

→ It hardens the final resist image.

→ Hard bake is done at temp. of 110°C for about 30 minutes.

→ It improves adhesion.

7) Develop inspection

→ The resist patterned wafer is inspected for particles, defects, CD, linewidth resolution and overlay accuracy.

8) Etching

→ It is the selective removal of upper layer of wafer.

→ It is performed either by using wet chemicals (acids) or in a dry plasma environment.

9) Strip

→ It is the process of removing photoresist.

→ Wet acid strip or dry plasma strip.

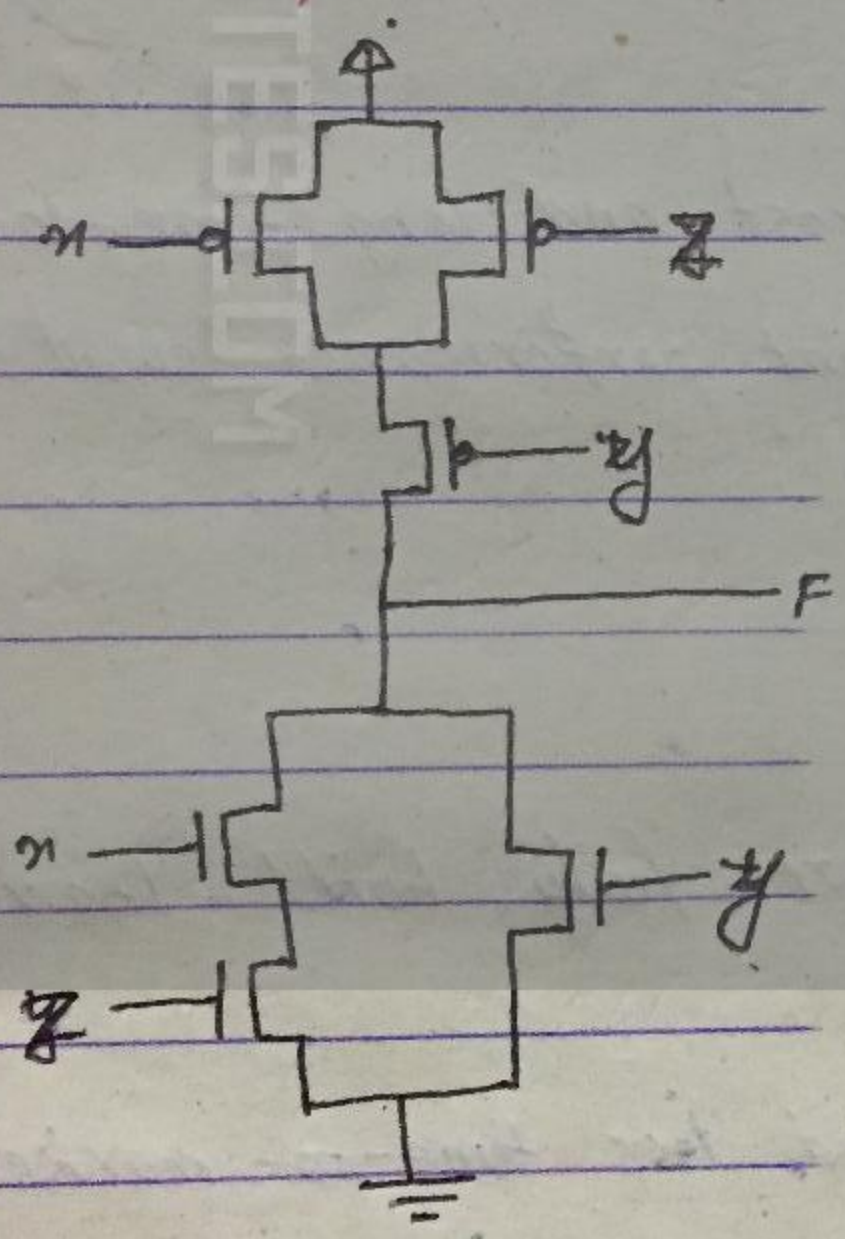
10> Final Inspection

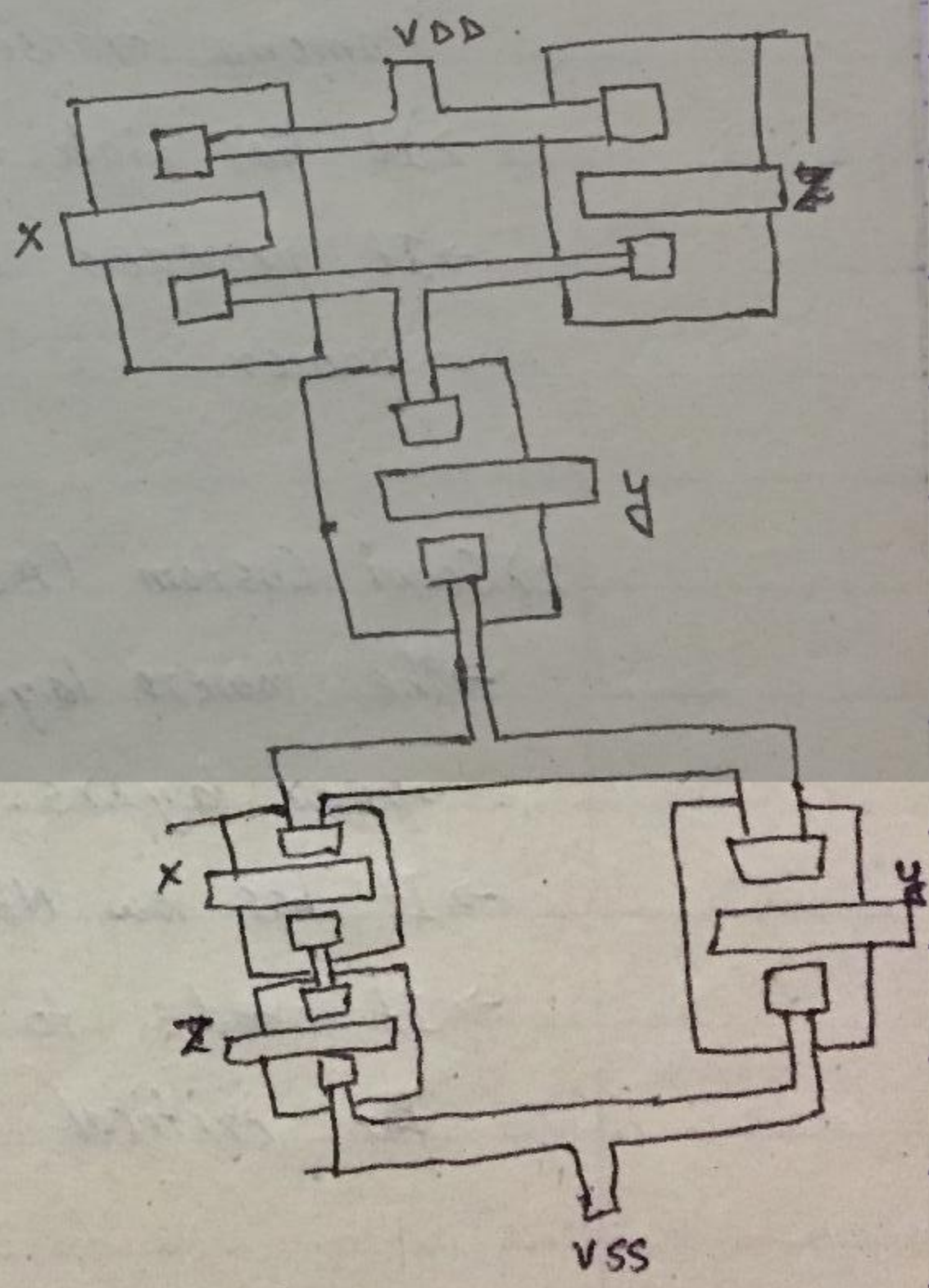
→ Complete removal of photoresist.

→ Pattern on wafer to be correct.

→ Defects, particles, step height and CD are checked.

Q. Show top-down view of circuit $F = xzty$ on an IC.





3.2 Types of IC Technology

1> Full Custom (VLSI)

→ All layers are optimized for a particular ES implementation.

→ It involves placement of transistors to minimize interconnection length, sizing transistor to optimize signal transmission and routing wires among transistors.

→ It has high NRE cost and long time-to-market.

→ It provides excellent performance, small size and power.

2> Semi Custom (ASIC)

→ The lower layers are fully built leaving the upper layers.

→ It has low NRE and less time-to-market.

→ It needs to integrate with full custom IC for critical regions of design.

3) Programming Logic Device (PLD)

→ All layers already exists with a programmable circuit.

→ It involves creation or destruction of connection between wires that connect gates.

→ It has very low NRE and instant time-to-market.

→ FPGA is a complex PLD that offers general connectivity among blocks of logic.

Chapter 9

8051 Microcontroller

1) Continuously toggles the value of port 0.

```
#include<reg51.h>
void main(void)
{
while(1)
  {
     P0 = 0x00;
     P0 = 0xFF;
  }
}
```

```
loop: MOV A, #00H
      MOV P0, A
      MOV A, #0FFH
      MOV P0, A
      SJMP loop
```

2) Get data from P0 and set to P1.

```
#include <reg51.h>
void main (void)
{   P0 = 0xFF;
    P1 = 0x00;
while(1)
{
    P1 = P0;
}
}
```

```
MOV   A, #0FFH
MOV   P0, A
MOV   A, #00H
MOV   P1, A
loop: MOV   A, P0
      MOV   P1, A
      SJMP  loop
```

3) Create 50% duty cycle on bit 0 of port 1.

```
#include<reg51.h>
sbit mybit = P1^0;
void delay(unsigned int n)
{ unsigned int x, y;
  x=0;
  y=0;
  while(x<n)
  { while(y<1275)
    { y++;
    }
    x++;
  }
}
void main(void)
{ while(1)
  {
    mybit = ~mybit;
    delay(250);
  }
}
```

```
loop: SETB  P1.0
      LCALL DELAY
      CLR   P1.0
      LCALL DELAY
      SJMP  loop
```

4) Get status of switch at P1.0 and sent it to LED at P2.7

```
#include <reg51.h>
sbit in_bit = P1^0;
sbit out_bit = P2^7;
void main(void)
{
while (1)
{
   out_bit = in_bit;
}
}
```

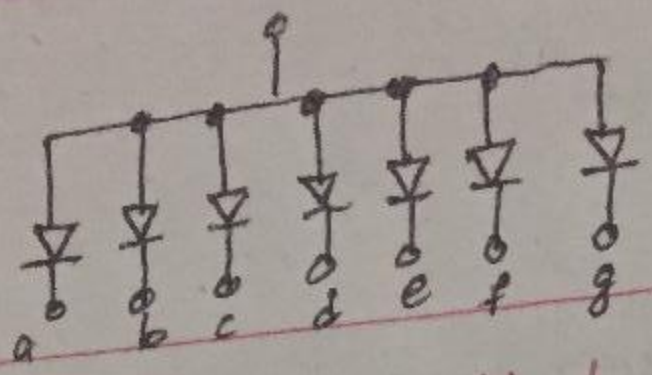
```
        SETB   P1.0
        CLR    P2.7
loop:   MOV    C, P1.0
        MOV    P2.7, C
        SJMP   loop
```

5) Blink 8 LEDs connected at Port 2.

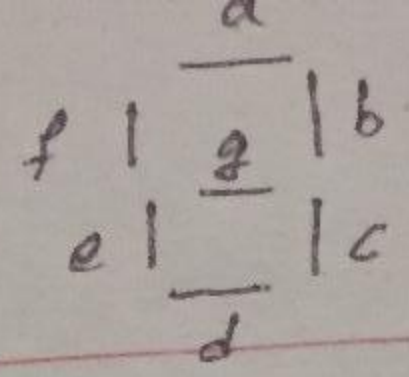
```
#include <reg51.h>
void main(void)
{
   P2 = 0X00;
   while (1)
   {
      P2 = 0X00;
      P2 = 0XFF;
   }
}
```

```
      MOV A, #00H
      MOV P2, A
loop: MOV A, #00H
      MOV P2, A
      MOV A, #0FFH
      MOV P2, A
      SJMP loop
```

Common Anode



ON → 0
OFF → 1



6) 7 segment display to show 0 to 9.

```
MOV A, #00H
MOV P2, A
MOV P2, #C0
ACALL DELAY
MOV P2, #F9
ACALL DELAY
MOV P2, #A4
ACALL DELAY
MOV P2, #B0
ACALL DELAY
MOV P2, #99
ACALL DELAY
MOV P2, #92
ACALL DELAY
MOV P2, #82
ACALL DELAY
MOV P2, #F8
ACALL DELAY
MOV P2, #80
ACALL DELAY
MOV P2, #90
ACALL DELAY
```

```
#include <reg51.h>
void main(void)
{ P2 = 0x00;
  P2 = 0xC0;
  Delay (200);
  P2 = 0xF9;
  Delay (200);
  P3 = 0xA4;
  Delay (200);
  .
  .
  .
}
```

7) 7 segment display to show 99 to 00.

```
#include<reg51.h>
void main (void){
{ int i, j;
int array[10];
array[0] = 0xE0;
array[1] = 0xF9;
array[2] = 0xA4;
array[3] = 0xB0;
array[4] = 0x99;
array[5] = 0x92;
array[6] = 0x82;
array[7] = 0xF8;
array[8] = 0x80;
array[9] = 0x90;
P1 = 0x00;
P2 = 0x00;
for(i=9; i>=0; i--)
{
P1 = array[i];
for(j=9; j>=0; j--)
{
P2 = array[j];
}
}
}
```

8) Sum of two 8-bit BCD in RAM at 50H and 51H and store BCD sum at RAM 52H and 53H.

```
~~MOV R0, #50H~~
MOV A,50H
MOV B, 51H
CLR R7
ADD A, B
DA A
JNC next
INC R7
next: MOV 52H, R7
MOV 53H, A
```

```
#include <reg51.h>
void main (void)
{
P0 = 0x50;
P1 = 0x51;
```

Q) Compute precise 5-ms delay.

For 5ms;

$n \times 1.085\,\mu s = 5\,ms$

$\therefore n = 4608$ ~~$= 9C80\,H$~~

~~$\Rightarrow FFFF - x = 9C80 \Rightarrow x = 6377F$~~

$1 + (65535 - 4608) = 60927 + 1$

$= EE00\,H$

```
#include <reg51.h>
void delay (void);
void main (void)
{ while(1)
   {
      delay();
   }
}
void delay (void)
{ TMOD = 0x01;
   TL0 = 0x00;
   TH0 = 0x EE;
   TR0 = 1;
   while (TF0 == 0);
   TR0 = 0;
   TF0 = 0;
}
```

```
DELAY:  MOV TMOD, #01
here:   MOV TL0, #0
        MOV TH0, #0EEH
        SETB TR0
again   JNB TF0, again
        CLR TR0
        CLR TF0
```

Common Anode 7-Segment Display

	dp	g	f	e	d	c	b	a	
0	1	1	0	0	0	0	0	0	= C0
1	1	1	1	1	1	0	0	1	= F9
2	1	0	1	0	0	1	0	0	= A4
3	1	0	1	1	0	0	0	0	= B0
4	1	0	0	1	1	0	0	1	= 99
5	1	0	0	1	0	0	1	0	= 92
6	1	0	0	0	0	0	1	0	= 82
7	1	1	1	1	1	0	0	0	= F8
8	1	0	0	0	0	0	0	0	= 80
9	1	0	0	1	0	0	0	0	= 90

Chapter - 10

VHDL

1) NOT Gate

x	F
0	1
1	0

```
library ieee;
use ieee.std_logic_1164.all;

ENTITY not1 IS
    PORT ( x : IN STD_LOGIC;
           F : OUT STD_LOGIC;
    );
END not1;

ARCHITECTURE behv1 OF not1 IS
BEGIN
    PROCESS (x)
    BEGIN
        IF (x = '1') THEN
            F <= '0';
        ELSE
            F <= '1';
        END IF;
    END PROCESS;
END behv1;
```

x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

2) OR Gate

```
library iece;
use ieee.std_logic_1164.all;

ENTITY or2 IS
  PORT ( x, y : IN STD_LOGIC;
         F : OUT STD_LOGIC;
  );
END or2;

ARCHITECTURE behv1 OF or2 IS
BEGIN
   process (x, y)
   BEGIN
   IF ((X = '0') AND (Y = '0')) THEN
        F <= '0';
   ELSE
        F <= '1';
   ENDIF;
   END process;
END behv1;
```

3) AND Gate

X	Y	F
0	0	0
0	1	0
1	0	0
1	1	1

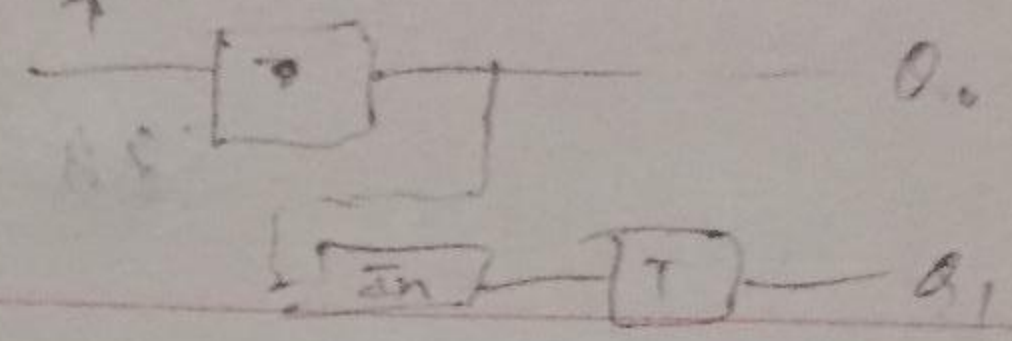
```
library ieee;
use ieee.std_logic_1164.all;

ENTITY and2 IS
  PORT (x, y : IN STD_LOGIC;
        F : OUT STD_LOGIC;
  );
END and2;

ARCHITECTURE behv1 OF and2 IS
BEGIN
    process (x, y)
  BEGIN
      IF ((X='1') AND (Y='1')) THEN
          F <= '1';
      ELSE
          F<='0';
      END IF;
    END process;
END behv1;
```

x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

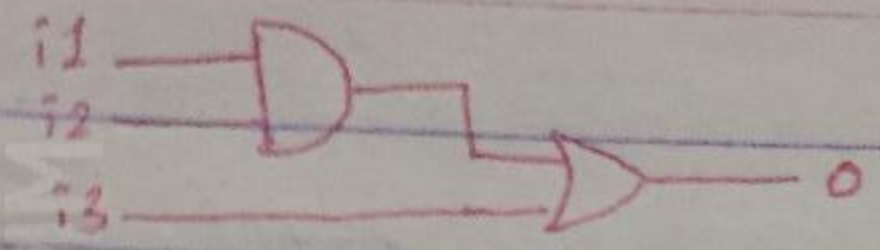
4) XOR Gate

```
library ieee;
use ieee.std_logic_1164.all;

ENTITY xor2 IS
    PORT ( x, y : IN STD_LOGIC;
           F : OUT STD_LOGIC;
    );
END xor2;

ARCHITECTURE behv1 OF xor2 IS
BEGIN
    process (x, y)
    BEGIN
    IF ((X = '0') AND (Y = '0')) THEN
        F <= '0';
    ELSE IF ((X = '1') AND (Y = '1')) THEN
        F <= '0';
    ELSE
        F <= '1';
    END IF;
  END process;
END behv1;
```

79



5)




```
-- component 1
library ieee;
use ieee.std_logic_1164.all
ENTITY or2 IS
    PORT ( x, y : IN STD_LOGIC;
           F1 : OUT STD_LOGIC;
    );
END or2;
ARCHITECTURE behv1 OF or2 IS
BEGIN
    process (x,y)
    BEGIN
        IF ((x = '0') AND (y= '0'))
            F1 <= '0';
        ELSE
            F1 <= '1';
        END IF;
    END process;
END behv1;
```

```
-- component 2
library ieee;
use ieee.std_logic_1164.all;
ENTITY and2 IS
    PORT ( A, B : IN STD_LOGIC;
           F2 : OUT STD_LOGIC;
    );
END and2;
ARCHITECTURE behv2 OF and2 IS
BEGIN
    process (A, B)
    BEGIN
        F2 <= A AND B;
    END process;
END behv2;
```

```
--Top level circuit
library ieee;
use ieee.std_logic_1164.all;
use work.all;
ENTITY comb_ckt IS
PORT (i1, i2, i3 : IN STD_LOGIC;
      O : OUT STD_LOGIC;
);
END comb_ckt;
ARCHITECTURE behv OF comb_ckt IS
   COMPONENT and2 IS
      PORT (A, B : IN STD_LOGIC;
            F2 : OUT STD_LOGIC;
      );
   END COMPONENT;
   COMPONENT or2 IS
      PORT (X, Y : IN STD_LOGIC;
            F1 : OUT STD_LOGIC;
      );
   END COMPONENT;
   SIGNAL wire : STD_LOGIC;
```

```
BEGIN
    Gate1 : and2 PORT MAP (A => i1, B => i2, F2 => wire);
    Gate2 : or2 PORT MAP (x => wire, y => i3, F1 => o);
END behv;
```

6) MUX

```
ENTITY mux IS
PORT (I3, I2, I1, I0 : IN STD_LOGIC_VECTOR (2 DOWNTO 0)
      S : IN STD_LOGIC_VECTOR (1 DOWN TO 0);
      O : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
);
END mux;
ARCHITECTURE behv OF mux IS
BEGIN
    proces (I3, I2, I1, I0, S)
    BEGIN
        CASE S IS
            WHEN "00" => O <= I0;
            WHEN "01" => O <= I1;
            WHEN "10" => O <= I2;
            WHEN "11" => O <= I3;
            WHEN OTHERS => O <= "ZZZ";
        END CASE;
    END process;
END behv;
```

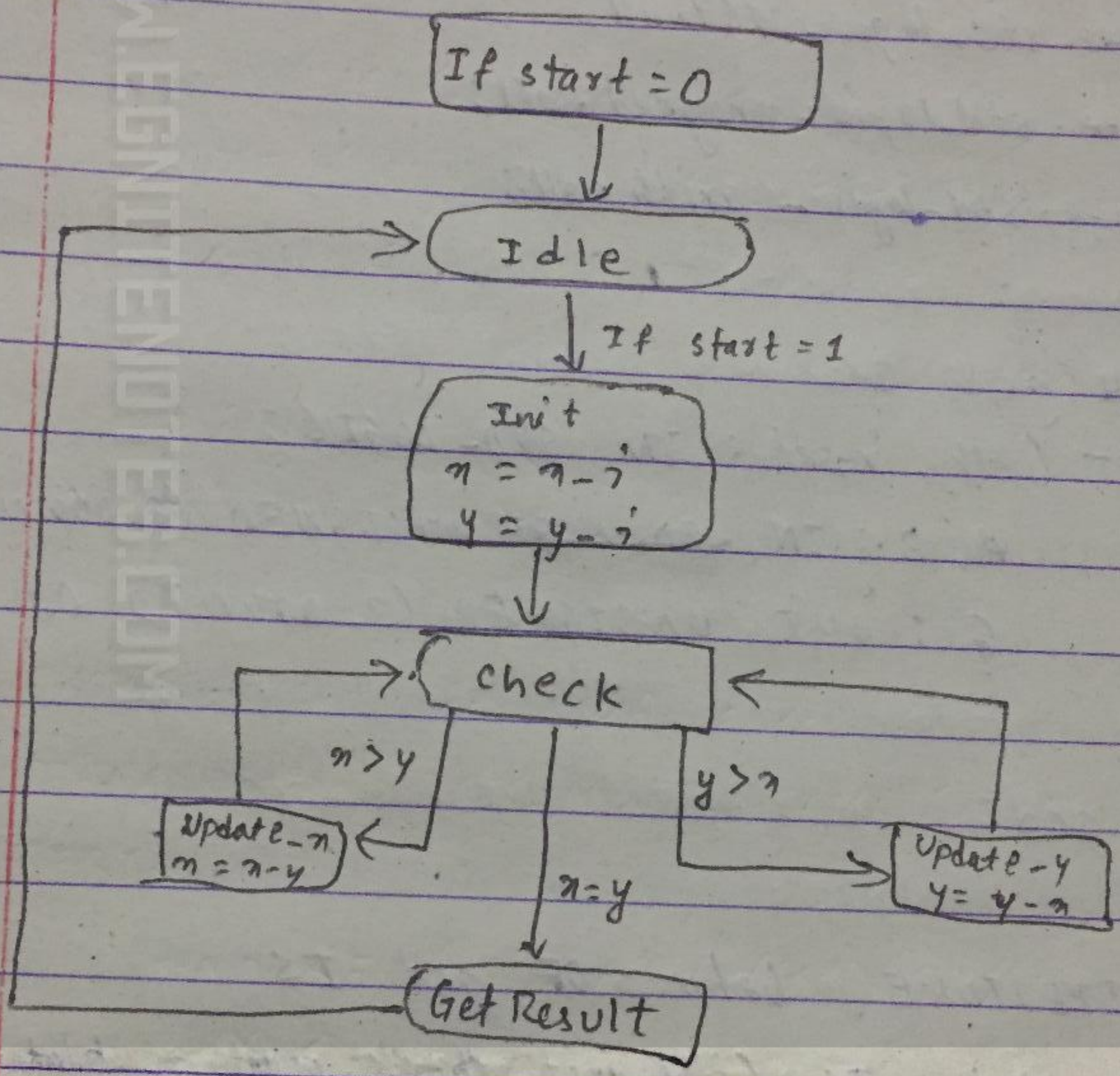
7) DECODER

```
ENTITY decoder IS
    PORT (I : IN STD_LOGIC_VECTOR (1 DOWNTO 0)
          O : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
    );
END decoder;

ARCHITECTURE behv OF decoder IS
BEGIN
    process (I)
    BEGIN
        CASE I IS
            WHEN "00" => O <= "0001";
            WHEN "01" => O <= "0010";
            WHEN "10" => O <= "0100";
            WHEN "11" => O <= "1000";
            WHEN OTHERS => O <= "zzzz";
        END CASE;
    END process;
END behv;
```

8) ADDER

```
ENTITY adder IS
    GENERIC (n : natural := 2);
    PORT ( A, B : IN STD_LOGIC_VECTOR (n-1 DOWNTO 0);
           C : OUT STD_LOGIC;
           S : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0)
    );
END adder

ARCHITECTURE behv OF adder IS
SIGNAL temp : STD_LOGIC_VECTOR (n DOWNTO 0);
BEGIN
    temp <= ('0' & A) + ('0' & B);
    S <= temp (n-1 DOWNTO 0);
    C <= temp (n);
END behv;
```

a) Calculate GCD



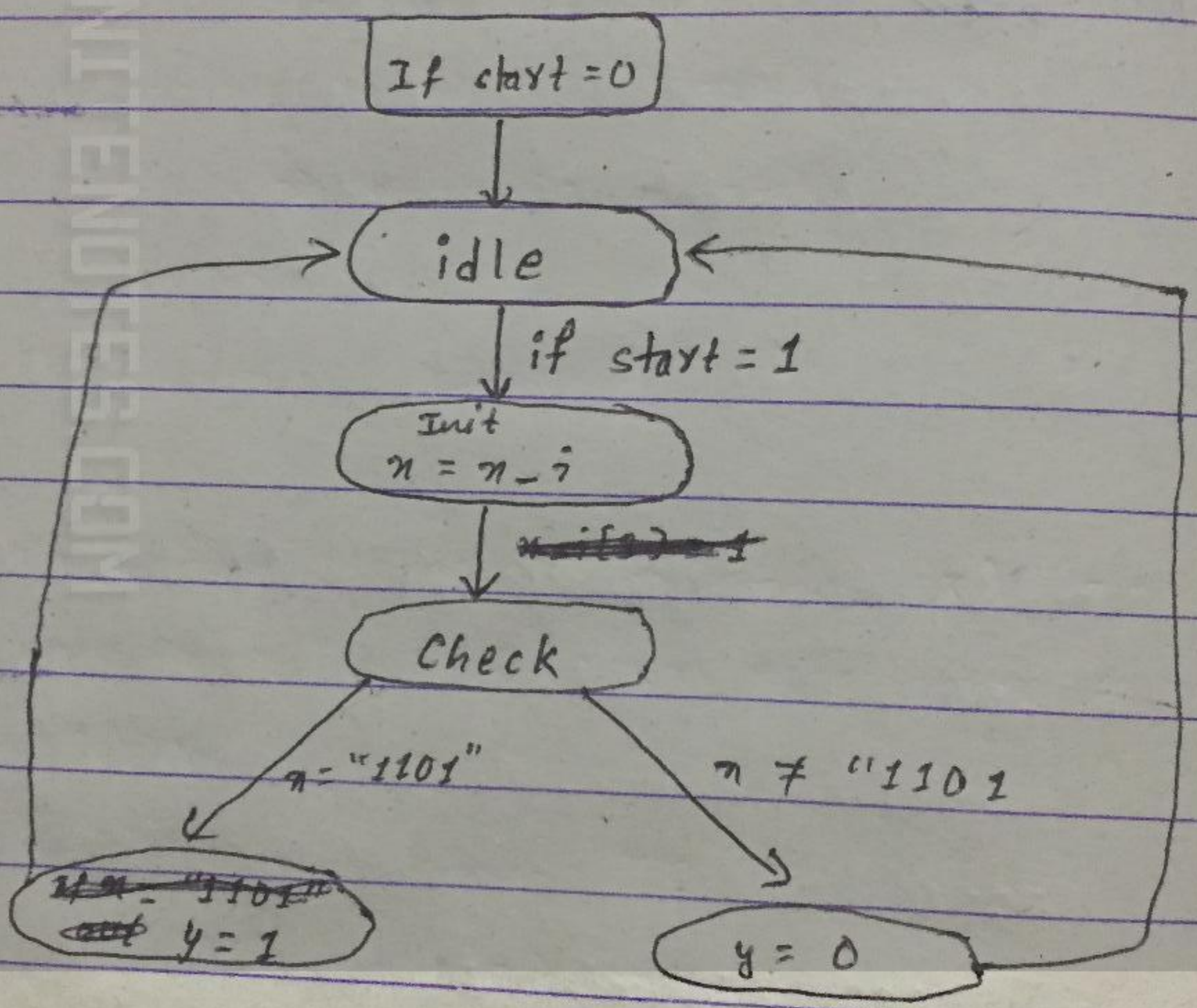
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

ENTITY gcd IS
  PORT (clk, reset : IN STD_LOGIC;
        A, B : IN UNSIGNED (3 DOWNTO 0);
        G : OUT UNSIGNED (3 DOWNTO 0)
  );
END gcd;

ARCHITECTURE behv OF gcd IS
TYPE state IS (idle, init, check, update_x, update_y, get_result);
SIGNAL pr_state : state := idle;
SIGNAL n_state : state;
SIGNAL start : STD_LOGIC := '1';

BEGIN
```

```
sequential: process (clk, reset) IS
BEGIN
    IF (reset = '1') THEN
        pr_state <= idle;
    ELSE IF (clk = '1') THEN
        pr_state <= n_state;
    END IF;
END process sequential;

combinational : process (pr_state, A, B) IS
~~BEGIN~~
    variable tempx, tempy : UNSIGNED(3 DOWNTO 0) := '0000';
    BEGIN
    CASE pr_state IS
    WHEN idle =>
        IF (start = '1') THEN
            n_state <= init;
            start <= '0';
        ELSE
            n-state <= idle;
        END IF
```

```
WHEN  init =>
      temp x <= A;
      temp y <= B;
      n_state <= check;
WHEN  check =>
   IF (temp x = temp y) THEN
      n_state <= get_result;
   ELSEIF (temp_x > temp y) THEN
      n_state <= update_x;
   ELSE
      n_state <= update_y;
   END IF
WHEN  update_x =>
      temp x := temp_x - temp y;
      n_state => check;
When  update_y =>
      temp y := tempy - temp x;
      n_state => check;
When  get_reuIf =>
      G <= temp x;
      n_state <= idle;
```

```
            start <= '1';
        END CASE;
    END process combinational;
END behav;
```

10) Sequence Detector for string "1101", that outputs '1' when input matches the string.



```
library ieee
use ieee.std_logic_1164.all;
use std_logic_arith.all;
use std_logic_unsigned.all;

ENTITY seq_det IS
PORT ( x : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
       y : OUT STD_LOGIC;
       clk : IN STD_LOGIC;
       reset : IN STD_LOGIC
);
END seq_det;

ARCHITECTURE behv OF seq_det IS
TYPE state IS (idle, init, check, pos_out, neg_out);
SIGNAL pr_state : state := idle;
SIGNAL n_state : state;
SIGNAL start : STD_LOGIC := '1';

BEGIN
```

```
sequential : process (clk, reset) IS
BEGIN
    IF (reset = '1') THEN
        pr_state <= idle;
    ELSE IF (clk = '1') THEN
        pr_state <= n_state;
    END IF;
END process sequential;

combinational : process ( pr_state, x ) IS
    variable temp n : STD_LOGIC_VECTOR (3 DOWNTO 0);
    BEGIN
    CASE pr_state IS
    WHEN idle =>
        IF (start = '1') THEN
            n_state <= init;
            start <= '0';
        ELSE
            n_state <= idle;
```

```
WHEN init =>
    tempx <= X"0";
    n_state <= check;
WHEN check =>
    If (tempx = "1101") THEN
        y <= '1';
    ELSE
        y <= '0';
    END IF
    n_state <= idle;
    start <= '1';
END CASE;
END process combinational;
END behv;
```

11> T- Flip Flop

T	Q	Qnext
0	0	0
0	1	1
1	0	1
1	1	0

```
library ieee;
use ieee.std_logic_1164.all;

ENTITY t_ff IS
 PORT ( T : IN STD_LOGIC;
        clk : IN STD_LOGIC;
        Q : OUT STD_LOGIC);
END t_ff;

ARCHITECTURE behv OF t_ff IS
SIGNAL tmp : STD_LOGIC;
BEGIN
     process (clk)
BEGIN
     IF clk = '1' THEN
          IF T = '0' THEN
               tmp <= tmp;
          ELSE
               tmp <= NOT (tmp);
          END IF;
     END IF;
END process;
```

Q <= temp ;

END behv ;

12> JK Flip Flop

	J	K	Q_{next}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

J = 0, K = 0 ; $Q_n = Q$

J = 1, K = 1 ; $Q_n = \bar{Q}$

J = 0, K = 1 ; $Q_n = 0$

J = 1, K = 0 ; $Q_n = 1$

13> D Flip Flop

Q = D

14> SR Flip Flop

S = 0, R = 0 ; $Q_n = Q$

S = 1, R = 1 ; Q_n = unknown

S = 0, R = 1 ; $Q_n = 0$

S = 1, R = 0 ; $Q_n = 1$