DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

COURSE FILE

SUBJECT	:	EMBEDDED SYSTEM DESIGN
FACULTY	:	J. SRAVANTHI
REGULATION	:	R16
BATCH	:	2017
YEAR	:	IV
SEMESTER	:	II

Signature of the staff

Head of the Department

HEAD OF THE DEPARTMENT
DEPT.OF ELECTRONICS AND COMMUNICATION ENGG.
NARASARAOPETA ENGINEERING COLLEGE
NARASARAOPET-522 601

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

S.No	CONTENTS
1	Institute, Department Vision and Mission
2	Programme Educational Objectives and Programme Specific Outcomes
3	Program Outcomes
4	Bloom's Taxonomy levels
5	Course Objectives & Course Outcomes
6	Course Information Sheet
7	Academic calendar
8	Time tables
9	Syllabus copy
10	Lesson Plan
11	CO-PO& PSO Mapping and assessment
12	Web references & other pedagogical initiates details
13	Student's Roll list
14	Hand Written / Printed Lecture Notes / Material given to the Students
15	Power Point Presentation Slides
16	Mid & Assignment Examination Question Papers with scheme and solutions (for problems)
17	Unit wise important questions
18	Previous University Question Papers
19	Missing Topics(Course gaps) and Topics beyond Syllabus
20	Remedial/corrective actions

Submitted By

Approved By

HEAD OF THE DEPARTMENT
DEPT.OF ELECTRONICS AND COMMUNICATION ENGG.
NARASARAOPETA ENGINEERING COLLEGE
NARASARAOPET-522 601

Institute Vision and Mission

Department of Electronics and Communication Engineering

Institute Vision and Mission

Vision

To emerge a Centre of excellence in technical education with a blend of effective student centric teaching learning practices as well as research for the transformation of lives and community,

Mission

M1:Provide the best class infra- structure to explore the field of engineering and research

M2:Build a passionate and a determined team of faculty with student centric teaching, imbibing experiential, innovative skills

M3:Imbibe lifelong learning skills, entrepreneurial skills and ethical values in students for addressing societal problems

PRINCIPAL

PRINCIPAL
NARASARAOPETA ENGINEERING COLLEGE
(AUTONOMOUS)
NARASARAOPET - 522 601
Guntur (Dist.), A.P.

Department Vision and Mission

Department of Electronics and Communication Engineering

Vision and Mission of the Department

DEPARTMENT VISION

To emerge as a **centre of excellence** in Electronics and Communication Engineering through **student centric education** and **research focus** to cater the current and future needs of **society**.

DEPARTMENT MISSION

M1: To provide best infrastructure for empowering the students with quality education to motivate them towards higher studies and **research**

M2: To provide qualified and experienced faculty for **student centric teaching** in order to mould the students as successful professionals in modern Electronics industry

M3: To inculcate leadership qualities, professional etiquette, **ethical values** and **social** responsibilities

Head of the Department

HEAD OF THE DEPARTMENT
DEPT.OF ELECTRONICS AND COMMUNICATION ENGG.
NARASARAOPETA ENGINEERING COLLEGE
NARASARAOPET-522 601

Department of Electronics and Communication Engineering

PROGRAM EDUCATIONAL OBJECTIVES:

PEO1: To train the students to design and analyze the electronic circuits and equipment for societal benefits.

PEO2: To inculcate in the students the desire for lifelong learning to obtain thorough knowledge in their chosen fields and also to motivate them for higher studies/research.

PEO3: To train the students so that they can effectively perform the duties assigned to them as team leaders or project managers in the industry/organization with ethical and moral values.

PROGRAM SPECIFIC OUTCOMES:

PSO1: Analyze and Design Analog and Digital circuits for a given specification and function.

PSO2: Design a variety of Electronic Systems for applications including Signal Processing, Communications, Computer Networks and Control Systems.

HOD

HEAD OF THE DEPARTMENT
DEPT.OF ELECTRONICS AND COMMUNICATION ENGG
NARASARAOPETA ENGINEERING COLLEGE
NARASARAOPET-522 601

PROGRAM OUTCOMES

Department of Electronics and Communication Engineering

Narasaraopeta Engineering College follows the **Program Outcomes (PO)** as defined by NBA Engineering Graduates will be able to:

PO1	**1. Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	**2. Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences
PO3	**3. Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	**4.Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions
PO5	**5. Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	**6. The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice
PO7	**7. Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	**8. Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice
PO9	**9. Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings
PO10	**10. Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions
PO11	**11. Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments
PO12	**12. Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Head of the Department

HEAD OF THE DEPARTMENT
DEPT.OF ELECTRONICS AND COMMUNICATION ENGG.
NARASARAOPETA ENGINEERING COLLEGE
NARASARAOPET-522 601

BLOOMS TAXONOMY LEVELS

Bloom's Taxonomy

Level	Description	Verbs
create	Produce new or original work	Design,assemble, construct, conjecture, develop, formulate, author, investigate
evaluate	Justify a stand or decision	appraise, argue, defend, judge, select, support, value, critique, weigh
analyze	Draw connections among ideas	differentiate, organize, relate, compare, contrast, distinguish, examine, experiment, question, test
apply	Use information in new situations	execute, implement, solve, use, demonstrate, interpret, operate, schedule, sketch
understand	Explain ideas or concepts	classify, describe; discuss, explain, identify, licate, recognize, report, select, translate
remember	Recall facts and basic concepts	define, duplicate, list, memorize, repeat, state

COURSE OBJECTIVES AND OUTCOMES

Department of Electronics and Communication Engineering

COURSE OBJECTIVES:

1. The method of designing a real time systems
2. Implementation and testing an embedded system
3. Summarize special concerns that real-time systems present and how these concerns are addressed

COURSE OUTCOMES: Students are able to

CO1: Recall the fundamentals of Core of the Embedded System. [K1]

CO2: Define Process model and technologies to design an Embedded system. [K1]

CO3: Demonstrate the customization of Hardware/Software. [K2]

CO4: Delineate the unique Characteristics of Embedded Systems. [K2]

CO5: Make use of system design techniques to develop Hardware/Software for Embedded systems [K3]

CO6: Develop an Embedded system with real time constraints. [K3]

Faculty Signature

COURSE INFORMATION SHEET

Narasaraopeta Engineering College
(Autonomous)
Yallmanda(Post), Narasaraopet- 522601
Department of Electronics and Communication Engineering

COURSE INFORMATION SHEET

PROGRAMME: B.Tech Electronics and Communication Engineering	
COURSE: EMBEDDED SYSTEM DESIGN	Semester : VIII CREDITS: 3
COURSE CODE: R16EC4211 REGULATION: R16	COURSE TYPE (CORE /ELECTIVE / BREADTH/ S&H): CORE
COURSE AREA/DOMAIN: ELECTRONIC CIRCUITS	PERIODS: 6 Per Week.

COURSE PRE-REQUISITES:

C.CODE	COURSE NAME	DESCRIPTION	SEM
R16EC4102	Microcontrollers and Embedded Systems	Know the building blocks of typical embedded system, memory devices and supporting devices.	VII

COURSE OUTCOMES:

SNO	Course Outcome Statement
CO1	Recall the fundamentals of Core of the Embedded System. [K1]
CO2	Define Process model and technologies to design an Embedded system. [K1]
CO3	Demonstrate the customization of Hardware/Software. [K2]
CO4	Delineate the unique Characteristics of Embedded Systems. [K2]
CO5	Make use of system design techniques to develop Hardware/Software for Embedded systems [K3]
CO6	Develop an Embedded system with real time constraints. [K3]

SYLLABUS:

UNIT	DETAILS
I	INTRODUCTION Embedded systems overview, Design Challenges, Processor Technology, IC Technology, Design Technology, Trade-offs.
II	CUSTOM SINGLE PURPOSE PROCESSORS: HARDWARE Combinational logic, Sequential logic, Custom single-purpose processor Design, RT-level Custom single-purpose processor Design, Optimizing Custom single-purpose processor.
III	GENERAL PURPOSE PROCESSORS: SOFTWARE Basic Architecture, Operation, Programmer's view, Development Environment, Application specific Instruction set processors, Selecting a processor, General purpose processor design.
IV	MEMORY Common memory types, composing memory, memory hierarchy and cache. INTERFACING: Arbitration, Multilevel bus architectures, advanced communication principles.
V	STATE MACHINE AND CONCURRENT PROCESS MODELS Models vs Languages, Basic state machine model, HCFSM and state charts language, program state machine model, Role of appropriate model and language, concurrent process model,

	communication among processes, Synchronization among processes, Implementation, Real time systems.
VI	**IC TECHNOLOGY** Full custom IC technology, Semi-custom IC technology, PLD IC technology. DESIGN TECHNOLOGY: Automation: Synthesis, Verification: Hardware/Software co-simulation, Reuse: Intellectual property cores, Design process models.

TEXT BOOKS	
T	**BOOK TITLE/AUTHORS/PUBLISHER**
T1	Frank Vahid, Tony D. Givargis "Embedded System Design: A Unified Hardware/Software Introduction", Wiley India Edition, 2002.
REFERENCE BOOKS	
R	**BOOK TITLE/AUTHORS/PUBLISHER**
R1	KVKK Prasad, "Embedded / Real Time Systems", Dreamtech Press, 2005.
R2	Shibu K.V, "Introduction to Embedded Systems ", Tata Mc Graw Hill, 2009.
R3	David E. Simon, "An Embedded Software Primer", Pearson Education, Eighth Impression 2009.

WEB SOURCE REFERENCES:

S.No.	Name	URL
1	Embedded System Design and Foundations	https://drive.google.com/file/d/198yvRnbtwdRRnSjVoO9p3CxvtweyHzhM/view
2	Modern Embedded Computing Designing	https://drive.google.com/file/d/10Meik77CrPZyTLXyYi4xpQuFEtXZola1/view
3	Embedded Systems	https://drive.google.com/file/d/1IM7auyswcLIHb4WnFAZLzim7bBh81uCp/view
4	Embedded System Design: A Unified Hardware/Software Approach	http://dsp-book.narod.ru/ESDUA.pdf
5	Introduction to Embedded Systems	https://ptolemy.berkeley.edu/books/leeseshia/releases/LeeSeshia_DigitalV1_08.pdf

DELIVERY/INSTRUCTIONAL METHODOLOGIES:

☐Chalk & Talk	☐ PPT	☐Active Learning
☐Web Resources	☐ Students Seminars	☐Case Study
☐Blended Learning	☐ Quiz	☐Tutorials
☐Project based learning	☐NPTEL/MOOCS	☐ Simulation
☐Flipped Learning	☐Industrial Visit	☐Model Demonstration
☐Brain storming	☐Role Play	☐Virtual Labs

MAPPING CO'S WITH POs & PSOs

COs	POs													
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
C423.1	2													
C423.2	2	3	1										3	2
C423.3	3	3	3	2									3	2
C423.4	2	2	2	3									3	2
C423.5	2	2											3	2
C423.6		2	3										3	2
C423	2.2	2.4	2.25	2.5									3	2

ATTAINING COURSE WITH POs & PSOs

Course	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
C423	2.2	2.4	2.25	2.5									3	2

COURSE OUTCOME RUBRIC (ASSESMENT PER STUDENT):					
ASSESMENT TOOL WITH WEIGHTAGE	METHOD	ATTAINMENT LEVEL 3 (EXCELLENT)	ATTAINMENT LEVEL 2 (GOOD)	ATTAINMENT LEVEL 1 (AVERAGE)	ATTAINMENT LEVEL 0 (POOR)
Internal tests (40%)	Direct	Student secured ≥ 60% marks of allocated marks for that CO	Student secured ≥ 60% and < 50% marks of allocated marks for that CO	Student secured ≥ 50% and <40% marks of allocated marks for that CO	Student secured < 40% marks of allocated marks for that CO
Assignments (20%)	Direct	Student secured ≥ 80% marks allocated for that CO	Student secured ≥ 70% and <80% marks allocated for that CO	Student secured ≥ 60% and <70% marks allocated for that CO	Student secured < 60% of marks allocated for that CO
End Semester Examination (30%)	Direct	Student secured grades A*&S* in External Exam	Student secured grades C*&B* in External Exam	Student secured grades D*&E* in External Exam	Student secured grades F* in External Exam
Course end Survey (10%)	Indirect	Student selected option	Student selected option	Student selected option	Student selected option
* Grade Definition: S: >= 90%; A: 80%-89%; B: 70%-79%; C: 60%-69%; D: 50%-59%; E: 40%-49%; F: <40%					

Course Coordinator

Module Coordinator

Head of the Department

HEAD OF THE DEPARTMENT
DEPT.OF ELECTRONICS AND COMMUNICATION ENGG.

ACADEMIC CALENDAR

Narasaraopeta Engineering College (Autonomous)
Kotappakonda Road, Yellamanda (P.O), Narasaraopet- 522601, Guntur District, AP.

ACADEMIC CALENDAR

(B.Tech. 2019, 2018 and 2017 admitted batches, Academic Year 2020-21)

2019 Batch 2nd Year 1st Semester, 2018 Batch 3rd Year 1st Semester and 2017 Batch 4th Year 1st Semester			
Description	**From Date**	**To Date**	**Duration**
Commencement of Class Work	02-11-2020		4 Weeks
1st Spell of Instructions	**02-11-2020**	**30-11-2020**	
I Mid examinations	01-12-2020	05-12-2020	1 Week
2nd Spell of Instructions	**07-12-2020**	**20-02-2021**	11 Weeks
II Mid examinations	22-02-2021	27-02-2021	1 Week
Preparation & Practicals	01-03-2021	06-03-2021	1 Week
Semester End Examinations	08-03-2021	20-03-2021	2 Weeks
2019 Batch 2nd Year 2nd semester, 2018 Batch 3rd Year 2nd Semester and 2017 Batch 4th Year 2nd Semester			
Commencement of Class Work	22-03-2021		7 Weeks
1st Spell of Instructions	**22-03-2021**	**08-05-2021**	
I Assignment Test	**12-04-2021**	**17-04-2021**	
II Assignment Test	**26-04-2021**	**30-04-2021**	
I Mid examinations	10-05-2021	15-05-2021	1 Week
2nd Spell of Instructions	**17-05-2021**	**03-07-2021**	7 Weeks
III Assignment Test	**31-05-2021**	**05-06-2021**	
IV Assignment Test	**21-06-2021**	**26-06-2021**	
II Mid examinations	05-07-2021	10-07-2021	1 Week
Preparation & Practicals	12-07-2021	17-07-2021	1 Week
Semester End Examinations	19-07-2021	31-07-2021	2 Weeks

PRINCIPAL

TIME TABLES

(AUTONOMOUS)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

IV B.Tech., II Semester, ECE-A Class Time Table for the A.Y. 2020-21

Room No.: 3317 w.e.f: 24-03-2021

DAY	1 9:10 to 10:00	2 10:00 to 10:50	10:50 to 11:00	3 11:00 to 11:50	4 11:50 to 12:40	12:40 to 1:30	5 1:30 to 2:20	6 2:20 to 3:10	7 3:10 to 4:00
MON	CMC	ESD	BREAK	WSN	ESD	LUNCH BREAK	Project Work/ Pracrical Training		
TUE	WSN	ESD		WSN	CMC		Project Work/ Pracrical Training		
WED	ESD	WSN		CMC	WSN		Project Work/ Pracrical Training		
THU	ESD	CMC		WSN	ESD		Project Work/ Pracrical Training		
FRI	WSN	CMC		ESD	CMC		Project Work/ Pracrical Training		
SAT	CMC	ESD		WSN	CMC		Project Work/ Pracrical Training		

ESD : Embedded System Design — Ms. J. Sravanthi

WSN : Wireless Sensor Networks — Ms. SK. Ayesha

CMC : Cellular and Mobile Communications — Dr. Sk. Bajid Vali

Project Work/ Pracrical Training : Ms. J. Sravanthi Ms. SK. Ayesha Dr. Sk. Bajid Vali, Dr V.Venkat Rao

HEAD OF THE DEPARTMENT
(Dr. V. VENKATA RAO)

PRINCIPAL
(Dr. M. SREENIVASA KUMAR)

HEAD OF THE DEPARTMENT
DEPT.OF [illegible] COMMUNICATION
NARASARA[illegible] COLLEGE
NARASARAOPET 522 601

PRINCIPAL
NARASARAOPETA ENGINEERING COLLEGE
(AUTONOMOUS)
NARASARAOPET - 522 601.
Guntur (Dist.), A.P.

(AUTONOMOUS)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

IV B.Tech., II Semester, ECE-B Class Time Table for the A.Y. 2020-21

Room No.: 3319 w.e.f: 24-03-2021

DAY	1	2		3	4		5	6	7
	9:10 to 10:00	10:00 to 10:50	10:50 to 11:00	11:00 to 11:50	11:50 to 12:40	12:40 to 1: 30	1:30 to 2:20	2:20 to 3:10	3:10 to 4:00
MON	WSN	CMC	BREAK	ESD	WSN	LUNCH BREAK	Project Work/ Pracrical Training		
TUE	CMC	WSN		CMC	ESD		Project Work/ Pracrical Training		
WED	CMC	CMC		WSN	ESD		Project Work/ Pracrical Training		
THU	WSN	ESD		CMC	WSN		Project Work/ Pracrical Training		
FRI	ESD	WSN		CMC	ESD		Project Work/ Pracrical Training		
SAT	ESD	WSN		CMC	ESD		Project Work/ Pracrical Training		

ESD : Embedded System Design Mr. P. Shankar

WSN : Wireless Sensor Networks Mr. Ch. Adi Babu

CMC : Cellular and Mobile Communications Mr. J.V.K.Ratnam

Project Work/ Pracrical Training : J.V.K.Ratnam, Dr. Ayesha, Mr. Ch. Adi Babu, Dr V.Venkat Rao

Mr. P. Shankar, Dr. Ayesha, Dr. SK. SD.Basha

HEAD OF THE DEPARTMENT
(Dr. V. VENKATA RAO)

PRINCIPAL
(Dr. M. SREENIVASA KUMAR)

HEAD OF THE DEPARTMENT
DEPT.OF ELECTRONICS AND COMMUNICATION
NARASARAOPETA ENGINEERING COLLEGE
NARASARAOPET-522 601

PRINCIPAL
NARASARAOPETA ENGINEERING COLLEGE
(AUTONOMOUS)
NARASARAOPET - 522 601
Guntur (Dist.), A.P.

(AUTONOMOUS)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

IV B.Tech., II Semester, ECE-C Class Time Table for the A.Y. 2020-21

Room No.: 3322 w.e.f: 24-03-2021

DAY	1	2		3	4		5	6	7
	9:10 to 10:00	10:00 to 10:50	10:50 to 11:00	11:00 to 11:50	11:50 to 12:40	12:40 to 1: 30	1:30 to 2:20	2:20 to 3:10	3:10 to 4:00
MON	CMC	ESD	BREAK	WSN	CMC	LUNCH BREAK	Project Work/ Pracrical Training		
TUE	WSN	ESD		WSN	CMC		Project Work/ Pracrical Training		
WED	ESD	CMC		WSN	CMC		Project Work/ Pracrical Training		
THU	ESD	CMC		WSN	CMC		Project Work/ Pracrical Training		
FRI	CMC	ESD		ESD	WSN		Project Work/ Pracrical Training		
SAT	ESD	WSN		WSN	ESD		Project Work/ Pracrical Training		

ESD : Embedded System Design — Dr. N. Veda Kumar

WSN : Wireless Sensor Networks — Dr. S. MeghaSyam Reddy

CMC : Cellular and Mobile Communications — Dr. K. Anjaneyulu

Project Work/ Pracrical Training :Dr. K. Anjaneyulu Dr. N. Veda Kumar , Dr. D. Subba Rao, Dr. R. Siva Kumar, Dr. J. Narendra Babui Dr V.Venkat Rao Dr. S. MeghaSyam Reddy

HEAD OF THE DEPARTMENT
(Dr. V. VENKATA RAO)

HEAD OF THE DEPARTMENT
DEPT.OF ELE... ...COMMUNICATION
NARASARAO... ...EERING COLLEGE
NARASARAOPET-522 601

PRINCIPAL
(Dr. M. SREENIVASA KUMAR)

PRINCIPAL
NARASARAOPETA ENGINEERING COLLEGE
(AUTONOMOUS)
NARASARAOPET - 522 601.
Guntur (Dist.), A.P.

(AUTONOMOUS)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

IV B.Tech., II Semester, ECE-D Class Time Table for the A.Y. 2020-21

Room No.: 3323 w.e.f: 24-03-2021

DAY	1 9:10 to 10:00	2 10:00 to 10:50	10:50 to 11:00	3 11:00 to 11:50	4 11:50 to 12:40	12:40 to 1: 30	5 1:30 to 2:20	6 2:20 to 3:10	7 3:10 to 4:00
MON	WSN	CMC	BREAK	ESD	WSN	LUNCH BREAK	Project Work/ Pracrical Training		
TUE	CMC	WSN		ESD	ESD		Project Work/ Pracrical Training		
WED	WSN	ESD		CMC	ESD		Project Work/ Pracrical Training		
THU	CMC	WSN		ESD	WSN		Project Work/ Pracrical Training		
FRI	ESD	CMC		WSN	CMC		Project Work/ Pracrical Training		
SAT	WSN	CMC		ESD	CMC		Project Work/ Pracrical Training		

ESD	: Embedded System Design	Mr. N. Rajeev Reddy
WSN	: Wireless Sensor Networks	Dr. S. MeghaSyam Reddy
CMC	: Cellular and Mobile Communications	Dr. K. Laxma Reddy

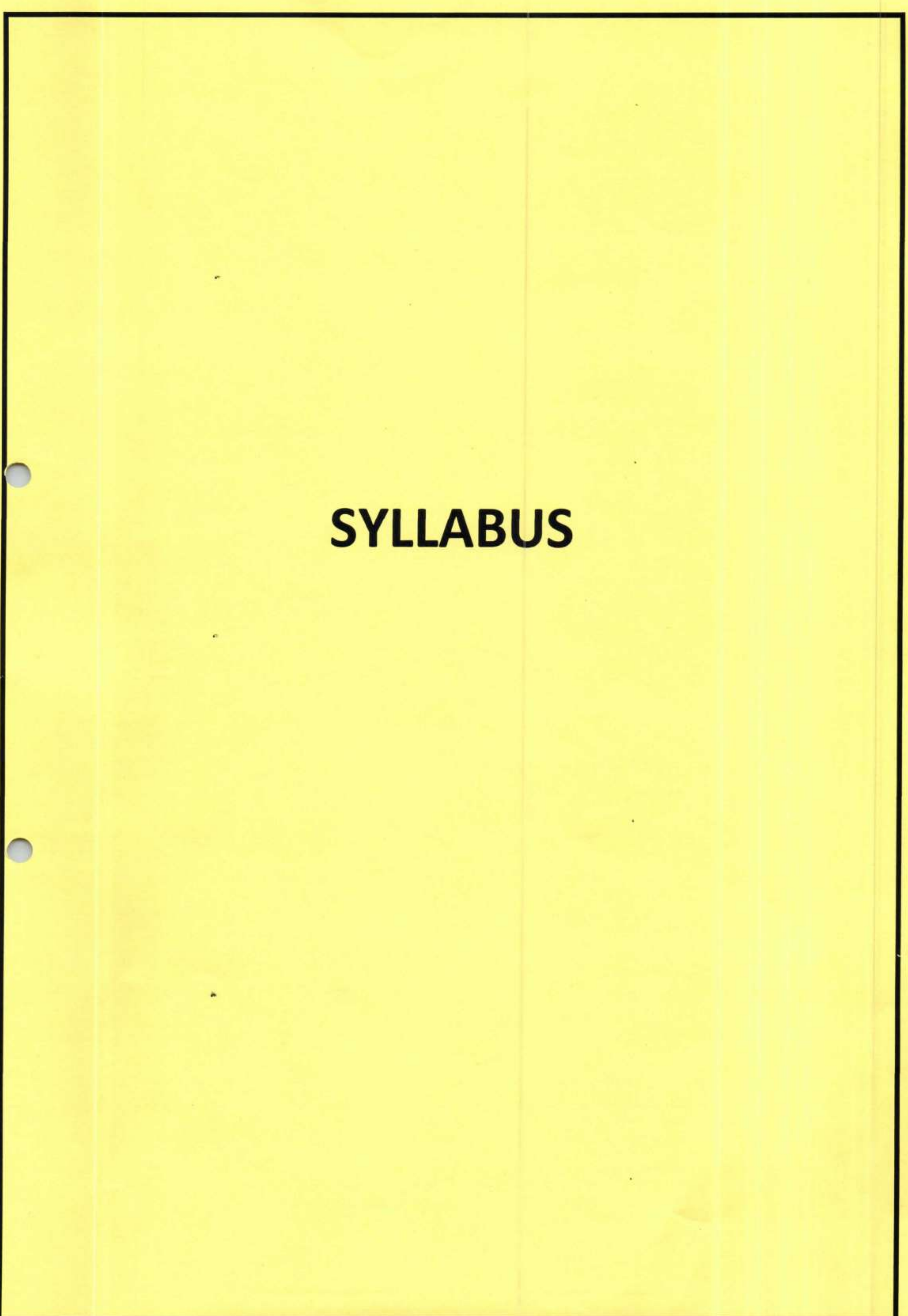
Project Work/ Pracrical Training : Dr. K. Laxma Reddy Mr. N. Rajeev Reddy , Dr V.Venkat Rao Dr. G. Lakshmi Narayana

HEAD OF THE DEPARTMENT
(Dr. V. VENKATA RAO)

PRINCIPAL
(Dr. M. SREENIVASA KUMAR)

HEAD OF THE DEPARTMENT
DEPT.OF ELE[illegible] COMMUNICATION
NAR[illegible]EERING COLLEGE
NARASARAOPET-522 601

PRINCIPAL
NARASARAOPETA ENGINEERING COLLEGE
(AUTONOMOUS)
NARASARAOPET - 522 601.
Guntur (Dist.), A.P.



SYLLABUS

IV B.TECH- II-SEMESTER	L	T	P	INTERNAL MARKS	EXTERNAL MARKS	TOTAL MARKS	CREDITS
	4	0	0	40	60	100	3
EMBEDDED SYSTEM DESIGN *(Professional Elective – V)*							

COURSE OBJECTIVES:

1. The method of designing a real time systems
2. Implementation and testing an embedded system
3. Summarize special concerns that real-time systems present and how these concerns are addressed

COURSE OUTCOMES:

After completion of the course, the student will be able to

CO1: Recall the fundamentals of Core of the Embedded system.

CO2: Define process models and technologies to design an Embedded system.

CO3: Demonstrate the customization of Hardware/Software.

CO4: Delineate the unique characteristics of Embedded systems.

CO5: Make use of system design techniques to develop Hardware/Software for embedded systems.

CO6: Develop an embedded system with real time constraints.

UNIT-I: INTRODUCTION

Embedded systems overview, Design Challenges, Processor Technology, IC Technology, Design Technology, Trade-offs.

UNIT-II: CUSTOM SINGLE PURPOSE PROCESSORS: HARDWARE

Combinational logic, Sequential logic, Custom single-purpose processor Design, RT-level Custom single-purpose processor Design, Optimizing Custom single-purpose processor.

UNIT-III: GENERAL PURPOSE PROCESSORS: SOFTWARE

Basic Architecture, Operation, Programmer's view, Development Environment, Application specific Instruction set processors, Selecting a processor, General purpose processor design.

UNIT-IV: MEMORY

Common memory types, composing memory, memory hierarchy and cache.

INTERFACING: Arbitration, Multilevel bus architectures, advanced communication principles.

UNIT-V: STATE MACHINE AND CONCURRENT PROCESS MODELS

Models vs Languages, Basic state machine model, HCFSM and state charts language, program state machine model, Role of appropriate model and language, concurrent process model, communication among processes, Synchronization among processes, Implementation, Real time systems.

UNIT-VI: IC TECHNOLOGY

Full custom IC technology, Semi-custom IC technology, PLD IC technology.

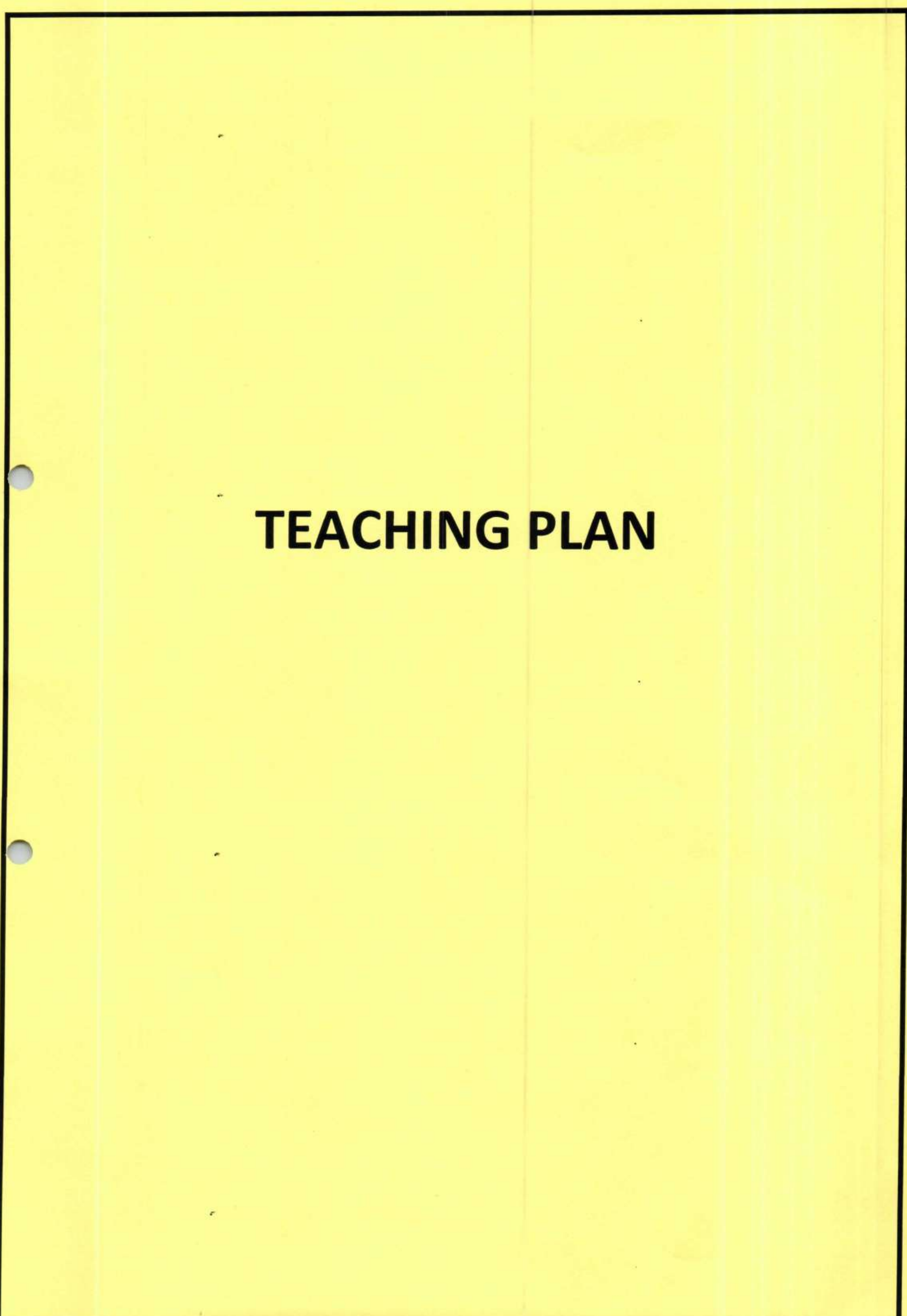
DESIGN TECHNOLOGY: Automation: Synthesis, Verification: Hardware/Software co-simulation, Reuse: Intellectual property cores, Design process models.

Text Book:

1. Frank Vahid, Tony D. Givargis "Embedded System Design: A Unified Hardware/Software Introduction", Wiley India Edition, 2002.

Reference Books:

1. KVKK Prasad, "Embedded / Real Time Systems", Dreamtech Press, 2005.
2. Shibu K.V, "Introduction to Embedded Systems ", Tata Mc Graw Hill, 2009.
3. David E. Simon, "An Embedded Software Primer", Pearson Education, Eighth Impression 2009.



TEACHING PLAN

Narasaraopeta Engineering College
(Autonomous)
Yallmanda(Post), Narasaraopet- 522601

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
TEACHING PLAN

Subject Code	Course Title (Regulation)	Sem	Branch	Contact Periods/Week	Sections
R16EC4211	Embedded System Design	VIII	Electronics and Communication Engineering	6	A, B, C & D

COURSE OUTCOMES: Students are able to

CO1: Recall the fundamentals of Core of the Embedded System. [K1]
CO2: Define Process model and technologies to design an Embedded system. [K1]
CO3: Demonstrate the customization of Hardware/Software. [K2]
CO4: Delineate the unique Characteristics of Embedded Systems. [K2]
CO5: Make use of system design techniques to develop Hardware/Software for Embedded systems [K3]
CO6: Develop an Embedded system with real time constraints. [K3]

Unit No	Outcome		Topics/Activity	Ref Text book	Total Periods	Delivery Method
1	CO 1. Recall the fundamentals of the Embedded System. [K1]		**UNIT–I: INTRODUCTION**			
		1.1	Embedded systems overview	T1, R1	10	Chalk & Talk, PPT, Active Learning & Tutorial
		1.2	Design Challenges	T1, R1		
		1.3	Processor Technology, IC Technology	T1, R1		
		1.4	Design Technology, Trade-offs.	T1		
2	CO2. Define Process model and echnologies to design an Embedded system. [K1]		**UNIT–II:CUSTOM SINGLE PURPOSE PROCESSORS: HARDWARE**			
		2.1	Combinational logic, Sequential logic	T1, R1	10	Chalk & Talk, PPT Tutorial, Active Learning & Case Study
		2.2	Combinational logic, Sequential logic	T1, R1		
		2.3	RT-level Custom single-purpose processor Design	T1, R1		
		2.4	Optimizing Custom single-purpose processor.	T1, R1		
MID I EXAMINATION DURING SIXTH WEEK						
3	CO 3. Demonstrate the customization of Hardware/Software. [K2]		**UNIT–III: GENERAL PURPOSE PROCESSORS: SOFTWARE**			
		3.1	Basic Architecture, Operation	T1, R1		Chalk & Talk, PPT, Tutorial
		3.2	Programmer's view, Development Environment	T1, R1		
		3.3	Application specific Instruction set processors	T1, R1		

		3.4	Selecting a processor, General purpose processor design	T1, R1	10	
4	CO 4. Delineate the unique aracteristics of Embedded Systems. [K2]	UNIT–IV: MEMORY				
		4.1	Common memory types, composing memory	T1, R1	10	Chalk & Talk, PPT & Tutorial.
		4.2	memory hierarchy and cache	T1, R1		
		4.3	INTERFACING: Arbitration	T1, R1		
		4.4	Multilevel bus architectures	T1, R1		
		4.5	Advanced communication principles.	T1		
MID II EXAMINATION DURING TWELTH						
5	CO 5. Make use of system design techniques to develop Hardware/Software for Embedded systems [K3]	UNIT–V: STATE MACHINE AND CONCURRENT PROCESS MODELS				
		5.1	Models vs Languages, Basic state machine model, HCFSM and state charts language	T1, R1	10	Chalk & Talk, PPT, Active Learning & Seminars
		5.2	program state machine model, Role of appropriate model and language	T1, R1		
		5.3	concurrent process model, communication among processes	T1, R1		
		5.4	Synchronization among processes, Implementation, Real time systems.	T1, R1		
6	CO 6. Develop an Embedded system with real time constraints. [K3]	UNIT–VI: IC TECHNOLOGY				
		6.1	Full custom IC technology, Semi-custom IC technology, PLD IC technology.	T1	15	Chalk & Talk, PPT Tutorial, Active Learning & Seminars
		6.2	DESIGN TECHNOLOGY: Automation: Synthesis, Verification	T1		
		6.3	Hardware/Software co-simulation	T1		
		6.4	Reuse: Intellectual property cores, Design process models.	T1		
				Total	65	
END EXAMINATIONS						

Text Books:

T1 Frank Vahid, Tony D. Givargis "Embedded System Design: A Unified Hardware/Software Introduction", Wiley India Edition, 2002.

Reference Books:

R1 KVKK Prasad, "Embedded / Real Time Systems", Dreamtech Press, 2005.

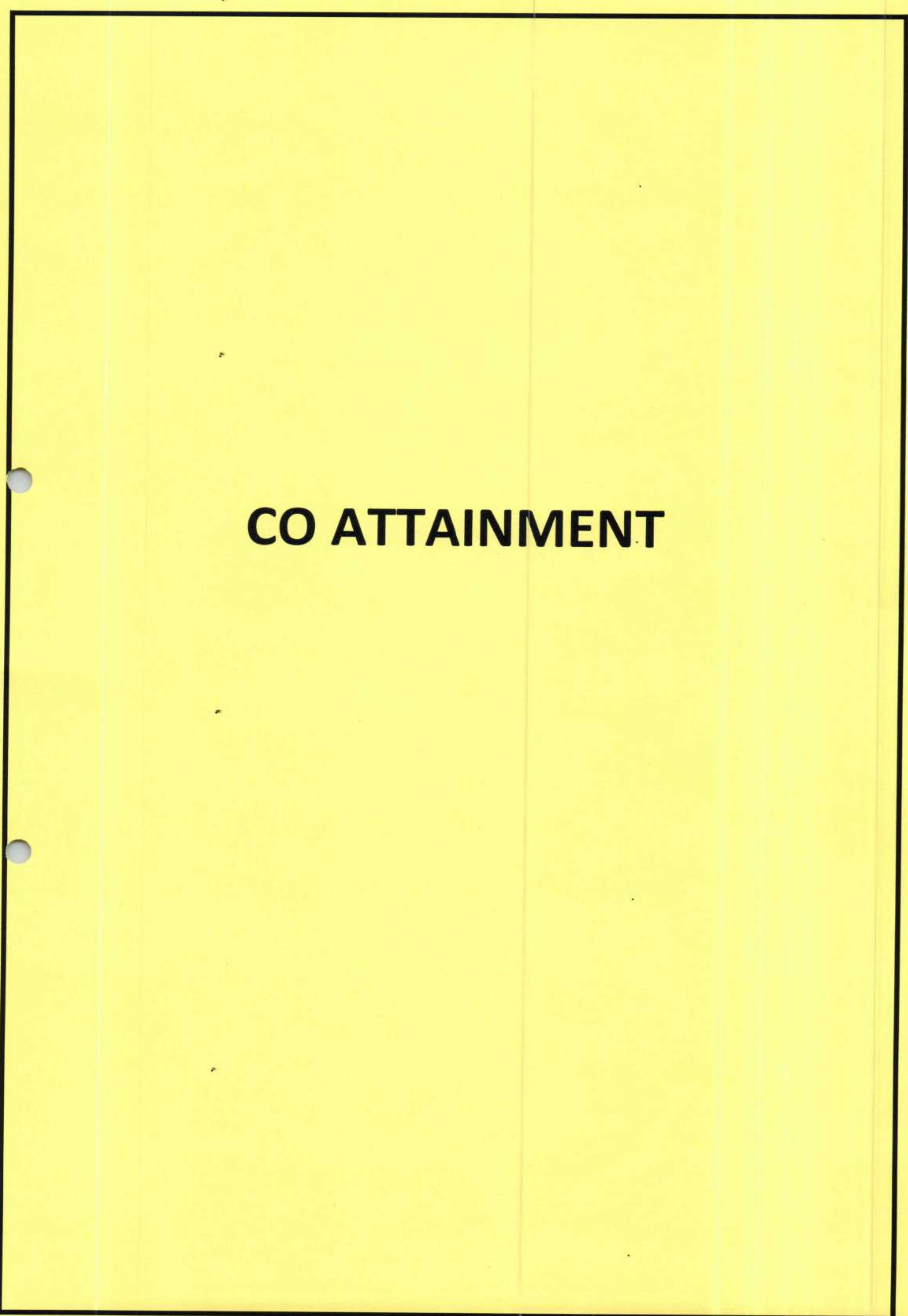
R2 Shibu K.V, "Introduction to Embedded Systems ", Tata Mc Graw Hill, 2009.

R3 David E. Simon, "An Embedded Software Primer", Pearson Education, Eighth Impression 2009

Faculty

HOD

CO-PO& PSO Mapping and assessment

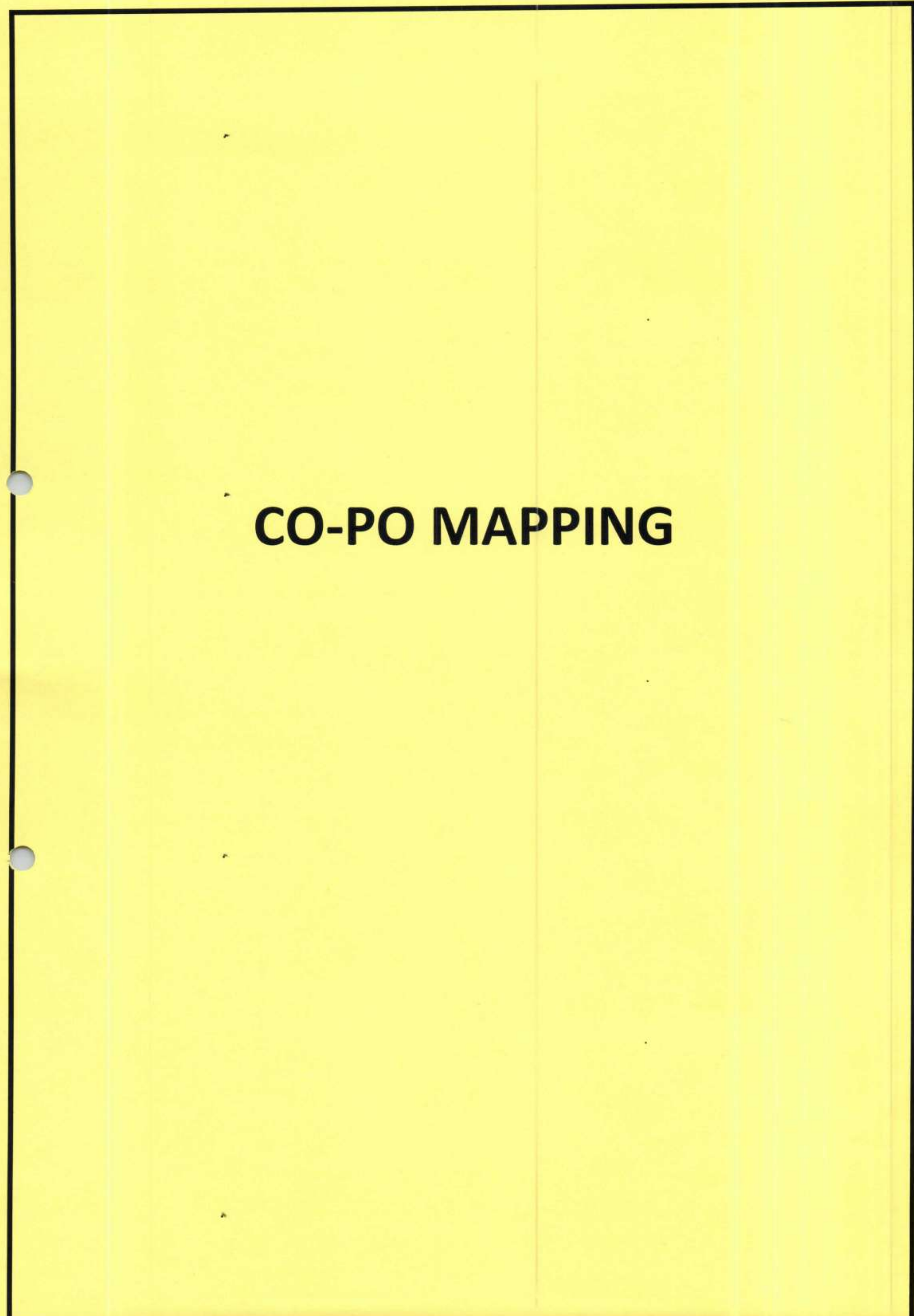


CO ATTAINMENT

CO Attainment through Direct and Indirect Assessment

	CO Attainment Level (Internal)	CO Attainment Level (External)	Direct CO Attainment Level (Internal * 30%) + (External * 70%)	Indirect CO Attainment Level	Total CO Attainment Level (Direct CO Attainment * 90% + Indirect CO Attainment * 10%)
CO1: Recall the fundamentals of Core of the Embedded system	3	3	3	3	3
CO2: Define process models and technologies to design an Embedded system.	3	2	3	3	3
CO3: Demonstrate the customization of Hardware/Software.	3	2	3	3	3
CO4: Delineate the unique characteristics of Embedded systems.	3	2	3	3	3
CO5: Make use of system design techniques to develop Hardware/Software for embedded systems	2	1	2	3	3
CO6: Develop an embedded system with real time constraints.	2	3	3	3	3
					3

Faculty Signature


CO-PO MAPPING

CO-PO Mapping

COs	POs											
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
C423.1	2											
C423.2	2	3	1									
C423.3	3	3	3	2								
C423.4	2	2	2	3								
C423.5	2	2										
C423.6		2	3									
C423	2.2	2.4	2.25	2.5								

Total CO Attainment through Direct & Indirect Assessment

CO Attainment	3

PO Attainment

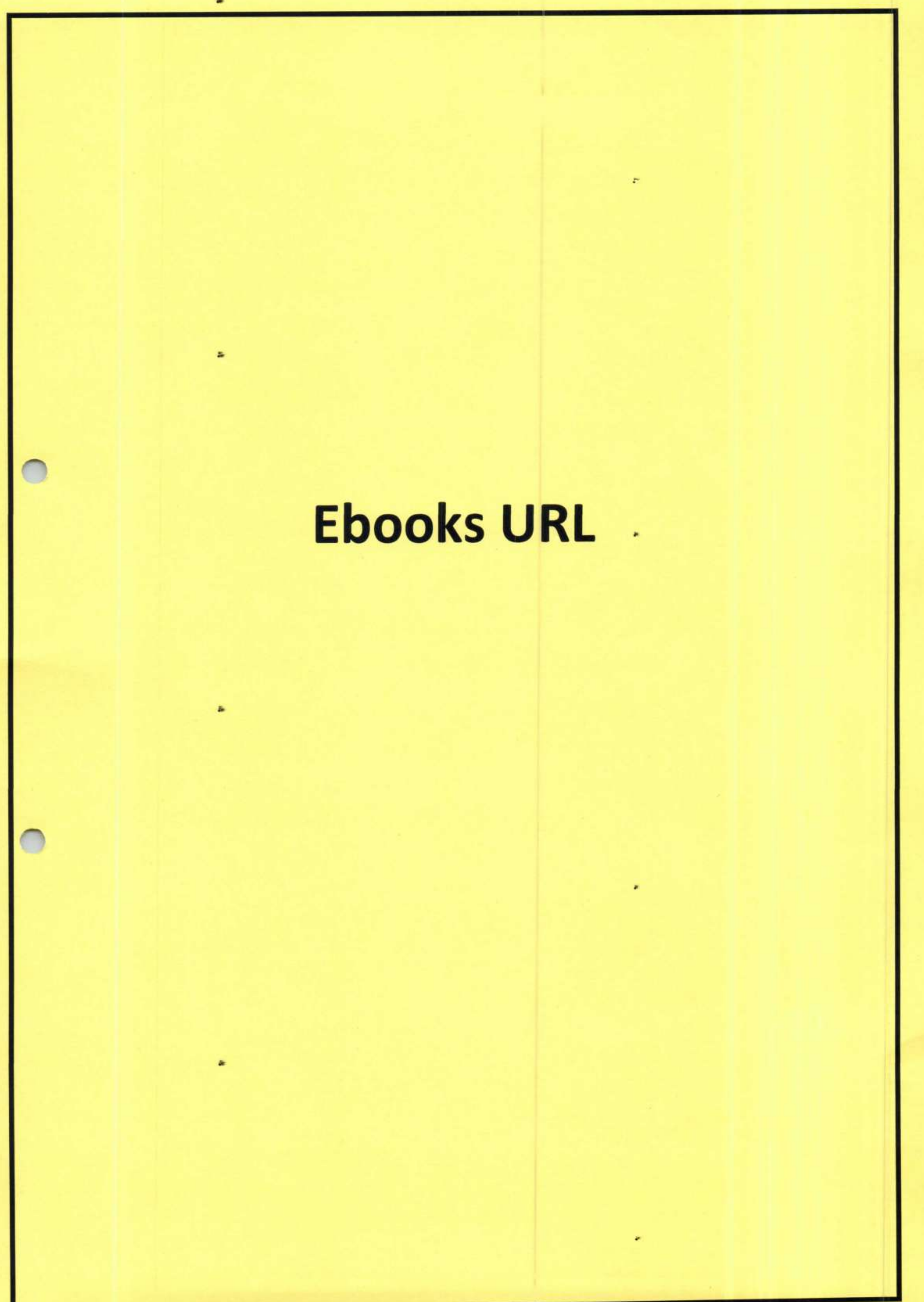
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
PO Attainment	2.2	2.4	2.25	2.5								

1. Copy CO - PO matrix and CO attainment matrix from previous pages and find PO attainment.

2. PO attainment is calculated as per the following formula:

POi * Total CO attainment Level / 3 where 'i' ranges from 1 to 12

Faculty Signature


Ebooks URL

Narsaraopeta Engineering College (Autonomous): Narasaraopet

Department of Electronics and Communication Engineering

Subject: **Embedded System Design (ECE) [4-2]**

S.No.	Name	URL
1	Embedded System Design and Foundations	https://drive.google.com/file/d/198vvRnbtwdRRnSjVoO9p3CxvtweyHzhM/view
2	Modern Embedded Computing Designing	https://drive.google.com/file/d/10Meik77CrPZyTLXyYi4xpQuFEtXZola1/view
3	Embedded Systems	https://drive.google.com/file/d/1IM7auyswcLlHb4WnFAZLzim7bBh81uCp/view
4	Embedded System Design: A Unified Hardware/Software Approach	http://dsp-book.narod.ru/ESDUA.pdf
5	Introduction to Embedded Systems	https://ptolemy.berkeley.edu/books/leeseshia/releases/LeeSeshia_DigitalV1_08.pdf

STUDENT'S

ROLL LIST

BRANCH/SEC - ECE/A		
SL.NO.	H.T.NO.	STUDENT NAME
1	16471A0405	NELAKUDITI VENKATA SIVA SAI PRUDHVI KUMAR
2	17471A0401	KOLLA CHAKRI SAI VIJAYACHANDRA
3	17471A0403	MANAM YASWANTH CHOWDARY
4	17471A0404	CHINTAGUNTLA KALYAN KUMAR
5	17471A0405	YERUVA SUDHEER KUMAR REDDY
6	17471A0406	KOLISETTY BABA SRI RAM KUMAR
7	17471A0407	BATTULA CHANDAN
8	17471A0408	KOTABHATTAR V V S PRATHYUSHA
9	17471A0409	CHERUKULA KASI MAITHRI
10	17471A0410	KANCHETI VINAY
11	17471A0411	YAKKALA NAGA MADAN DATHA KUMAR
12	17471A0412	SANAMPUDI VENKATA NARASIMHA REDDY
13	17471A0413	GADAM RAM BHUPAL REDDY
14	17471A0414	YANDAPALLI SAI VAMSIKRISHNA
15	17471A0415	MAMILLAPALLI SAI RAM
16	17471A0416	NEMALIDINNE VENKATAJAHNAVI
17	17471A0417	NEMALIDINNE VENKATA YASHASWINI

18	17471A0418	MANDALA SAI BHARGAV REDDY
19	17471A0419	MAMIDIPAKA SAI SRIDHAR
20	17471A0420	POTHURI YASWANTH GUPTHA
21	17471A0421	POPURI VENU
22	17471A0422	KALANGI KRISHNA AKHIL
23	17471A0423	DESABOINA PRUDHVISAI
24	17471A0424	CHINNI ESWAR RAO
25	17471A0425	YAKKALA PRATHAP
26	17471A0426	PATHURI SAIPAVAN
27	17471A0427	KOPPURAVURI JEEVAN JITHENDRA
28	17471A0428	SANKARAPU SEKHAR BABU
29	17471A0429	NUTHALAPATI DURGA PRASAD
30	17471A0430	BOGGAVARAPU YASWANTH AMARESH
31	17471A0432	CHANDRAGIRI SAI PRAGNA
32	17471A0433	SYED MANISHA
33	17471A0434	GADDAM VAMSI
34	17471A0435	MINDYALA NAGASAI
35	17471A0436	IRUVANTI SATYA SITA RAMA SASTRY
36	17471A0437	PANGA SRINIVASA RAO

37	17471A0438	POTHRALA RAMANJI
38	17471A0439	PASUPULATI SURESH
39	17471A0440	GUDIPATI CHARITHA
40	17471A0441	SHAIK SALMAN
41	17471A0442	MANDALAPU AKHIL SURYA
42	17471A0443	PABBA VENKATESH NAIDU
43	17471A0444	NANDHYALA LINGA REDDY
44	17471A0445	GANGAVARAPU TEJESWAR REDDY
45	17471A0446	TALLAPANENI VYSHNAVI
46	17471A0448	YELURI NAVYA
47	17471A0449	DORAGACHARLA PAVAN KUMAR REDDY
48	17471A0450	GOPALAM NAVYASRI
49	17471A0451	SHAIK ABTHAB
50	17471A0452	TADIKAMALLA SURESH BABU
51	17471A0453	THUMATI MUKESH CHOWDARY
52	17471A0454	ANEKALLA LAKSHMAN REDDY
53	17471A0455	RAGHUVU VENKAT SIVA RAMA NAGENDRA
54	17471A0456	NANDIKONDA ANJI REDDY
55	17471A0457	KAKUMANU GANESH KRISHNA SAI

56	17471A0459	RAMIDEVI SUMANTH

BRANCH/SEC - ECE/B		
SL.NO.	H.T.NO.	STUDENT NAME
1	17471A0461	PONUGOTI RAMESH
2	17471A0462	MATTRAM VISHNU BABU
3	17471A0463	RAMISETTY RAMCHARAN
4	17471A0464	ANANTHA DURGA
5	17471A0465	KAMMA NAGA SAI RITHVIK
6	17471A0467	DANDE NAGALAKSHMI
7	17471A0468	JANAPATI YASASWINI JAYA BHARATHI SAHITHI
8	17471A0469	JANAPATI SAILAKSHMI SRAVANI
9	17471A0470	KUNISETTY GOPINADH
10	17471A0471	SHAIK MD YASIN
11	17471A0472	RAMISETTI LAKSHMISAITEJA
12	17471A0473	B. MANI DEEPAK
13	17471A0474	BOKKA JOHN VICTOR
14	17471A0475	GODUGUNURI VIJAYA SAI DILEEP KUMAR REDDY
15	17471A0476	GANAPATHI JYOTHI PRAKASH
16	17471A0477	MUNAGAPATI MANOJKUMAR

17	17471A0478	SYED MD GOUSE
18	17471A0479	GOLI SRINIVASARAO
19	17471A0480	PATHI VENKATESWARI
20	17471A0481	VANUKURI HARIVARDHAN VEERA REDDY
21	17471A0482	PANCHUMARTHI DILEEP KUMAR
22	17471A0483	KOLLURU KRISHNA MOHAN
23	17471A0484	RACHUMALLU SASIDHAR
24	17471A0485	KARNATI HEMANTH SAI
25	17471A0486	THUMATI VENKATA SUNIL
26	17471A0487	KOPPURAVURI AKHILA
27	17471A0488	NAIDU RACHANA
28	17471A0489	TELAPROLU PAVAN KALYAN
29	17471A0490	BODEMPUDI SRI HARSHA
30	17471A0491	SHAIK RUKSANA
31	17471A0492	KATTAMURI SATYANARAYANA
32	17471A0493	KOPPULA GANESH REDDY
33	17471A0494	SYED MAHABOOB JANI BASHA
34	17471A0495	PERUMALLA PREETHI KOUMIKA
35	17471A0496	SHAIK JANI BASHA

36	17471A0497	SHAIK MOHAMMED ALTHAF
37	17471A0498	JANGALA KIRAN BABU
38	17471A0499	RAMA CHANDRULA KAVYASRI
39	17471A04A0	MUVVA MANOJ KUMAR
40	17471A04A1	KAKUMANU SUMANTH
41	17471A04A2	YANDAPALLI N V S L MALLIKA BRAMARAMBIKA
42	17471A04A3	TUMMALACHERUVU SAITEJA
43	17471A04A4	JAKKIREDDY KEERTHI
44	17471A04A5	SHAIK AFRID
45	17471A04A6	YERRAMSETTY SAI PAVAN
46	17471A04A7	DESABOYINA HEMARAMACHANDRA VASU
47	17471A04A8	SHAIK TANGEDA CHINA BAJI
48	17471A04A9	KOLLA SIVA HEMANTH
49	17471A04B0	AKULA ASHOK KUMAR
50	17471A04B1	MOHAMMED ZAKIR HUSSAIN KHAN
51	17471A04B2	BALUPUNURI KASU VASU DEVA VENKATA REDDY
52	17471A04B3	GORANTLA SRAVAN KRISHNA
53	17471A04B4	JAMMULA CHANDRIKA
54	17471A04B5	BONDE RAJENDRA

55	17471A04B6	NANNEM VEENA VATSALYA
56	17471A04B7	GOGULA NAVEEN KUMAR
57	17471A04B8	KURANGI MUKUNDA SAI
58	17471A04B9	GOUSE MOMITH BAIG
59	17471A04C0	BUSSI JOSEPH BALA YASWANTH BABU

BRANCH/SEC - ECE/C		
SL.NO.	H.T.NO.	STUDENT NAME
1	17471A04C1	VATTIKONDA SIVA RAMAKRISHNA
2	17471A04C2	GUNDA PRATHYUSHA
3	17471A04C3	ALLA SARATH SAI
4	17471A04C4	GARIKAPATI PAVAN KALYAN
5	17471A04C5	GUTHA VENKAT RAO
6	17471A04C6	PULUKURI SRI PRASANNA
7	17471A04C7	BANDI CHINNAPA REDDY
8	17471A04C8	PINNIKA SRIVANI
9	17471A04C9	MADDI KOTI KIRAN KUMAR
10	17471A04D0	DODDAKULA PRASANTH
11	17471A04D1	SHAIK ARSHAD
12	17471A04D2	PALADUGU SRINIVASULU

18	17471A04D8	GANGASANI ASHOK REDDY
19	17471A04D9	KOSANA PRATAP
20	17471A04E0	G RAGA VENKATA DEEPTHI
21	17471A04E1	POTHURI MANIKANTA
22	17471A04E2	SHAIK NANNU SHAIDA
23	17471A04E3	POLU DIVYA
24	17471A04E4	VIBHARAMPATTAPU MAMATHA
25	17471A04E5	MANDAVA SRI BHARATHI
26	17471A04E6	BHOJANAPALLI TEJASWINI
27	17471A04E7	SHAIK JAVEED
28	17471A04E8	KOLLIKONDA GANGABHAVANI
29	17471A04E9	GUNTA ROHITHA REDDY
30	17471A04F0	LAKSHMISETTY VENKATA SAI VYSHNAVI
31	17471A04F1	RAVULAPALLI SRINU
32	17471A04F2	SADINENI SOWJANYA
33	17471A04F3	GANJI KRANTHI
34	17471A04F4	CHEVALA PRITHVI RAJ
35	17471A04F5	DEVARAPALLI NAGA POOJA SAI SRI
36	17471A04F6	MEKAPOTHULA GOPI KRISHNA
37	17471A04F7	REPALLE PRATHYUSHA

38	17471A04F8	KOMMANABOYINA NAGA ANIL
39	17471A04F9	BATCHU DURGA BHAVANI
40	17471A04G0	DIRISALA SRAVANI
41	17471A04G1	KOMIREDDY MANJU BHARGAVI
42	17471A04G2	POLA VENKATA MALLIKHARJUNA RAO
43	17471A04G3	KOLIPAKULA DEVI CHAMUNDESWARI
44	17471A04G4	BOBBA PRASANTH
45	17471A04G5	G JAGADEESH CHANDRA BOSE
46	17471A04G6	GUNTU NAVEEN CHOWDARY
47	17471A04G8	YELLANURU JAHNAVI
48	17471A04G9	MAMIDIPAKA NAGASUSHMA
49	17471A04H0	DUGGARAJU GOWTHAMY
50	17471A04H1	P RAMA KRISHNA
51	17471A04H2	NALAGANGULA KOTIREDDY
52	17471A04H3	GUNTUPALLI THIRUMALA PRASANNA SANKAR
53	17471A04H4	RAJARAPU SRILAKSHMI TIRUMALESWARI
54	17471A04H5	PAMURU DIVYA
55	17471A04H6	SHAIK SAMEER
56	17471A04H7	KUNCHALA GOPI KRISHNA
57	17471A04H8	PUSALA MADHU KUMAR

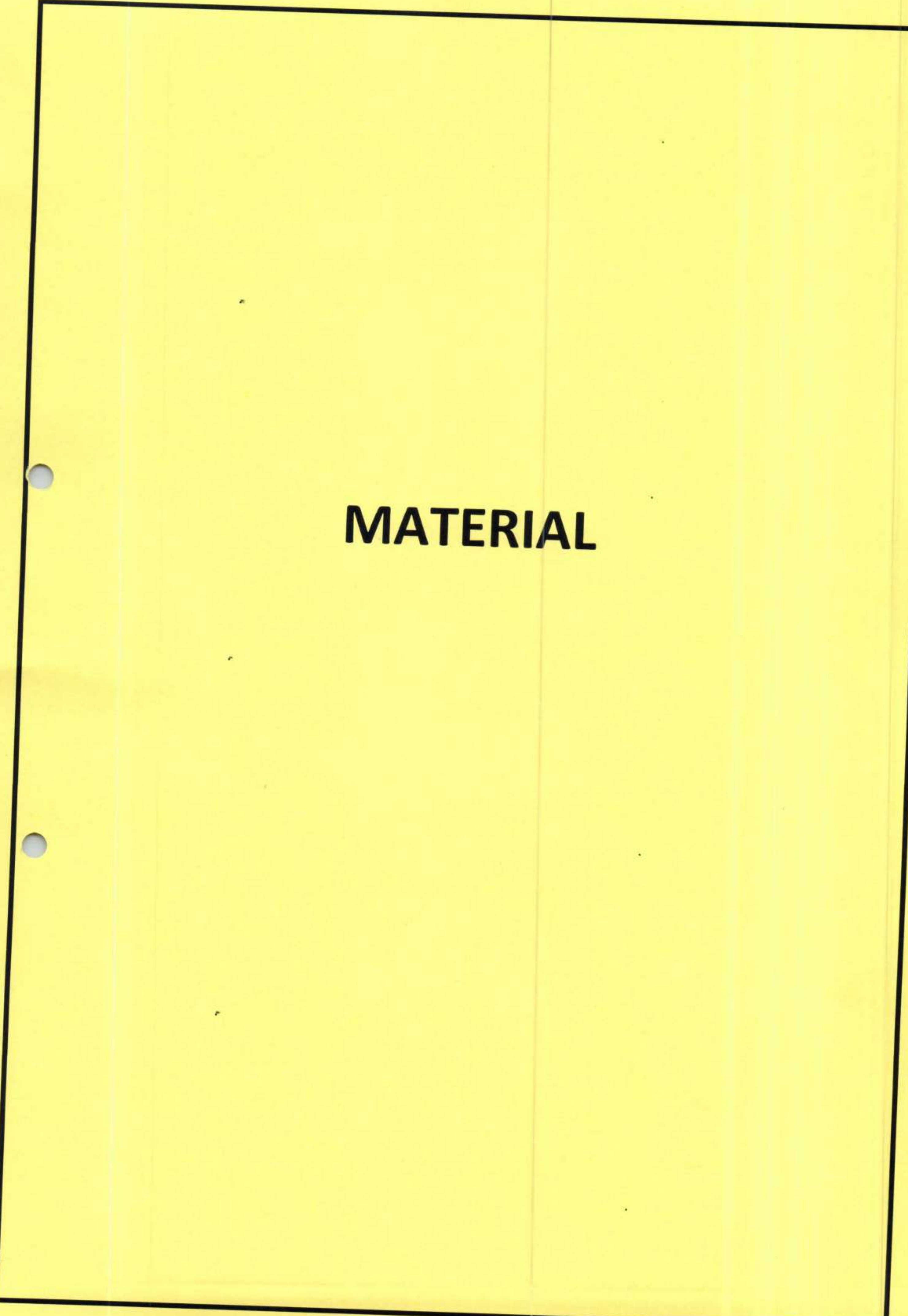
58	17471A04H9	GAJJALA MARUTHI VENKATA KRISHNA REDDY
59	17471A04I0	MEDA RAVI TEJA
60	18475A0401	MARELLA VAMSI
61	18475A0402	RACHAKONDA SHYAM PREMKUMAR
62	18475A0403	SURISETTI ARCHANA
63	18475A0404	PALADUGU GANESH
64	18475A0405	VUYYURU SRILAKSHMI
65	18475A0406	NUNNA VENKATA SIVASAI
66	18475A0407	PARITALA HARITHA
67	18475A0408	MADDINA VENKATA SANDEEP
68	18475A0409	CHERUKURI BRAHMA VENKATESWARLU

BRANCH/SEC - ECE/D		
SL.NO.	H.T.NO.	STUDENT NAME
1	16475A0415	POLLA SIVA
2	16471A0424	GOLLA VENKATESWARI
3	16471A04D2	THALLAPALLI SIVANAGARAJU
4	16471A04G1	SRIRAM NAVEEN KUMAR
5	17471A04I1	CHERUKURI RAVI KUMAR
6	17471A04I2	KOLA PAVAN KALYAN
7	17471A04I3	TEMPALLI PRABHU KUMAR
8	17471A04I4	SHAIK MASTANVALI
9	17471A04I5	TANNIRU AVINASH BABU

10	17471A04I6	TELLAGORLA MANIKANTA GOPALA KRISHNA
11	17471A04I7	KOLA RAKESH
12	17471A04I8	KESANUPALLI PRIYANKA
13	17471A04I9	NARISETTI UMAMAHESWARI
14	17471A04J0	MANYAM UDAY BHASKAR
15	17471A04J1	Y SUPRAJA
16	17471A04J2	GOSULA THIRUPATHI RAO
17	17471A04J3	NARE TEJASWI
18	17471A04J4	VAKA GOPI CHAND
19	17471A04J5	DOPPALAPUDI NELSON RAJU
20	17471A04J6	SHAIK AFRIN
21	17471A04J7	MOLAMANTI SAIKALYAN
22	17471A04J8	SHAIK KANDIPATI MOULALI
23	17471A04K0	PALLEPOGU SHARONU
24	17471A04K1	CHILAKA VAMSI KRISHNA
25	17471A04K2	GOCHIPATHALA RAJ KAMAL
26	17471A04K3	KESENAPALLI MARIYA BABU
27	17471A04K4	ANGALAKURTHI SUMA PRIYA
28	17471A04K5	PERUMALLA VINAY KUMAR
29	17471A04K6	SHAIK SAJID HASAN
30	17471A04K7	USAA PAVAN KALYAN
31	17471A04K8	GORANTLA ASHOK
32	17471A04K9	NALLAMOLU KUSUMA

33	17471A04L0	JUPUDI RAJU
34	17471A04L1	MOGAL IRFAN
35	17471A04L2	VELISALA MANISH PREETHAM
36	17471A04L4	V PREETHI MANISHA
37	17471A04L5	JAMPANI KRISHNAVAMSI
38	17471A04L6	MEDATATI DIVYA
39	17471A04L8	SADHU KOTESWA RAO
40	17471A04M0	BAPATLA VIJAYA LAKSHMI
41	17471A04M1	PATIBANDLA NARESH
42	17471A04M2	KANDULA GURU KIRAN
43	17471A04M3	DARSI ANIL KUMAR
44	17471A04M4	BATRAJU NAGA UMAMAHESH
45	17471A04M5	PALLEMPATI DURGA PRASAD
46	17471A04M6	PUTTA SRIKANTH
47	17471A04M7	MEKALA YASHWANTH KUMAR
48	17471A04M8	SHAIK IMRAN
49	17471A04N0	SHAIK MAHAMOOD SHAREEF
50	17471A04N1	MADHIREDDY ANIL KUMAR REDDY
51	17471A04N2	KADIYAM SUDHAKAR
52	17471A04N3	VEMULURI YASASWI
53	18475A0410	THOTA BHARATH
54	18475A0411	SIDDEALA DILEEP SAGAR
55	18475A0412	DOPPALAPUDI SARATH CHANDRA

56	18475A0413	JANGA MAHENDRA
57	18475A0414	AINAOLU GOPIKRISHNA
58	18475A0415	THUNGALA PRAMOD
59	18475A0416	SURUBULA LEELA PAVANKUMAR
60	18475A0417	SARIKONDA RAMA KRISHNAM RAJU
61	18475A0418	ATTULURI SYAM PRASAD
62	18475A0419	KUNDA JASHUVA
63	18475A0420	YALAVARTHI MADHU BABU
64	18475A0421	M SUBRAHMANYAM
65	18475A0422	GUDISE VENKATESH
66	18475A0423	VARIKUTI KRISHNANJANEYULU



MATERIAL

ESD UNIT-1

INTRODUCTION

INTRODUCTION:

Computing systems are everywhere. It's probably no surprise that millions of computing systems are built every year destined for desktop computers (Personal Computers, or PC's), workstations, mainframes and servers. What may be surprising is that billions of computing systems are built every year for a very different purpose: they are embedded within larger electronic devices, repeatedly carrying out a particular function, often going completely unrecognized by the device's user. Creating a precise definition of such embedded computing systems, or simply embedded systems, is not an easy task. We might try the following definition: An embedded system is nearly any computing system other than a desktop, laptop, or mainframe computer. That definition isn't perfect, but it may be as close as we'll get. We can better understand such systems by examining common examples and common characteristics. Such examination will reveal major challenges facing designers of such systems.

APPLICATIONS OF EMBEDDED SYSTEMS:

Embedded systems are found in a variety of common electronic devices, such as:

(a) consumer electronics -- cell phones, pagers, digital cameras, camcorders, videocassette recorders, portable video games, calculators, and personal digital assistants

(b) home appliances -- microwave ovens, answering machines, thermostat, home security, washing machines, and lighting systems

(c) office automation -- fax machines, copiers, printers, and scanners

(d) business equipment -- cash registers, curb side check-in, alarm systems, card readers, product scanners, and automated teller machines

(e) automobiles -- transmission control, cruise control, fuel injection, anti-lock brakes, and active suspension

CHARACTERISTICS OF EMBEDDED SYSTEMS:

Embedded systems have several common characteristics:

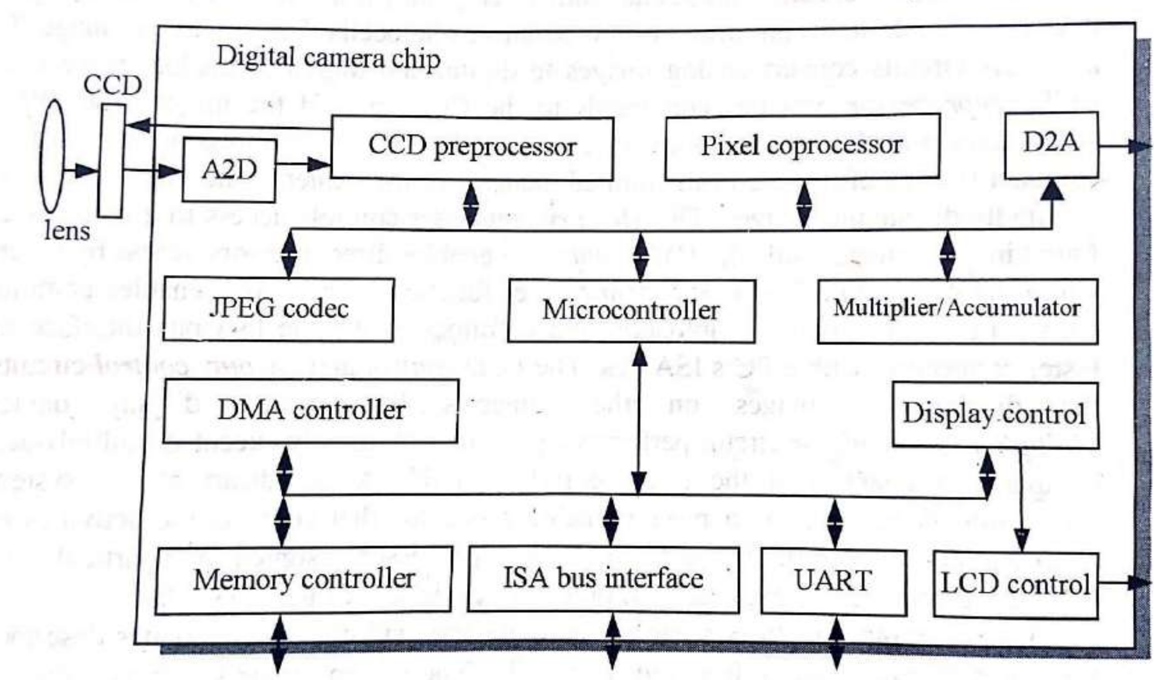
1) **Single-functioned**: An embedded system usually executes only one program, repeatedly. For example, a pager is always a pager. In contrast, a desktop system executes a variety of programs, like spreadsheets, word processors, and video games, with new programs added frequently.
2) **Tightly constrained**: All computing systems have constraints on design metrics, but those on embedded systems can be especially tight. A design metric is a measure of an implementation's features, such as cost, size, performance, and power. Embedded systems often must cost just a few dollars, must be sized to fit on a single chip, must perform fast enough to process data in real-time, and must consume minimum power to extend battery life or prevent the necessity of a cooling fan.

There are some exceptions. One is the case where an embedded system's program is updated with a newer program version. For example, some cell phones can be updated in such a manner. A second is the case where several programs are swapped in and out of a system due to size limitations. For example, some missiles run one program while in cruise mode, then load a second program for locking onto a target.

3) **Reactive and real-time:** Many embedded systems must continually react to changes in the system's environment, and must compute certain results in real time without delay. For example, a car's cruise controller continually monitors and reacts to speed and brake sensors. It must compute acceleration or decelerations amounts repeatedly within a limited time; a delayed computation result could result in a failure to maintain control of the car. In contrast, a desktop system typically focuses on computations, with relatively infrequent (from the computer's perspective) reactions to input devices. In addition, a delay in those computations, while perhaps inconvenient to the computer user, typically does not result in a system failure.

There are some exceptions. One is the case where an embedded system's program is updated with a newer program version. For example, some cell phones can be updated in such a manner. A second is the case where several programs are swapped in and out of a system due to size limitations. For example, some missiles run one program while in cruise mode, then load a second program for locking onto a target.

EXAMPLE:



For example, consider the digital camera system shown in Figure 1.1. The A2D and D2A circuits convert analog images to digital and digital to analog, respectively. The CCD pre-processor is a charge-coupled device pre-processor. The JPEG codec compresses and

decompresses an image using the JPEG 2 compression standard, enabling compact storage in the limited memory of the camera. The Pixel coprocessor aids in rapidly displaying images. The Memory controller controls access to a memory chip also found in the camera, while the DMA controller enables direct memory access without requiring the use of the microcontroller. The UART enables communication with a PC's serial port for uploading video frames, while the ISA bus interface enables a faster connection with a PC's ISA bus. The LCD ctrl and Display ctrl circuits control the display of images on the camera's liquid-crystal display device. A Multiplier/Accum circuit assists with certain digital signal processing. At the heart of the system is a microcontroller, which is a processor that controls the activities of all the other circuits. We can think of each device as a processor designed for a particular task, while the microcontroller is a more general processor designed for general tasks. This example illustrates some of the embedded system characteristics described above. First, it performs a single function repeatedly. The system always acts as a digital camera, wherein it captures, compresses and stores frames, decompresses and displays frames, and uploads frames. Second, it is tightly constrained. The system must be low cost since consumers must be able to afford such a camera. It must be small so that it fits within a standard-sized camera. It must be fast so that it can process numerous images in milliseconds. It must consume little power so that the camera's battery will last a long. JPEG is short for the Joint Photographic Experts Group. The 'joint' refers to its status as a committee working on both ISO and ITU-T standards. Their best known standard is for still image compression.

1.2 Design Challenge — Optimizing Design Metrics

The embedded-system designer must of course construct an implementation that fulfills desired functionality, but a difficult challenge is to construct an implementation that simultaneously optimizes numerous design metrics.

Common Design Metrics

For our purposes, an implementation consists either of a microprocessor with an accompanying program, a connection of digital gates, or some combination thereof. A *design metric* is a measurable feature of a system's implementation. Commonly used metrics include:

[1] *JPEG* is short for *Joint Photographic Experts Group*. "Joint" refers to the group's status as a committee working on both ISO and ITU-T standards. Their best-known standard is for still-image compression.

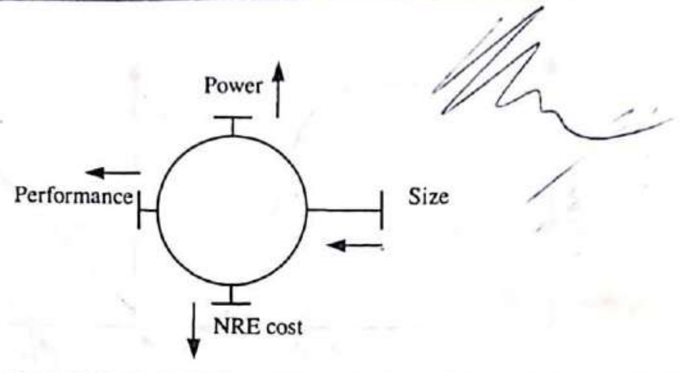


Figure 1.3: Design metric competition — improving one may worsen others.

- *NRE cost* (nonrecurring engineering cost): The one-time monetary cost of designing the system. Once the system is designed, any number of units can be manufactured without incurring any additional design cost; hence the term *nonrecurring*.
- *Unit cost*: The monetary cost of manufacturing each copy of the system, excluding NRE cost.
- *Size*: The physical space required by the system, often measured in bytes for software, and gates or transistors for hardware.
- *Performance*: The execution time of the system.
- *Power*: The amount of power consumed by the system, which may determine the lifetime of a battery, or the cooling requirements of the IC, since more power means more heat.
- *Flexibility*: The ability to change the functionality of the system without incurring heavy NRE cost. Software is typically considered very flexible.
- *Time-to-prototype*: The time needed to build a working version of the system, which may be bigger or more expensive than the final system implementation, but it can be used to verify the system's usefulness and correctness and to refine the system's functionality.
- *Time-to-market*: The time required to develop a system to the point that it can be released and sold to customers. The main contributors are design time, manufacturing time, and testing time.
- *Maintainability*: The ability to modify the system after its initial release, especially by designers who did not originally design the system.
- *Correctness*: Our confidence that we have implemented the system's functionality correctly. We can check the functionality throughout the process of designing the system, and we can insert test circuitry to check that manufacturing was correct.
- *Safety*: The probability that the system will not cause harm.

Metrics typically compete with one another: Improving one often leads to worsening of another. For example, if we reduce an implementation's size, the implementation's performance may suffer. Some observers have compared this phenomenon to a wheel with numerous pins, as illustrated in Figure 1.3. If you push one pin in, such as size, then the other

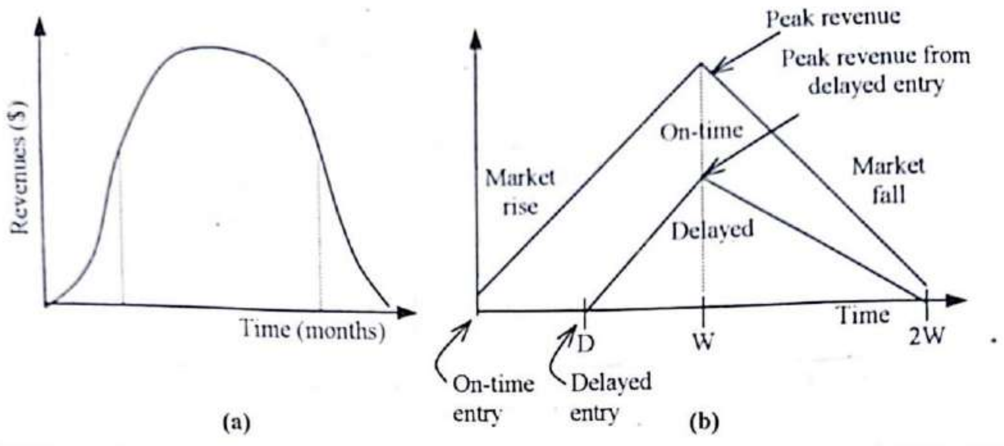


Figure 1.4: Time-to-market: (a) market window, (b) simplified revenue model for computing revenue loss from delayed entry.

pins pop out. To best meet this optimization challenge, the designer must be comfortable with a variety of hardware and software implementation technologies, and must be able to migrate from one technology to another, in order to find the best implementation for a given application and constraints. Thus, a designer cannot simply be a hardware expert or a software expert, as is commonly the case today; the designer must have expertise in both areas.

The Time-to-Market Design Metric

Most of these metrics are heavily constrained in an embedded system. The time-to-market constraint has become especially demanding in recent years. Introducing an embedded system to the marketplace early can make a big difference in the system's profitability, since market windows for products are becoming quite short, with such windows often measured in months. For example, Figure 1.4(a) shows a sample market window during which time a product would have highest sales. Missing this window, which means that the product begins being sold further to the right on the time scale, can mean significant loss in sales. In some cases, each day that a product is delayed from introduction to the market can translate to a one-million-dollar loss. The average time-to-market constraint has been reported as having shrunk to only 8 months!

Adding to the difficulty of meeting the time-to-market constraint is the fact that embedded system complexities are growing due to increasing IC capacities, as we will see later in this chapter. Such rapid growth in IC capacity translates into pressure on designers to add more functionality to a system. Thus, designers today are being asked to do more in less time.

Let's investigate the loss of revenue that can occur due to delayed entry of a product in the market. We'll use a simplified model of revenue that is shown in Figure 1.4(b). This model assumes the peak of the market occurs at the halfway point, denoted as W, of the product life, and that the peak is the same even for a delayed entry. The revenue for an

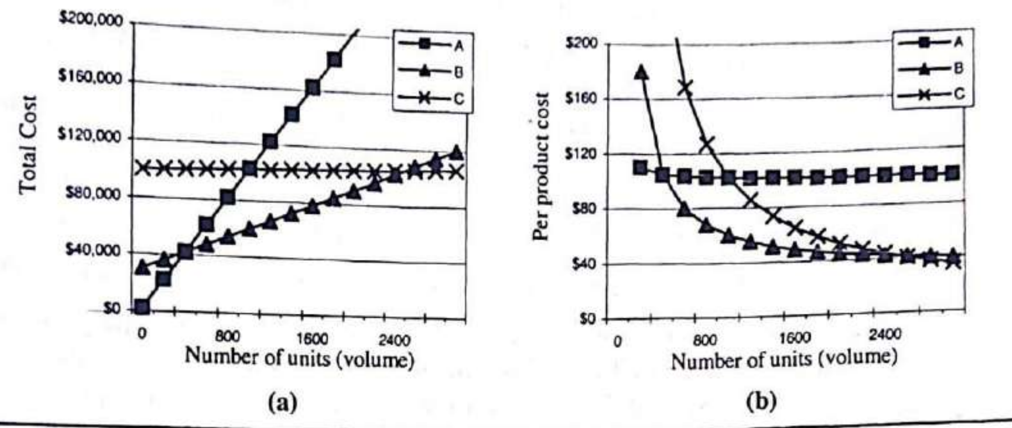


Figure 1.5: Costs for technologies A, B, and C as a function of volume: (a) total cost, (b) per-product cost.

on-time market entry is the area of the triangle labeled *On-time*, and the revenue for a delayed entry product is the area of the triangle labeled *Delayed*. The revenue loss for a delayed entry is just the difference of these two triangles' areas. Let's derive an equation for percentage revenue loss, which equals ((*On-time* – *Delayed*) / *On-time*) * 100%. For simplicity, we'll assume the market rise angle is 45 degrees, meaning the height of the triangle is W, and we leave as an exercise the derivation of the same equation for any angle. The area of the *On-time* triangle, computed as ½ * *base* * *height*, is thus ½ * $2W$ * W, or W^2. The area of the *Delayed* triangle is $\frac{1}{2}(W - D + W) * (W - D)$. After algebraic simplification, we obtain the following equation for percentage revenue loss:

$$\text{percentage revenue loss} = (D(3W - D) / 2W^2) * 100\%$$

Consider a product whose lifetime is 52 weeks, so $W = 26$. According to the preceding equation, a delay of just $D = 4$ weeks results in a revenue loss of 22%, and a delay of $D = 10$ weeks results in a loss of 50%. Some studies claim that reaching market late has a larger negative effect on revenues than development cost overruns or even a product price that is too high.

The NRE and Unit Cost Design Metrics

As another exercise, let's consider NRE cost and unit cost in more detail. Suppose three technologies are available for use in a particular product. Assume that implementing the product using technology A would result in an NRE cost of $2,000 and unit cost of $100, that technology B would have an NRE cost of $30,000 and unit cost of $30, and that technology C would have an NRE cost of $100,000 and unit cost of $2. Ignoring all other design metrics, like time-to-market, the best technology choice will depend on the number of units we plan to produce. We illustrate this concept with the plot of Figure 1.5(a). For each of the three technologies, we plot total cost versus the number of units produced, where:

total cost = NRE cost + unit cost * # of units

We see from the plot that, of the three technologies, technology A yields the lowest total cost for low volumes, namely for volumes between 1 and 400. Technology B yields the lowest total cost for volumes between 400 and 2500. Technology C yields the lowest cost for volumes above 2500.

Figure 1.5(b) illustrates how larger volumes allow us to amortize NRE costs such that lower per-product costs result. The figure plots per-product cost versus volume, where:

per-product cost = total cost / # of units = NRE cost / # of units + unit cost

For example, for technology C and a volume of 200,000, the contribution to the per-product cost due to NRE cost is $100,000 / 200,000, or $0.50. So the per-product cost would be $0.50 + $2 = $2.50. The larger the volume, the lower the per-product cost, since the NRE cost can be distributed over more products. The per-product cost for each technology approaches that technology's unit cost for very large volumes. So for very large volumes, numbering in the hundreds of thousands, we can approach a per-product cost of just $2 — quite a bit less than the per-product cost of over $100 for small volumes.

Clearly, one must consider the revenue impact of both time-to-market and per-product cost, as well as all the other relevant design metrics when evaluating different technologies.

The Performance Design Metric

Performance of a system is a measure of how long the system takes to execute our desired tasks. Performance is perhaps the most widely used design metric in marketing an embedded system, and also one of the most abused. Many metrics are commonly used in reporting system performance, such as clock frequency or instructions per second. However, what we really care about is how long the system takes to execute our application. For example, in terms of performance, we care about how long a digital camera takes to process an image. The camera's clock frequency or instructions per second are not the key issues — one camera may actually process images faster but have a lower clock frequency than another camera.

With that said, there are several measures of performance. For simplicity, suppose we have a single task that will be repeated over and over, such as processing an image in a digital camera. The two main measures of performance are:

- *Latency*, or *response time*: The time between the start of the task's execution and the end. For example, processing an image may take 0.25 second.
- *Throughput*: The number of tasks that can be processed per unit time. For example, a camera may be able to process 4 images per second.

However, note that throughput is not always just the number of tasks times latency. A system may be able to do better than this by using parallelism, either by starting one task before finishing the next one or by processing each task concurrently. A digital camera, for example, might be able to capture and compress the next image, while still storing the previous image to memory. Thus, our camera may have a latency of 0.25 second but a throughput of 8 images per second.

In embedded systems, performance at a very detailed level is also often of concern. In particular, two signal changes may have to be generated or measured within some number of nanoseconds.

Speedup is a common method of comparing the performance of two systems. The speedup of system A over system B is determined simply as:

speedup of A over B = performance of A / performance of B.

Performance could be measured either as latency or as throughput, depending on what is of interest. Suppose the speedup of camera A over camera B is 2. Then we also can say that A is 2 times faster than B and B is 2 times slower than A.

Embedded processor technology:

Processor technology involves the architecture of the computation engine used to implement a system's desired functionality. While the term "processor" is usually associated with programmable software processors, we can think of many other, non-programmable, digital systems as being processors also. Each such processor differs in its specialization towards a particular application (like a digital camera application), thus manifesting different design metrics. We illustrate this concept graphically in Figure 1.5. The application requires a specific embedded functionality, represented as a cross, such as the summing of the items in an array, as shown in Figure 1.5(a). Several types of processors can implement this functionality, each of which we now describe. We often use a collection of such processors to best optimize our system's design metrics, as was the case in our digital camera example.

General-purpose processors – software

The designer of a general-purpose processor builds a device suitable for a variety of applications, to maximize the number of devices sold. One feature of such a processor is a program memory – the designer does not know what program will run on the processor, so cannot build the program into the digital circuit. Another feature is a general datapath – the datapath must be general enough to handle a variety of computations, so typically has a large register file and one or more general-purpose arithmetic-logic units (ALUs). An embedded system designer, however, need not be concerned about the design of a general-purpose processor. An embedded system designer simply uses a general-purpose processor, by programming the processor's memory to carry out the required functionality. Many people refer to this portion of an implementation simply as the "software" portion.

Using a general-purpose processor in an embedded system may result in several design-metric benefits. Design time and NRE cost are low, because the designer must only write a program, but need not do any digital design. Flexibility is high, because changing functionality requires only changing the program. Unit cost may be relatively low in small quantities, since the processor manufacturer sells large quantities to other customers and hence distributes the NRE cost over many units. Performance may be fast for computation-intensive applications, if using a fast processor, due to advanced architecture features and leading edge IC technology.

However, there are also some design-metric drawbacks. Unit cost may be too high for large quantities. Performance may be slow for certain applications. Size and power may be large due to unnecessary processor hardware.

For example, we can use a general-purpose processor to carry out our arraysumming functionality from the earlier example. Figure 1.5(b) illustrates that a general-purpose covers the desired functionality, but not necessarily efficiently. Figure 1.6(a) shows a simple architecture of a general-purpose processor implementing the arraysumming functionality. The functionality is stored in a program memory. The controller fetches the current instruction, as indicated by the program counter (PC), into the instruction register (IR). It then configures the data path for this instruction and executes the instruction. Finally, it determines the appropriate next instruction address, sets the PC to this address, and fetches again.

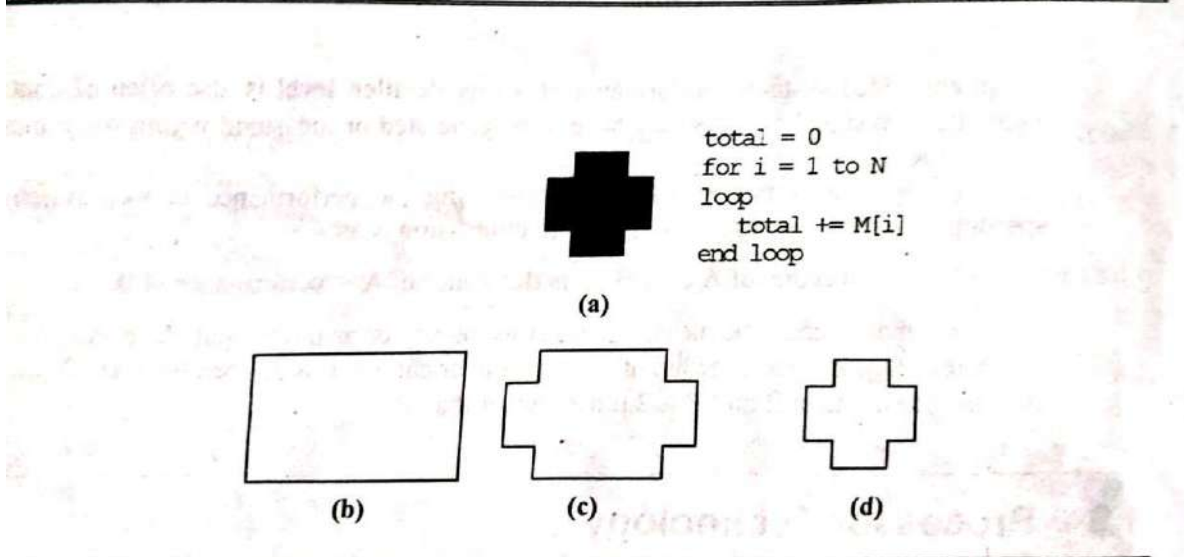


Figure 1.6: Processors vary in their customization for the problem at hand: (a) desired functionality, (b) general-purpose processor, (c) application-specific processor, (d) single-purpose processor.

Single-purpose processors – hardware

A single-purpose processor is a digital circuit designed to execute exactly one program. For example, consider the digital camera example . All of the components other than the microcontroller are single-purpose processors. The JPEG codec, for example, executes a single program that compresses and decompresses video frames. An embedded system designer creates a single-purpose processor by designing a custom digital circuit, as discussed in later chapters. Many people refer to this portion of the implementation simply as the "hardware" portion (although even software requires a hardware processor on which to run). Other common terms include coprocessor and accelerator.

Using a single-purpose processor in an embedded system results in several designmetric benefits and drawbacks, which are essentially the inverse of those for generalpurpose processors. Performance may be fast, size and power may be small, and unit-cost may be low for large quantities, while design time and NRE costs may be high, flexibility is low, unit cost may be high for small quantities, and performance may not match general-purpose processors for some applications.

For example, Figure 1.5(d) illustrates the use of a single-purpose processor in our embedded system example, representing an exact fit of the desired functionality, nothing more, nothing less. Figure 1.6(c) illustrates the architecture of such a single-purpose processor for the example. Since the example counts from one to N, we add an index register. The index register will be loaded with N, and will then count down to zero, at which time it will assert a status line read by the controller. Since the example has only one other value, we add only one register labelled total to the data path. Since the example's only arithmetic operation is addition, we add a single adder to the data path. Since the processor only executes this one program, we hardwire the program directly into the control logic.

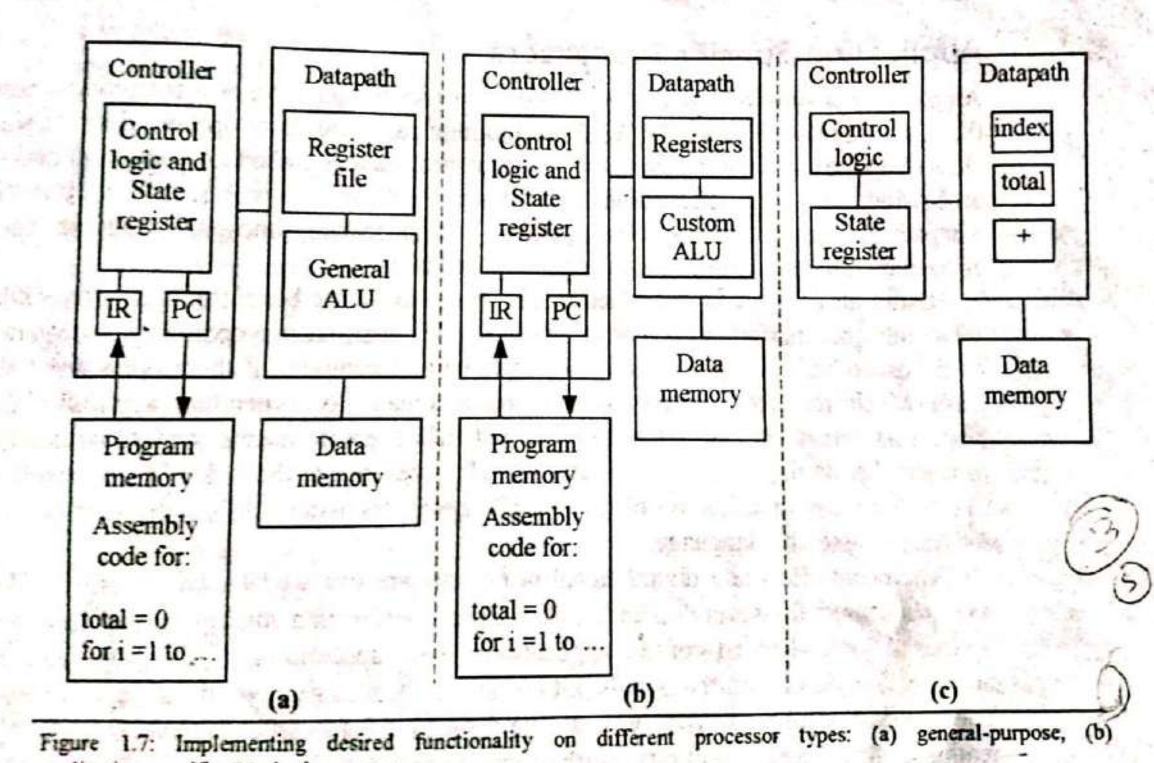


Figure 1.7: Implementing desired functionality on different processor types: (a) general-purpose, (b) application-specific, (c) single-purpose.

Application-specific processors

An application-specific instruction-set processor (or ASIP) can serve as a compromise between the above processor options. An ASIP is designed for a particular class of applications with common characteristics, such as digital-signal processing, telecommunications, embedded control, etc. The designer of such a processor can optimize the datapath for the application class, perhaps adding special functional units for common operations, and eliminating other infrequently used units.

Using an ASIP in an embedded system can provide the benefit of flexibility while still achieving good performance, power and size. However, such processors can require large NRE cost to build the processor itself, and to build a compiler, if these items don't already exist. Much research currently focuses on automatically generating such processors and associated retargetable compilers. Due to the lack of retargetable compilers that can exploit the unique features of a particular ASIP, designers using ASIPs often write much of the software in assembly language.

Digital-signal processors (DSPs) are a common class of ASIP, so demand special mention. A DSP is a processor designed to perform common operations on digital signals, which are the digital encodings of analog signals like video and audio. These operations carry out common signal processing tasks like signal filtering, transformation, or combination. Such operations are usually math-intensive, including operations like multiply and add or shift and add. To support such operations, a DSP may have specialpurpose datapath components such a multiply-accumulate unit, which can perform a computation like T = T + M[i]*k using only one instruction. Because DSP programs often manipulate large arrays of data, a DSP may also include special hardware to fetch sequential data memory locations in parallel with other operations, to further speed execution.

Figure 1.5(c) illustrates the use of an ASIP for our example; while partially customized to the desired functionality, there is some inefficiency since the processor also contains features to support reprogramming. Figure 1.6(b) shows the general architecture of an ASIP for the example. The datapath may be customized for the example. It may have an auto-incrementing register, a path that allows the add of a register plus a memory location in one instruction, fewer registers, and a simpler controller.

IC technology:

Every processor must eventually be implemented on an IC. IC technology involves the manner in which we map a digital (gate-level) implementation onto an IC. An IC (Integrated Circuit), often called a “chip,” is a semiconductor device consisting of a set of connected transistors and other devices. A number of different processes exist to build semiconductors, the most popular of which is CMOS (Complementary Metal Oxide Semiconductor). The IC technologies differ by how customized the IC is for a particular implementation. For lack of a better term, we call these technologies “IC technologies.” IC technology is independent from processor technology; any type of processor can be mapped to any type of IC technology.

To understand the differences among IC technologies, we must first recognize that semiconductors consist of numerous layers. The bottom layers form the transistors. The middle layers form logic gates. The top layers connect these gates with wires. One way to create these layers is by depositing photo-sensitive chemicals on the chip surface and then shining light through masks to change regions of the chemicals. Thus, the task of building the layers is actually one of designing appropriate masks. A set of masks is often called a layout. The narrowest line that we can create on a chip is called the feature size, which today is well below one micro-meter (sub-micron). For each IC technology, all layers must eventually be built to get a working IC.

Full-custom/VLSI:

In a full-custom IC technology, we optimize all layers for our particular embedded system’s digital implementation. Such optimization includes placing the transistors to minimize interconnection lengths, sizing the transistors to optimize signal transmissions and routing wires among the transistors. Once we complete all the masks, we send the mask specifications to a fabrication plant that builds the actual ICs. Full-custom IC design, often referred to as VLSI (Very Large Scale Integration) design, has very high NRE cost and long turnaround times (typically months) before the IC becomes available, but can yield excellent

performance with small size and power. It is usually used only in high-volume or extremely performance-critical applications.

Semi-custom ASIC (gate array and standard cell):

In an ASIC (Application-Specific IC) technology, the lower layers are fully or partially built, leaving us to finish the upper layers. In a gate array technology, the masks for the transistor and gate levels are already built (i.e., the IC already consists of arrays of gates). The remaining task is to connect these gates to achieve our particular implementation. In a standard cell technology, logic-level cells (such as an AND gate or an AND-OR-INVERT combination) have their mask portions pre-designed, usually by hand. Thus, the remaining task is to arrange these portions into complete masks for the gate level, and then to connect the cells. ASICs are by far the most popular IC technology, as they provide for good performance and size, with much less NRE cost than full-custom IC's.

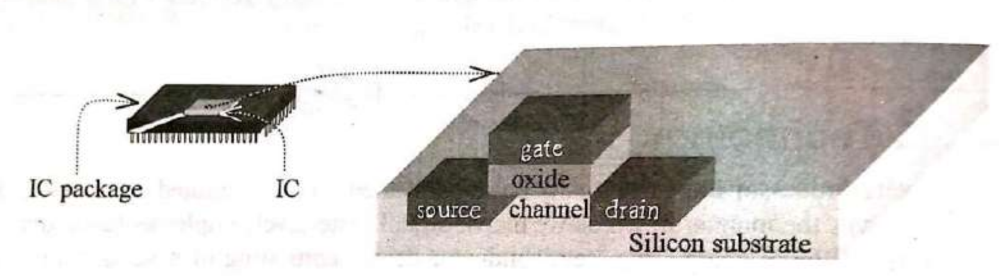


Figure 1.8: ICs consist of several layers. Shown is a simplified CMOS transistor; an IC may possess millions of these, connected above by many layers of metal (not shown).

PLD:

In a PLD (Programmable Logic Device) technology, all layers already exist, so we can purchase the actual IC. The layers implement a programmable circuit, where programming has a lower-level meaning than a software program. The programming that takes place may consist of creating or destroying connections between wires that connect gates, either by blowing a fuse, or setting a bit in a programmable switch. Small devices, called programmers, connected to a desktop computer can typically perform such programming. We can divide PLD's into two types, simple and complex. One type of simple PLD is a PLA (Programmable Logic Array), which consists of a programmable array of AND gates and a programmable array of OR gates. Another type is a PAL (Programmable Array Logic), which uses just one programmable array to reduce the number of expensive programmable components. One type of complex PLD, growing very rapidly in popularity over the past decade, is the FPGA (Field Programmable Gate Array), which offers more general connectivity among blocks of logic, rather than just arrays of logic as with PLAs and PALs, and are thus able to implement far more complex designs. PLDs offer very low NRE cost and almost instant IC availability. However, they are typically bigger than ASICs, may have higher unit cost, may consume more power, and may be slower (especially FPGAs). They still provide reasonable performance, though, so are especially well suited to rapid prototyping.

The choice of an IC technology is independent of processor types. For example, a general-purpose processor can be implemented on a PLD, semi-custom, or full-custom IC. In fact, a company marketing a commercial general-purpose processor might first market a semi-custom implementation to reach the market early, and then later introduce a full-custom implementation. They might also first map the processor to an older but more reliable technology, like 0.2 micron, and then later map it to a newer technology, like 0.08 micron. These two evolutions of mappings to a large extent explain why a processor's clock speed improves on the market over time.

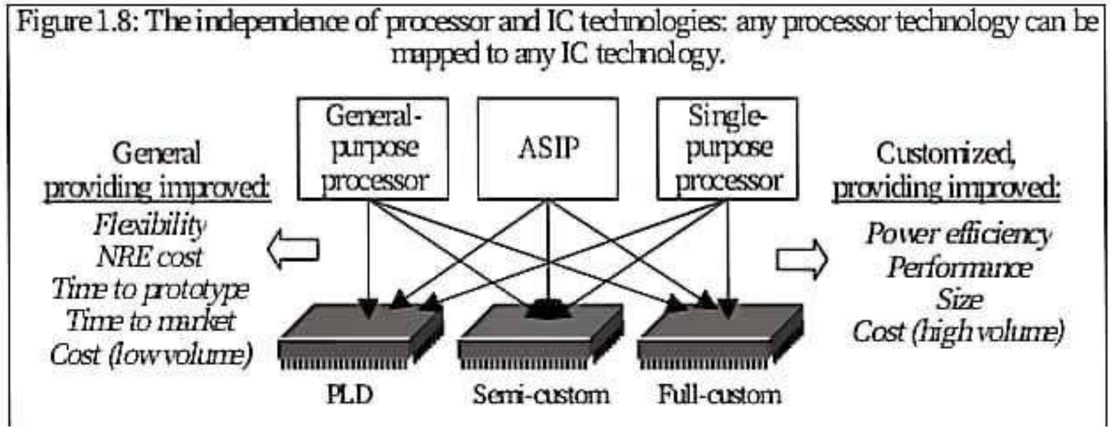


Figure 1.8: The independence of processor and IC technologies: any processor technology can be mapped to any IC technology.

Trends

We should be aware of what is by far the most important trend in embedded systems, a trend related to ICs: *IC transistor capacity has doubled roughly every 18 months for the past several decades.*

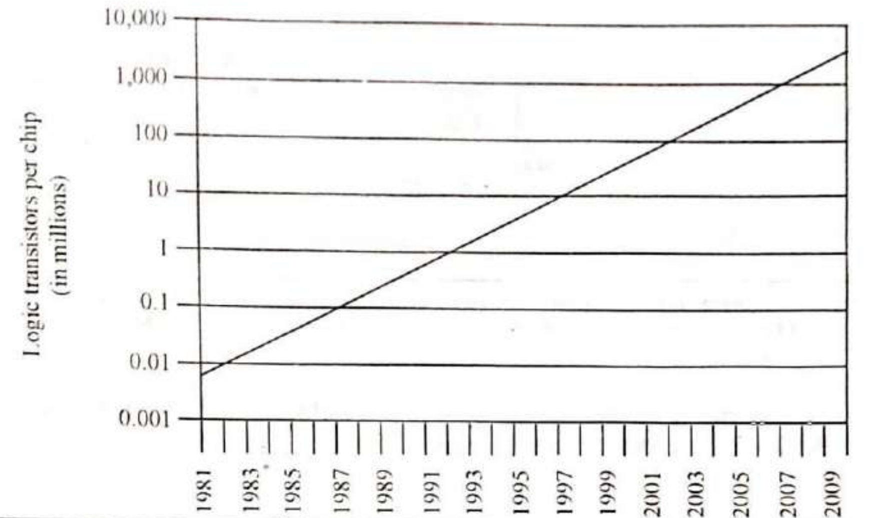


Figure 1.9: IC capacity exponential increase, following "Moore's Law." Source: The International Technology Roadmap for Semiconductors.

This trend, illustrated in Figure 1.9, was actually predicted way back in 1965 by Intel co-founder Gordon Moore. He predicted that semiconductor transistor density would double every 18 to 24 months. The trend is therefore known as *Moore's Law*. Moore recently predicted about another decade before such growth slows down. The trend is mainly caused by improvements in IC manufacturing that result in smaller parts, such as transistor parts and wires, on the surface of the IC. The minimum part size, commonly known as feature size, for a CMOS IC in 2002 is about 130 nanometers.

Figure 1.9 shows leading-edge chip approximate capacity per year from 1981 to 2010, using predicted data for years 2000–2010. Note that chip capacity, shown in millions of transistors per chip, is plotted on a logarithmic scale. People often underestimate and are somewhat amazed by the actual growth of something that doubles over short time periods, in this case 18 months. For example, this underestimation in part explains the popularity of so-called pyramid schemes. It is the key to the popular trick question of asking someone to choose between a salary of $1,000/day for a year, or a penny on day one, 2 pennies on day two, with continued doubling each day for a year. While many people would choose the first option, the second option results in more money than exists in the world. Many people are also surprised to discover that just 20 generations ago, meaning a few hundred years, we find that we each have one million ancestors.

Figure 1.10 shows that in 1981, a leading-edge chip could hold about 10,000 transistors, which is roughly the complexity of an 8-bit microprocessor. In 2002, a leading-edge chip can hold about 150,000,000 transistors, the equivalent of 15,000 8-bit microprocessors! For comparison, if automobile fuel efficiency had improved at this rate since 1981, cars in 2002 would get about 500,000 miles per gallon.

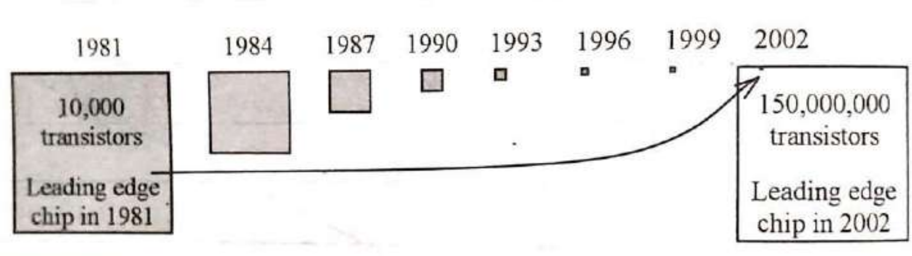


Figure 1.10: Graphical demonstration of the rapid growth in transistor density. The shaded region symbolizes the area required by a 10,000-transistor design over the years. Note that the area occupies an incredibly tiny portion of a leading edge chip in 2002.

This trend of increasing chip capacity has enabled the proliferation of low-cost, high-performance embedded systems that we see today.

Design technology:

Design technology involves the manner in which we convert our concept of desired system functionality into an implementation. We must not only design the implementation to optimize design metrics, but we must do so quickly. As described earlier, the designer must be able to produce larger numbers of transistors every year, to keep pace with IC technology. Hence, improving design technology to enhance productivity has been a focus of the software and hardware design communities for decades.

To understand how to improve the design process, we must first understand the design process itself. Variations of a top-down design process have become popular in the past decade, an ideal form of which is illustrated in Figure 1.9. The designer refines the system through several abstraction levels. At the system level, the designer describes the desired functionality in some language, often a natural language like English, but preferably an executable language like C; we shall call this the system specification. The designer refines this specification by distributing portions of it among chosen processors (general or single purpose), yielding behavioral specifications for each processor. The designer refines these specifications into register-transfer (RT) specifications by converting behavior on general-purpose processors to assembly code, and by converting behavior on single-purpose processors to a connection of register-transfer components and state machines. The designer then refines the register-transfer-level specification of a single-purpose processor into a logic specification consisting of Boolean equations. Finally, the designer refines the remaining specifications into an implementation, consisting of machine code for general-purpose processors, and a gate-level netlist for single-purpose processors.

There are three main approaches to improving the design process for increased productivity, which we label as compilation/synthesis, libraries/IP, and test/verification. Several other approaches also exist. We now discuss all of these approaches. Each approach can be applied at any of the four abstraction levels.

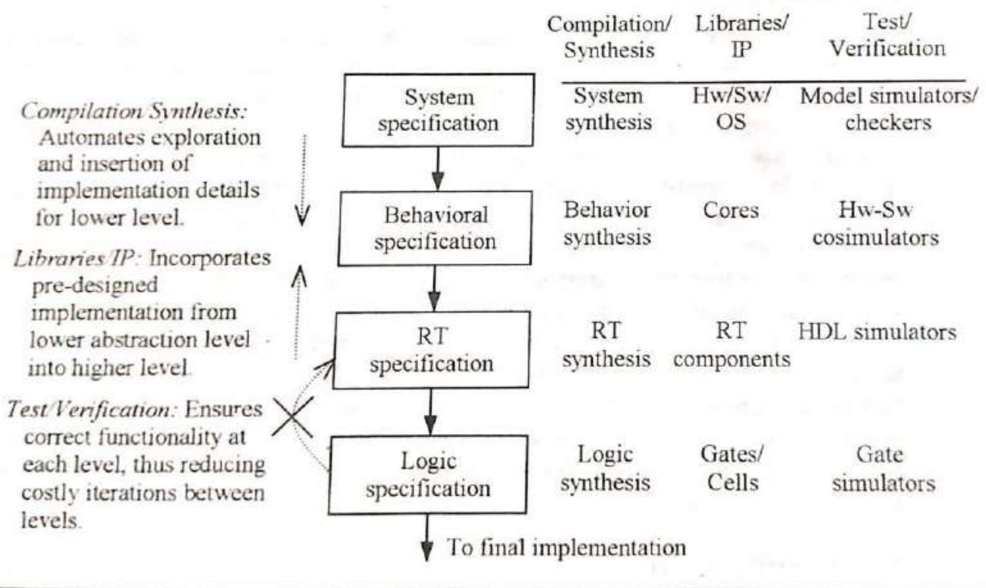


Figure 1.11: Ideal top-down design process, and productivity improvers.

Compilation/Synthesis:

Compilation/Synthesis lets a designer specify desired functionality in an abstract manner, and automatically generates lower-level implementation details. Describing a system at high abstraction levels can improve productivity by reducing the amount of details, often by an order of magnitude, that a design must specify.

A logic synthesis tool converts Boolean expressions into a connection of logic gates (called a netlist). A register-transfer (RT) synthesis tool converts finite-state machines and register-transfers into a datapath of RT components and a controller of Boolean equations. A behavioral synthesis tool converts a sequential program into finite-state machines and register transfers. Likewise, a software compiler converts a sequential program to assembly code, which is essentially register-transfer code. Finally, a system synthesis tool converts an abstract system specification into a set of sequential programs on general and single-purpose processors.

Libraries/IP:

Libraries involve re-use of pre-existing implementations. Using libraries of existing implementations can improve productivity if the time it takes to find, acquire, integrate and test a library item is less than that of designing the item oneself.

A logic-level library may consist of layouts for gates and cells. An RT-level library may consist of layouts for RT components, like registers, multiplexors, decoders, and functional units. A behavioral-level library may consist of commonly used components, such as compression components, bus interfaces, display controllers, and even generalpurpose processors. The advent of system-level integration has caused a great change in this level of library. Rather than these components being IC's, they now must also be available in a form, called cores, that we can implement on just one portion of an IC. This change from behavioral-level libraries of IC's to libraries of cores has prompted use of the term Intellectual Property (IP), to emphasize the fact that cores exist in a "soft" form that must be

protected from copying. Finally, a system-level library might consist of complete systems solving particular problems, such as an interconnection of processors with accompanying operating systems and programs to implement an interface to the Internet over an Ethernet network.

Test/Verification:

Test/Verification involves ensuring that functionality is correct. Such assurance can prevent time-consuming debugging at low abstraction levels and iterating back to high abstraction levels.

Simulation is the most common method of testing for correct functionality, although more formal verification techniques are growing in popularity. At the logic level, gatelevel simulators provide output signal timing waveforms given input signal waveforms. Likewise, general-purpose processor simulators execute machine code. At the RT-level, hardware description language (HDL) simulators execute RT-level descriptions and provide output waveforms given input waveforms. At the behavioral level, HDL simulators simulate sequential programs, and co-simulators connect HDL and generalpurpose processor simulators to enable hardware/software co-verification. At the system level, a model simulator simulates the initial system specification using an abstract computation model, independent of any processor technology, to verify correctness and completeness of the specification. Model checkers can also verify certain properties of the specification, such as ensuring that certain simultaneous conditions never occur, or that the system does not deadlock.

Other productivity improvers:

There are numerous additional approaches to improving designer productivity. Standards focus on developing well-defined methods for specification, synthesis and libraries. Such standards can reduce the problems that arise when a designer uses multiple tools, or retrieves or provides design information from or to other designers. Common standards include language standards, synthesis standards and library standards.

Languages focus on capturing desired functionality with minimum designer effort. For example, the sequential programming language of C is giving way to the objectoriented language of C++, which in turn has given some ground to Java. As another example, state-machine languages permit direct capture of functionality as a set of states and transitions, which can then be translated to other languages like C.

Frameworks provide a software environment for the application of numerous tools throughout the design process and management of versions of implementations. For example, a framework might generate the UNIX directories needed for various simulators and synthesis tools, supporting application of those tools through menu selections in a single graphical user interface.

Trends

The combination of compilation/synthesis, libraries/IP, test/verification, standards, languages, and frameworks has improved designer productivity over the past several decades, as shown in Figure 1.12. Productivity is measured as the number of transistors that one designer can produce in one month. As the figure shows, the growth has been impressive. A designer in 1981 could produce only about 100 transistors per month, whereas in 2002 a designer should be able to produce about 5,000 transistors per month.

1.6 Trade-offs

Perhaps the key embedded system design challenge is the simultaneous optimization of competing design metrics. To address this challenge, the designer trades off among the

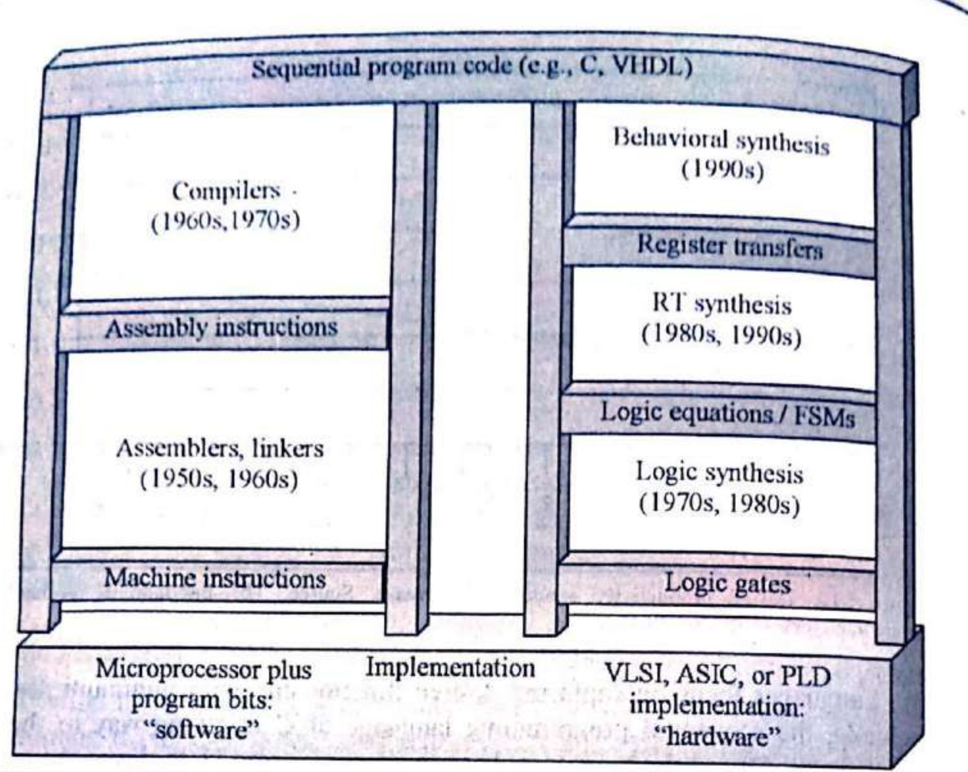


Figure 1.13: The co-design ladder: recent maturation of synthesis enables a unified view of hardware and software.

advantages and disadvantages of the various available processor technologies and IC technologies. To optimize a system, the designer must therefore be familiar with and comfortable with the various technologies — the designer must be a "renaissance engineer," in the words of some. In the past and to a large extent in the present, however, most designers had expertise with either general-purpose processors or with single-purpose processors but not both — they were either software designers or hardware designers. Because of this separation of design expertise, systems had to be separated into the software and hardware subsystems very early in the design process, separately designed, and then integrated near the end of the process. However, such early and permanent separation clearly doesn't allow for the best optimization of design metrics. Instead, being able to move functions between hardware and software, at any stage of the design process, provides for better optimization.

The relatively recent maturation of RT and behavioral synthesis tools has enabled a unified view of the design process for hardware and software. In the past, the design processes were radically different — software designers wrote sequential programs, while hardware designers connected components. But today, synthesis tools have changed the hardware designer's task essentially into one of writing sequential programs, albeit with some knowledge of how the hardware will be synthesized from such programs. We can think of abstraction levels as being the rungs of a ladder, and compilation and synthesis as enabling us to step up the ladder and hence enabling designers to focus their design efforts at higher levels

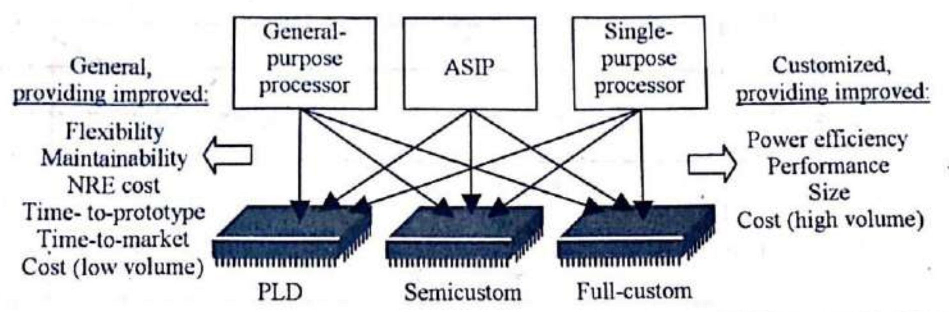


Figure 1.14: The independence of processor and IC technologies: Any processor technology can be mapped to any IC technology.

of abstraction, as illustrated in Figure 1.13. Thus, the starting point for either hardware or software is sequential programs, enhancing the view that system functionality can be implemented in hardware, software, or some combination thereof, leading to the following important point:

> *The choice of hardware versus software for a particular function is simply a trade-off among various design metrics, like performance, power, size, NRE cost, and especially flexibility; there is no fundamental difference between what hardware or software can implement.*

Hardware/software codesign is the field that emphasizes a unified view of hardware and software, and develops synthesis tools and simulators that enable the co-development of systems using both hardware and software.

In general, we can view the basic design trade-off as general versus customized implementation, with respect to either processor technology or IC technology, as illustrated in Figure 1.14. The more general, programmable technologies on the left of the figure provide greater flexibility (a design can be reprogrammed relatively easily), reduced NRE cost (designing using those technologies is generally cheaper), faster time-to-prototype and time-to-market (since designing takes less time), and lower cost in low volumes (since the IC manufacturer distributes its IC NRE cost over large quantities of ICs). On the other hand, more customized technologies provide for better power efficiency, faster performance, reduced size, and lower cost in high volumes.

Recall that each of the three processor technologies can be implemented in any of the three IC technologies. For example, a general-purpose processor can be implemented on a PLD, semicustom, or full-custom IC. In fact, a company marketing a product, such as a set-top box or even a general-purpose processor, might first market a semicustom implementation to reach the market early, and then later introduce a full-custom implementation. They might also first map the processor to an older but more reliable technology, like 0.2 micron, and then later map it to a newer technology, like 0.08 micron. These two evolutions of mappings to a large extent explain why a general-purpose processor's

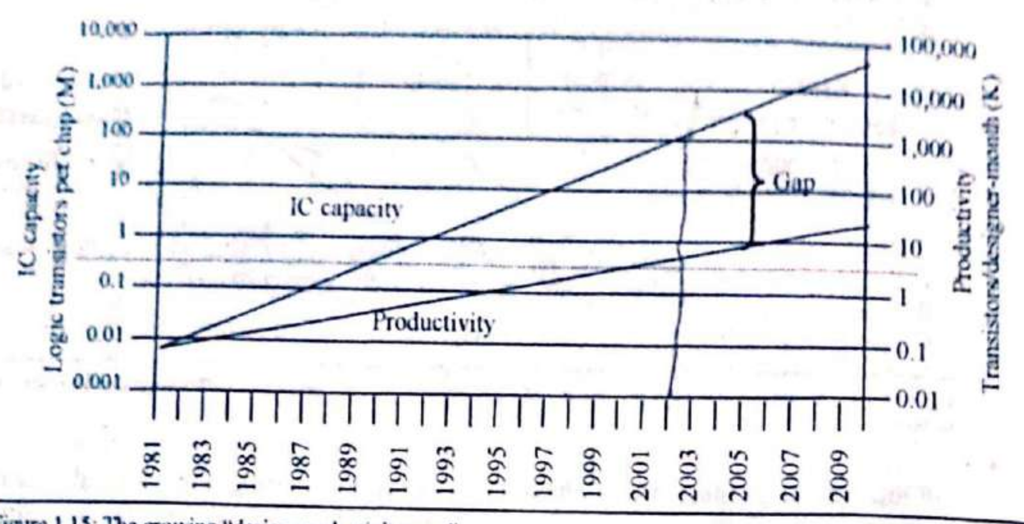


Figure 1.15: The growing "design productivity gap."

clock speed improves on the market over time. Likewise, a designer of an embedded system may use PLDs for prototyping a product, and even for the first few hundred instances of the product to speed its time-to-market, switching to ASICs for larger-scale production.

Furthermore, we often implement multiple processors of different types on the same IC. Figure 1.2 was an example of just such a situation — the digital camera included a microcontroller plus numerous single-purpose processors on the same IC. A single chip with multiple processors is often referred to as a *system-on-a-chip*. In fact, we can even implement more than one IC technology on a single IC — a portion of the IC may be custom, another portion semicustom, and yet another portion programmable logic. The need for designers comfortable with the variety of processor and IC technologies thus becomes evident.

Design Productivity Gap

While designer productivity has grown at an impressive rate over the past decades, the rate of improvement has not kept pace with chip capacity growth. Figure 1.15 shows the productivity growth plot superimposed on the chip capacity growth plot, illustrating the growing design productivity gap. For example, in 1981, a leading-edge chip required about 100 designer-months to design, since 100 designer-months * 100 transistors/designer-month = 10,000 transistors. However, in 2002, a leading-edge chip would require about 30,000 designer months, since 30,000 designer-months * 5,000 transistors/designer-month = 150,000,000 transistors. So the design productivity gap has resulted in an increase from 100 to 30,000 designer-months to build a leading-edge chip. Assuming a designer costs $10,000 per month, the cost of building a leading-edge chip has risen from $1,000,000 in 1981 to an incredible $300,000,000 in 2002. Few products can justify such large investment in a chip. Thus, most designs do not even come close to using potential chip capacity.

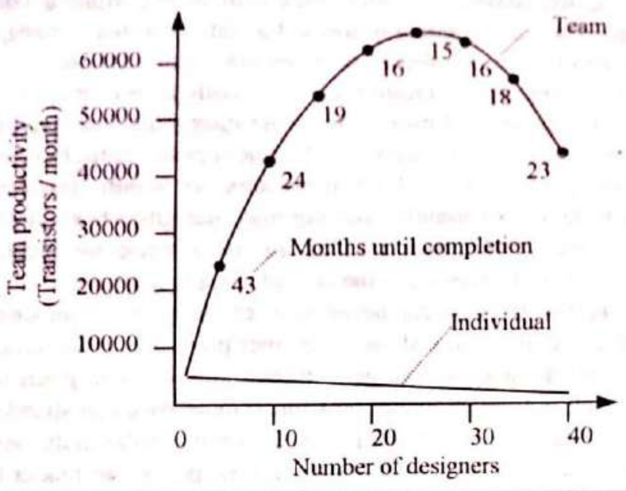


Figure 1.16: The "mythical man-month": Adding designers can decrease individual productivity and at some point can actually delay the project completion time.

The situation is even worse than stated before, because the discussion assumes that designer productivity is independent of project team size, whereas in reality adding more designers to a project team can actually decrease productivity. Suppose 10 designers work together on a project, and each produces 5,000 transistors/month, so that their combined output is 10 * 5,000 = 50,000 transistors/month. Would 100 designers on a project then produce 100 * 5,000 = 500,000 transistors/month? Probably not. The complexity of having 100 designers work together is far greater than having 10 designers work together. Even calling a meeting of 100 designers is a fairly complex task, whereas a 10-designer meeting is quite straightforward. Furthermore, a 100-designer team would likely be decomposed into groups, each group having a group leader that meets with other group leaders and reports back to his or her group, thus introducing extra layers of communication and hence more likelihood of misunderstandings and time-consuming mistakes.

This decrease in productivity as designers are added to a project was reported by Frederick Brooks in his classic 1975 book entitled *The Mythical Man-Month*. His book focused on writing software, but the same principle applies to designing hardware. The decrease in productivity due to team-size complexity can at some point actually lengthen the time to complete a project. For example, consider a hypothetical 1,000,000 transistor project, in which a designer working alone can produce 5,000 transistors per month, and each additional designer added to the project results in a productivity decrease of 100 transistors per designer, due to the added complexities of team communication and management. So a designer can complete the project in 1,000,000 / 5,000 = 200 months. 10 designers can produce 4,100 transistors per month each, meaning 10 * 4,100 = 41,000 transistors per month total, requiring 1,000,000 / 41,000 = 24.3 months to complete the project. Figure 1.16 plots individual designer productivity as designers are added to the project. The figure also plots

team productivity, computed simply as the number of designers multiplied by their individual productivity. Project completion times for different team sizes, computed as 1,000,000 transistors divided by team-transistors/month, are also shown. A 25-designer team can produce 25 * 2,600 = 65,000 transistors per month, requiring 1,000,000/65,000 = 15.3 months to complete the project. However, a 26-designer team also produces 26 * 2,500 = 65,000 transistors per month, so adding a 26th designer doesn't help. Furthermore, a 27-designer team produces only 27 * 2,400 = 64,800 transistors per month, thus actually delaying the project completion time to 15.4 months. Adding more designers beyond 26 only worsens the project completion time. Hence, man-months are in a sense mythical: We cannot always add designers to a project to decrease the project completion time.

Therefore, the growing gap between IC capacity and designer productivity in Figure 1.15 is even worse than the figure shows. Designer productivity decreases as we add designers to a project, making the gap even larger. Furthermore, at some point we simply cannot decrease project completion time no matter how much money we can spend on designers, since adding designers will decrease the project team's overall productivity. And therefore, leading-edge chips cannot always be designed in a given time period, no matter how much money we have to spend on designers.

Thus, a pressing need exists for new design technologies that will shrink the design gap. One partial solution proposed by many people is to educate designers not just in one subarea of embedded systems, like hardware design or software design, but instead to educate them to be comfortable with both hardware and software design. This book is intended to contribute to this solution.

ESD UNIT-2

CUSTOM SINGLE PURPOSE PROCESSORS: HARDWARE

INTRODUCTION:

A single-purpose processor is a digital system intended to solve a specific computation task. While a manufacturer builds a standard single-purpose processor for use in a variety of applications, we build a custom single- purpose processor to execute a specific task within our embedded system. An embedded system designer choosing to use a custom single-purpose, rather than a general-purpose, processor to implement part of a system's functionality may achieve several benefits, similar to some of those of the previous chapter.

First, performance may be fast, due to fewer clock cycles resulting from a customized data path, and due to shorter clock cycles resulting from simpler functional units, less multiplexors, or simpler control logic. Second, size may be small, due to a simpler data path and no program memory. In fact, the processor may be faster and smaller than a standard one implementing the same functionality, since we can optimize the implementation for our particular task.

However, because we probably won't manufacture as many of the custom processor as a standard processor, we may not be able to invest as much NRE, unless the embedded system we are building will be sold in large quantities or does not have tight cost constraints. This fact could actually penalize performance and size.

COMBINATIONAL LOGIC:

Transistors and Logic Gates:

A transistor is the basic electrical component of digital systems. Combinations of transistors form more abstract components called logic gates, which designers primarily use when building digital systems. Thus, we begin with a short description of transistors before discussing logic design.

A transistor acts as a simple on/off switch. One type of transistor (CMOS -- Complementary Metal Oxide Semiconductor) is shown in Figure 4.1(a). The gate (not to be confused with logic gate) controls whether or not current flows from the source to the drain. When a high voltage (typically +5 Volts, which we'll refer to as logic 1) is applied to the gate, the transistor conducts, so current flows. When low voltage (which we'll refer to as logic 0, typically ground, which is drawn as several horizontal lines of decreasing width) is applied to the gate, the transistor does not conduct. We can also build a transistor with the opposite functionality, illustrated in in Figure 4.1(b). When logic 0 is applied to the gate, the transistor conducts, and when logic 1 is applied, the transistor does not conduct. Given these two basic transistors, we can easily build a circuit whose output inverts its gate input, as shown in in Figure 4.1(c). When the input x is logic 0, the top transistor conducts (and the bottom does not), so logic 1 appears at the output F. We can also easily build a circuit whose output is logic 1 when at least one of its inputs is logic 0, as shown in Figure 4.1(d). When at least one of the inputs x and y is logic 0, then at least one of the top transistors conducts (and the bottom transistors do not), so logic 1 appears at F. If both inputs are logic 1, then neither of the top transistors conducts, but both of the bottom ones do, so logic 0 appears at F. Likewise,

we can easily build a circuit whose output is logic 1 when both of its inputs are logic 0, as illustrated in Figure 4.1(e). The three circuits shown implement three basic logic gates: an inverter, a NAND gate, and a NOR gate.

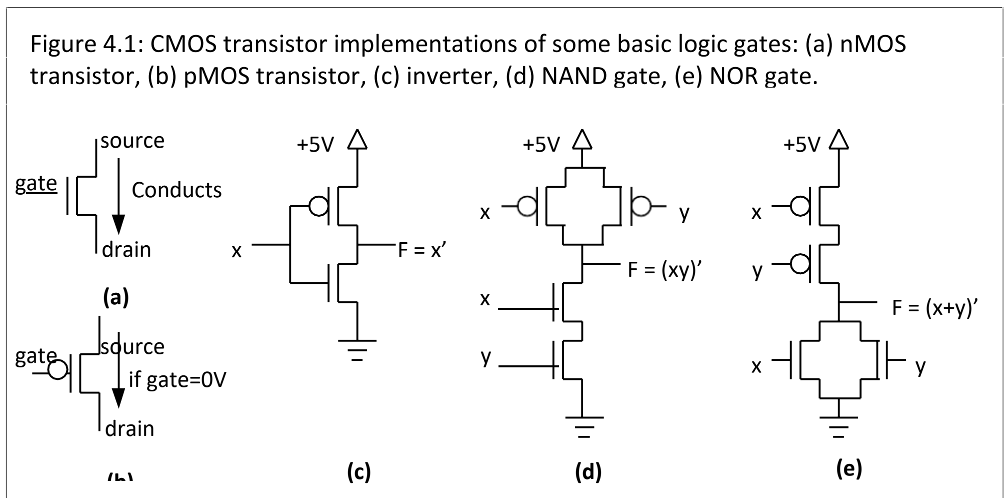


Figure 4.1: CMOS transistor implementations of some basic logic gates: (a) nMOS transistor, (b) pMOS transistor, (c) inverter, (d) NAND gate, (e) NOR gate.

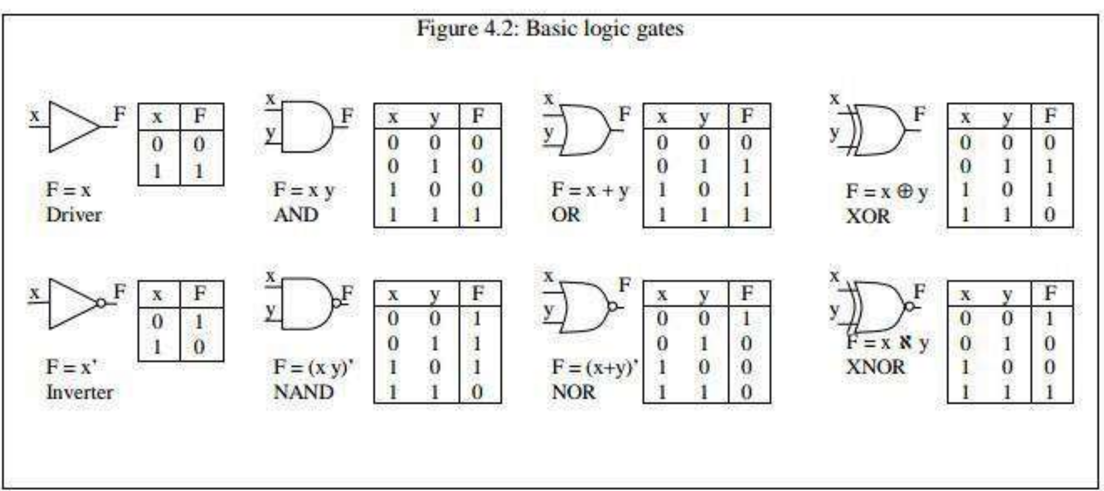
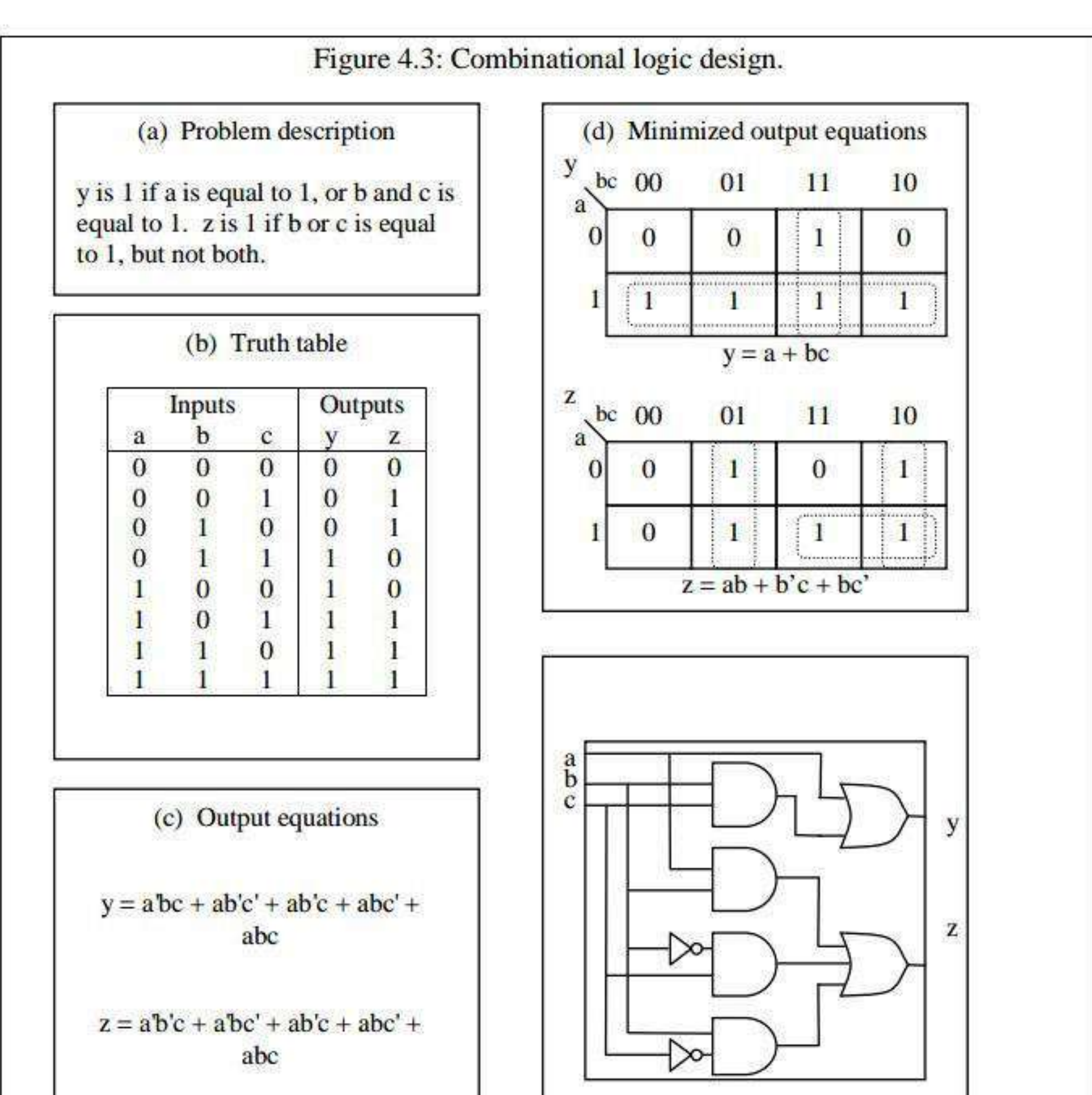


Figure 4.2: Basic logic gates

Digital system designers usually work with logic gates, not transistors. Figure 4.2 describes 8 basic logic gates. Each gate is represented symbolically, with a Boolean equation, and with a truth table. The truth table has inputs on the left, and output on the right. The AND gate outputs 1 if and only if both inputs are 1. The OR gate outputs 1 if and only if at least one of the inputs is 1. The XOR (exclusive-OR) gate outputs 1 if and only if exactly one of its two inputs is 1. The NAND, NOR, and XNOR gates output the complement of AND, OR, and XOR, respectively. As you might have noticed from our transistor implementations, the NAND and NOR gates are actually simpler to build than AND and OR gates.

Basic Combinational Logic Design:

A combinational circuit is a digital circuit whose output is purely a function of its current inputs; such a circuit has no memory of past inputs. We can apply a simple technique to design a combinational circuit using our basic logic gates, as illustrated in Figure 4.3. We start with a problem description, which describes the outputs in terms of the inputs. We translate that description to a truth table, with all possible combinations of input values on the left, and desired output values on the right. For each output column, we can derive an output equation, with one term per row. However, we often want to minimize the logic gates in the circuit. We can minimize the output equations by algebraically manipulating the equations. Alternatively, we can use Karnaugh maps, as shown in the figure. Once we've obtained the desired output equations (minimized or not), we can draw the circuit diagram.



Inputs			Outputs	
a	b	c	y	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

y: a \ bc	00	01	11	10
0	0	0	1	0
1	1	1	1	1

z: a \ bc	00	01	11	10
0	0	1	0	1
1	0	1	1	1

Figure 4.3: Combinational logic design.

R-T Level Combinational Components:

Although we can design all combinational circuits in the above manner, large circuits would be very complex to design. For example, a circuit with 16 inputs would have 2 16 , or 64K, rows in its truth table. One way to reduce the complexity is to use components that are more abstract than logic gates. Figure 4.4 shows several such combinational components. We now describe each briefly

A multiplexor, sometimes called a selector, allows only one of its data inputs Im to pass through to the output O. Thus, a multiplexor acts much like a railroad switch, allowing only one of multiple input tracks to connect to a single output track. If there are m data inputs, then there are log2(m) select lines S, and we call this an m-by-1 multiplexor (m data inputs, one data output). The binary value of S determines which data input passes through; 00...00 means I0 may pass, 00...01 means I1 may pass, 00...10 means I2 may pass, and so on. For example, an 8x1 multiplexor has 8 data inputs and thus 3 select lines. If those three select lines have values of 110, then I6 will pass through to the output. So if I6 is 1, then the output would be 1; if I6 is 0, then the output would be 0. We commonly use a more complex device called an n-bit multiplexor, in which each data input, as well as the output, consists of n lines. Suppose the previous example used a 4-bit 8x1 multiplexor. Thus, if I6 is 0110, then the output would be 0110. Note that n does not affect the number of select lines.

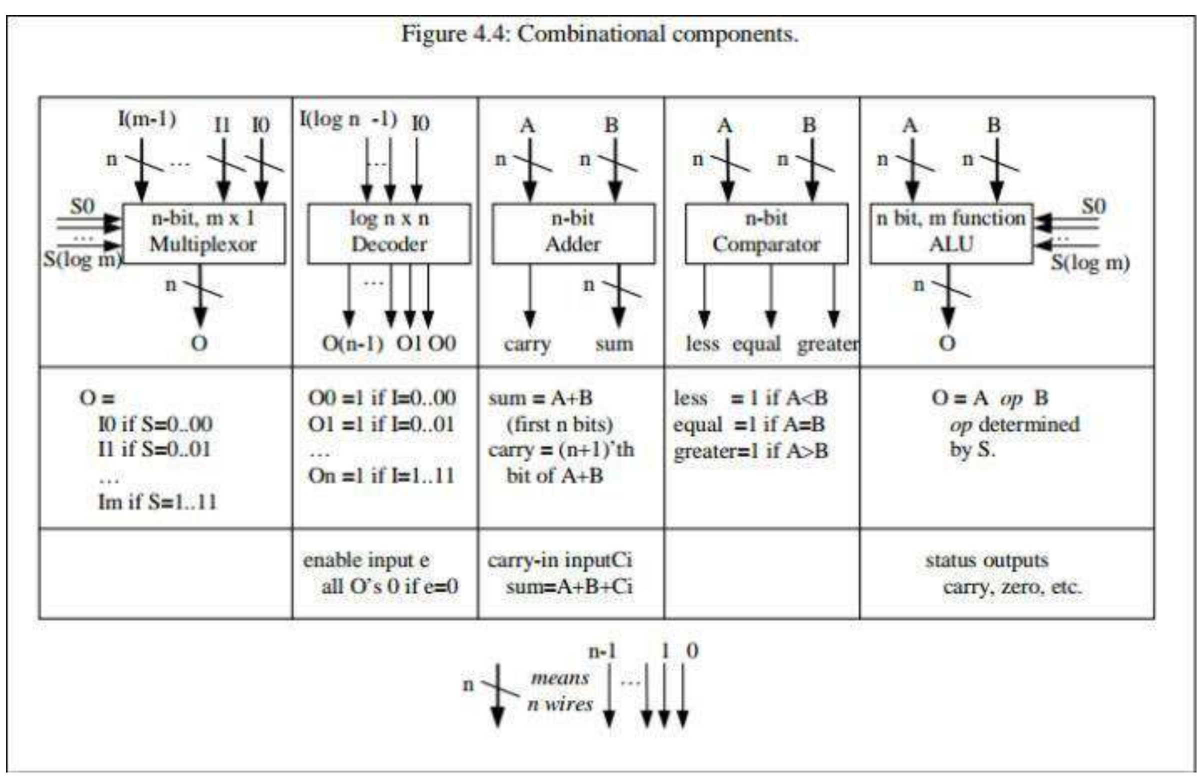


Figure 4.4: Combinational components.

. A decoder converts its binary input I into a one-hot output O. "One-hot" means that exactly one of the output lines can be 1 at a given time. Thus, if there are n outputs, then there must be log2(n) inputs. We call this a log2(n)xn decoder. For example, a 3x8 decoder has 3 inputs and 8 outputs. If the input is 000, then the output O0 will be 1. If the input is 001, then the output O1 would be 1, and so on. A common feature on a decoder is an extra input called enable. When enable is 0, all outputs are 0. When enable is 1, the decoder functions as before.

An adder adds two n-bit binary inputs A and B, generating an n-bit output sum along with an output carry. For example, a 4-bit adder would have a 4-bit A input, a 4-bit B input, a 4-bit sum output, and a 1-bit carry output. If A is 1010 and B is 1001, then sum would be 0011 and carry would be 1.

A comparator compares two n-bit binary inputs A and B, generating outputs that indicate whether A is less than, equal to, or greater than B. If A is 1010 and B is 1001, then less would be 0, equal would be 0, and greater would be 1.

An ALU (arithmetic-logic unit) can perform a variety of arithmetic and logic functions on its n-bit inputs A and B. The select lines S choose the current function; if there are m possible functions, then there must be at least log2(m) select lines. Common functions include addition, subtraction, AND, and OR.

SEQUENTIAL LOGIC

FLIP-FLOPS:

A sequential circuit is a digital circuit whose outputs are a function of the current as well as previous input values. In other words, sequential logic possesses memory. One of the most basic sequential circuits is the flip-flop. A flip-flop stores a single bit. The simplest type of flip-flop is the D flip-flop. It has two inputs: D and clock. When clock is 1, the value of D is stored in the flip-flop, and that value appears at an output Q. When clock is 0, the value of D is ignored; the output Q maintains its value. Another type of flip-flop is the SR flip-flop, which has three inputs: S, R and clock. When clock is 0, the previously stored bit is maintained and appears at output Q. When clock is 1, the inputs S and R are examined. If S is 1, a 1 is stored. If R is 1, a 0 is stored. If both are 0, there's no change. If both are 1, behaviour is undefined. Thus, S stands for set, and R for reset. Another flip-flop type is a JK flip-flop, which is the same as an SR flip-flop except that when both J and K are 1, the stored bit toggles from 1 to 0 or 0 to 1. To prevent unexpected behaviour from signal glitches, flip-flops are typically designed to be edgetriggered, meaning they only pay attention to their non-clock inputs when the clock is rising from 0 to 1, or alternatively when the clock is falling from 1 to 0.

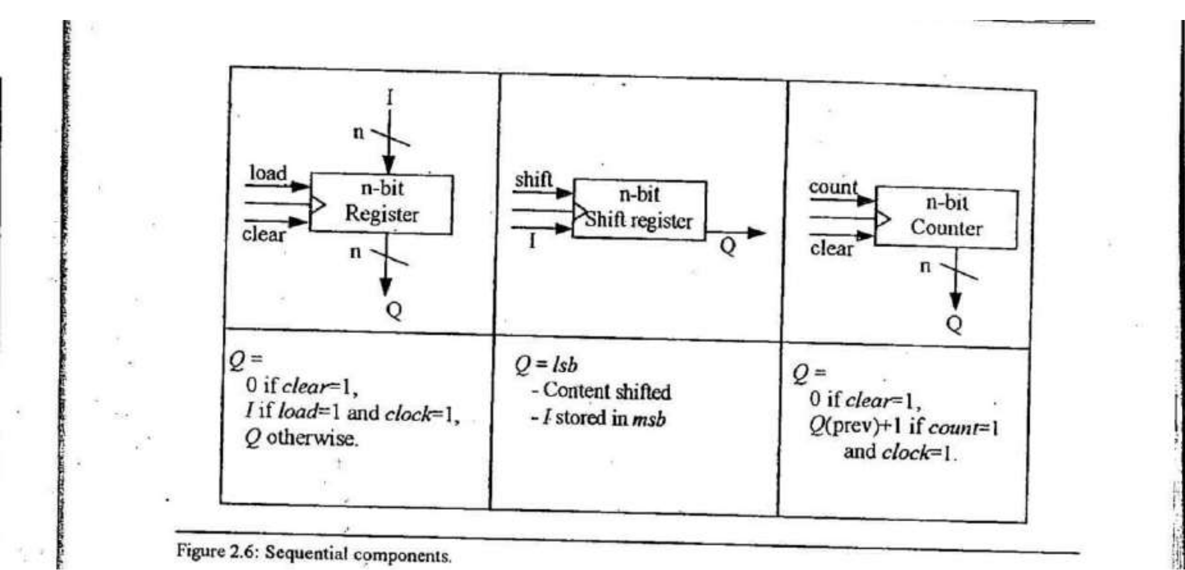


Figure 2.6: Sequential components.

R-T-Level Sequential Components:

Just as we used more abstract combinational components to implement complex combinational systems, we also use more abstract sequential components for complex sequential systems. Figure above illustrates several sequential components, which we now describe.

A register stores n bits from its n-bit data input I, with those stored bits appearing at its output O. A register usually has at least two control inputs, clock and load. For a rising-edge-triggered register, the inputs I are only stored when load is 1 and clock is rising from 0 to 1. The clock input is usually drawn as a small triangle, as shown in the figure. Another common register control input is clear, which resets all bits to 0, regardless of the value of I. Because all n bits of the register can be stored in parallel, we often refer to this type of register as a parallel-load register, to distinguish it from a shift register.

A shift register stores n bits, but these bits cannot be stored in parallel. Instead, they must be shifted into the register serially, meaning one bit per clock edge. A shift register has a one-bit data input I, and at least two control inputs clock and shift. When clock is rising and shift is 1, the value of I is stored in the (n)'th bit, while the (n)'th bit is stored in the (n-1)'th bit, and likewise, until the second bit is stored in the first bit. The first bit is typically shifted out, meaning it appears over an output Q.

A counter is a register that can also increment (add binary 1) to its stored binary value. In its simplest form, a counter has a clear input, which resets all stored bits to 0 and a count input, which enables incrementing on the clock edge. A counter often also has a parallel load data input and associated control signal. A common counter feature is both up and down counting (incrementing and decrementing), requiring an additional control input to indicate the count direction.

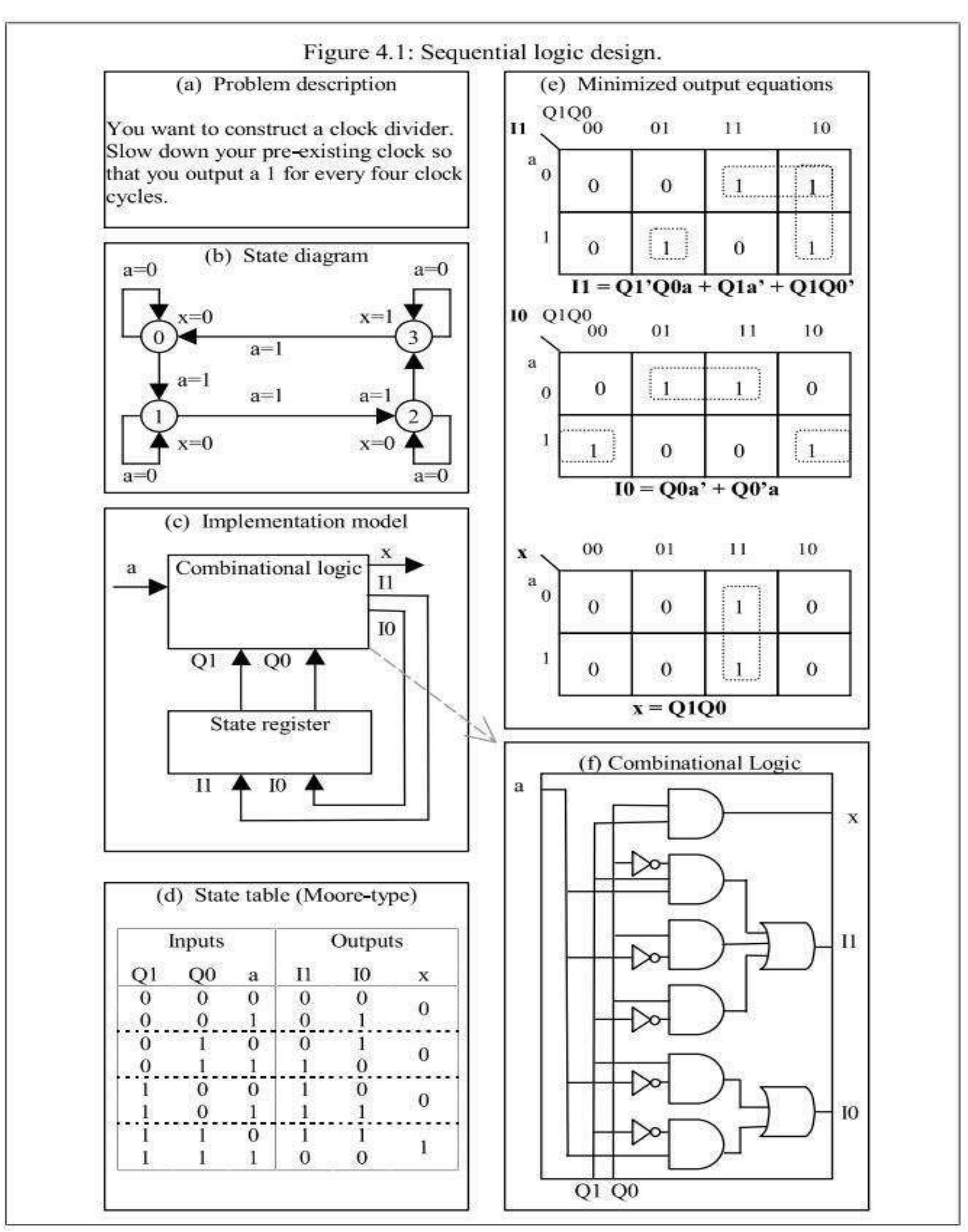
The control inputs discussed above can be either synchronous or asynchronous. A synchronous input's value only has an effect during a clock edge. An asynchronous input's value affects the circuit independent of the clock. Typically, clear control lines are asynchronous.

Sequential Logic Design:

Sequential logic design can be achieved using a straightforward technique, whose steps are illustrated in Figure 4.1. We again start with a problem description. We translate this description to a state diagram. We describe state diagrams further in a later chapter. Briefly, each state represents the current "mode" of the circuit, serving as the circuit's memory of past input values. The desired output values are listed next to each state. The input conditions that cause a transition from one state to another are shown next to each arc. Each arc condition is implicitly AND with a rising (or falling) clock edge. In other words, all inputs are synchronous. State diagrams can also describe asynchronous systems, but we do not cover such systems in this book, since they are not common.

We will implement this state diagram using a register to store the current state, and combinational logic to generate the output values and the next state. We assign each state with a unique binary value, and we then create a truth table for the combinational logic. The inputs for the combinational logic are the state bits coming from the state register, and the

external inputs, so we list all combinations of these inputs on the left side of the table. The outputs for the combinational logic are the state bits to be loaded into the register on the next clock edge (the next state), and the external output values, so we list desired values of these outputs for each input combination on the right side of the table. Because we used a state diagram for which outputs were a function of the current state only, and not of the inputs, we list an external output value only for each possible state, ignoring the external input values. Now that we have a truth table, we proceed with combinational logic design as described earlier, by generating minimized output equations, and then drawing the combinational logic circuit.



Inputs			Outputs		
Q1	Q0	a	I1	I0	x
0	0	0	0	0	0
0	0	1	0	1	
0	1	0	0	1	0
0	1	1	1	0	
1	0	0	1	0	0
1	0	1	1	1	
1	1	0	1	1	1
1	1	1	0	0	

Figure 4.1: Sequential logic design.

CUSTOM SINGLE-PURPOSE PROCESSOR DESIGN:

We now have the knowledge needed to build basic processor. A basic processor consists of a controller and data path shown in below figure. The data path stores and manipulates a

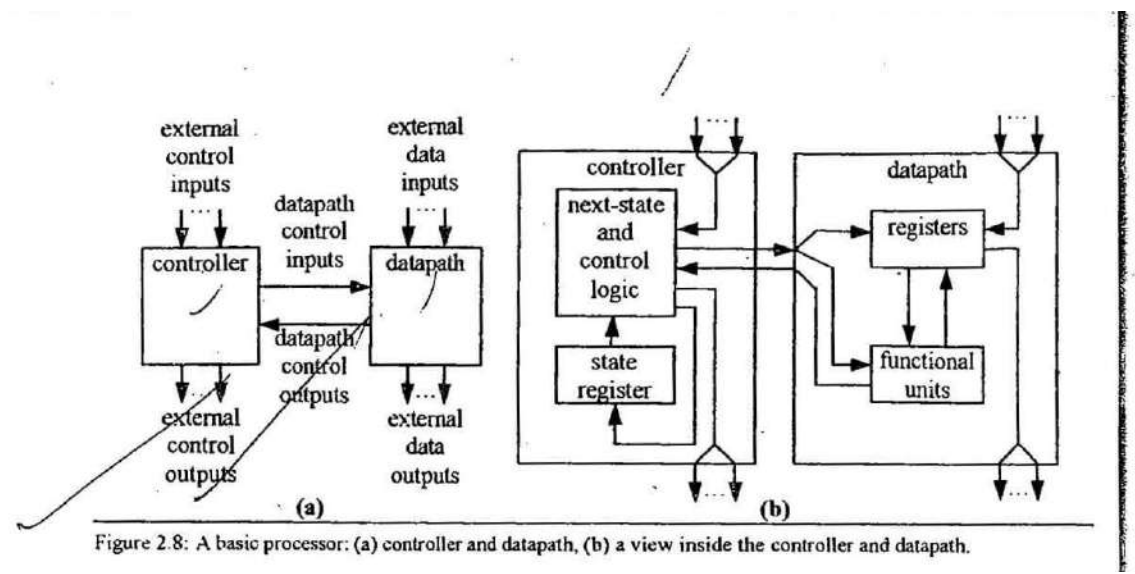


Figure 2.8: A basic processor: (a) controller and datapath, (b) a view inside the controller and datapath.

system's data. Examples of data in an embedded system include binary numbers representing external conditions like temperature or speed, characters to be displayed on a screen, or a digitized photographic image to be stored and compressed. The datapath contains register units, functional units, and connection units like wires and multiplexors. The datapath can be configured to read data from particular registers, feed that data through functional units configured to carry out particular operations like add or shift, and store the operation results back into particular registers. A *controller* carries out such configuration of the datapath. It sets the datapath control inputs, like register load and multiplexor select signals, of the register units, functional units, and connection units to obtain the desired configuration at a particular time. It monitors external control inputs as well as datapath control outputs, known as status signals, coming from functional units, and it sets external control outputs as well.

We can apply the above combinational and sequential logic design techniques to build data path components and controllers. Therefore, we have nearly all the knowledge we need to build a custom single-purpose processor for a given program, since a processor consists of a controller and a data path. We now describe a technique for building such a processor.

We begin with a sequential program we must implement. Figure 4.3 provides a example based on computing a greatest common divisor (GCD). Figure 4.3(a) shows a black-box diagram of the desired system, having x_i and y_i data inputs and a data output d_i. The system's functionality is straightforward: the output should represent the GCD of the inputs. Thus, if the inputs are 12 and 8, the output should be 4. If the inputs are 13 and 5, the output should be 1. Figure 4.3(b) provides a simple program with this functionality. The reader might trace this program's execution on the above examples to verify that the program does indeed compute the GCD.

To begin building our single-purpose processor implementing the GCD program, we first convert our program into a complex state diagram, in which states and arcs may include arithmetics expressions, and these expressions may use external inputs and outputs or variables. In contrast, our earlier state diagrams only included boolean expressions, and these

expressions could only use external inputs and outputs, not variables. Thus, these more complex state diagram looks like a sequential program in which statements have been scheduled into states.

We can use templates to convert a program to a state diagram, as illustrated in Figure 4.2. First, we classify each statement as an assignment statement, loop statement, or branch (if-then-else or case) statement. For an assignment statement, we create a state with that statement as its action. We add an arc from this state to the state for the next statement, whatever type it may be. For a loop statement, we create a condition state C and a join state J, both with no actions. We add an arc with the loop's condition from the condition state to the first statement in the loop body. We add a second arc with the complement of the loop's condition from the condition state to the next statement after the loop body. We also add an arc from the join state back to the condition state. For a branch statement, we create a condition state C and a join state J, both with no actions. We add an arc with the first branch's condition from the condition state to the branch's first statement. We add another arc with the complement of the first branch's condition AND with the second branches condition from the condition state to the branches first statement. We repeat this for each branch. Finally, we connect the arc leaving the last statement of each branch to the join state, and we add an arc from this state to the next statement's state.

Using this template approach, we convert our GCD program to the complex state diagram of Figure 4.3(c). We are now well on our way to designing a custom single-purpose processor that executes the GCD program. Our next step is to divide the functionality into a data path part and a controller part, as shown in Figure 4.4. The data path part should consist of an interconnection of combinational and sequential components. The controller part should consist of a basic state diagram, i.e., one containing only boolean actions and conditions.

We construct the datapath through a four-step process:

1. First, we create a register for any declared variable. In the example, these are x and y. We treat an output port as having an implicit variable, so we create a register d and connect it to the output port. We also draw the input and output ports.

2. Second, we create a functional unit for each arithmetic operation in the state diagram. In the example, there are two subtractions, one comparison for less than, and one comparison for inequality, yielding two subtractors and two comparators, as shown in the figure.

3. Third, we connect the ports, registers and functional units. For each write to a variable in the state diagram, we draw a connection from the write's source (an input port, a functional unit, or another register) to the variable's register. For each arithmetic and logical operation, we connect sources to an input of the operation's corresponding functional unit. When more than one source is connected to a register, we add an appropriately-sized multiplexor.

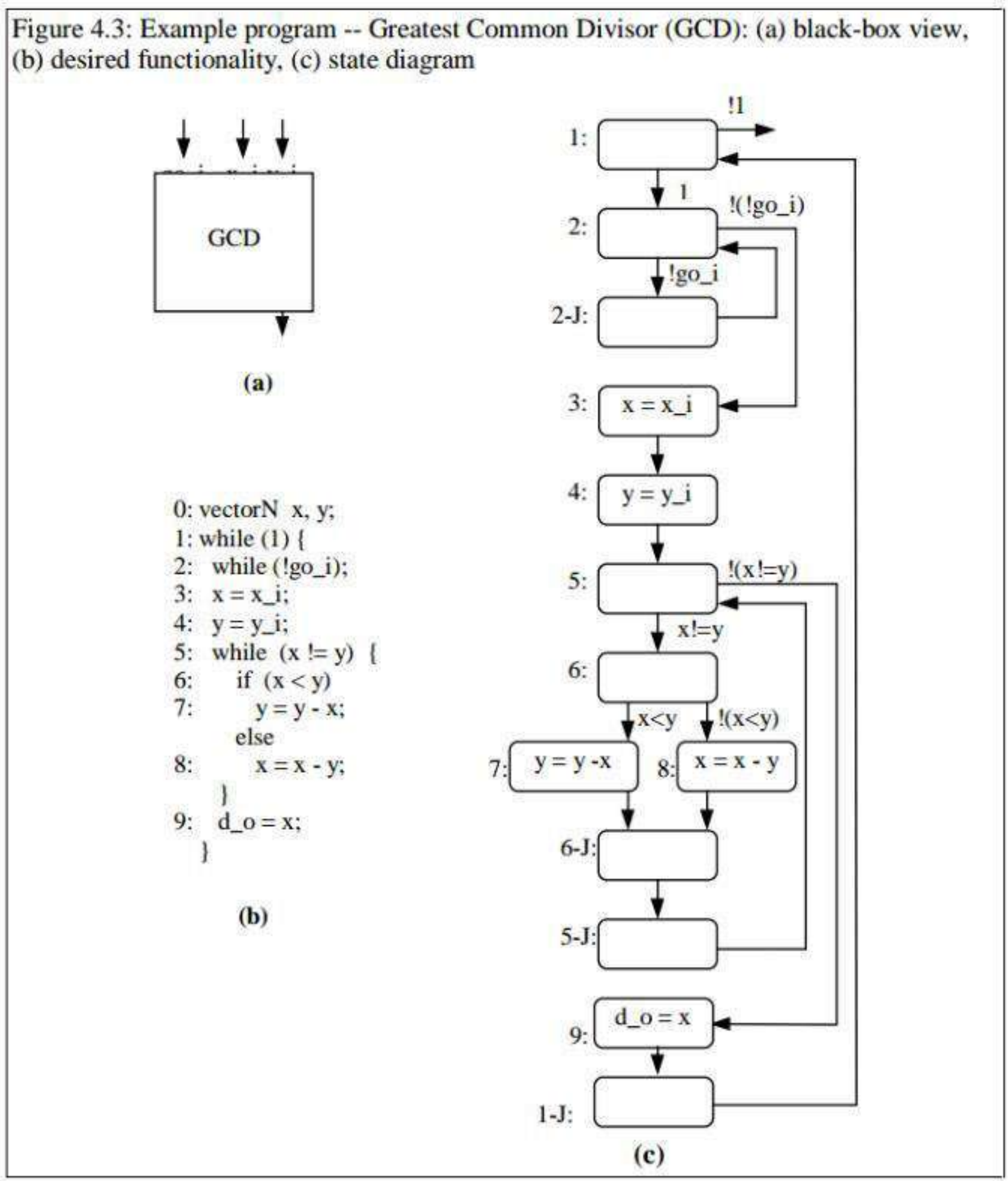


Figure 4.3: Example program -- Greatest Common Divisor (GCD): (a) black-box view, (b) desired functionality, (c) state diagram

4. Finally, we create a unique identifier for each control input and output of the data path components.

Now that we have a complete data path, we can build a state diagram for our controller. The state diagram has the same structure as the complex state diagram. However, we replace complex actions and conditions by boolean ones, making use of our data path. We replace every variable write by actions that set the select signals of the multiplexor in front of the variable's register's such that the write's source passes through, and we assert the load signal of that register. We replace every logical operation in a condition by the corresponding functional unit control output.

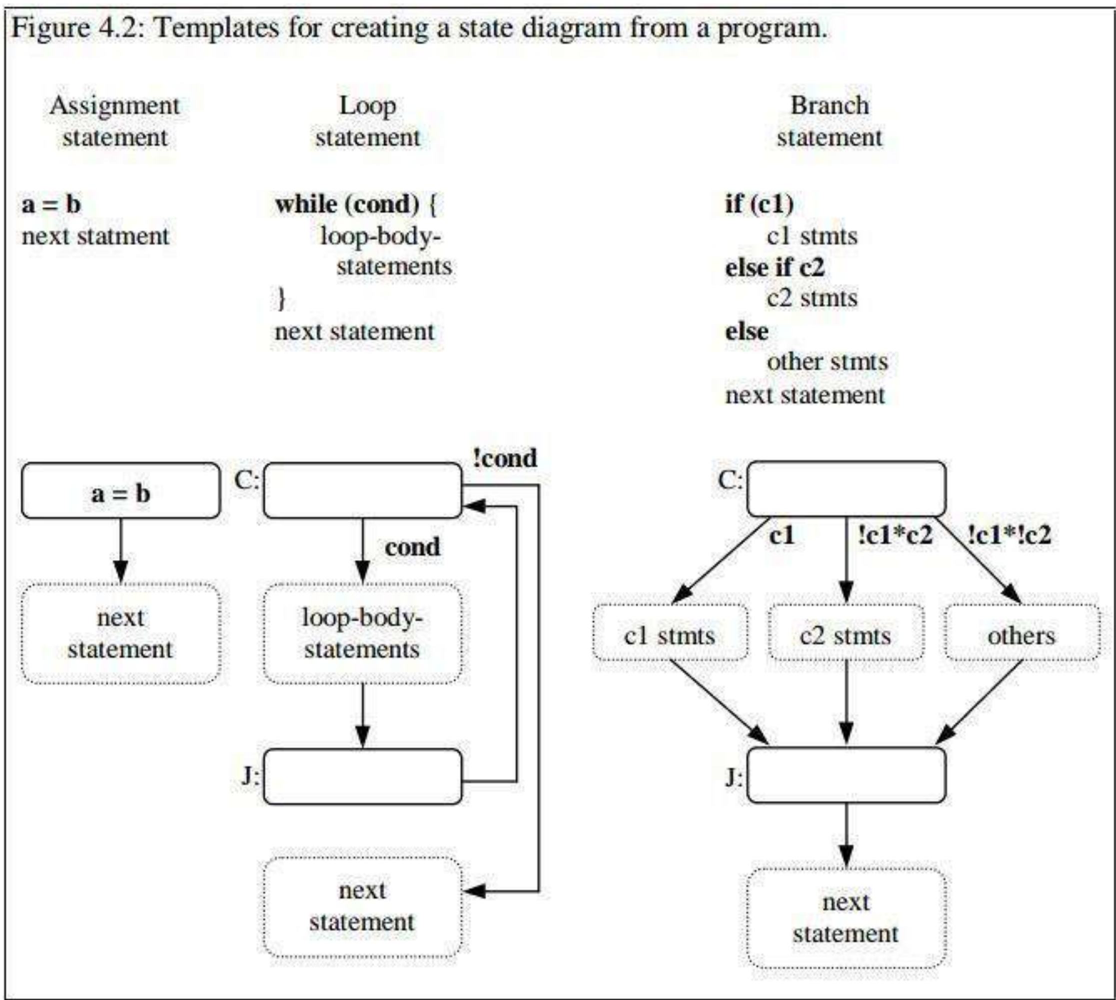
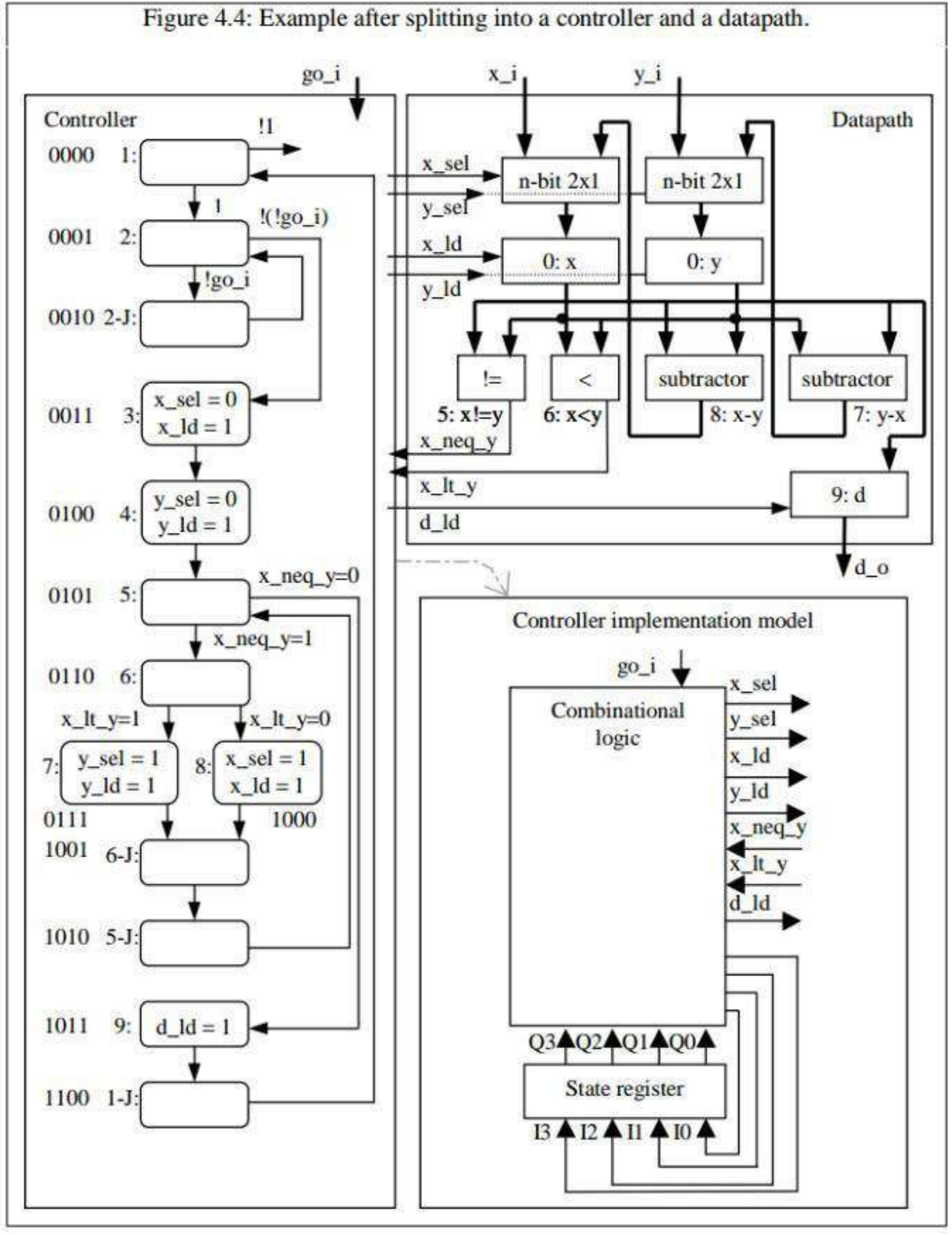


Figure 4.2: Templates for creating a state diagram from a program.

We can then complete the controller design by implementing the state diagram using our sequential design technique described earlier. Figure 4.4 shows the controller implementation model, and Figure 4.5 shows a state table.

Note that there are 7 inputs to the controller, resulting in 128 rows for the table. We reduced rows in the state table by using don't cares for some input combinations, but we can still see that optimizing the design using hand techniques could be quite tedious. For this reason, computer-aided design (CAD) tools that automate the combinational as well as sequential logic design can be very helpful.



Also, note that we could perform significant amounts of optimization to both the data path and the controller. For example, we could merge functional units in the data path, resulting in fewer units at the expense of more multiplexors. We could also merge states in the data path.

Figure 4.5: State table for the GCD example.

State table

Inputs							Outputs								
Q3	Q2	Q1	Q0	x_neq_y	x_lt_y	go_i	I3	I2	I1	I0	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
0	0	0	1	*	*	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
0	1	0	1	1	*	*	0	1	1	0	X	X	0	0	0
0	1	1	0	*	0	*	1	0	0	0	X	X	0	0	0
0	1	1	0	*	1	*	0	1	1	1	X	X	0	0	0
0	1	1	1	*	*	*	1	0	0	1	X	1	0	1	0
1	0	0	0	*	*	*	1	0	0	1	1	X	1	0	0
1	0	0	1	*	*	*	1	0	1	0	X	X	0	0	0
1	0	1	0	*	*	*	0	1	0	1	X	X	0	0	0
1	0	1	1	*	*	*	1	1	0	0	X	X	0	0	1
1	1	0	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	0	1	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	1	*	*	*	0	0	0	0	X	X	0	0	0

* - indicates all possible combinations of 0's and 1's

X - indicates don't cares

Remember that we could alternatively implement the GCD program by programming a microcontroller, thus eliminating the need for this design process, but possibly yielding slower and bigger design.

RT-Level Custom Single-Purpose Processor Design

Section 2.4 described a basic technique for converting a sequential program into a custom single-purpose processor, by first converting the program to an FSMD using the provided templates for each language construct, splitting the FSMD into a simple FSM controlling a datapath, and performing sequential logic design on the FSM. However, in many cases, we prefer not to start with a program, but instead directly with an FSMD. The reason is that often the cycle-by-cycle timing of a system is central to the design, but programming languages don't typically support cycle-by-cycle description. FSMDs, in contrast, make cycle-by-cycle timing explicit.

For example, consider the design problem in Figure 2.13(a). We want one device (the sender) to send an 8-bit number to another device (the receiver). The problem is that while the receiver can receive all 8 bits at once, the sender sends 4 bits at a time; first it sends the low-order 4 bits, then the high-order 4 bits. So we need to design a bridge that will enable to two devices to communicate.

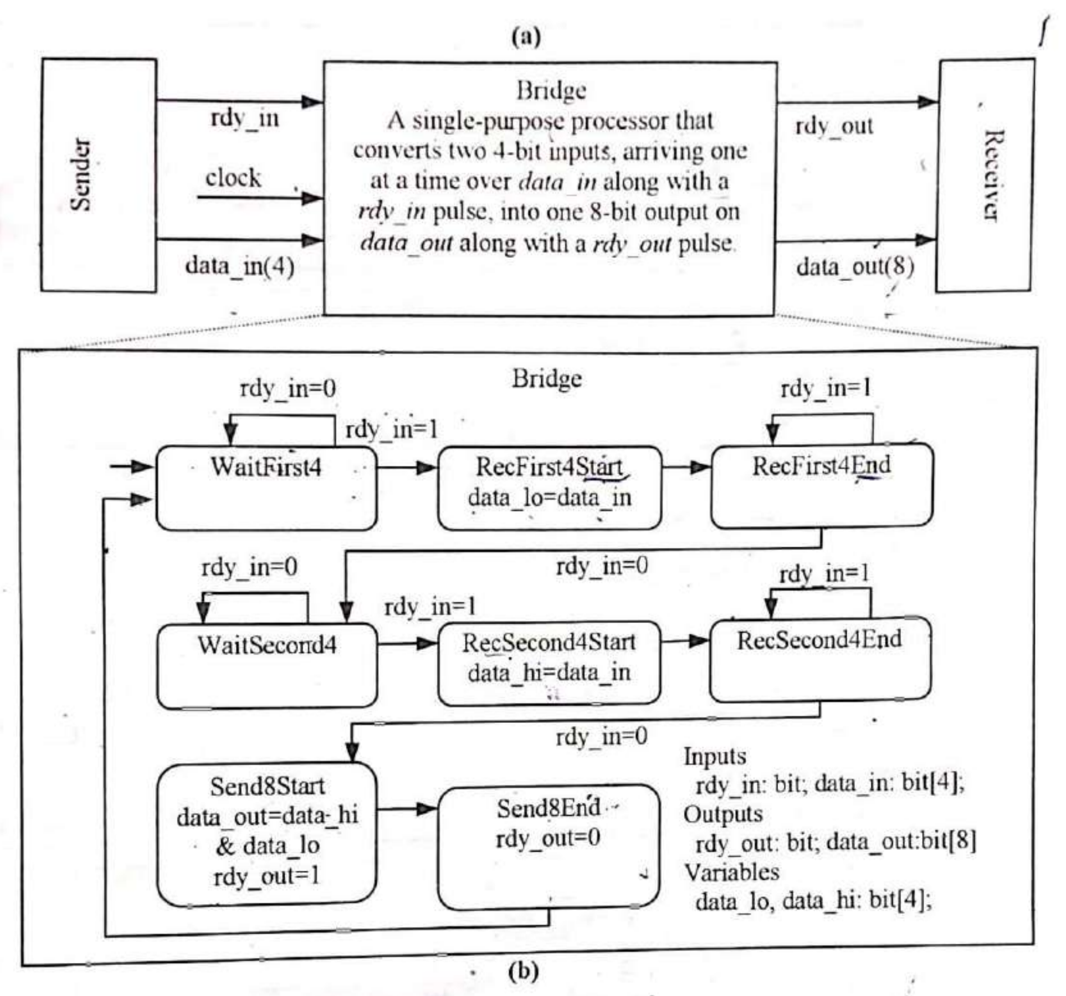


Figure 2.13: RT-level custom single-purpose processor design example: (a) problem specification, (b) FSMD.

Different designers might attack this problem at different levels of abstraction. One designer might start thinking in terms of registers, multiplexors, and flip-flops. Another might try to describe the bridge as a sequential program. But perhaps the most natural level is to describe the bridge as an FSMD, as shown in Figure 2.13(b). We begin by creating a state *WaitFirst4* that waits for the first 4 bits, whose presence on *data_in* will be indicated by a pulse on *rdy_in*. Once the pulse is detected, we transition to a state *RecFirst4Start* that saves the contents of *data_in* in a variable called *data_lo*. We then wait for the pulse on *rdy_in* to end, and then wait for the other 4 bits, indicated by a second pulse on *rdy_in*. We save the contents of *data_in* in a variable called *data_hi*. After waiting for the second pulse on *rdy_in* to end, we write the full 8 bits of data to the output *data_out*, and we pulse *rdy_out*. We

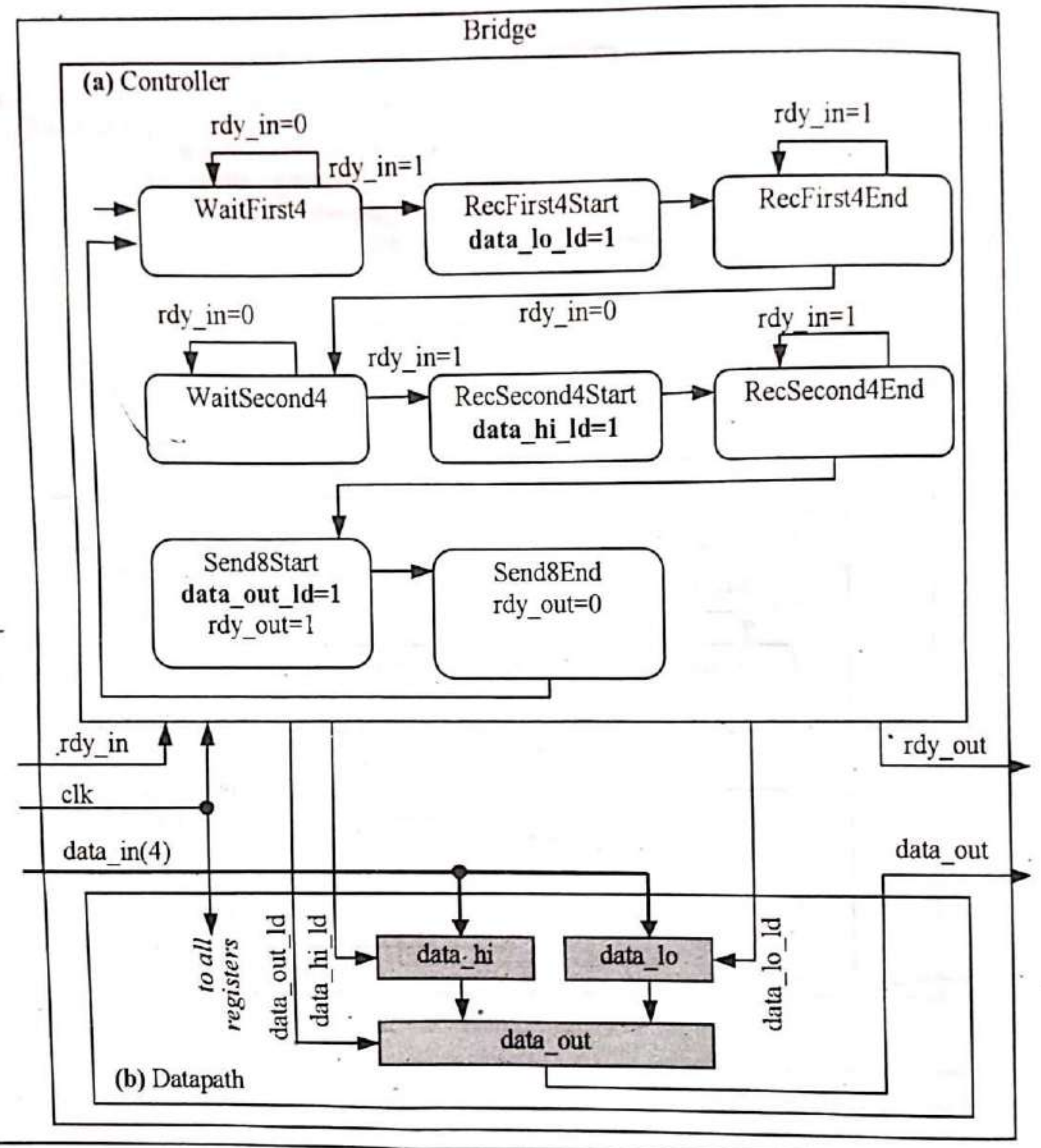


Figure 2.14: RT-level custom single-purpose processor design example continued: (a) controller, (b) datapath.

assume we are building a synchronous circuit, so the bridge has a clock input — in our FSMD, every transition is implicitly ANDed with the clock.

We apply the same methods as before to convert this FSMD to a controller and a datapath implementation, as illustrated in Figure 2.14. We build a datapath, shown in Figure 2.14(b), using the four-step process outlined before. We add registers for *data_hi* and *data_lo*, as well as for the output *data_out*. We don't add any functional units since there are no arithmetic operations. We connect the registers according to the assignments in the FSMD; no multiplexors are necessary. We create unique identifiers for the register control signals. Having completed the datapath, we convert the FSMD into an FSM that uses the datapath, as

shown in Figure 2.14(a). This conversion requires only three simple changes, as shown in bold in the figure. Having obtained the FSM, we can convert the FSM into a state-register and combinational logic using the same technique as in Figure 2.7; we omit this conversion here.

This example demonstrates how a problem that consists mostly of waiting for or making changes on signals, rather than consisting mostly of performing computations on data, might most easily be described as an FSMD. The FSMD would be even more appropriate if specific numbers of clock cycles were specified (e.g., the input pulse would be held high exactly two cycles and the output pulse would have to be held high for three cycles). On the other hand, if a problem consists mostly of an algorithm with lots of computations, the detailed timing of which are not especially important, such as the GCD computation in the earlier example, then a program might be the best starting point.

The FSMD level is often referred to as the register-transfer (RT) level, since an FSMD describes in each state which registers should have their data transferred to which other registers, with that data possibly being transformed along the way. The RT-level is probably the most common starting point for custom single-purpose processor design today.

Some custom single-purpose processors do not manipulate much data. These processors consist primarily of a controller, with perhaps no datapath or a trivial one with just a couple registers or counters, as in our bridge example of Figure 2.14. Likewise, other custom single-purpose processors do not exhibit much control. These processors consist primarily of a datapath configured to do one or a few things repeatedly, with no controller or a trivial one with just a couple flip-flops and gates. Nevertheless, we can still think of these circuits as processors.

2.6 Optimizing Custom Single-Purpose Processors

You may have noticed in the GCD example of Figure 2.11 that we ignored several opportunities to simplify the resulting design. For example, the FSM had several states that obviously do nothing and could have been removed. Likewise, the datapath has two adders whereas one would have been sufficient. We intentionally did not perform such optimizations so as not to detract from the basic idea that programs can be converted to custom single-purpose processors through a series of straightforward steps. However, when we really design such processors, we will usually also want to optimize them whenever possible. Optimization is the task of making design metric values the best possible. Optimization is an extensive subject, and we do not intend to cover it in depth here. Instead, we point out some simple optimizations that can be applied, and refer the reader to textbooks on the subject.

Optimizing the Original Program

Let us start with optimizing the initial program, such as the GCD program in Figure 2.9. At this level, we can analyze the number of computations and size of variables that are required by the algorithm. In other words, we can analyze the algorithm in terms of time complexity and space complexity. We can try to develop alternative algorithms that are more efficient. In

the GCD example, if we assume we can make use of a modulo operation %, we could write an algorithm that would use fewer steps. In particular, we could use the following algorithm:

```
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;} // x must be the larger number
   while  (y != 0)  {
      r = x % y;
      x = y;
      y = r;
   }
   d_o = x;
}
```

Let us compare this second algorithm with the earlier one when computing the GCD of 42 and 8. The earlier algorithm would step through its inner loop with x and y values as follows: (42,8), (34,8), (26,8), (18,8), (10,8), (2,8), (2,6), (2,4), (2,2), thus outputting 2. The second algorithm would step through its inner loop with x and y values as follows: (42,8), (8,2), (2,0), thus outputting 2. The second algorithm is far more efficient in terms of time. Analysis of algorithms and their efficient design is a widely researched area. The choice of algorithm can have perhaps the biggest impact on the efficiency of the designed processor.

Optimizing the FSMD

Once an algorithm is settled upon, we convert the program describing that algorithm to an FSMD. Use of the template-based method introduced in this chapter will result in a rather inefficient FSMD. In particular, many states in the resulting FSMD could likely be merged into fewer states.

Scheduling is the task of assigning operations from the original program to states in an FSMD. The scheduling obtained using the template-based method can be improved. Consider the original FSMD for the GCD, which is redrawn in Figure 2.15(a). State 1 is clearly not necessary since its outgoing transitions have constant values. States 2 and 2-J can be merged into a single state since there are no loop operations in between them. States 3 and 4 can be merged since they perform assignment operations that are independent of one another. States 5 and 6 can be merged. States 6-J and 5-J can be eliminated, with the transitions from states 7 and 8 pointing directly to state 5. Likewise, state 1-J can be eliminated. The resulting reduced FSMD is shown in Figure 2.15(b). We reduced the FSMD from thirteen states to only six states. Be careful, though, to avoid the common mistake of assuming that a variable assigned in a state can have the newly assigned value read on an outgoing arc of that state!

The original FSMD could also have had too few states to be efficient in terms of hardware size. Suppose a particular program statement had the operation $a = b * c * d * e$. Generating a single state for this operation will require us to use three multipliers in our datapath. However, multipliers are expensive, and thus we might instead want to break this

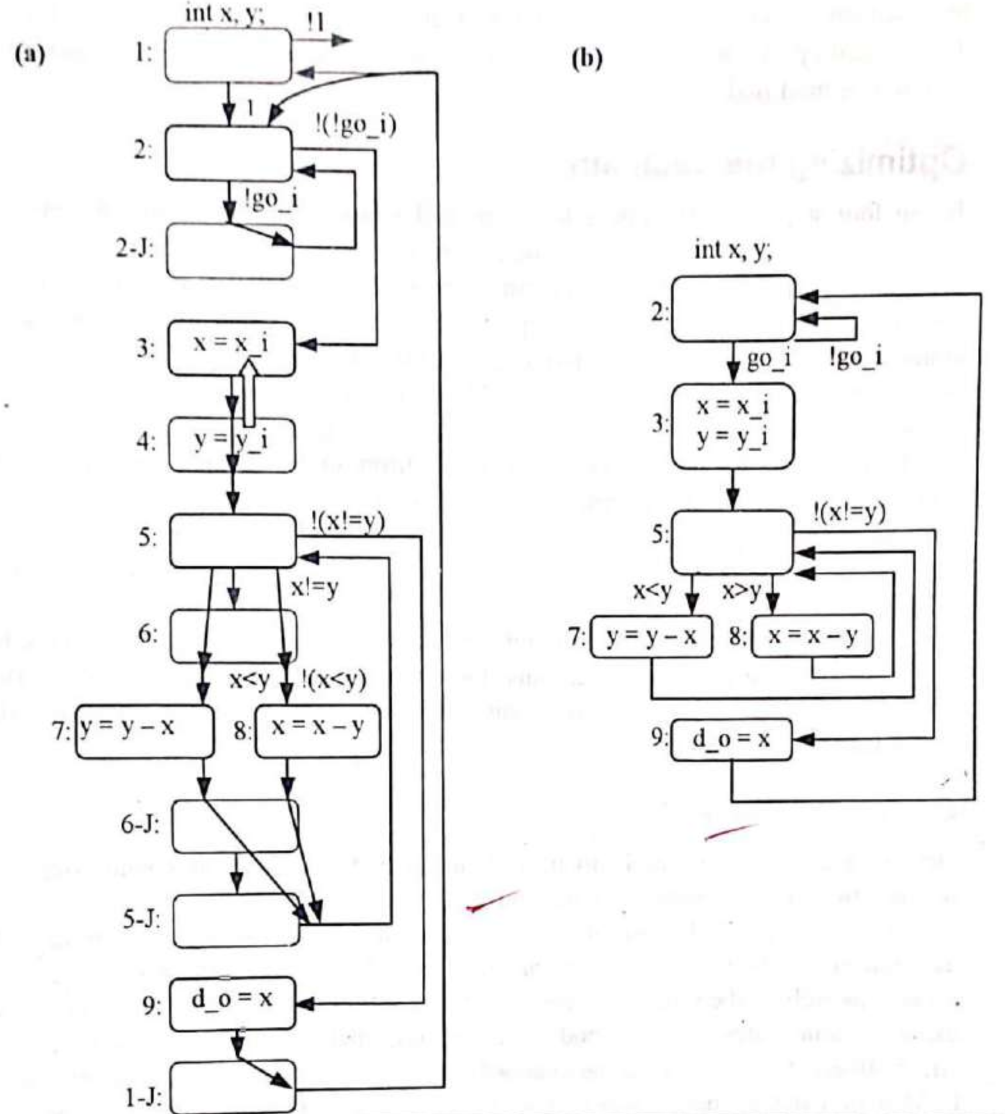


Figure 2.15: Optimizing the FSMD for the GCD example: (a) original FSMD and optimizations, and (b) optimized FSMD.

operation down into smaller operations, like $t1 = b * c$, $t2 = d * e$, and $a = t1 * t2$, with each smaller operation having its own state. Thus, only one multiplier would be needed in the datapath, since the three multiplications could share multiplier; sharing will be discussed in the next section.

In this scenario, we assumed that the timing of output operations could be changed. For example, the reduced FSMD will generate the GCD output in fewer clock cycles than the original FSMD. In many cases, changing the timing is not acceptable. For example, in our earlier clock divider example, changing the timing clearly would not be acceptable, since we

intended for the cycle-by-cycle behavior of the original FSM to be preserved during design. Thus, when optimizing the FSMD, a design must be aware of whether output timing may or may not be modified.

Optimizing the Datapath

In our four-step datapath approach, we created a unique functional unit for every arithmetic operation in the FSMD. However, such a one-to-one-mapping is often not necessary. Many arithmetic operations in the FSMD can share a single functional unit if that functional unit supports those operations, and those operations occur in different states. In the GCD example, states 7 and 8 both performed subtractions. In the datapath of Figure 2.11, each subtraction got its own subtractor. Instead, we could use a single subtractor and use multiplexors to choose whether the subtractor inputs are x and y, or instead y and x.

Furthermore, we often have a number of different RT components from which we can build our datapath. For example, we have fast and slow adders available. We may have multifunction components, like ALUs, also. *Allocation* is the task of choosing which RT components to use in the datapath. *Binding* is the task of mapping operations from the FSMD to allocated components.

Scheduling, allocation, and binding are highly interdependent. A given schedule will affect the range of possible allocations, for example. An allocation will affect the range of possible schedules. And so on. Thus, we sometimes want to consider these tasks simultaneously.

Optimizing the FSM

Designing a sequential circuit to implement an FSM also provides some opportunities for optimization, namely, state encoding and state minimization.

State encoding is the task of assigning a unique bit pattern to each state in an FSM. Any assignment in which the encodings are unique will work properly, but the size of the state register as well as the size of the combinational logic may differ for different encodings. For example, four states A, B, C, and D can be encoded as 00, 01, 10, and 11, respectively. Alternatively, those states can be encoded as 11, 10, 00, and 01, respectively. In fact, for an FSM with n states where n is a power of 2, there are $n!$ possible encodings. We can see this easily if we treat encoding as an ordering problem — we order the states and assign a straightforward binary encoding, starting with 00...00 for the first state, 00...01 for the second state, and so on. There are $n!$ possible orderings of n items, and thus $n!$ possible encodings. $n!$ is a very large number for large n, and thus checking each encoding to determine which yields the most efficient controller is a hard problem. Even more encodings are possible, since we can use more than $\log_2(n)$ bits to encode n states, up to n bits to achieve a one-hot encoding. CAD tools are therefore a great aid in searching for the best encoding.

State minimization is the task of merging equivalent states into a single state. Two states are equivalent if, for all possible input combinations, those two states generate the same outputs and transition to the same next state. Such states are clearly equivalent, since merging them will yield exactly the same output behavior.

The state merging that we did when optimizing our FSMD was not the same as state minimization as defined here. The reason is that our state merging in the FSMD actually changed the output behavior, in particular the output timing, of the FSMD. Typically, by the time we arrive at an FSM, we assume output timing cannot be changed. State minimization does not change the output behavior in any way.

ESD UNIT-3

GENERAL PURPOSE PROCESSORS: SOFTWARE

BASIC ARCHITECHTURE:

A general-purpose processor, sometimes called a CPU (Central Processing Unit) or a microprocessor, consists of a data path and a controller, tightly linked with a memory. We now discuss these components briefly. Figure 2.1 illustrates the basic architecture.

Figure 2.1: General-purpose processor basic architecture.

Datapath: The data path consists of the circuitry for transforming data and for storing temporary data. The data path contains an arithmetic-logic unit (ALU) capable of transforming data through operations such as addition, subtraction, logical AND, logical OR, inverting, and shifting. The ALU also generates status signals, often stored in a status register (not shown), indicating particular data conditions. Such conditions include indicating whether data is zero, or whether an addition of two data items generates a carry. The data path also contains registers capable of storing temporary data. Temporary data may include data brought in from memory but not yet sent through the ALU, data coming from the ALU that will be needed for later ALU operations or will be sent back to memory, and data that must be moved from one memory location to another. The internal data bus is the bus over which data travels within the data path, while the external data bus is the bus over which data is brought to and from the data memory.

We typically distinguish processors by their size, and we usually measure size as the bit-width of the data path components. A bit, which stands for binary digit, is the processor's basic data unit, representing either a 0 (low or false) or a 1 (high or true), while we refer to 8 bits as a byte. An N-bit processor may have N-bit wide registers, an N-bit wide ALU, an N-bit wide internal bus over which data moves among data path components, and an N-bit wide external bus over which data is brought in and out of the data path. Common processor sizes

include 4-bit, 8-bit, 16-bit, 32-bit and 64-bit processors. However, in some cases, a particular processor may have different sizes among its registers, ALU, internal bus, or external bus, so the processor-size definition is not an exact one. For example, a processor may have a 16-bit internal bus, ALU and registers, but only an 8-bit external bus to reduce pins on the processor's IC.

CONTROLLER: The controller consists of circuitry for retrieving program instructions, and for moving data to, from, and through the data path according to those instructions. The controller contains a program counter (PC) that holds the address in memory of the next program instruction to fetch. The controller also contains an instruction register (IR) to hold the fetched instruction. Based on this instruction, the controller's control logic generates the appropriate signals to control the flow of data in the data path. Such flows may include inputting two particular registers into the ALU, storing ALU results into a particular register, or moving data between memory and a register. Finally, the next-state logic determines the next value of the PC. For a non-branch instruction, this logic increments the PC. For a branch instruction, this logic looks at the data path status signals and the IR to determine the appropriate next address.

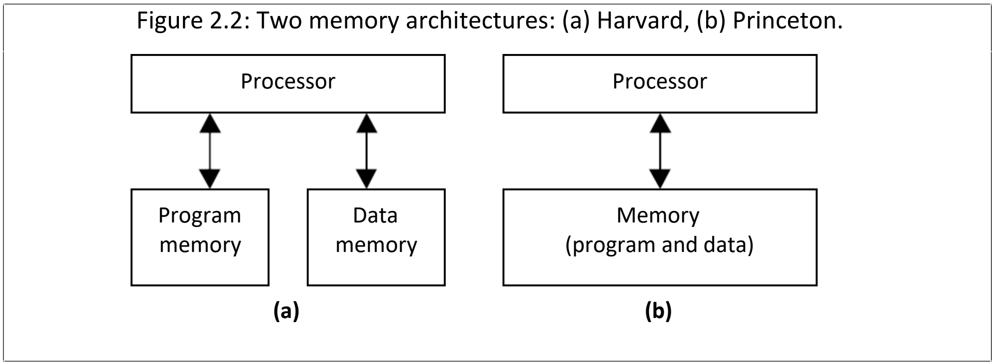
The PC's bit-width represents the processor's address size. The address size is independent of the data word size; the address size is often larger. The address size determines the number of directly accessible memory locations, referred to as the address space or memory space. If the address size is M, then the address space is 2 M. Thus, a processor with a 16-bit PC can directly address 2 16 = 65,536 memory locations. We would typically refer to this address space as 64K, although if 1K = 1000, this number would represent 64,000, not the actual 65,536. Thus, in computer-speak, 1K = 1024.

For each instruction, the controller typically sequences through several stages, such as fetching the instruction from memory, decoding it, fetching operands, executing the instruction in the data path, and storing results. Each stage may consist of one or more clock cycles. A clock cycle is usually the longest time required for data to travel from one register to another. The path through the data path or controller that results in this longest time (e.g., from a data path register through the ALU and back to a data path register) is called the critical path. The inverse of the clock cycle is the clock frequency, measured in cycles per second, or Hertz (Hz). For example, a clock cycle of 10 nanoseconds corresponds to a frequency of 1/10x10 -9 Hz, or 100 M Hz. The shorter the critical path, the higher the clock frequency. We often use clock frequency as one means of comparing processors, especially different versions of the same processor, with higher clock frequency implying faster program execution (though this isn't always true).

MEMORY: While registers serve a processor's short term storage requirements, memory serves the processor's medium and long-term information-storage requirements. We can classify stored information as either program or data. Program information consists of the sequence of instructions that cause the processor to carry out the desired system functionality. Data information represents the values being input, output and transformed by the program.

We can store program and data together or separately. In a Princeton architecture, data and program words share the same memory space. In a Harvard architecture, the program memory space is distinct from the data memory space. Figure 2.2 illustrates these two methods. The Princeton architecture may result in a simpler hardware connection to memory, since only one connection is necessary. A Harvard architecture, while requiring two connections, can perform instruction and data fetches simultaneously, so may result in improved performance. Most machines have a Princeton architecture. The Intel 8051 is a well-known Harvard architecture.

Figure 2.2: Two memory architectures: (a) Harvard, (b) Princeton.



Memory may be read-only memory (ROM) or readable and writable memory (RAM). ROM is usually much more compact than RAM. An embedded system often uses ROM for program memory, since, unlike in desktop systems, an embedded system's program does not change. Constant-data may be stored in ROM, but other data of course requires RAM.

Memory may be on-chip or off-chip. On-chip memory resides on the same IC as the processor, while off-chip memory resides on a separate IC. The processor can usually access on-chip memory must faster than off-chip memory, perhaps in just one cycle, but finite IC capacity of course implies only a limited amount of on-chip memory.

To reduce the time needed to access (read or write) memory, a local copy of a portion of memory may be kept in a small but especially fast memory called cache, as illustrated in Figure 2.3. Cache memory often resides on-chip, and often uses fast but expensive static RAM technology rather than slower but cheaper dynamic RAM (see Chapter 5). Cache memory is based on the principle that if at a particular time a processor accesses a particular memory location, then the processor will likely access that location and immediate neighbours of the location in the near future. Thus, when we first access a location in memory, we copy that location and some number of its neighbours (called a block) into cache, and then access the copy of the location in cache. When we access another location, we first check a cache table to see if a copy of the location resides in cache. If the copy does reside in cache, we have a cache hit, and we can read or write that location very quickly. If the copy does not reside in cache, we have a cache miss, so we must copy the location's block into cache, which takes a lot of time. Thus, for a cache to be effective in improving performance, the ratio of hits to misses must be very high, requiring intelligent caching schemes. Caches are used for both program memory (often called instruction cache, or I-cache) as well as data memory (often called D-cache).

Figure 2.3: Cache memory.

Fast/expensive technology, usually on the same chip

Processor

Cache

Memory

Slower/cheaper technology, usually on a different chip

OPERATION:

Instruction execution: We can think of a microprocessor's execution of instructions as consisting of several basic stages:

1. Fetch instruction: the task of reading the next instruction from memory into the instruction register.

2. Decode instruction: the task of determining what operation the instruction in the instruction register represents (e.g., add, move, etc.).

3. Fetch operands: the task of moving the instruction's operand data into appropriate registers.

4. Execute operation: the task of feeding the appropriate registers through the ALU and back into an appropriate register.

5. Store results: the task of writing a register into memory. If each stage takes one clock cycle, then we can see that a single instruction may take several cycles to complete.

Pipelining: Pipelining is a common way to increase the instruction throughput of a microprocessor. We first make a simple analogy of two people approaching the chore of washing and drying 8 dishes. In one approach, the first person washes all 8 dishes, and then the second person dries all 8 dishes. Assuming 1 minute per dish per person, this approach requires 16 minutes. The approach is clearly inefficient since at any time only one person is working and the other is idle. Obviously, a better approach is for the second person to begin drying the first dish immediately after it has been washed. This approach requires only 9 minutes -- 1 minute for the first dish to be washed, and then 8 more minutes until the last dish is finally dry. We refer to this latter approach as pipelined.

Each dish is like an instruction, and the two tasks of washing and drying are like the five stages listed above. By using a separate unit (each akin a person) for each stage, we can pipeline instruction execution. After the instruction fetch unit fetches the first instruction, the decode unit decodes it while the instruction fetch unit simultaneously fetches the next instruction. The idea of pipelining is illustrated in Figure 2.4. Note that for pipelining to work

well, instruction execution must be decomposable into roughly equal length stages, and instructions should each require the same number of cycles.

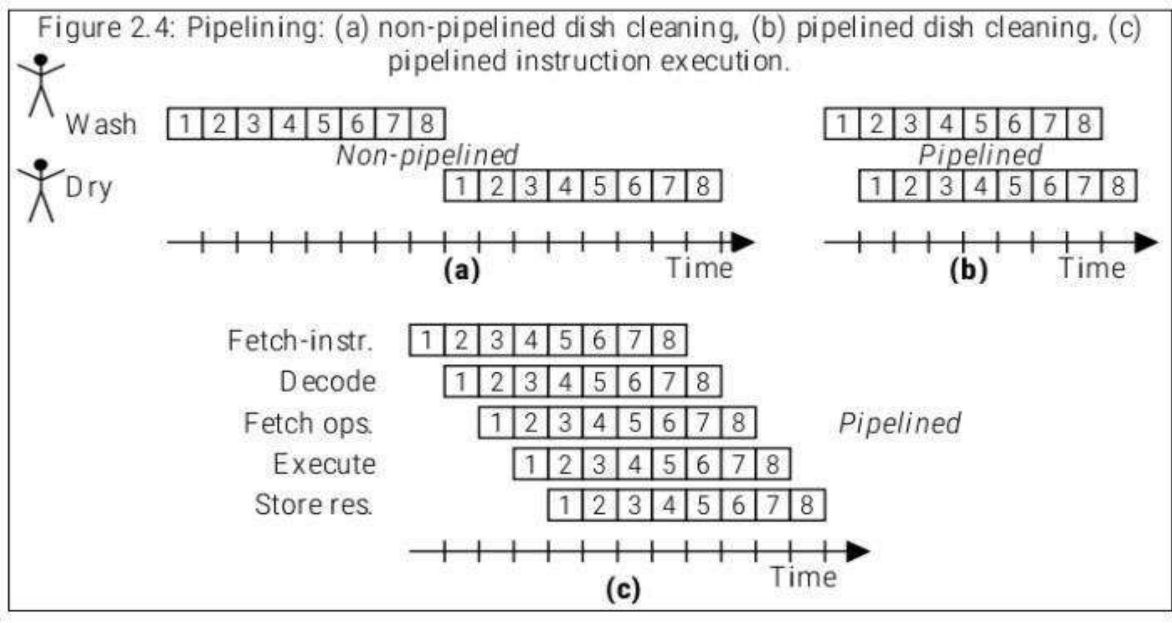


Figure 2.4: Pipelining: (a) non-pipelined dish cleaning, (b) pipelined dish cleaning, (c) pipelined instruction execution.

Branches pose a problem for pipelining, since we don't know the next instruction until the current instruction has reached the execute stage. One solution is to stall the pipeline when a branch is in the pipeline, waiting for the execute stage before fetching the next instruction. An alternative is to guess which way the branch will go and fetch the corresponding instruction next; if right, we proceed with no penalty, but if we find out in the execute stage that we were wrong, we must ignore all the instructions fetched since the branch was fetched, thus incurring a penalty. Modern pipelined microprocessors often have very sophisticated branch predictors built in.

PROGRAMMER'S VIEW:

A programmer writes the program instructions that carry out the desired functionality on the general-purpose processor. The programmer may not actually need to know detailed information about the processor's architecture or operation, but instead may deal with an architectural abstraction, which hides much of that detail. The level of abstraction depends on the level of programming. We can distinguish between two levels of programming. The first is assembly-language programming, in which one programs in a language representing processor-specific instructions as mnemonics. The second is structured-language programming, in which one programs in a language using processorindependent instructions. A compiler automatically translates those instructions to processor-specific instructions. Ideally, the structured-language programmer would need no information about the processor architecture, but in embedded systems, the programmer must usually have at least some awareness, as we shall discuss.

Actually, we can define an even lower-level of programming, machine-language programming, in which the programmer writes machine instructions in binary. This level of

programming has become extremely rare due to the advent of assemblers. Machinelanguage programmed computers often had rows of lights representing to the programmer the current binary instructions being executed. Today's computers look more like boxes or refrigerators, but these do not make for interesting movie props, so you may notice that in the movies, computers with rows of blinking lights live on.

Instruction set: The assembly-language programmer must know the processor's instruction set. The instruction set describes the bit-configurations allowed in the IR, indicating the atomic processor operations that the programmer may invoke. Each such configuration forms an assembly instruction, and a sequence of such instructions forms an assembly program

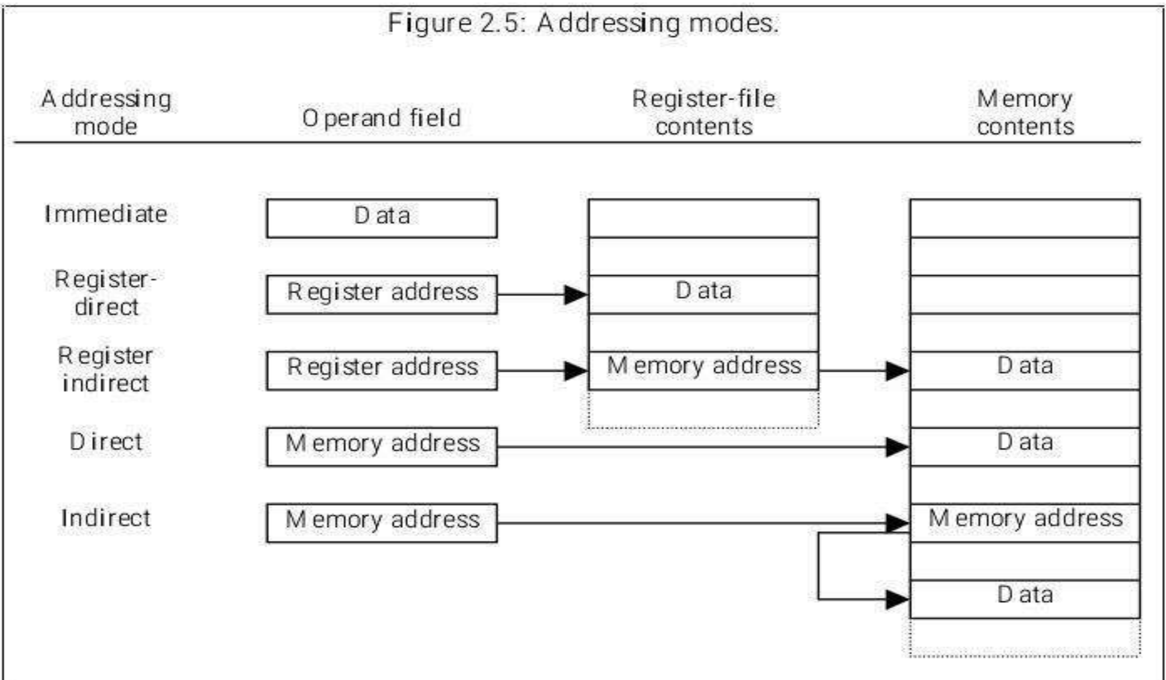


Figure 2.5: Addressing modes.

An instruction typically has two parts, an opcode field and operand fields. An opcode specifies the operation to take place during the instruction. We can classify instructions into three categories. Data-transfer instructions move data between memory and registers, between input/output channels and registers, and between registers themselves. Arithmetic/logical instructions configure the ALU to carry out a particular function, channel data from the registers through the ALU, and channel data from the ALU back to a particular register. Branch instructions determine the address of the next program instruction, based possibly on data path status signals.

Branches can be further categorized as being unconditional jumps, conditional jumps or procedure call and return instructions. Unconditional jumps always determine the address of the next instruction, while conditional jumps do so only if some condition evaluates to true, such as a particular register containing zero. A call instruction, in addition to indicating the address of the next instruction, saves the address of the current instruction 1 so that a subsequent return instruction can jump back to the instruction immediately following the

most recent invoked call instruction. This pair of instructions facilitates the implementation of procedure/function call semantics of high-level programming languages.

An operand field specifies the location of the actual data that takes part in an operation. Source operands serve as input to the operation, while a destination operand stores the output. The number of operands per instruction varies among processors. Even for a given processor, the number of operands per instruction may vary depending on the instruction type.

The operand field may indicate the data's location through one of several addressing modes, illustrated in Figure 2.5. In immediate addressing, the operand field contains the data itself. In register addressing, the operand field contains the address of a data path register in which the data resides. In register-indirect addressing, the operand field contains the address of a register, which in turn contains the address of a memory location in which the data resides. In direct addressing, the operand field contains the address of a memory location in which the data resides. In indirect addressing, the operand field contains the address of a memory location, which in turn contains the address of a memory location in which the data resides. Those familiar with structured languages may note that direct addressing implements regular variables, and indirect addressing implements pointers. In inherent or implicit addressing, the particular register or memory location of the data is implicit in the opcode; for example, the data may reside in a register called the "accumulator." In indexed addressing, the direct or indirect operand must be added to a particular implicit register to obtain the actual operand address. Jump instructions may use relative addressing to reduce the number of bits needed to indicate the jump address. A relative address indicates how far to jump from the current address, rather than indicating the complete address – such addressing is very common since most jumps are to nearby instructions.

Figure 2.6: A simple (trivial) instruction set.

Assembly instruct.	First byte		Second byte		Operation
MOV Rn, direct	0000	Rn	direct		Rn = M(direct)
MOV direct, Rn	0001	Rn	direct		M(direct) = Rn
MOV @Rn, Rm	0010		Rn	Rm	M(Rn) = Rm
MOV Rn, #immed.	0011	Rn	immediate		Rn = immediate
ADD Rn, Rm	0100		Rn	Rm	Rn = Rn + Rm
SUB Rn, Rm	0101		Rn	Rm	Rn = Rn - Rm
JZ Rn, relative	1000	Rn	relative		PC = PC+ relative (only if Rn is 0)

Ideally, the structured-language programmer would not need to know the instruction set of the processor. However, nearly every embedded system requires the programmer to write at least some portion of the program in assembly language. Those portions may deal with low-level input/output operations with devices outside the processor, like a display device. Such a device may require specific timing sequences of signals in order to receive data, and the programmer may find that writing assembly code achieves such timing most conveniently. A driver routine is a portion of a program written specifically to communicate with, or drive, another device. Since drivers are often written in assembly language, the structured-language programmer may still require some familiarity with at least a subset of the instruction set.

Figure 2.6 shows a (trivial) instruction set with 4 data transfer instructions, 2 arithmetic instructions, and 1 branch instruction, for a hypothetical processor. Figure 2.7(a) shows a program, written in C, that adds the numbers 1 through 10. Figure 2.7(b) shows that same program written in assembly language using the given instruction set.

Figure 2.7: Sample programs: (a) C program, (b) equivalent assembly program.

```
int total = 0;
for (int i=10; i!=0; i--)
  total += i;
// next instructions...
```

(a)

```
       MOV R0, #0;     // total = 0
       MOV R1, #10;    // i = 10
       MOV R2, #1;     // constant 1
       MOV R3, #0;     // constant 0
Loop:  JZ R1, Next;    // Done if i=0
       ADD R0, R1;     // total += i
       SUB R1, R2;     // i--
       JZ R3, Loop;    // Jump always
Next:  // next instructions...
```

(b)

Program and data memory space: The embedded systems programmer must be aware of the size of the available memory for program and for data. For example, a particular processor may have a 64K program space, and a 64K data space. The programmer must not exceed these limits. In addition, the programmer will probably want to be aware of on-chip program and data memory capacity, taking care to fit the necessary program and data in on-chip memory if possible.

Registers: The assembly-language programmer must know how many registers are available for general-purpose data storage. He/she must also be familiar with other registers that have special functions. For example, a base register may exist, which permits the programmer to use a data-transfer instruction where the processor adds an operand field to the base register to obtain an actual memory address.

Other special-function registers must be known by both the assembly-language and the structured-language programmer. Such registers may be used for configuring built-in timers, counters, and serial communication devices, or for writing and reading external pins.

I/O: The programmer should be aware of the processor's input and output (I/O) facilities, with which the processor communicates with other devices. One common I/O facility is parallel I/O, in which the programmer can read or write a port (a collection of external pins) by reading or writing a special-function register. Another common I/O facility is a system bus, consisting of address and data ports that are automatically activated by certain addresses or types of instructions.

Interrupts: An interrupt causes the processor to suspend execution of the main program, and instead jump to an Interrupt Service Routine (ISR) that full fills a special, short-term processing need. In particular, the processor stores the current PC, and sets it to the address of the ISR. After the ISR completes, the processor resumes execution of the main program by restoring the PC. The programmer should be aware of the types of interrupts supported by the processor (we describe several types in a subsequent chapter), and must write ISRs when necessary. The assembly-language programmer places each ISR at a specific address in program memory. The structured-language programmer must do so also; some compilers allow a programmer to force a procedure to start at a particular memory location, while recognize pre-defined names for particular ISRs.

For example, we may need to record the occurrence of an event from a peripheral device, such as the pressing of a button. We record the event by setting a variable in memory when that event occurs, although the user's main program may not process that event until later. Rather than requiring the user to insert checks for the event throughout the main program, the programmer merely need write an interrupt service routine and associate it with an input pin connected to the button. The processor will then call the routine automatically when the button is pressed.

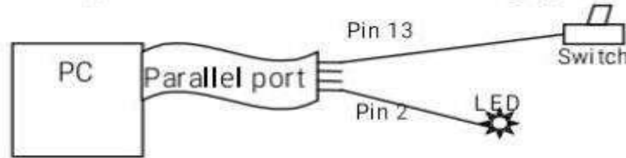
Example: Assembly-language programming of device drivers

This example provides an application of assembly language programming of a low-level driver, showing how the parallel port of an x86 based PC (Personal Computer) can be used to perform digital I/O. Writing and reading three special registers accomplishes parallel communication on the PC. Those three register are actually in an 8255A Peripheral Interface Controller chip. In unidirectional mode, (default power-on-reset mode), this device is capable of driving 12 output and five input lines. In the following table, we provide the parallel port (known as LPT) connector pin numbers and the corresponding register location.

Parallel port signals and associated registers.

LPT Connector Pin	I/O Direction	Register Address
1	Output	0th bit of register #2
2-9	Output	0th-7th bit of register #0
10	Input	6th bit of register #1
11	Input	7th bit of register #1
12	Input	5th bit of register #1
13	Input	4th bit of register #1
14	Output	1st bit of register #2
15	Input	3rd bit of register #1
16	Output	2nd bit of register #2
17	Output	3rd bit of register #2

In our example, we are to build the following system:



A switch is connected to input pin number 13 of the parallel port. An LED (lightemitting diode) is connected to output pin number 2. Our program, running on the PC, should monitor the input switch and turn on/off the LED accordingly.

Figure 2.8 gives the code for such a program, in x86 assembly language. Note that the in and out assembly instructions read and write the internal registers of the 8255A. Both instructions take two operands, address and data. Address specifies the the register we are trying to read or write. This address is calculated by adding the address of the device, called the base address, to the address of the particular register as given in Figure 2.8. In most PCs, the base address of LPT1 is at 3BC hex (though not always). The second operand is the data. For the in instruction, the content of this eight-bit operand will be written to the addressed register. For the out instruction, the content of the addressed eight-bit register will be read into this operand

The program makes use of masking, something quite common during low-level I/O. A mask is a bit-pattern designed such that ANDing it with a data item D yields a specific part of D. For example, a mask of 00001111 can be used to yield bits 0 through 3, e.g., 00001111 AND 10101010 yields 00001010. A mask of 00010000, or 10h in hexadecimal format, would yield bit 4.

In Figure 2.8, we have broken our program in two source files, assembly and C. The assembly program implements the low-level I/O to the parallel port and the C program implements the high-level application. Our assembly program is a simple form of a device driver program that provides a single procedure to the high-level application. While the trend is for embedded systems to be written in structured languages, this example shows that some small assembly program may still need to be written for low-level drivers.

Figure 2.8: Parallel port example.

```
;
; This program consists of a sub-routine that reads
; the state of the input pin, determining the on/off state
; of our switch and asserts the output pin, turning the LED
; on/off accordingly.
;
        .386

CheckPort       proc
        push    ax              ; save the content
        push    dx              ; save the content
        mov     dx, 3BCh + 1    ; base + 1 for register #1
        in      al, dx          ; read register #1
        and     al, 10h         ; mask out all but bit # 4
        cmp     al, 0           ; is it 0?
        jne     SwitchOn        ; if not, we need to turn the LED on

SwitchOff:
        mov     dx, 3BCh + 0    ; base + 0 for register #0
        in      al, dx          ; read the current state of the port
        and     al, f7h         ; clear first bit (masking)
        out     dx, al          ; write it out to the port
        jmp     Done            ; we are done

SwitchOn:
        mov     dx, 3BCh + 0    ; base + 0 for register #0
        in      al, dx          ; read the current state of the port
        or      al, 01h         ; set first bit (masking)
        out     dx, al          ; write it out to the port

Done:   pop     dx              ; restore the content
        pop     ax              ; restore the content
CheckPort       endp
```

```
extern "C" CheckPort(void);        // defined in assembly above

void main(void) {

        while( 1 ) {

                CheckPort();
        }
}
```

Operating system: An operating system is a layer of software that provides low-level services to the application layer, a set of one or more programs executing on the CPU consuming and producing input and output data. The task of managing the application layer involves the loading and executing of programs, sharing and allocating system resources to these programs, and protecting these allocated resources from corruption by non-owner programs. One of the most important resource of a system is the central processing unit (CPU), which is typically shared among a number of executing programs. The operating system, thus, is responsible for deciding what program is to run next on the CPU and for how long. This is called process/task scheduling and is determined by the operating system's preemption policy. Another very important resource is memory, including disk storage, which is also shared among the applications running on the CPU.

In addition to implementing an environment for management of high-level application programs, the operating system provides the software required for servicing various hardware-interrupts, and provides device drivers for driving the peripheral devices present in the system. Typically, on startup, an operating system initializes all peripheral devices, such as disk controllers, timers and input/output devices and installs hardware interrupt (interrupts generated by the hardware) service routines (ISR) to handle various signals generated by these devices 2. Then, it installs software interrupts (interrupts generated by the software) to

process system calls (calls made by high-level applications to request operating system services) as described next.

Figure 2.9: System call invocation.

```
DB file_name "out.txt"      -- store file name

     MOV R0, 1324           -- system call "open" id
     MOV R1, file_name      -- address of file-name
     INT 34                 -- cause a system call
     JZ  R0, L1             -- if zero -> error

     . . . read the file
     JMP L2                 -- bypass error cond.
L1:
     . . . handle the error

L2:
```

A system call is a mechanism for an application to invoke the operating system. This is analogous to a procedure or function call, as in high-level programming languages. When a program requires some service from the operating system, it generates a predefined software interrupt that is serviced by the operating system. Parameters specific to the requested services are typically passed from (to) the application program to (from) the operating system through CPU registers. Figure 2.9 illustrates how the file "open" system call may be invoked, in assembly, by a program. Languages like C and Pascal provide wrapper functions around the system-calls to provide a high-level mechanism for performing system calls.

DEVELOPMENT ENVIRNORNMENT:

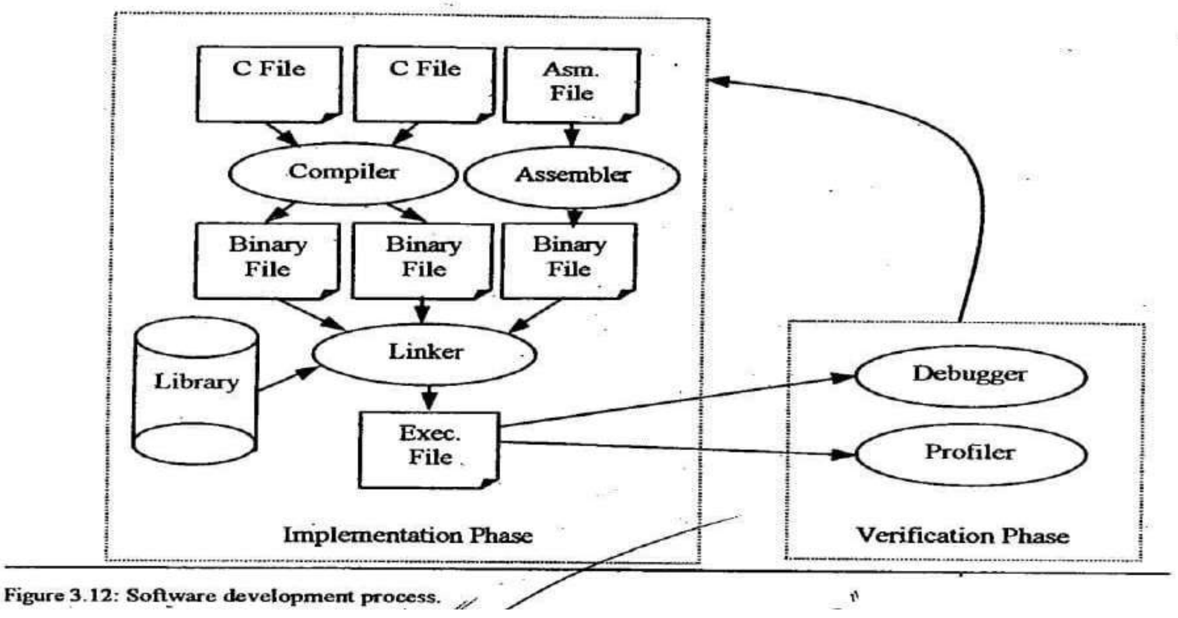


Figure 3.12: Software development process.

Several software and hardware tools commonly support the programming of general-purpose processors. First, we must distinguish between two processors we deal with when developing

an embedded system. One processor is the development processor, on which we write and debug our program. This processor is part of our desktop computer. The other processor is the target processor, to which we will send our program and which will form part of our embedded system's implementation. For example, we may develop our system on a Pentium processor, but use a Motorola 68HC11 as our target processor. Of course, sometimes the two processors happen to be the same, but this is mostly a coincidence.

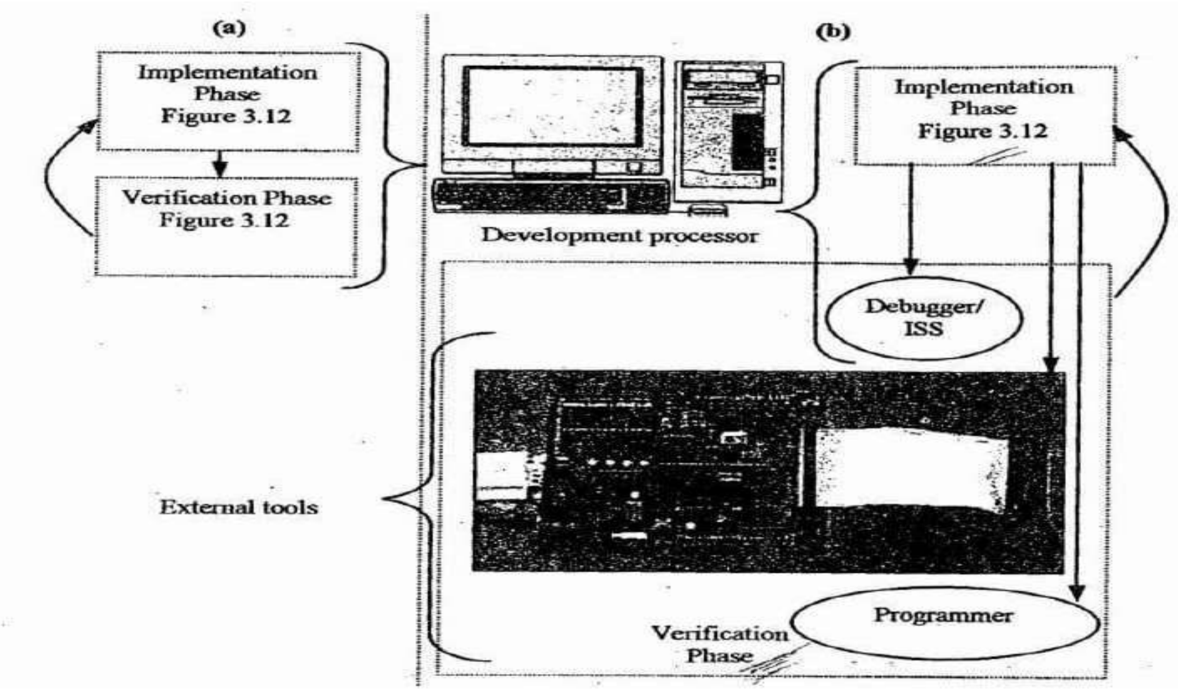


Fig: software development a) desktop, b) embedded system

Assemblers translate assembly instructions to binary machine instructions. In addition to just replacing opcode and operand mnemonics by binary equivalents, an assembler may also translate symbolic labels into actual addresses. For example, a programmer may add a symbolic label END to an instruction A, and may reference END in a branch instruction. The assembler determines the actual binary address of A, and replaces references to END by this address. The mapping of assembly instructions to machine instructions is one-to-one. A linker allows a programmer to create a program in separately-assembled files; it combines the machine instructions of each into a single program, perhaps incorporating instructions from standard library routines.

Compilers translate structured programs into machine (or assembly) programs. Structured programming languages possess high-level constructs that greatly simplify programming, such as loop constructs, so each high-level construct may translate to several or tens of machine instructions. Compiler technology has advanced tremendously over the past decades, applying numerous program optimizations, often yielding very size and performance efficient code. A cross-compiler executes on one processor (our development processor), but generates code for a different processor (our target processor). Cross-compilers are extremely common in embedded system development.

Debuggers help programmers evaluate and correct their programs. They run on the development processor and support stepwise program execution, executing one instruction and then stopping, proceeding to the next instruction when instructed by the user. They permit execution up to user-specified breakpoints, which are instructions that when encountered cause the program to stop executing. Whenever the program stops, the user can examine values of various memory and register locations. A source-level debugger enables step-by-step execution in the source program language, whether assembly language or a structured language. A good debugging capability is crucial, as today's programs can be quite complex and hard to write correctly.

Device programmers download a binary machine program from the development processor's memory into the target processor's memory.

Emulator's support debugging of the program while it executes on the target processor. An emulator typically consists of a debugger coupled with a board connected to the desktop processor via a cable. The board consists of the target processor plus some support circuitry (often another processor). The board may have another cable with a device having the same pin configuration as the target processor, allowing one to plug this device into a real embedded system. Such an in-circuit emulator enables one to control and monitor the program's execution in the actual embedded system circuit. Incircuit emulators are available for nearly any processor intended for embedded use, though they can be quite expensive if they are to run at real speeds.

The availability of low-cost or high-quality development environments for a processor often heavily influences the choice of a processor.

SELECTING A PROCESSOR:

The embedded system designer must select a microprocessor for use in an embedded system. The choice of a processor depends on technical and non-technical aspects. From a technical perspective, one must choose a processor that can achieve the desired speed within certain power, size and cost constraints. Non-technical aspects may include prior expertise with a processor and its development environment, special licensing arrangements, and so on.

Figure 2.10: General Purpose Processors

Processor	Clock Speed	Peripherals	Bus Width	MIPS	Power	Transistor	Price
Intel SA110	233 MHz	32K cache	32	268	360 mW	2.1 M	$49
VLSI ARM710	25 MHz	8K cache	32	30	120 mW	341 K	$35
IBM 401GF	50 MHz	3K cache	32	52	140 mW	345 K	$11
Mistubishi M32R/D	66 MHz	4K cache	16	52	180 mW	7 M	$80
PIC 12508	8 MHz	512 ROM, 25 RAM, 5 I/O	8	~ .8	NA	~10 K	$6
Intel 8051	12 MHz	4K ROM, 128 RAM, 32 I/O, Timer, UART	8	~ 1	NA	~10 K	$7

Sources: Embedded Systems Programming, Nov. 1998; PIC and Intel datasheets.

Speed is a particularly difficult processor aspect to measure and compare. We could compare processor clock speeds, but the number of instructions per clock cycle may differ greatly among processors. We could instead compare instructions per second, but the complexity of each instruction may also differ greatly among processors -- e.g., one processor may require 100 instructions, and another 300 instructions, to perform the same computation. One attempt to provide a means for a fairer comparison is the Dhrystone benchmark. A benchmark is a program intended to be run on different processors to compare their performance. The Dhrystone benchmark was developed in 1984 by Reinhold Weicker specifically as a performance benchmark; it performs no useful work. It focuses on exercising a processor's integer arithmetic and string-handling capabilities. It is written in C and in the public domain. Since most processors can execute it in milliseconds, it is typically executed thousands of times, and thus a processor is said to be able to execute so many Dhrystones per second.

Another commonly-used speed comparison unit, which happens to be based on the Dhrystone, is MIPS. One might think that MIPS simply means Millions of Instructions Per Second, but actually the common use of the term is based on a somewhat more complex notion. Specifically, its origin is based on the speed of Digital's VAX 11/780, thought to be the first computer able to execute one million instructions per second. A VAX 11/780 could execute 1,757 Dhrystones/second. Thus, for a VAX 11/780, 1 MIPS = 1,757 Dhrystones/second. This unit for MIPS is the one used today. So if a machine today is said to run at 750 MIPS, that actually means it can execute 750*1757 = 1,317,750 Dhrystones/second.

The use and validity of benchmark data is a subject of great controversy. There is also a clear need for benchmarks that measure performance of embedded processors. Numerous general-purpose processors have evolved in the recent years and are in common use today. In Figure 2.10, we summarize some of the features of several popular processors.

Application-Specific Instruction-Set Processors (ASIPs)

Today's embedded applications, such as high definition TV, require high computing power and very specific functionality. The performance, power, cost, or size demands of these applications cannot always be dealt with efficiently by using general-purpose processors. Nonetheless, the inflexibility of custom single-purpose processors is often too prohibitive. A solution is to use an instruction-set processor that is specific to that application or application domain. Because these ASIPs are instruction-set processors, they can be programmed by writing software, resulting in short time-to-market and good flexibility, while the performance and other constraints may be efficiently satisfied.

As with most other aspects of embedded systems design, there is a trade-off here. Instruction-set processors and the associated software tools (compilers, linkers, etc.) are very expensive to develop; therefore, they are expensive to integrate into low-cost embedded systems. In contrast, the large applicability and resulting cost amortization of general-purpose processors make them very cost effective solutions in most embedded systems. ASIPs tend to come in three major varieties, namely, microcontrollers, which are specific to applications that perform a large amount of control-oriented tasks, digital signal processors (DSPs), which are specific to applications that process large amounts of data, and everything else, which are less general ASIPs.

Microcontrollers

Numerous processor IC manufacturers market devices specifically for the control-dominated embedded systems domain. These devices may include several features. First, they may include several peripheral devices, such as timers, analog-to-digital converters, and serial communication devices, on the same IC as the processor. Second, they may include some program and data memory on the same IC. Third, they may provide the programmer with direct access to a number of pins of the IC. Fourth, they may provide specialized instructions for common embedded system control operations, such as bit-manipulation operations. A *microcontroller* is a device possessing some or all of these features.

Incorporating peripherals and memory onto the same IC reduces the number of required ICs, resulting in compact and low-power implementations. Providing pin access allows programs to easily monitor sensors, to set actuators, and to transfer data with other devices. Providing specialized instructions improves performance for embedded systems applications. Thus, microcontrollers can be considered ASIPs to some degree.

Many manufacturers market devices referred to as "embedded processors." The difference between embedded processors and microcontrollers is not clear, although we note that the former term seems to be used more for large (32-bit) processors.

Digital Signal Processors (DSP)

Digital signal processors (DSPs) are processors that are highly optimized for processing large amounts of data. The source of this large amount of data is some form of digitized signal, like a photo image captured by a digital camera, a voice packet going through a network router, or an audio clip played by a digital keyboard. A DSP may contain numerous register files, memory blocks, multipliers, and other arithmetic units. In addition, DSPs often provide instructions that are central to digital signal processing, such as filtering and transforming vectors or metrics of data. In a DSP, frequently used arithmetic functions, such as multiply-and-accumulate, are implemented in hardware and thus execute orders of magnitude faster than a software implementation running on a general-purpose processor. In addition, DSPs may allow for execution of some functions in parallel, resulting in a boost in performance.

As with microcontrollers, DSPs also tend to incorporate many peripherals that are useful in signal processing on a single IC. As an example, a DSP device may contain a number of analog-to-digital and digital-to-analog converters, pulse-width-modulators, direct-memory-access controllers, timers, and counters.

Many companies offer a variety of commonly used DSPs that are well supported in terms of compiler and other development tools, making them easy and cheap to integrate into most embedded systems.

Less-General ASIP Environments

In contrast to microcontrollers and DSPs, which can be used in a variety of embedded systems, IC manufacturers have designed ASIPs that are less general in nature. These ASIPs are designed to perform some very domain specific processing while allowing some degree of programmability. For example, an ASIP designed for networking hardware may be designed to be programmable with different network routing, checksum, and packet processing protocols.

General-Purpose Processor Design

A general-purpose processor is really just a single-purpose processor whose purpose is to process instructions stored in a program memory. Therefore, we can design a general-purpose processor using the single-purpose processor design technique described in Chapter 2. While real microprocessors intended for mass production are more commonly designed using custom methods rather than the general technique of this section, using the the general technique here may prove a useful exercise that will illustrate the basic unity between single-purpose and general-purpose processors.

Suppose we want to design a general-purpose processor having the basic architecture of Figure 3.1 and supporting the instruction set of Figure 3.7. We can begin by creating the FSMD shown in Figure 3.16(a), which describes the desired processor's behavior. The FSMD declares several variables for storage: a 16-bit program counter *PC*, a 16-bit instruction register *IR*, a 64K × 16 bit memory *M*, and a 16 × 16 bit register file *RF*. The FSMD's initial state, *Reset*, clears *PC* to 0. The *Fetch* state reads *M[PC]* into *IR*. The *Decode* state does nothing but adds the extra cycle necessary for *IR* to get updated so we can then read it on an arc. Each arc leaving the *Decode* state detects a particular instruction opcode, causing a transition to the corresponding execute state for that opcode. Each execute state, like *Mov1*,

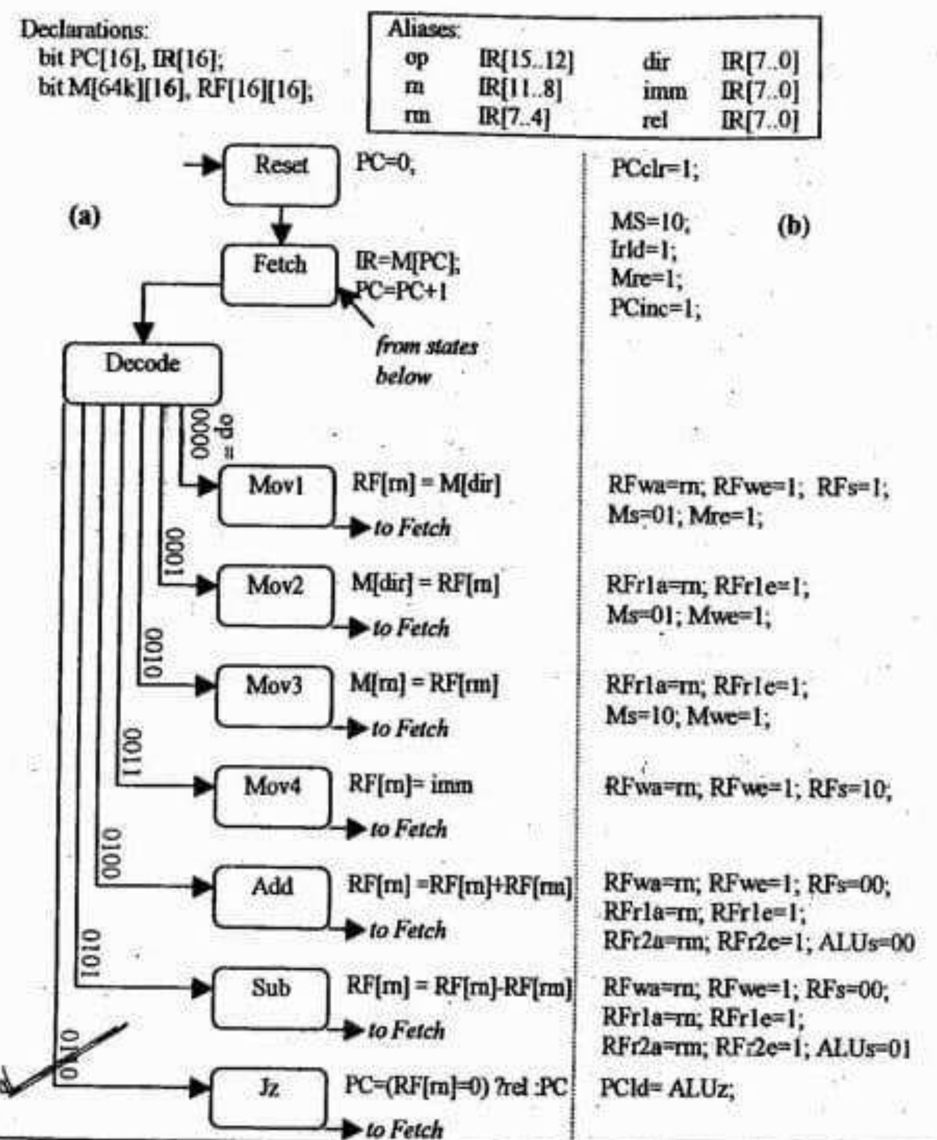


Figure 3.16: A simple microprocessor: (a) FSMD, (b) FSM operations that replace the FSMD operations after we create the datapath of Figure 3.17.

Add, and *Jz*, carries out the actual instruction operations by moving data between storage devices, modifying data, or updating *PC*.

We can now build a datapath that can carry out the operation of this FSMD, as described in Chapter 2. The datapath we create using the following steps is shown in Figure 3.17. The first step is to instantiate a storage device for each declared variable, so we instantiate

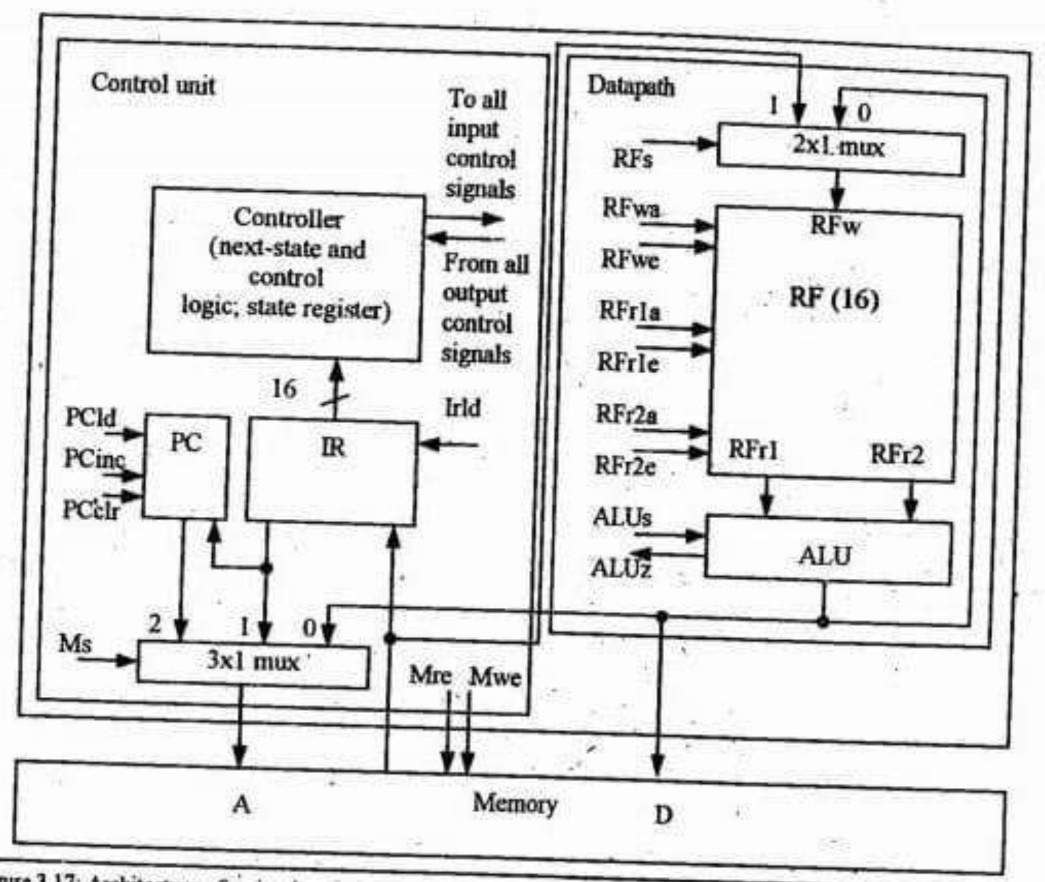


Figure 3.17: Architecture of a simple microprocessor.

registers *PC* and *IR*, memory *M*, and register file *RF*. The second step is to instantiate functional units to carry out the FSMD operations. We'll use a single ALU capable of carrying out all the operations. The third step is to add the connections among the components' ports as required by the FSMD operations, adding multiplexors when there is more than one connection being input to a port. Finally, we create unique identifiers for every control signal.

Given this datapath, we can now rewrite the FSMD as an FSM representing the datapath's controller. Each FSMD operation must be replaced by binary operations on control signals, as shown in Figure 3.16(b). The states and arcs are identical for the FSMD and FSM, and only the operations change, so we do not redraw the states and arcs in the figure. As an example of operation replacement, we replace the assignment $PC = 0$ in state *Reset* by the control signal setting $PCclr = 1$.

We can use the FSM design technique of Chapter 2 to design a controller, consisting of a state register and next-state/control logic. We omit this step here.

Having just designed a simple general-purpose processor using the same technique we used to design a single-purpose processor, we can see the similarity between the two processor types. The key difference is that a single-purpose processor puts the "program" inside of its control logic, whereas a general-purpose processor keeps it in an external memory. So the program of a single-purpose processor cannot be changed once the processor has been implemented. But nevertheless, both processor types process programs. A second difference is that we design the datapath in a general-purpose processor without knowledge of what program will be put in the memory, whereas we know this program in a single-purpose processor. So the datapath of a single-purpose processor can be optimized to the program. We see that single-purpose and general-purpose processors both implement programs. Though they may differ in terms of design metrics like flexibility, power, performance, and cost, they fundamentally do the same thing.

ESD UNIT-4

MEMORY&INTERFACING

COMMON MEMORY TYPES: ROM&RAM

READ ONLY MEMORY—ROM:

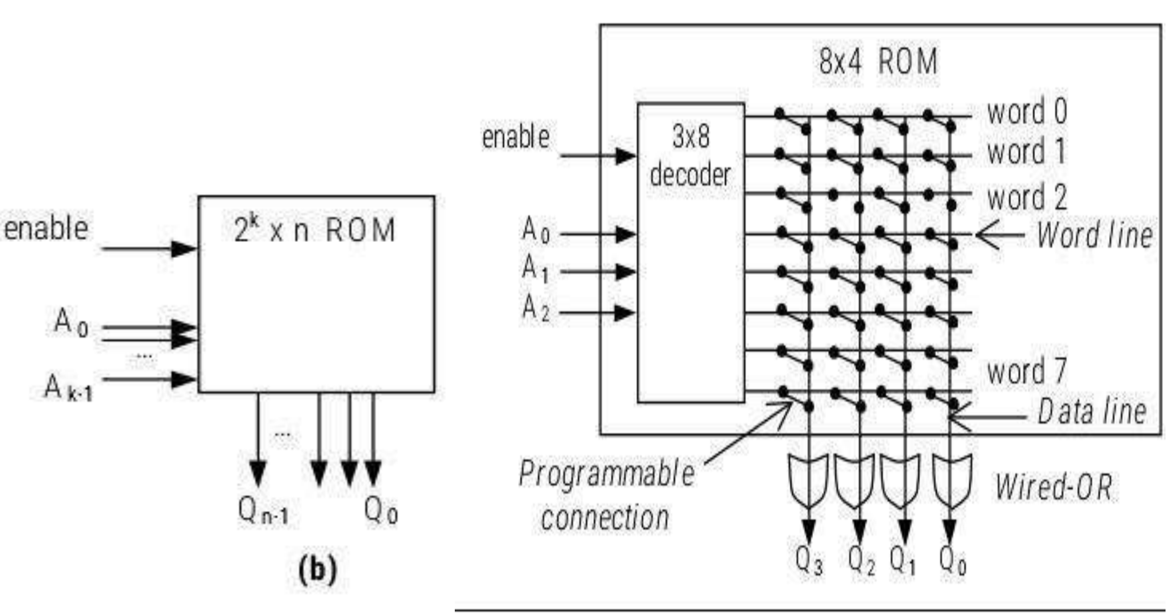
ROM, or read-only memory, is a memory that can be read from, but not typically written to, during execution of an embedded system. Of course, there must be a mechanism for setting the bits in the memory (otherwise, of what use would the read data serve?), but we call this "programming," not writing. Such programming is usually done off-line, i.e., when the memory is not actively serving as a memory in an embedded system. We usually program a ROM before inserting it into the embedded system. Figure 1(b) provides a block diagram of a ROM.

We can use ROM for various purposes. One use is to store a software program for a general-purpose processor. We may write each program instruction to one ROM word. For some processors, we write each instruction to several ROM words. For other processors, we may pack several instructions into a single ROM word. A related use is to store constant data, like large lookup tables of strings or numbers.

Another common use is to implement a combinational circuit. We can implement any combinational function of k variables by using a 2k x 1 ROM, and we can implement n functions of the same k variables using a 2k x n ROM. We simply program the ROM to implement the truth table for the functions, as shown in Figure 2.

Figure below provides a symbolic view of the internal design of an 8x4 ROM. To the right of the 3x8 decoder in the figure is a grid of lines, with word lines running horizontally and data lines vertically; lines that cross without a circle in the figure are not connected. Thus, word lines only connect to data lines via the programmable connection lines shown. The figure shows all connection lines in place except for two connections in word 2. To see how this device acts as a read-only memory, consider an input address of "010." The decoder will thus set word 2's line to 1. Because the lines connecting this word line with data lines 2 and 0 do not exist, the ROM output will read "1010." Note that if the ROM enable input is 0, then no word is read. Also note that each data line is shown as a wired-OR, meaning that the wire itself acts to logically OR all the connections to it.

Figure 5.3: ROM internals



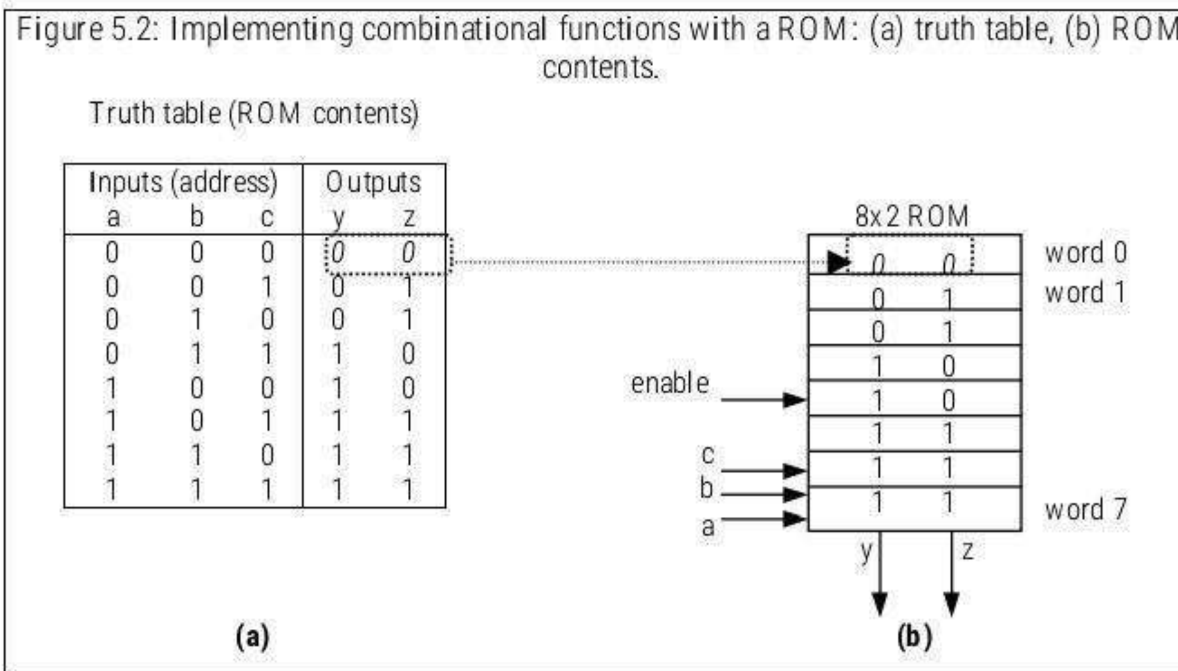


Figure 5.2: Implementing combinational functions with a ROM: (a) truth table, (b) ROM contents.

Truth table (ROM contents)

Inputs (address)			Outputs	
a	b	c	y	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

MASK-PROGRAMMED ROM: In a mask-programmed ROM, the connection is made when the chip is being fabricated (by creating an appropriate set of masks). Such ROM types are typically only used in high-volume systems, and only after a final design has been determined.

PROGRAMMABLE ROM: PROM, which can be programmed by the chip's user, well after the chip has been manufactured. These devices are better suited to prototyping and to low-volume applications. To program a PROM device, the user provides a file indicating the desired ROM contents. A piece of equipment called a ROM programmer (note: the programmer is a piece of equipment, not a person who writes software) then configures each programmable

connection according to the file. A basic PROM uses a fuse for each programmable connection. The ROM programmer blows fuses by passing a large current wherever a connection should not exist. However, once a fuse is blown, the connection can never be re-established. For this reason, basic PROM is often referred to as one-time-programmable device, or OTP.

ERASABLE PROM or EPROM: This device uses a MOS transistor as its programmable component. The transistor has a "floating gate," meaning its gate is not connected. An EPROM programmer injects electrons into the floating gate, using higher than normal voltage (usually 12V to 25V) that causes electrons to "tunnel" into the gate. When that high voltage is removed, the electrons can-not escape, and hence the gate has been charged and programming has occurred. Standard EPROMs are guaranteed to hold their programs for at least 10 years. To erase the program, the electrons must be excited enough to escape from the gate. Ultra-violet (UV) light is used to fulfil this role of erasing. The device must be placed under a UV eraser for a period of time, typically ranging from 5 to 30 minutes, after which the device can be programmed again. In order for the UV light to reach the chip, EPROM's come with a small quartz window in the package through which the chip can be seen. For this reason, EPROM is often referred to as a windowed ROM device.

ELECTRICALLY-ERASABLE PROM, or EEPROM: is designed to eliminate the time consuming and sometimes impossible requirement of exposing an EPROM to UV light to erase the ROM. An EEPROM is not only programmed electronically, but is also erased electronically. These devices are typically more expensive the EPROM's, but far more convenient to use. EEPROM's are often called " E square's " for short. Flash memory is a type of EEPROM in which reprogramming can be done to certain regions of the memory, rather than the entire memory at once.

Flash Memory

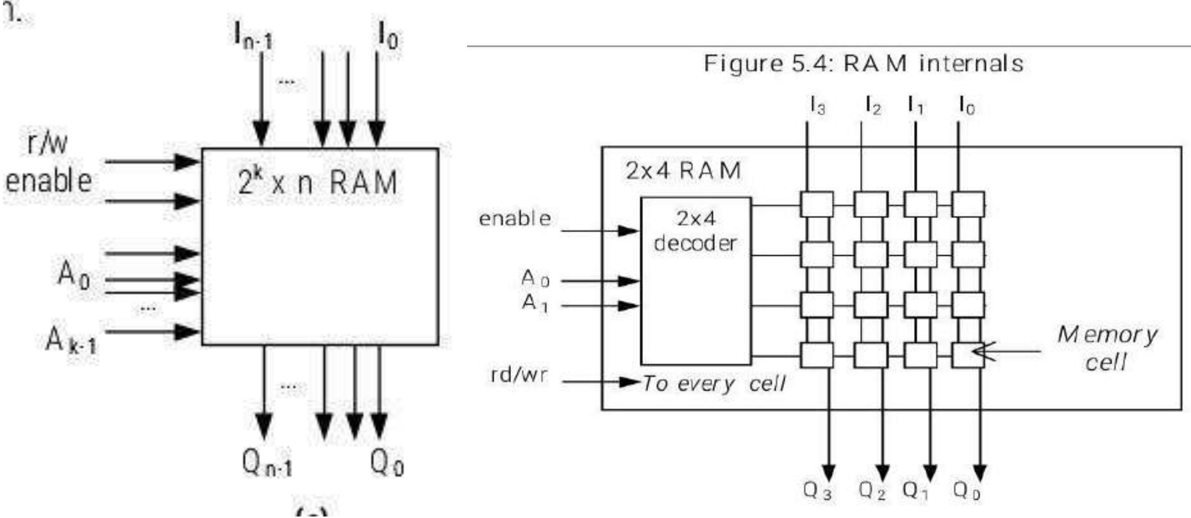
Flash memory is an extension of EEPROM that was developed in the late 1980s. While also using the floating-gate principle of EEPROM, flash memory is designed such that large blocks of memory can be erased all at once, rather than just one word at a time as in traditional EEPROM. A block is typically several thousand bytes large. This fast erase ability can vastly improve the performance of embedded systems where large data items must be stored in nonvolatile memory, systems like digital cameras, TV set-top boxes, cell phones, and medical monitoring equipment. It can also speed manufacturing throughput, since programming the complete contents of flash may be faster than programming a similar-sized EEPROM.

Like EEPROM, each block in a flash memory can typically be erased and reprogrammed tens of thousands of times before the block loses its ability to store data, and can store its data for 10 years or more.

A drawback of flash memory is that writing to a single word in flash may be slower than writing to a single word in EEPROM, since an entire block will need to be read, the word within it updated. and than the block written back.

READ WRITE MEMORY – RAM:

RAM, or random-access memory, is a memory that can be both read and written. In contrast to ROM, a RAM's content is not "programmed" before being inserted into an embedded system. Instead, the RAM contains no data when inserted in the embedded system; the system writes data to and then reads data from the RAM during its execution. Below provides a block diagram of a RAM. A RAM's internal structure is somewhat more complex than a ROM's, as shown in Figure. which illustrates a 4x4 RAM (note: RAMs typically have thousands of words, not just 4 as in the figure). Each word consists of a number of memory cells, each storing one bit. In the figure, each input data connects to every cell in its column. Likewise, each output data line connects to every cell in its column, with the output of a memory cell being OR with the output data line from above. Each word enable line from the decoder connects to every cell it's row. The read/write input (readd/write) is assumed to be connected to every cell. The memory cell must possess logic such that it stores the input data bit when read/write indicates write and the row is enabled, and such that it outputs this bit when read/write indicates read and the row is enabled.



There are two basic types of RAM, static and dynamic. Static RAM is faster but bigger than dynamic RAM.

STATIC RAM: Static RAM, or SRAM, uses a memory cell consisting of a flip-flop to store a bit. Each bit thus requires about 6 transistors. This RAM type is called static because it will hold its data as long as power is supplied, in contrast to dynamic RAM. Static RAM is typically used for high-performance parts of a system.

DYNAMIC RAM: Dynamic RAM, or DRAM, uses a memory cell consisting of a MOS transistor and capacitor to store a bit. Each bit thus requires only 1 transistor, resulting in more compact memory than SRAM. However, the charge stored in the capacitor leaks gradually, leading to discharge and eventually to loss of data. To prevent loss of data, each cell must regularly have its charge "refreshed." A typical DRAM cell minimum refresh rate is once every 15.625 microseconds. Because of the way DRAMs are designed, reading a DRAM word refreshes that word's cells. In particular, accessing a DRAM word results in the word's data

being stored in a buffer and then being written back to the word's cells. DRAMs tend to be slower to access than SRAMs.

PSRAM: Pseudo-Static RAMs, or PSRAMs, are DRAMs with a refresh controller built-in. Thus, since the RAM user need not worry about refreshing, the device appears to behave much like an SRAM. However, in contrast to true SRAM, a PSRAM may be busy refreshing itself when accessed, which could slow access time and add some system complexity. Nevertheless, PSRAM is a popular low-cost alternative to SRAM in many embedded systems.

NONVOLATILE RAM: Non-volatile RAM, or NVRAM, is another RAM variation. Non-volatile storage is storage that can hold its data even after power is no longer being supplied. Note that all forms of ROM are non-volatile, while normal forms of RAM (static or dynamic) are volatile. One type of NVRAM contains a static RAM along with its own permanently connected battery. A second type contains a static RAM and its own (perhaps flash) EEPROM. This type stores RAM data into the EEPROM just before power is turned off (or whenever instructed to store the data), and reloads that data from EEPROM into RAM after power is turned back on. NVRAM is very popular in embedded systems. For example, a digital camera must digitize, store and compress an image in a fraction of a second when the camera's button is pressed, requiring writes to a fast RAM (as opposed to programming of a slower EEPROM). But it also must store that image so that the image is saved even when the camera's power is shut off, requiring EEPROM. Using NVRAM accomplishes both these goals, since the data is originally and quickly stored in RAM, and then later copied to EEPROM, which may even take a few seconds.

COMPOSING MEMORY:

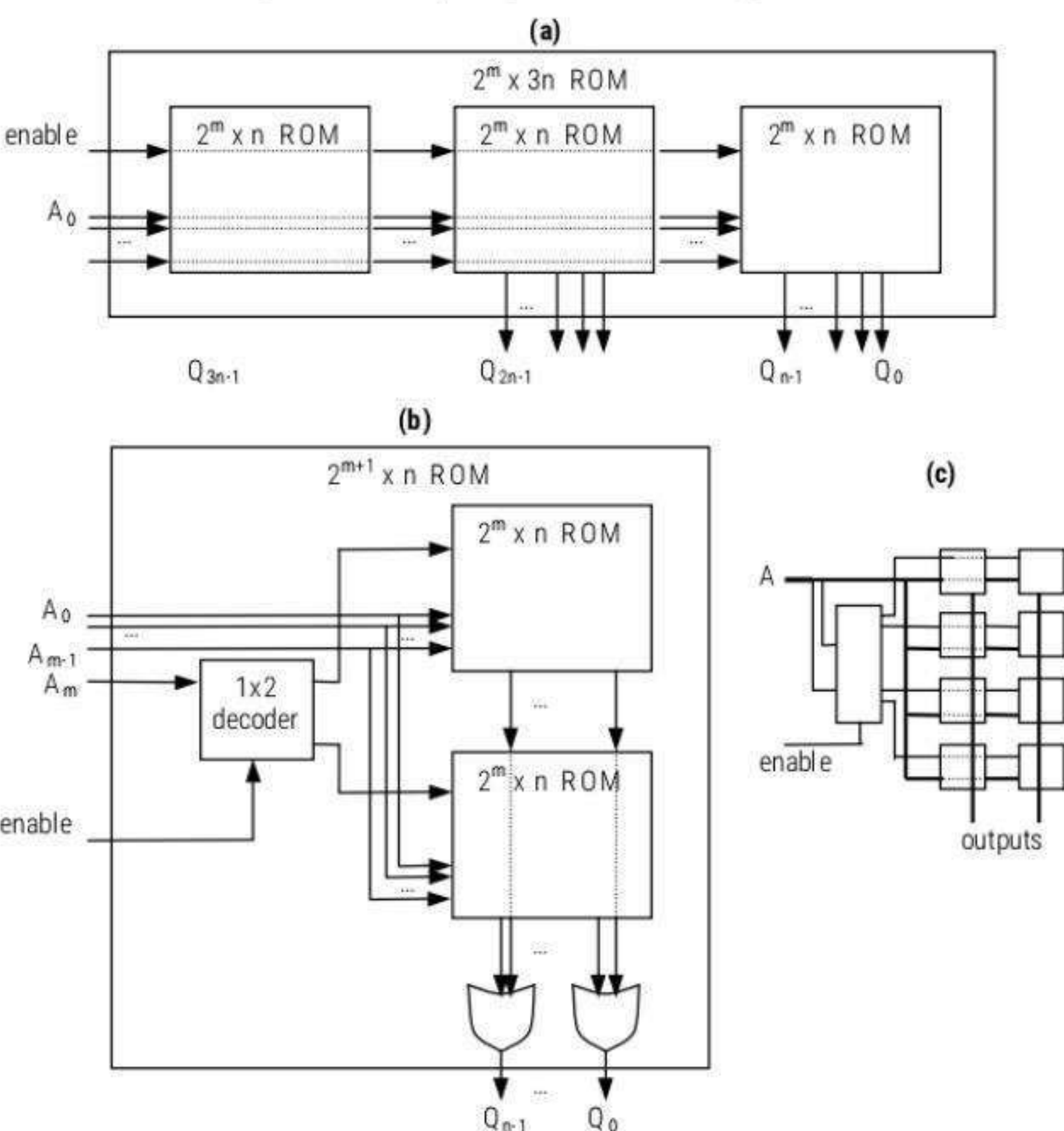
An embedded system designer is often faced with the situation of needing a particular-sized memory (ROM or RAM), but having readily available memories of a different size. For example, the designer may need a 2k x 8 ROM, but may have 4k x 16 ROMs readily available. Alternatively, the designer may need a 4k x 16 ROM, but may have 2k x 8 ROMs available for use.

The case where the available memory is larger than needed is easy to deal with. We simply use the needed lower words in the memory, thus ignoring unneeded higher words and their high-order address bits, and we use the lower data input/output lines, thus ignoring unneeded higher data lines. (Of course, we could use the higher data lines and ignore the lower lines instead).

The case where the available memory is smaller than needed requires more design effort. In this case, we must compose several smaller memories to behave as the larger memory we need. Suppose the available memories have the correct number of words, but each word is not wide enough. In this case, we can simply connect the available memories side-by-side. For example, Figure 5(a) illustrates the situation of needing a ROM three-times wider than that available. We connect three ROMs side-by-side, sharing the same address and enable lines among them, and concatenating the data lines to form the desired word width.

Suppose instead that the available memories have the correct word width, but not enough words. In this case, we can connect the available memories top-to-bottom. For example, Figure 5(b) illustrates the situation of needing a ROM with twice as many words, and hence needing one extra address line, than that available. We connect the ROMs top-to-bottom, the corresponding data lines of each. We use the extra high-order address line to select the higher or lower ROM (using a 1x2 decoder), and the remaining address lines to offset into the selected ROM. Since only one ROM will ever be enabled at a time, the data lines never actually in volves more than one 1.

Figure 5.5: Composing memories into larger ones.



If we instead needed four times as many words, and hence two extra address lines, we would instead use four ROMs. A 2x4 decoder having the two high-order address lines as input would select which of the four ROMs to access. Finally, suppose the available memories have a smaller word with as well as fewer words than necessary. We then combine the above two

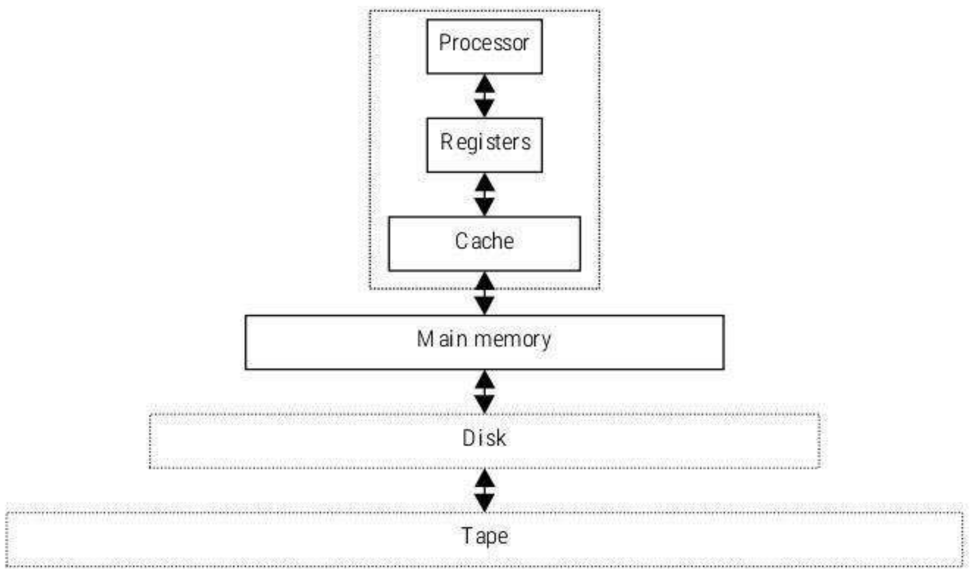
techniques, first creating the number of columns of memories necessary to achieve the needed word width, and then creating the number of rows of memories necessary, along with a decoder, to achieve the needed number of words. The approach is illustrated in Figure 5(c).

MEMORY HEIRARCHY AND CACHE:

When we design a memory to store an embedded system's program and data, we often face the following dilemma: we want an inexpensive and fast memory, but inexpensive memories tend to be slow, whereas fast memories tend to be expensive. The solution to this dilemma is to create a memory hierarchy, as illustrated in Figure 5.6. We use an inexpensive but slow main memory to store all of the program and data. We use a small amount of fast but expensive cache memory to store copies of likely-accessed parts of main memory. Using cache is analogous to posting on a wall near a telephone a short list of important phone numbers rather than posting the entire phonebook.

Some systems include even larger and less expensive forms of memory, such as disk and tape, for some of their storage needs. However, we do not consider these further as they are not especially common in embedded systems. Also, although the figure shows only one cache, we can include any number of levels of cache, those closer to the processor being smaller and faster than those closer to main memory. A two-level cache scheme is common.

Figure 5.6: An example memory hierarchy.



CACHE: Cache is usually designed using static RAM rather than dynamic RAM, which is one reason that cache is more expensive but faster than main memory. Because cache usually appears on the same chip as a processor, where space is very limited, cache size is typically only a fraction of the size main memory. Cache access time may be as low as just one clock cycle, whereas main memory access time is typically several cycles.

A cache operates as follows. When we want the processor to access (read or write) a main memory address, we first check for a copy of that location in cache. If the copy is in the cache, called a cache hit, then we can access it quickly. If the copy is not there, called a cache miss,

then we must first read the address (and perhaps some of its neighbors) into the cache. This description of cache operation leads to several cache design choices: cache mapping, cache replacement policy, and cache write techniques. These design choices can have significant impact on system cost, performance, as well as power, and thus should be evaluated carefully for a given application.

Cache mapping techniques: Cache mapping is the method for assigning main memory addresses to the far fewer number of available cache addresses, and for determining whether a particular main memory address' contents are in the cache. Cache mapping can be accomplished using one of three basic techniques:

1. **Direct mapping**: In this technique, the main memory address is divided into two fields, the index and the tag. The index represents the cache address, and thus the number of index bits is determined by the cache size, i.e., index size = log2(cache size). Note that many different main memory addresses will map to the same cache address. When we store a main memory address' content in the cache, we also store the tag. To determine if a desired main memory address is in the cache, we go to the cache address indicated by the index, and we then compare the tag there with the desired tag.

2. **Fully-associative mapping:** In this technique, each cache address contains not only a main memory address' content, but also the complete main memory address. To determine if a desired main memory address is in the cache, we simultaneously (associatively) compare all the addresses stored in the cache with the desired address.

3. **Set-associative mapping**: This technique is a compromise between direct and fully-associative mapping. As in direct-mapping, an index maps each main memory address to a cache address, but now each cache address contains the content and tags of two or more memory locations, called a set or a line. To determine if a desired main memory address is in the cache, we go to the cache address indicated by the index, and we then simultaneously (associatively) compare all the tags at that location (i.e., of that set) with the desired tag. A cache with a set of size N is called an N-way set-associative cache. 2-way, 4- way and 8-way set associative caches are common.

Direct-mapped caches are easy to implement, but may result in numerous misses if two or more words with the same index are accessed frequently, since each will bump the other out of the cache. Fully-associative caches on the other hand are fast but the comparison logic is expensive to implement. Set-associative caches can reduce misses compared to direct-mapped caches, without requiring nearly as much comparison logic as fully-associative caches. Caches are usually designed to treat collections of a small number of adjacent mainmemory addresses as one indivisible block, typically consisting of about 8 address.

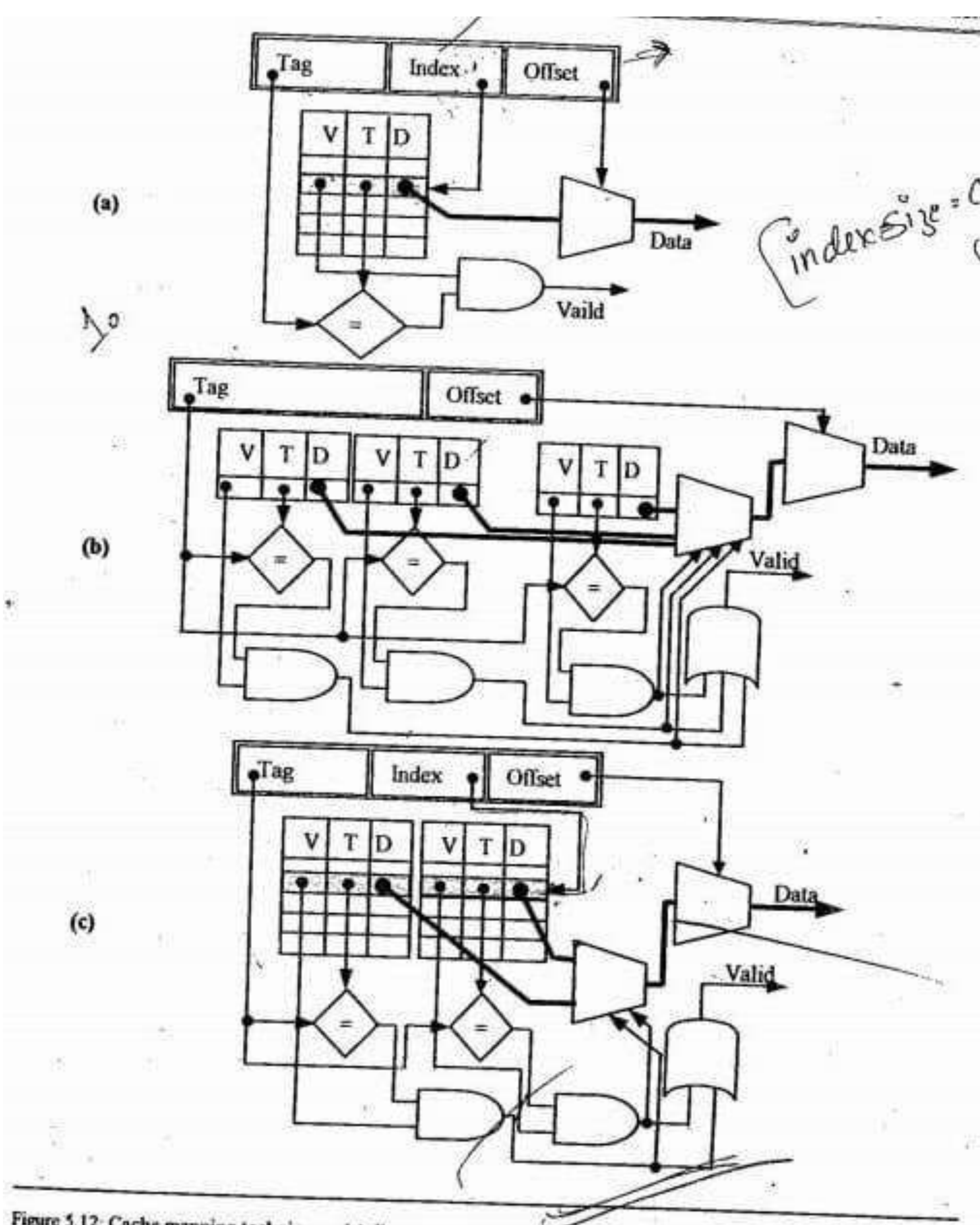


Figure 5.12: Cache mapping techniques: (a) direct-mapped, (b) fully associative, (c) two-way set associative.

Cache replacement policy: The cache-replacement policy is the technique for choosing which cache block to replace when a fully-associative cache is full, or when a set-associative cache's line is full. Note that there is no choice in a direct-mapped cache; a main memory address always maps to the same cache address and thus replaces whatever block is already there. There are three common replacement policies. A random replacement policy chooses the block to replace randomly. While simple to implement, this policy does nothing to prevent replacing block that's likely to be used again soon. A least-recently used (LRU) replacement policy replaces the block that has not been accessed for the longest time, assuming that this means that it is least likely to be accessed in the near future. This policy provides for an excellent hit/miss ratio but requires expensive hardware to keep track of the times blocks are accessed. A first-in-first-out (FIFO) replacement policy uses a queue of size N, pushing each block

address onto the queue when the address is accessed, and then choosing the block to replace by popping the queue.

Cache Write technique:

When we write to a cache, we must at some point update the memory. Such update is only an issue for data cache, since instruction cache is read-only. There are two common update techniques, write-through and write-back.

In the write-through technique, whenever we write to the cache, we also write to main memory, requiring the processor to wait until the write to main memory completes. While easy to implement, this technique may result in several unnecessary writes to main memory. For example, suppose a program writes to a block in the cache, then reads it, and then writes it again, with the block staying in the cache during all three accesses. There would have been no need to update the main memory after the first write, since the second write overwrites this first write.

The write-back technique reduces the number of writes to main memory by writing a block to main memory only when the block is being replaced, and then only if the block was written to during its stay in the cache. This technique requires that we associate an extra bit, called a dirty bit, with each block. We set this bit whenever we write to the block in the cache, and we then check it when replacing the block to determine if we should copy the block to main memory.

INTERFACING:

ARBITARATION: Several situations existed in which multiple peripherals might request service from a single resource. For example, multiple peripherals might share a single microprocessor that services their interrupt requests. As another example, multiple peripherals might share a single DMA controller that services their DMA requests. In such situations, two or more peripherals may request service simultaneously. We therefore must have some method to arbitrate among these contending requests, i.e., to decide which one of the contending peripherals gets service, and thus which peripherals need to wait. Several methods exist, which we now discuss.

PRIORITY ARBITARATION: One arbitration method uses a single-purpose processor, called a priority arbiter. We illustrate a priority arbiter arbitrating among multiple peripherals using vectored interrupt to request servicing from a microprocessor, as illustrated in Figure 6.9. Each of the peripherals makes its request to the arbiter. The arbiter in turn asserts the microprocessor interrupt, and waits for the interrupt acknowledgment. The arbiter then provides an acknowledgement to exactly one peripheral, which permits that peripheral to put its interrupt vector address on the data bus

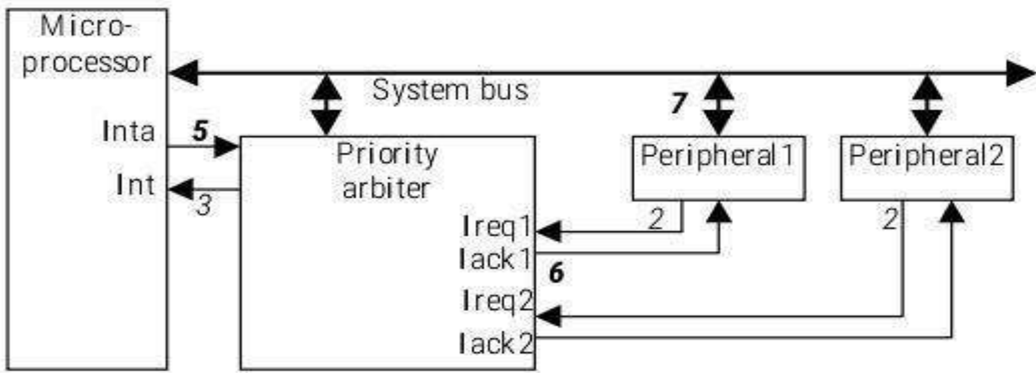
Priority arbiters typically use one of two common schemes to determine priority among peripherals: fixed priority or rotating priority. In fixed priority arbitration, each peripheral has a unique rank among all the peripherals. The rank can be represented as a number, so if there

are four peripherals, each peripheral is ranked 1, 2, 3 or 4. If two peripherals simultaneously seek servicing, the arbiter chooses the one with the higher rank.

In rotating priority arbitration (also called round-robin), the arbiter changes priority of peripherals based on the history of servicing of those peripherals. For example, one rotating priority scheme grants service to the least-recently serviced of the contending peripherals. This scheme obviously requires a more complex arbiter.

We prefer fixed priority when there is a clear difference in priority among peripherals. However, in many cases the peripherals are somewhat equal, so arbitrarily ranking them could cause high-ranked peripherals to get much more servicing than low ranked ones. Rotating priority ensures a more equitable distribution of servicing in this case.

Figure 6.9: Arbitration using a priority arbiter.

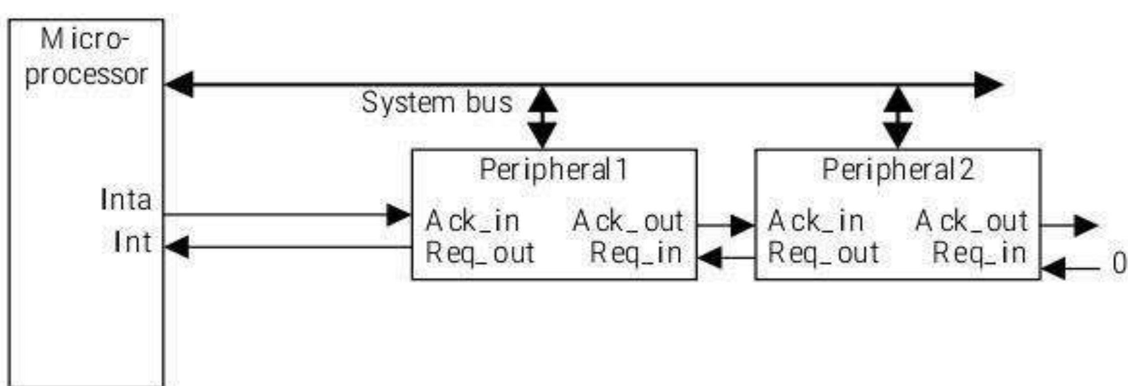


DAISY-CHAIN ARBITARATION: The daisy-chain arbitration method builds arbitration right into the peripherals. A daisy-chain configuration is shown in Figure 6.10, again using vectored interrupt to illustrate the method. Each peripheral has a request output and an acknowledge input, as before. But now each peripheral also has a request input and an acknowledge output. A peripheral-asserts its request output if it requires servicing, OR if its request input is asserted; the latter means that one of the "upstream" devices is requesting servicing. Thus, if any peripheral needs servicing, its request will flow through the downstream peripherals and eventually reach the microprocessor. Even if more than one peripheral request servicing, the microprocessor will see only one request. The microprocessor acknowledge connects to the first peripheral. If this peripheral is requesting service, it proceeds to put its interrupt vector address on the system bus. But if it doesn't need service, then it instead passes the acknowledgement upstream to the next peripheral, by asserting it's acknowledge output. In the same manner, the next peripheral may either begin being serviced or may instead pass the acknowledgement along. Obviously, the peripheral at the front of the chain, i.e., the one to which the microprocessor acknowledge is connected, has highest priority, and the peripheral at the end of the chain has lowest priority.

We prefer a daisy-chain priority configuration over a priority arbiter when we want to be able to add or remove peripherals from an embedded system without redesigning the system. Although conceptually we could add as many peripherals to a daisy-chain as we desired, in reality the servicing response time for peripherals at the end of the chain could become

intolerably slow. In contrast to a daisy-chain, a priority arbiter has a fixed number of channels; once they are all used, the system needs to be redesigned in order to accommodate more peripherals. However, a daisy-chain has the drawback of not supporting more advanced priority schemes, like rotating priority. A second drawback is that if a peripheral in the chain stops working, other peripherals may lose their access to the processor.

Figure 6.10: Arbitration using a daisy-chain configuration.



NETWORK-ORIENTED ARBITARATION METHODS: The arbitration methods described are typically used to arbitrate among peripherals in an embedded system. However, many embedded systems contain multiple microprocessors communicating via a shared bus; such a bus is sometimes called a network. Arbitration in such cases is typically built right into the bus protocol, since the bus serves as the only connection among the microprocessors. A key feature of such a connection is that a processor about to write to the bus has no way of knowing whether another processor is about to simultaneously write to the bus. Because of the relatively long wires and high capacitances of such buses, a processor may write many bits of data before those bits appear at another processor. For example, Ethernet and I2C use a method in which multiple processors may write to the bus simultaneously, resulting in a collision and causing any data on the bus to be corrupted. The processors detect this collision, stop transmitting their data, wait for some time, and then try transmitting again. The protocols must ensure that the contending processors don't start sending again at the same time, or must at least use statistical methods that make the chances of them sending again at the same time small.

As another example, the CAN bus uses a clever address encoding scheme such that if two addresses are written simultaneously by different processors using the bus, the higher-priority address will override the lower-priority one. Each processor that is writing the bus also checks the bus, and if the address it is writing does not appear, then that processor realizes that a higher-priority transfer is taking place and so that processor stops writing the bus.

MULTILEVEL BUS ARCHITECHTURES: A microprocessor-based embedded system will have numerous types of communications that must take place, varying in their frequencies and speed requirements. The most frequent and high-speed communications will likely be between the microprocessor and its memories. Less frequent communications, requiring less speed, will be between the microprocessor and its peripherals, like a UART. We

could try to implement a single high-speed bus for all the communications, but this approach has several disadvantages.

Figure 6.11: A two-level bus architecture.

First, it requires each peripheral to have a high-speed bus interface. Since a peripheral may not need such high-speed communication, having such an interface may result in extra gates, power consumption and cost. Second, since a highspeed bus will be very processor-specific, a peripheral with an interface to that bus may not be very portable. Third, having too many peripherals on the bus may result in a slower bus.

Therefore, we often design systems with two levels of buses: a high-speed processor local bus and a lower-speed peripheral bus, as illustrated in Figure 6.11. The processor local bus typically connects the microprocessor, cache, memory controllers, certain highspeed co-processors, and is highly processor specific. It is usually wide, as wide as a memory word.

The peripheral bus connects those processors that do not have fast processor local bus access as a top priority, but rather emphasize portability, low power, or low gate count. The peripheral bus is typically an industry standard bus, such as ISA or PCI, thus supporting portability of the peripherals. It is often narrower and/or slower than a processor local bus, thus requiring fewer gates and less power for interfacing. A bridge connects the two buses.

A bridge is a single-purpose processor that converts communication on one bus to communication on another bus. For example, the microprocessor may generate a read on the processor local bus with an address corresponding to a peripheral. The bridge detects that the address corresponds to a peripheral, and thus it then generates a read on the peripheral bus. After receiving the data, the bridge sends that data to the microprocessor. The microprocessor thus need not even know that a bridge exists -- it receives the data, albeit a few cycles later, as if the peripheral were on the processor local bus.

A three-level bus hierarchy is also possible, as proposed by the VSI Alliance. The first level is the processor local bus, the second level a system bus, and the third level a peripheral bus. The system bus would be a high-speed bus, but would offload much of the traffic from the processor local bus. It may be beneficial in complex systems with numerous co-processors.

ADVANCED COMMUNICATION PRINCIPALS:

Communication can take place over a number of different types of media, such as a single wire, a set of wires, radio waves, or infrared waves. We refer to the medium that is used to carry data from one device to another as the *physical layer*. Depending on the protocol, we may refer to an actor as a *device* or *node*. In either case, a device is simply a processor that uses the physical layer to send or receive data to and from another device.

In this section, we provide a general description of serial communication, parallel communication, and wireless communication. In addition, we describe communication principles such as layering, error detection and correction, data security, and plug and play.

Parallel Communication

Parallel communication takes place when the physical layer is capable of carrying multiple bits of data from one device to another. This means that the data bus is composed of multiple data wires, in addition to control and possibly power wires, running in parallel from one device to another. Each wire carries one of the bits. Parallel communication has the advantage of high data throughput, if the length of the bus is short. The length of a parallel bus must be kept short because long parallel wires will result in high capacitance values, and transmitting a bit on a bus with a higher capacitance value will require more time to charge or discharge. In addition, small variations in the length of the individual wires of a parallel bus can cause the received bits of the data word to arrive at different times. Such misalignment of data becomes more of a problem as the length of a parallel bus increases. Another problem with parallel buses is the fact that they are more costly to construct and may be bulky, especially when considering the insulation that must be used to prevent the noise from each wire from interfering with the other wires. For example, a 32-wire cable connecting two devices together will cost much more and be larger than a two-wire cable.

In general, parallel communication is used when connecting devices that reside on the same IC or devices that reside on the same circuit board. Since the length of such buses is short, the capacitance load, data misalignment and cost problems mentioned earlier do not play an important role.

Serial Communication

Serial communication involves a physical layer that carries one bit of data at a time. This means that the data bus is composed of a single data wire, along with control and possibly

power wires, running from one device to another. In serial communication, a word of data is transmitted one bit at a time. Serial buses are capable of higher throughputs than parallel buses when used to connect two physically distant devices. The reason for this is that a serial bus will have less average capacitance, enabling it to send more bits per unit of time. In addition, a serial bus cable is cheaper to build because it has fewer wires. The disadvantage of a serial bus is that the interfacing logic and communication protocol will be more complex. On the sending side, a transmitter must decompose data words into bits and on the receiving side, and the receiver must compose bits into words.

Most serial bus protocols eliminate the need for extra control signals, such as read and write signals, by using the same wire that carries data for this purpose. This is performed as follows. When data is to be sent, the sender first transmits a bit called a *start bit*. A start bit merely signals the receiver to wakeup and start receiving data. The start bit is then followed by N data bits, where N is the size of the word, and a *stop bit*. The stop bit signals to the receiver the end of the transmission. Often, both the transmitter and the receiver agree on the transmission speed used to send and receive data. After sending a start bit, the transmitter sends all N bits at the predetermined transmission speed. Likewise, on seeing a start bit, a receiver simply starts sampling the data at a predetermined frequency until all N bits are assembled. Another common synchronization technique is to use an additional wire for clocking purposes (see the I^2C bus protocol). Here, the transmitter and receiver devices use this clock line to determine when to send or sample the data.

Wireless Communication

Wireless communication eliminates the need for devices to be physically connected in order to communicate. The physical layer used in wireless communication is typically either an infrared (IR) channel or a radio frequency (RF) channel.

Infrared uses electromagnetic wave frequencies that are just below the visible light spectrum, thus undetectable by the human eye. These waves can be generated by using an infrared diode and detected by using an infrared transistor. An infrared diode is similar to a red or green diode except that it emits infrared light. An infrared transistor is a transistor that conducts (i.e., allows current to flow from its source to its drain), when exposed to infrared light. A simple transmitter can send 1s by turning on its infrared diode and can send 0s by turning off its infrared diode. Correspondingly, a receiver will detect 1s when current flows through its infrared transistor and 0s otherwise. The advantage of using infrared communication is that it is relatively cheap to build transmitters and receivers. The disadvantage of using infrared is the need for line of sight between the transmitter and receiver, resulting in a very restricted communication range.

Radio frequency (RF) uses electromagnetic wave frequencies in the radio spectrum. A transmitter here will need to use analog circuitry and an antenna to transmit data. Likewise, a receiver will need to use an antenna and analog circuitry to receive data. One advantage of using RF is that, generally, a line of sight is not necessary and thus longer distance communication is possible. The range of communication is, of course, dependent on the transmission power used by the transmitter.

Typically, RF transmitters and receivers must agree on a specific frequency in order to send and receive data. Using *frequency hopping*, it is possible for the transmitter and receiver to communicate while constantly changing the transmission frequency. Of course, both devices must have a common understanding of the sequence for frequency hops. Frequency hopping allows more devices to share a fixed set of frequencies and is commonly used in wireless communication protocols designed for networks of computers and other electronic devices.

Layering

Layering is a hierarchical organization of a communication protocol where lower levels of the protocol provide services to the higher levels. We have already discussed the physical layer. The physical layer provides the basic service of sending and receiving bits or words of data. The next higher-level protocol uses the physical layer to send and receive packets of data, where a packet of data is composed of possibly multiple bytes. The next higher level uses the packet transmission service of its lower level to perhaps send different type of data such as acknowledgments, special requests, and so on. Typically, the lowest level consists of the physical layer and the highest level consists of the application layer. The application layer provides abstract services to the application such as ftp or http.

Layering is a way to break the complexity of a communication protocol into independent pieces, thus making it easier to design and understand, much like a programmer abstracting away complexities of a program by creating objects or libraries of routines. In communication and networking, the concept of layering is very fundamental.

Error Detection and Correction

Error detection is the ability of a receiver to detect errors that may occur during the transmission of a data word or packet. The most common types of errors are bit errors and burst of bit errors. A *bit error* occurs when a single bit in a word or data packet is received as its inverted value. A *burst of bit error* occurs when consecutive bits of a word or data packet are received incorrectly. Given that an error is detected, *error correction* is the ability of a receiver and transmitter to cooperate in order to correct the problem. The ability to detect and correct errors is often part of a bus protocol. We will [illegible] parity and checksum error detection algorithms, which are commonly used in bus protocols.

ESD UNIT-5

STATE MACHINE AND CONCURRENT PROCESS MODELS

Models vs. Languages

A *computation model* describes desired system behavior, while a *language* captures models. A model is a conceptual notion, while a language captures that concept in a concrete form. A model can be captured in a variety of languages, while a language can capture a variety of models, as illustrated in Figure 8.1.

Let us consider an analogy involving cooking recipes. A recipe is like a model, a conceptual notion, consisting of a set of instructions for cooking something, and a notion of how to sequence among those instructions. For example, a particular recipe may include a requirement of first putting flour in a bowl and then mixing in two eggs. English is a language capable of capturing a recipe. This simple example illustrates three important points. First, a recipe can be captured faithfully in various languages, such as English, Spanish, or Japanese. In fact, a recipe exists independent of its capture in a particular language — some recipes are never written down! Second, a particular language can capture many different conceptual notions other than recipes, such as poetry or stories. Third, certain languages may be better at capturing recipes than others — while English works fine, a primitive language without words for "boil" or "simmer" may be cumbersome to use for capturing recipes.

Returning now from cooking to computing, consider sequential programs. A *sequential program* is a model, a conceptual notion, consisting of a set of program instructions for computing something, and a notion of how to sequence among those instructions. For example, a particular sequential program may include a requirement of first initializing a variable to 10, and then adding 2 to that variable. C is a language capable of capturing a sequential program. As in our analogy above, there are three important points to remember. First, a sequential program can be captured in any of various languages, such as C, C++, or Java. Second, a particular language can capture many different models other than sequential programs, such as state machines or dataflow. Third, certain languages may be better at capturing sequential programs than others — while C works fine, a primitive language like assembly without constructs for "loops" or "procedures" may be cumbersome to use for capturing sequential programs. As another example, C can be used to capture state machines, as we will see later, but a language intended specifically to capture state machines might be more convenient.

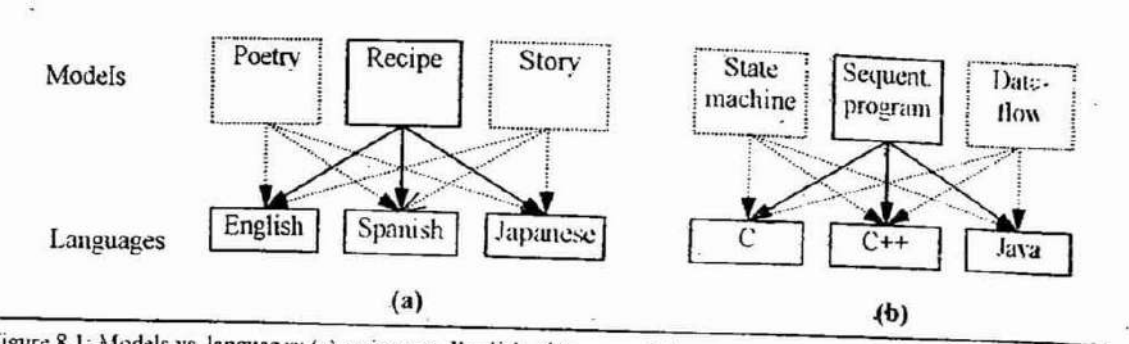
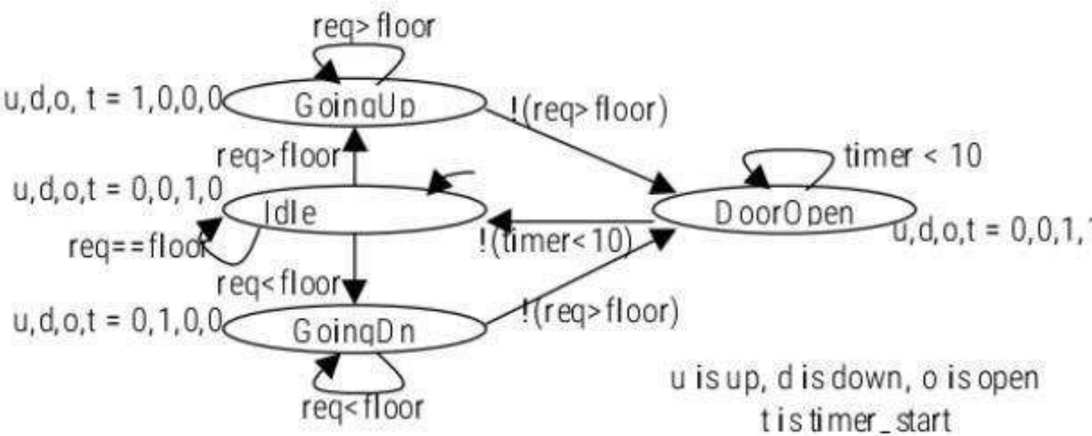


Figure 8.1: Models vs. languages: (a) recipes vs. English, (b) sequential programs vs. C.

BASIC STATE MACHINE MODEL: In a state machine model, we describe system behaviour as a set of possible states; the system can only be in one of these states at a given time. We also describe the possible transitions from one state to another depending on input values. Finally, we describe the actions that occur when in a state or when transitioning between states.

For example, Figure 8.2 shows a state machine description of the Unit Control part of our elevator example. The initial state, Idle, sets up and down to 0 and open to 1. The state machine stays in state Idle until the requested floor differs from the current floor. If the requested floor is greater, then the machine transitions to state Going Up, which sets up to 1, whereas if the requested floor is less, then the machine transitions to state Going Down, which sets down to 1. The machine stays in either state until the current floor equals the requested floor, after which the machine transitions to state Door Open, which sets open to 1. We assume the system includes a timer, so we start the timer while transitioning to Door Open. We stay in this state until the timer says 10 seconds have passed, after which we transition back to the Idle state.

Figure 8.2: The elevator's UnitControl process described using a state machine.



Finite-state machines: FSM

We have described state machines somewhat informally, but now provide a more formal definition. We start by defining the well-known finite-state machine computation model, or FSM, and then we'll define extensions to that model to obtain a more useful model for embedded system design. An FSM is a 6-tuple, where:

S is a set of states {s0, s1, … , s_i},
I is a set of inputs {i0, i1, …, i_m},
O is a set of outputs {o0, o1, …, o_n},
F is a next-state function (i.e., transitions), mapping states and inputs to states (S X I->S)
H is an output function, mapping current states to outputs (S O), and s0 is an initial state.

The above is a Moore-type FSM above, which associates outputs with states. A second type of FSM is a Mealy-type FSM, which associates outputs with transitions, i.e., H maps S x I -> O. You might remember that Moore outputs are associated with states by noting that the name Moore has two o's in it, which look like states in a state diagram. Many tools that support FSM's

support combinations of the two types, meaning we can associate outputs with states, transitions, or both.
We can use some shorthand notations to simplify FSM descriptions. First, there may be many system outputs, so rather than explicitly assigning every output in every state, we can say that any outputs not assigned in a state are implicitly assigned 0. Second, we often use an FSM to describe a single-purpose processor (i.e., hardware). Most hardware is synchronous, meaning that register updates are synchronized to clock pulses, e.g., registers are only updated on the rising (or falling) edge of a clock. Such an FSM would have every transition condition AND with the clock edge (e.g., clock rising and x and y). To avoid having to add this clock edge to every transition condition, we can simply say that the FSM is synchronous, meaning that every transition condition is implicitly AND with the clock edge.

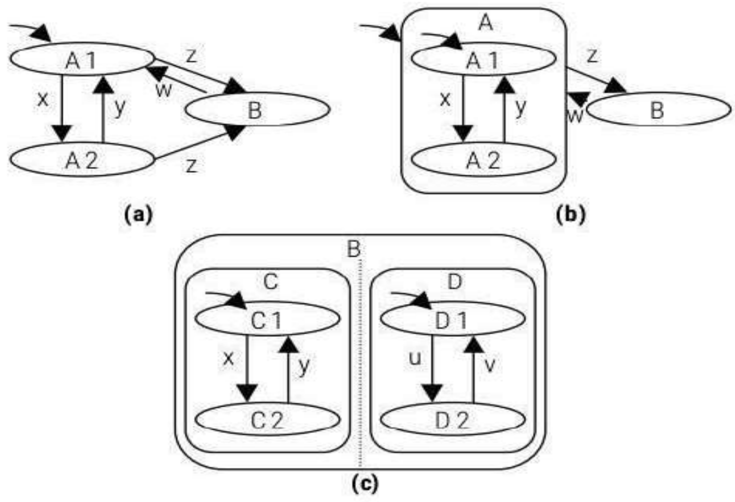
HCFSM AND STATE CHARTS LANGUAGE:

Harel proposed extensions to the state machine model to support hierarchy and concurrency, and developed State charts, a graphical state machine language designed to capture that model. We refer to the model as a hierarchical/concurrent FSM, or HCFSM.
The hierarchy extension allows us to decompose a state into another state machine, or conversely stated, to group several states into a new hierarchical state. For example, consider the state machine in Figure 8.5(a), having three states A1 (the initial state), A2, and B. Whenever we are in either A1 or A2 and event z occurs, we transition to state B. We can simplify this state machine by grouping A1 and A2 into a hierarchical state A, as shown in Figure 8.5(b). State A is the initial state, which in turn has an initial state A1. We draw the transition to B on event z as originating from state A, not A1 or A2. The meaning is that regardless of whether we are in A1 or A2, event z causes a transition to state B.

As another hierarchy example, consider our earlier elevator example, and suppose that we want to add a control input fire, along with new behaviour that immediately moves the elevator down to the first floor and opens the door when fire is true. As shown in Figure 8.6(a), we can capture this behaviour by adding a transition from every state originally in Unit Control to a new state called Fire Going Down, which moves the elevator to the first floor, followed by a state Fire Door Open, which holds the door open on the first floor. When fire becomes false, we go to the Idle state. While this new state machine captures the desired behaviour, it is becoming more complex due to many more transitions, and harder to comprehend due to more states. We can use hierarchy to reduce the number of transitions and enhance understandability. As shown in Figure 8.6(b), we can group the original state machine into a hierarchical state called Normal Mode, and group the fire-related states into a state called Fire Mode. This grouping reduces the number of transitions, since instead of four transitions from each original state to the fire-related states, we need only one transition, from Normal Mode to Fire Mode. This grouping also enhances understandability, since it clearly represents two main operating modes, one normal and one in case of fire

The concurrency extension allows us to use hierarchy to decompose a state into two concurrent states, or conversely stated, to group two concurrent states into a new hierarchical state. For example, Figure 8.5 (c), shows a state B decomposed into two concurrent states C and D. C happens to be decomposed into another state machine, as does D. Figure 8.7 shows the entire Elevator Controller behaviour captured as a HCFSM with two concurrent states.

Figure 8.5: Adding hierarchy and concurrency to the state machine model: (a) three-state example without hierarchy, (b) same example with hierarchy, (c) concurrency.



Therefore, we see that there are two methods for using hierarchy to decompose a state into substates. OR-decomposition decomposes a state into sequential states, in which only one state is active at a time (either the first state OR the second state OR the third state, etc.). AND-decomposition decomposes a state into concurrent states, all of which are active at a time (the first state AND the second state AND the third state, etc.).

The State-charts language includes numerous additional constructs to improve state machine capture. A timeout is a transition with a time limit as its condition. The transition is automatically taken if the transition source state is active for an amount of time equal to the limit. Note that this would have simplified the Unit Control state machine; rather than starting and checking an external timer, we could simply have created a transition from Door Open to Idle with the condition time out (10). History is a mechanism for remembering the last substate that an OR-decomposed state A was in before transitioning to another state B. Upon re-entering state A, we can start with the remembered substate rather than A's initial state. Thus, the transition leaving A is treated much like an interrupt and B as an interrupt service routine.

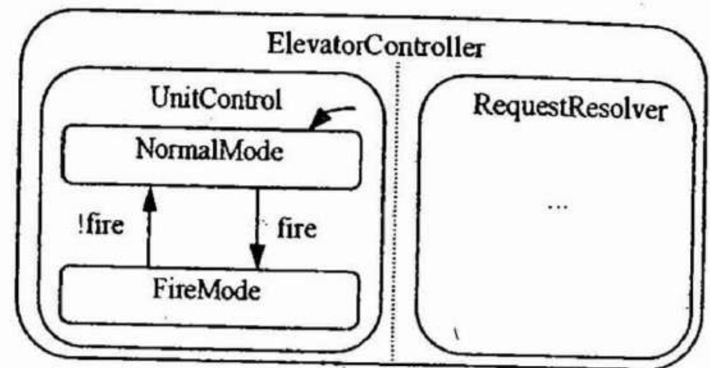


Figure 8.7: Using concurrency in an HCFSM to describe both processes of the Elevator Controller.

PROGRAM STATE MACHINE MODEL:

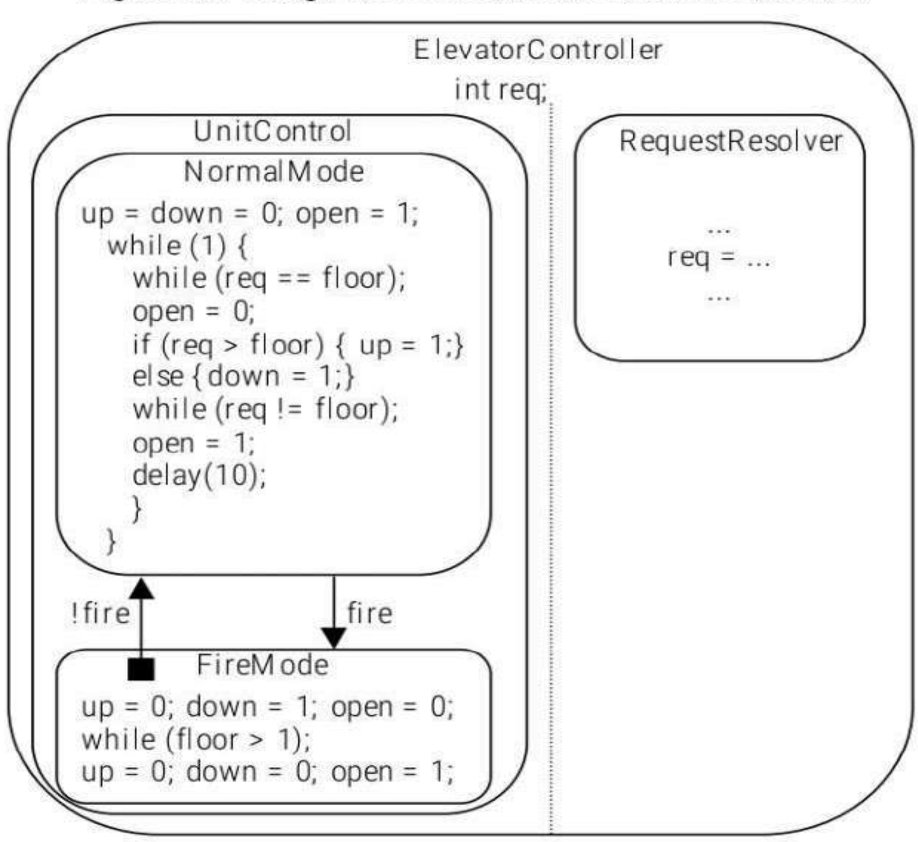
The program-state machine (PSM) model extends state machines to allow use of sequential program code to define a state's actions (including extensions for complex data types and variables), as well as including the hierarchy and concurrency extensions of HCFSM. Thus, PSM is a merger of the HCFSM and sequential program models, subsuming both models. A PSM having only one state (called a program-state in PSM terminology), where that state's actions are defined using a sequential program, is the same as a sequential program. A PSM having many states, whose actions are all just assignment statements, is the same as an HCFSM. Lying between these two extremes are various combinations of the two models.

For example, Figure 8.8 shows a PSM description of the Elevator Controller behaviour, which we AND-decompose into two concurrent program-states Unit Control and Request Resolver, as in the earlier HCFSM example. Furthermore, we OR-decompose Unit Control into two sequential program-states, Normal Mode and Fire Mode, again as in the HCFSM example. However, unlike the HCFSM example, we describe Normal Mode as a sequential program (identical to that of Figure 8.1(c)) rather than a state machine. Likewise, we describe Fire Mode as a sequential program. We didn't have to use sequential programs for those program-states,

and could have used state machines for one or both -- the point is that PSM allows the designer to choose whichever model is most appropriate.

PSM enforces a stricter hierarchy than the HCFSM model used in State charts. In State charts, transitions may point to or from a substate within a state, such as the transition in Figure 8.6(b) pointing from the substate of the state to the Normal Mode state. Having this transition start from Fire Door Open rather than Fire Mode causes the elevator to always go all the way down to the first floor when the fire input becomes true, even if the input is true just momentarily. PSM, on the other hand, only allows transitions between sibling states, i.e., between states with the same parent state. PSM's model of hierarchy is the same as in sequential program languages that use subroutines for hierarchy; namely, we always enter the subroutine from one point, and when we exit the sub-routine we do not specify to where we are exiting.

Figure 8.8: Using PSM to describe the ElevatorController.



As in the sequential programming model, but unlike the HCFSM model, PSM includes the notion of a program-state completing. If the program-state is a sequential program, then reaching the end of the code means the program-state is complete. If the program-state is OR-decomposed into substates, then a special complete substate may be added. Transitions may occur from a substate to the complete substate (but no transitions may leave the complete substate), which when entered means that the program-state is complete. Consequently, PSM introduces two types of transitions. A transitionimmediately (TI) transition is taken immediately if its condition becomes true, regardless of the status of the source program-state -- this is the same as the transition type in an HCFSM. A second, new type of transition, transition-on-completion (TOC), is taken only if the condition is true AND the source program-state is complete. Graphically, a TOC transition is drawn originating from a filled square inside

a state, rather than from the state's perimeter. We used a TOC transition in Figure 8.8 to transition from Fire Mode to Normal Mode only after Fire Mode completed, meaning that the elevator had reached the first floor. By supporting both types of transitions, PSM elegantly merges the reactive nature of HCFSM models (using TI transitions) with the transformational nature of sequential program models (using TOC transitions).

The Role of an Appropriate Model and Language

Specifying embedded system functionality can be a hard task, but an appropriate computation model can help. The model shapes the way we think of the system. The language should capture the model easily.

Consider how models shaped the way we thought about the elevator controller example's *UnitControl* behavior. In order to create the sequential program that we captured in Figure 8.2(c), we were thinking in terms of a *sequence of actions*. First, we wait for the requested floor to differ from the target floor, then we close the door, then we move up or down to the desired floor, then we open the door, and then we repeat this sequence. In contrast, in order to create the state machine that we captured in Figure 8.3, we were thinking in terms of possible system states and the transitions among those states. Many individuals say that, for this example, the state machine model feels more natural than the sequential program model. When a system must react to a variety of changing inputs, a state machine model may be a good choice. Furthermore, notice that the HCFSM model was able to describe the fire behavior nicely, while the FSM or FSMD models would have become somewhat complex.

The language should capture our chosen model easily. Ideally, the language would have constructs that directly capture features of the model — a language for capturing state machines should have constructs for capturing states and transitions, for example. However, such a model/language match is not always the case. As you may have already ascertained, the most common situation of a model/language mismatch in embedded systems is that of having a language designed to support the sequential program model, but wanting to capture a system using a state machine model. In this case, we can use structured techniques for capturing the state machine model in the sequential program language, as shown earlier. To see the benefit of using the best model, think of how the fire behavior would have been incorporated into the sequential program of Figure 8.2(c). We would have had to insert checks for the signal throughout the code, making the code very complex.

The moral of the story here is that often we cannot choose the language used to capture embedded system functionality — that choice is often dictated by other factors. But we need not be limited to using the model directly supported by that language. We can use a different model if that model provides an advantage, and then capture the model in the language using structured techniques.

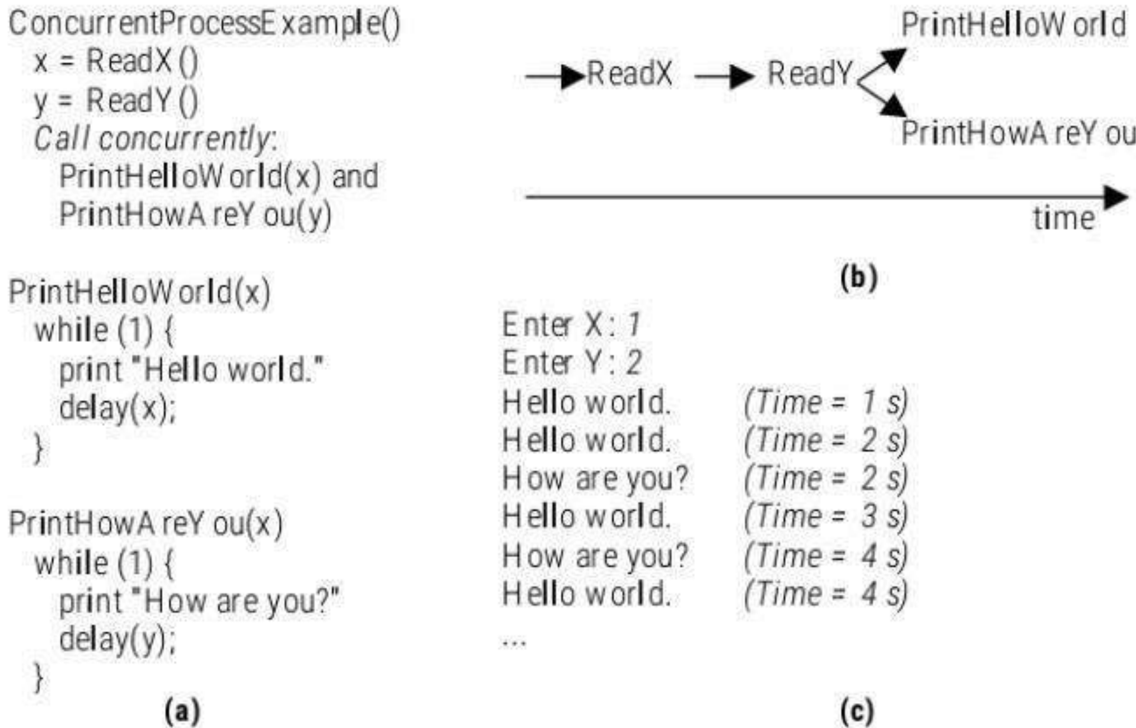
CONCURRENT PROCESS MODEL:

In a concurrent process model, we describe system behaviour as a set of processes, which communicate with one another. A process refers to a repeating sequential program. While many embedded systems are most easily thought of as one process, other systems are more easily thought of as having multiple processes running concurrently.

For example, consider the following made-up system. The system allows a user to provide two numbers X and Y. We then want to write "Hello World" to a display every X seconds, and "How are you" to the display every Y seconds. A very simple way to describe this system using concurrent processes is shown in Figure 8.9(a). After reading in X and Y, we call two subroutines concurrently. One subroutine print's "Hello World" every X seconds, the other prints "How are you" every Y seconds. (Note that you can't call two subroutines concurrently in a pure sequential program model, such as the model supported by the basic version of the C language). As shown in Figure 8.9(b), these two subroutines execute simultaneously. Sample output for X=1 and Y=2 is shown in Figure 8.9(c).

To see why concurrent processes were helpful, try describing the same system using a sequential program model (i.e., one process). You'll find yourself exerting effort figuring out how to schedule the two subroutines into one sequential program. Since this example is a trivial one, this extra effort is not a serious problem, but for a complex system, this extra effort can be significant and can detract from the time you have to focus on the desired system behaviour. Recall that we described our elevator controller using two "blocks." Each block is really a process. The controller was simply easier to comprehend if we thought of the two blocks independently.

Figure 8.9: A simple concurrent process example: (a) pseudo-code, (b) subroutine execution over time, (c) sample input and output.



COMMUNICATION AMONG PROCESSES:

Two common methods for communication among processors are SHARED MEMORY AND MESSAGE PASSING.

SHARED MEMORY: In the shared data technique, processes read and write variables that both processes can access, called global variables. For example, in the elevator example above, the Request Resolver process writes to a variable request, which is also read by the Unit Control process.

MESSAGE PASSING:

In message passing, communication occurs using send and receive constructs that are part of the computation model. Specifically, a process P explicitly sends data to another process Q, which must explicitly receive the data. In the elevator example, Request Resolver would include a statement: Send(Unit Controller request). Likewise, Unit Control would include statements of the form: Receive(Request Resolver, uc_request). rr_req and uc_req are variables local to each process.

Message passing may be blocking or non-blocking. In blocking message passing, a sending process must wait until the receiving process receives the data before executing the statement following the send. Thus, the processes synchronize at their send/receive points. In fact, a designer may use a send/receive with no actual message being passed, in order to achieve the synchronization. In non-blocking message passing, the sending process need not wait for the receive to occur before executing more statements. Therefore, a queue is implied in which the sent data must be stored before being received by the receiving process.

Example:

```
void processA() {
  while( 1 ) {
    produce(&data)
    send(B, &data);
    /* region 1 */
    receive(B, &data);
    consume(&data);
  }
}
```

```
void processB() {
  while( 1 ) {
    receive(A, &data);
    transform(&data)
    send(A, &data);
    /* region 2 */
  }
}
```

Communication among processes using send and receive.

The identifier uniquely identifies one of the processes that are currently executing in the system. An example of message passing is illustrated in Figure 8.16. Here process *A*, after producing a data packet, sends it to process *B*. Meanwhile, process *B* receives the packet, performs some transformation on the data and sends it back to *A*. Process *A*, after receiving the data packet, consumes it and the cycle repeats. Regions of code labeled 1 and 2 are segments that perform auxiliary functions in each process.

Note that receive operations are always blocking. That means the once a process executes a receive operation, it is blocked until another process executes the corresponding send operation. The send operations, on the other hand, may or may not be blocking. One reason for having nonblocking send operations is to allow a process that just performed a send operation to continue with its execution. In our example, the regions of code labeled 1 and 2 are executed immediately after a send operation, even though the receiving process may not have received the data item.

Synchronization among Processes

In order for two or more concurrent processes to accomplish a common task, they must at times synchronize their execution. Synchronization among processes means that one process must wait for another process to compute some value, reach a known point in its execution, or signal some condition, before it (the waiting process) proceeds.

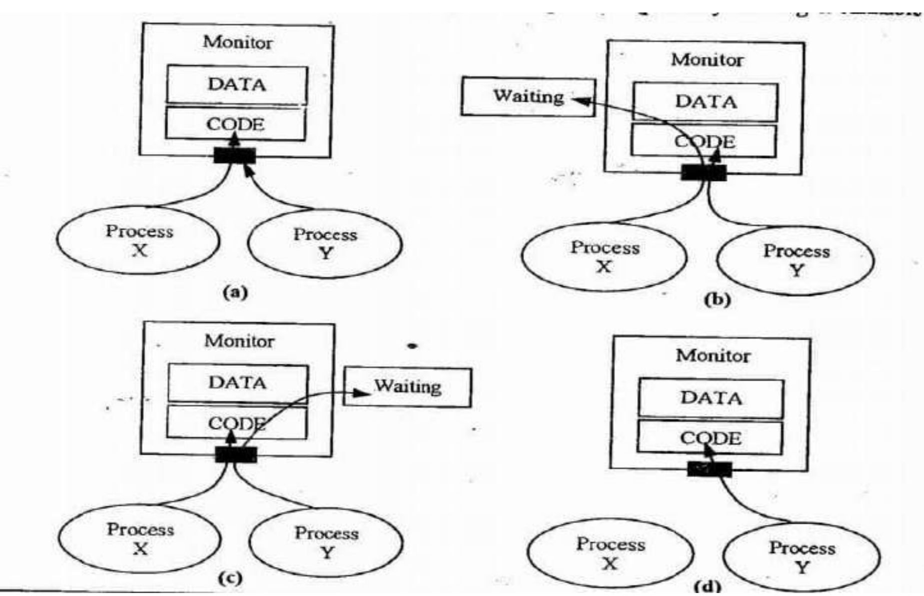
The join operation that we discussed earlier is a limited form of synchronization among two processes. Recall that here, one process performed a join operation on another process, indicating that it should be blocked until that other process terminates. The blocking send and receive protocols, a.k.a. synchronous send and receive, discussed in the previous section, also serve to synchronize processes. When one process performs a send or receive operation, it is blocked until the other process reaches its receive or send point, respectively, before the blocked process is allowed to continue. We will next describe *condition variables* and *monitors* as synchronization mechanisms.

Condition Variables

One way to achieve synchronization among concurrently executing processes is to use a special construct called a *condition variable*. A condition variable is an object that permits two kinds of operations, called *signal* and *wait*, to be performed on it. When wait is performed on a condition variable, the process that performed the wait operation is blocked until another process performs a signal operation on the same condition variable. The semantics of a wait operation is in fact a bit more complex. When a process, say, *A*, executes a wait operation, it passes it a mutex variable that it has already acquired the lock for. The wait operation will then cause the mutex to be unlocked such that another process, say, *B*, may be able to enter a critical section and compute some value or make some condition become true. Once the condition becomes true, process *B* will signal the condition variable causing process *A* to become runnable and implicitly reacquire the mutex lock.

Monitors

Another way to achieve synchronization among concurrently executing processes is to use a special construct called a monitor. A *monitor* is a collection of data and methods or subroutines that operate on this data similar to an object in an object-oriented paradigm. A special guarding property of a monitor guarantees that only one process is allowed to execute inside the monitor at a given time. In other words, one and only one of the methods of a monitor can be active at any given time. A process, say, *X*, is allowed to enter a monitor if there are no other processes executing in that monitor. This is shown in Figure 8.18(a). Once in a monitor, *X* has exclusive access to the data inside the monitor. If, and when, *X* executes a wait operation on a condition variable, also defined inside the monitor, it will be blocked waiting as shown in Figure 8.18 (b). At this point, another process, say *Y*, is allowed to enter the monitor. If *Y* signals the condition that *X* is currently waiting on, *Y* will be blocked and *X* will be allowed to reenter the monitor. This is shown in Figure 8.18 (c). Then, once X terminates, or waits on a condition, *Y* is allowed to reenter and finish its execution as shown in Figure 8.18 (d).



Fig;8.18 procedure consumer example with monitors

IMPLEMENTATION: The most straightforward method for implementing concurrent processes on processors is to implement each process on its own processor. This method is common when each process is to be implemented using a single-purpose processor.

CREATING AND TERMINATING PROCESSES:

One method for sharing a processor among multiple processes is to manually rewrite the processes as a single sequential program. For example, consider our Hello World program from earlier. We could rewrite the concurrent process model as a sequential one by replacing the concurrent running of the Print HelloWorld and Print How Are You routines by the following:

```
I = 1; T = 0;
while (1) {
    Delay(I); T = T + I
    if X modulo T is 0 then call PrintHelloWorld
    if Y modulo T is 0 then call PrintHowAreYou
}
```

Manually rewriting a model may be practical for simple examples, but extremely difficult for more complex examples. While some automated techniques have evolved to assist with such rewriting of concurrent processes into a sequential program, these techniques are not very commonly used.

a second, far more common method for sharing a processor among multiple processes is to rely on a multi-tasking operating system. An operating system is a lowlevel program that runs on a processor, responsible for scheduling processes, allocating storage, and interfacing to peripherals, among other things. A real-time operating system (RTOS) is an operating system that allows one to specify constraints on the rate of processes, and that guarantees that these

rate constraints will be met. In such an approach, we would describe our concurrent processes using either a language with processes builtin (such as Ada or Java), or a sequential program language (like C or C++) using a library of routines that extends the language to support concurrent processes. POSIX threads were developed for the latter purpose.

A third method for sharing a processor among multiple processes is to convert the processes to a sequential program that includes a process scheduler right in the code. This method results in less overhead since it does not rely on an operating system, but also yields code that may be harder to maintain.

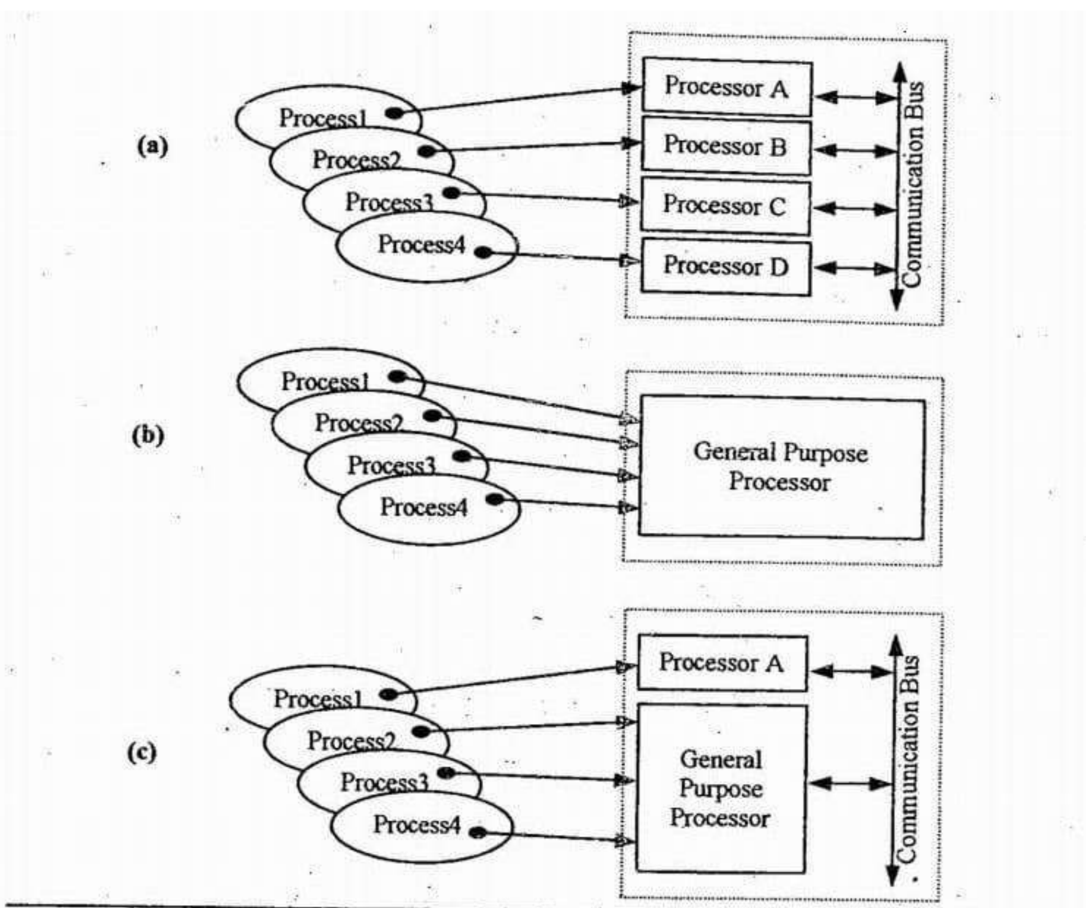


Figure 8.20: Mapping processes on processors: (a) processes mapped on multiple single-purpose processors, (b) processes mapped on one general-purpose processor, (c) processes mapped to a combination of single and general purpose processors.

Suspending and Resuming Processes

If multiple processes are implemented using single-purpose processors, suspending or resuming them must be built as part of the processor's implementation. For example, the processors may be designed having an extra input. When this input is asserted, the processor is suspended, otherwise it is executing. If multiple processors are implemented using a single general-purpose processor, than suspending or resuming the processes must be built into the programming language or multitasking library that is used to describe the processes. In both cases, the programming language or library may rely on the underlying operating system to handle these operations.

Joining a Process

If multiple processes are implemented using single-purpose processors, than for one process *X* to join another process *Y* would require building additional logic that will determine when *Y* has reached its completion point and in response resume *X*. Therefore, in addition to having input signals that signal when a processor should suspend, each processor must have output signals that indicate when that processor is done executing its task. If multiple processors are implemented using a single general-purpose processors, join must be built into the language or multitasking library that is used to describe the processes. In both cases, the programming language or library may rely on the underlying operating system to handle this operation.

Scheduling Processes

When multiple processes are implemented on a single general-purpose processor, the manner in which these processes are executed on a single shared processor plays an important role in meeting each process's timing requirements. This task of deciding when and for how long a processor executes a particular process is know as *process scheduling*. A *scheduler* is a special process that performs process scheduling. A scheduler can either be implemented as a *nonpreemptive scheduler* or *preemptive scheduler*. A nonpreemptive scheduler only decides on what process to select for execution, on the processor, once the currently executing process completes its execution. A preemptive scheduler is a scheduler that only allows a process to executed for a predetermined amount of time, called a *time quantum*, before preempting in order to allow another process to execute on the processor. This time quantum may be 10 to 100s of milliseconds long. The length of this time quantum greatly determines the response time of a system.

REAL TIME SYSTEMS:

We will discuss some operating systems that are designed to support real-time systems. Note that the term real-time system refers to a class of applications or embedded systems that exhibit the real-time characteristics and requirements mentioned above. Real-time operating systems, on the other hand, refer to underlying implementations or systems that supports real-time systems. In other words, real-time operating systems provide mechanisms, primitives, and guidelines for building embedded systems that are real-time in nature.

Windows CE

Windows CE was built specifically for the embedded system and the appliance market providing a scalable real-time 32-bit platform that can be used in a wide variety of embedded systems and products. One of the benefits of using Windows CE as an RTOS that supports the Windows application-programming interface API, which has gained great popularity. This operating system provides a set of Internet browsing and serving services that make it suitable

for systems that are designed to interface to the Internet. The Windows CE kernel allows for 256 priority levels per processes and implements preemptive priority scheduling. The size of the Windows CE kernel is 400 Kbytes.

QNX

The QNX RTOS architecture consists of a real-time micro-kernel surrounded by a collection of optional processes (called resource managers) that provide POSIX and UNIX compatible system services. A micro-kernel is a name given to a kernel that only supports the most basic services and operations that typical operating system's provide. However, by including or excluding resource manager processes the developer can scale QNX down for ROM-based embedded systems, or scale it up to encompass hundreds of processors connected by various networking and communication technologies. Resource manager processes are modules that can be added or removed from the basic micro-kernel to best fit the functionality provided by the operating system to that needed by the application. The micro-kernel of QNX occupies less than 10 Kbytes and complies with POSIX real-time standard. QNX supports up to 32 priority levels per process and implements preemptive process scheduling using either FIFO, round robin, adaptive, or priority-driven scheduling.

ESD UNIT-6

DESIGN&IC TECHNOLOGY

IC MANUFACTURING STEPS:

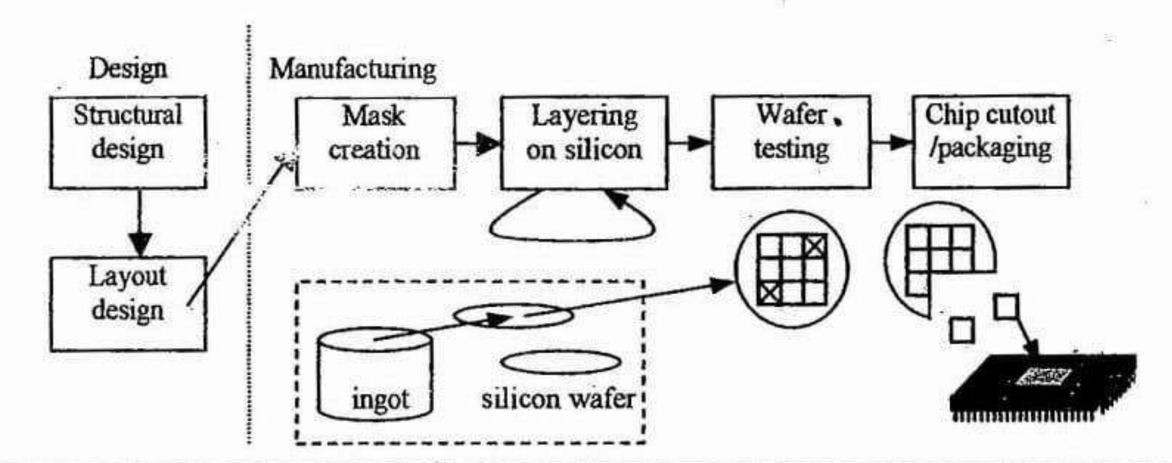


Figure 10.3: IC manufacturing steps.

Figure 10.3 illustrates IC manufacturing steps. During the design phase, a designer creates a structural design and then generates a layout for that design. The design phase may take many months. Once fully satisfied, the designer provides the layout to an IC manufacturer, also known as a fabrication plant, "fab," or foundry. Because the layout is often provided to the manufacturer on a magnetic tape commonly used for storing large quantities of digital data, providing the layout to a manufacturer is commonly referred to as "tape-out." Because part of the manufacturing process involves spinning molten silicon, generating ICs is also referred to as a "silicon spin." IC manufacturing may take months.

Manufacturing consists of several main steps. The first step is to create a set of masks corresponding to the layout. Hundreds of masks may be required. The second step is to use each of these masks to create the various layers on the silicon surface, consisting of several substeps per mask. We point out that this layering process doesn't just create a single IC, but rather numerous ICs at once. The reason is that ICs are built on a silicon wafer. A silicon wafer is a thin polished circle, sliced from a cylinder of silicon, like a pepperoni slice intended for a pizza is sliced from a cylinder of sausage. A silicon wafer may be tens of centimeters in diameter, whereas an IC is usually less than one centimeter on a side, thus meaning that a wafer can hold tens of ICs (perhaps 100). Thus, the masks actually contain tens of identical regions, so that tens of ICs are being created simultaneously on a silicon wafer, as shown in the figure. Think of this the next time that you watch a movie where everyone is trying to get their hands on some "prototype chip" that must be found lest the world be destroyed; if there's one chip, there's probably 50 or 100 more that were made on the same wafer, lying around somewhere! (Never mind — just enjoy the movie).

The third step is to test the ICs on the wafer. ICs determined to be bad are marked, literally, so that they will be thrown away later. The machines that perform such testing are known appropriately as *testers*. They use probes that contact the pads, or input and output ports, of a particular IC on the wafer. They then apply streams of input sequences and look for the appropriate output sequences. These testers are very expensive devices, and their cost per IC pin has actually increased. Unfortunately, with all the steps required to build an IC and because of the extremely small sizes of the transistors and wires involved, bad ICs are quite common. Yield is a measure of the percentage of good ICs versus bad ICs containing errors.

Finally, the last step is to cut out each IC and mount the good ones in an IC package, which of course gets tested again.

Full-Custom (VLSI) IC Technology

In a full-custom IC technology, the designer creates the complete layout — a task often called *physical design* or *VLSI design* (where VLSI stands for very large scale integrated circuit). The designer must design or obtain a transistor-level circuit for every processor and memory. After this point, there are several key physical design tasks necessary to obtain a good layout:

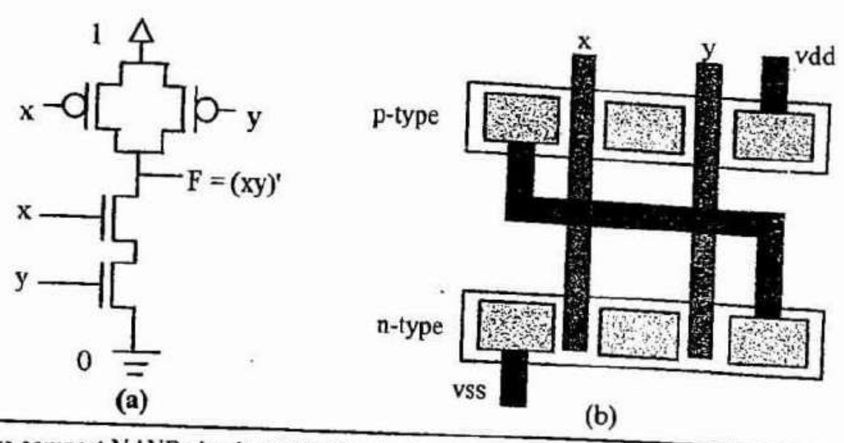


Figure 10.5: A more compact NAND circuit: (a) NAND circuit schematic, (b) compacted layout.

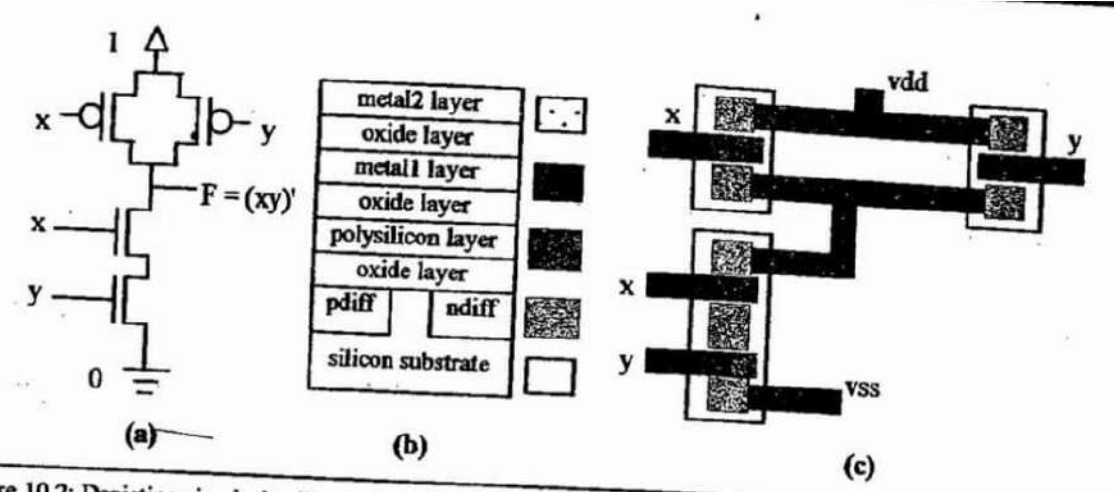


Figure 10.2: Depicting circuits in silicon: (a) a NAND circuit schematic, (b) layers, (c) top-down view of the NAND circuit on an IC.

- *Placement*: the task of placing and orienting every transistor somewhere on the IC.
- *Routing*: the task of running wires between the transistors, without intersecting other wires or transistors.
- *Sizing*: the task of deciding how big each wire and transistor will be. Larger wires and transistors provide better performance but consume more power and require more silicon area.

A good layout is typically defined by characteristics like speed and size. Speed is the longest path from input to output, or from register to register, typically measured in nanoseconds. Size is the total silicon area necessary to implement the complete circuit. Both of these features are usually improved when the circuit is highly compacted, namely, when transistors that are connected are placed close together and hence their connecting wires are shorter. Consider for example the NAND layout of Figure 10.2(c). In that example, we did not pay attention to creating a compact layout. Figure 10.5(b) shows a compacted version of the NAND circuit. Notice how much less area is wasted in this compacted version. However, such compaction must obey certain *design rules*. For example, two transistors must be spaced apart a minimum distance lest they electrically interfere with one another.

In the past, many transistor circuits were converted by hand into compact layouts. Such *circuit design* was a common job. However, ICs can now hold so many transistors, numbering in the hundreds of millions, that laying out complete ICs by hand would require an absurd amount of time. Thus, hand layout is usually used only for relatively small, critical components, like the ALU of a microprocessor, or for basic components like logic gates that will be heavily reused.

Instead of hand layout, most layout today is done using automated layout tools, known as *physical design* tools. These tools typically include powerful optimization algorithms that run for hours or days seeking to improve the speed and size of a layout.

The advantages of full-custom IC technology include its excellent efficiency with respect to power, performance, and size. Interconnected transistors can be placed near each other and thus be connected by very short wires, yielding good performance and power. Furthermore, only those transistors necessary for the circuit being designed appear on the IC, resulting in no wasted area due to unused transistors.

The main disadvantages of full-custom IC technology are its high NRE cost and long time-to-market. These disadvantages stem from having to design a complete layout, which even with the aid of tools can be time-consuming and error-prone. Furthermore, masks for every IC layer must be created, increasing NRE cost and delaying time-to-market. In addition, errors discovered after manufacturing the IC are common, often requiring several respins.

Semi-Custom (ASIC) IC Technology

As mentioned above, creating a full-custom layout can be quite challenging. A designer using a semi-custom IC technology has this burden partially relieved, since rather than creating a full-custom layout, the designer connects pre-layed-out building blocks. The common name for such a semi-custom IC is an *application specific integrated circuit* (ASIC). The term *application specific* was likely chosen to contrast with general-purpose processor ICs, since for many years a processor was implemented as its own IC. ASICs in contrast implemented a circuit specific to a particular application (i.e., a single-purpose processor). Today, however, a single ASIC may implement a combination of general-purpose and single-purpose processors. Needless to say, there is much confusion related to use of the term *ASIC* today. Thus, we prefer the term *semi-custom IC*.

The two main types of semi-custom IC technologies are gate array and standard cell. With either type, the main advantages versus full-custom are reduced NRE cost and faster time-to-market, since less layout and mask creation must be performed. The main disadvantage is reduced performance, power, and size efficiency. However, relative to programmable IC technology (yet to be discussed), semi-custom is extremely efficient in terms of performance, power, and size. Because of its good efficiency coupled with reduced NRE costs, semi-custom is the most popular IC technology today.

Gate Array Semi-Custom IC Technology

In a gate array IC technology, all of the logic gates of the IC have already been layed out with their placement on the IC known, leaving the designer with the task of connecting the gates (routing) in a manner implementing the desired circuit. Note that gate here refers to a logic gate (e.g., AND, OR) rather than a terminal of a CMOS transistor. A simplified gate array layout is shown Figure 10.6(a).

Because the IC's gates are placed beforehand, many of them may go unused, since we may not need all instances of each type of gate in our particular circuit. Furthermore, routing wires between gates may be quite long since the gate placement was decided before knowing what connections would be made.

Standard Cell Semi-Custom IC Technology

In standard cell IC technology, common logic functions, or *cells*, have already been compactly layed out. Examples of cells include a NAND gate, a NOR gate, a 2×1 multiplexor, and a combination of AND-OR-INVERT gates. The transistors within a cell are already layed out, but the placement of cells has not been determined. A designer thus must decide which cells to use, where to place them, and how to route among them. A standard cell layout is shown in Figure 10.6(b).

Standard cell therefore requires more NRE cost and longer time-to-market than gate array, since there is more layout remaining to be performed and all masks must still be made. However, NRE and time-to-market is still much less than full-custom, since the intricate layout within each cell is already completed. In addition, efficiency is very good compared to gate array, since only those cells needed are actually used, and their placement can be made so as to reduce interconnect. Furthermore, each cell may implement more complex functions than in gate arrays, leading to more compact designs.

A compromise between gate array and standard cell semi-custom ICs is known as a *cell array*, or cell-based array. A cell array is pretty much what we'd expect it to be based on its name. Cells, which you'll remember can be more complex than gates, have already been layed out, and have also already been placed. Thus, the designer need only connect the cells together.

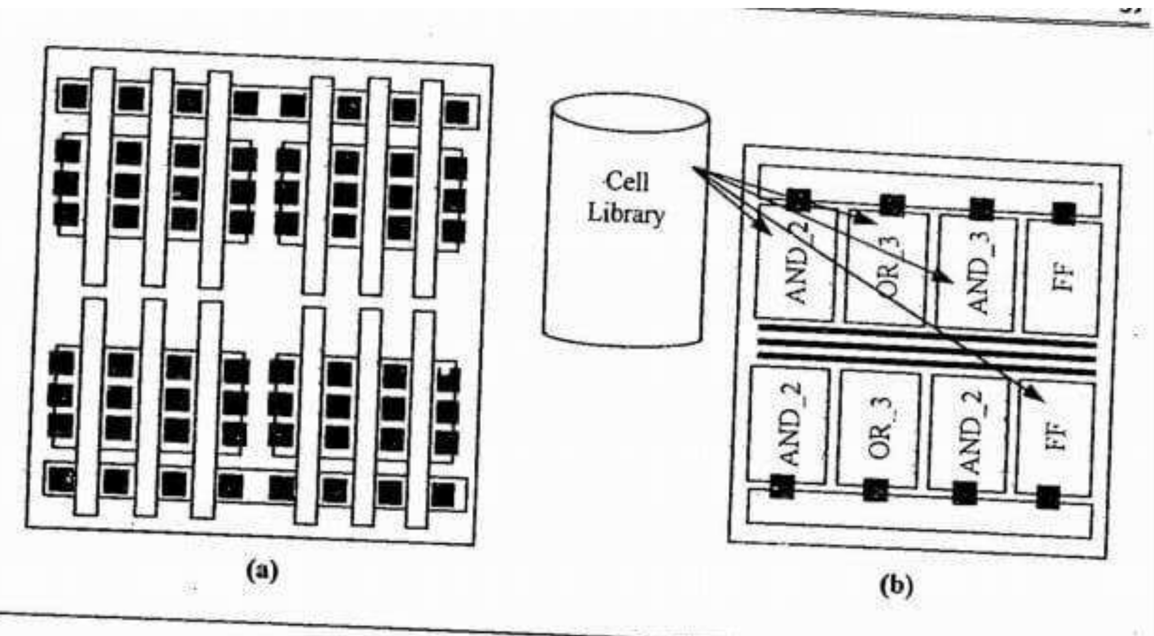


Figure 10.6: Semi-custom IC technology: (a) gate array, (b) standard cell.

Programmable Logic Device (PLD) IC Technology

The time required to manufacture an IC is measured in months, typically two to three months. While we may accept this time once we're ready to manufacture our final system, we probably can't wait so long to obtain a prototype of our system. Furthermore, the NRE cost to manufacture an IC (i.e., creating a layout and masks) may be too expensive to amortize over the number of ICs we plan to manufacture if that number is small. In addition, manufacturing an IC is risky, since we may discover after such manufacturing that an IC doesn't work properly in its target system, either due to manufacturing problems or due to an incorrect initial design. Thus, we never know how many respins will be necessary before we get a working IC; a recent study stated that the industry average was 3.5 spins. Therefore, we would like an IC technology that allows us to implement our system's structure on an IC, but that doesn't require us to manufacture that IC. Instead, we want an IC that we can program in the field, with the field being our lab or office. The term *program* here does not refer to writing software that executes on a microprocessor, but rather to configuring logic circuits and interconnection switches to implement a desired structural circuit.

Programmable logic device (PLD) technology satisfies this goal. A *PLD* is a pre-manufactured IC that we can purchase and then configure to implement our desired circuit.

An early example of a PLD was a programmable logic array (PLA), introduced in the early 1970s. A *PLA* was a small PLD with two levels of logic, a programmable AND array and a programmable OR array. Every PLA input and its complement was connected to every AND gate. So if a PLA had 10 inputs, every AND gate had 20 inputs. Any of these connections could be broken, meaning that each AND gate could generate any product term. Likewise, each OR gate could generate any sum of AND gate outputs. A PAL (programmable array logic) is another PLD type that eliminates the programmability of the OR array to reduce size and delay. PLAs and PALs are often referred to as simple PLDs, or *SPLDs*.

As IC capacity grew over the years, SPLDs could not simply be extended by adding more inputs, since the number of required connections to the AND array inputs would grow too high. Thus, the new capacity was taken advantage of instead by integrating numerous SPLDs on a single chip and adding programmable interconnect between them, resulting in what is known as a complex PLD, or *CPLD*. CPLDs often contain latches to enable implementation of sequential circuits also. Figure 10.7 illustrates a sample architecture for a CPLD. The top half of the figure is an SPLD that can implement any function of the chip's input signals as well as any SPLD output signal. The bottom half represents another identical SPLD. The array on the left consists of vertical lines that can be programmed to connect with any of the horizontal lines, so that any signal's true or complemented value can be fed into any gate. The output of each SPLD feeds into an IO cell. The IO cell can be programmed to pass the latched or unlatched, true or complemented, output to the CPLD's external output, and/or to the programmable array on the left as input to SPLDs.

While able to implement more complex circuits than SPLDs, CPLDs suffer from the problem of not scaling well as their sizes increase. For example, supposed the CPLD architecture of Figure 10.7 had 4 inputs and 2 outputs. Then there would be 6 signals in the programmable array, plus 6 more for those signals' complements, thus requiring 12-input AND gates. Likewise, suppose there were 12 inputs and 6 outputs. Then there would be 18 + 18 signals, requiring 36-input AND gates. Notice such an architecture doesn't scale well.

The logical solution is to build devices that are more modular in nature. In particular, there is no need to connect every input signal and every output signal to every AND gate. A more flexible approach can be used in which a subset of inputs and outputs are input to each SPLD. This more modular, more scalable approach to PLD design resulted in architectures

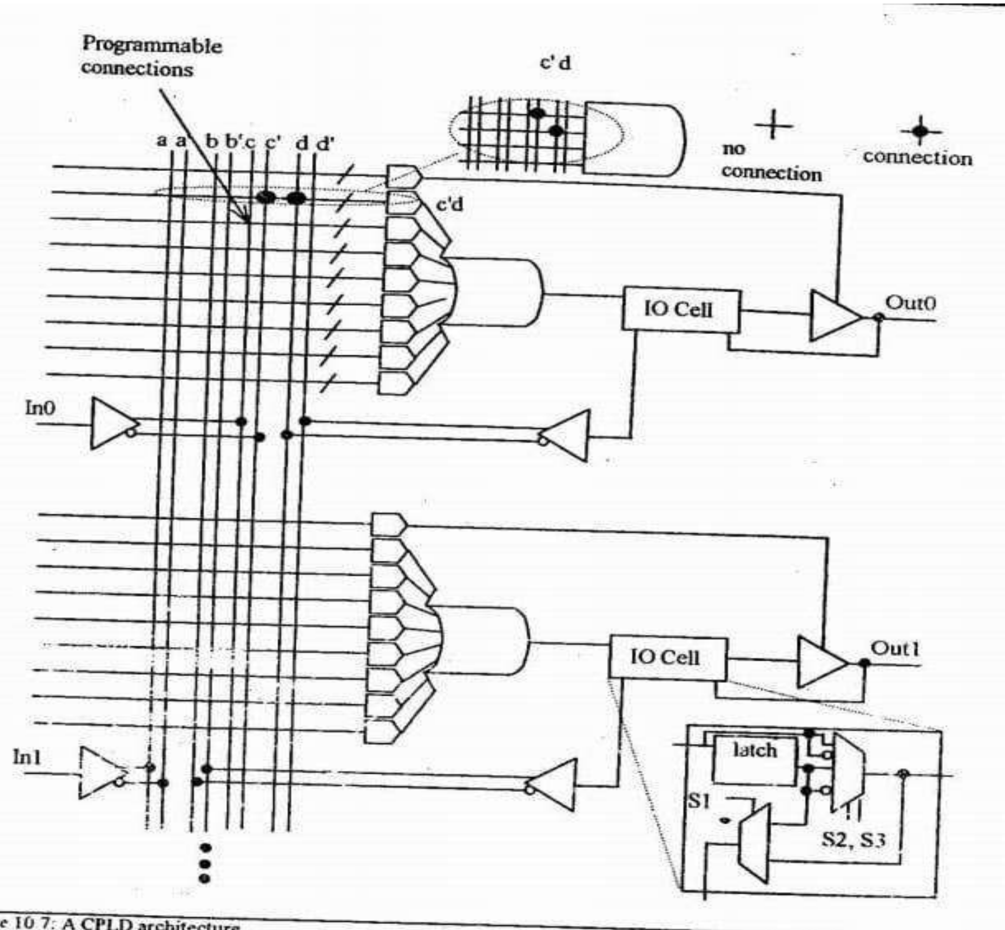


Figure 10.7: A CPLD architecture.

known as field-programmable gate arrays (FPGAs). An *FPGA* consists of arrays of programmable logic blocks connected by programmable interconnect blocks.

The name FPGA was intended to contrast these devices with traditional gate arrays, which need masks to create the interconnections between the already layed-out gates. FPGAs in contrast, have their interconnections as well as logic blocks programmed in the field, meaning in the designer's lab. However, most FPGA architectures do not have arrays of gates anywhere to be found, and thus the name FPGA can be somewhat misleading.

THREE IC TECHNOLOGIES:

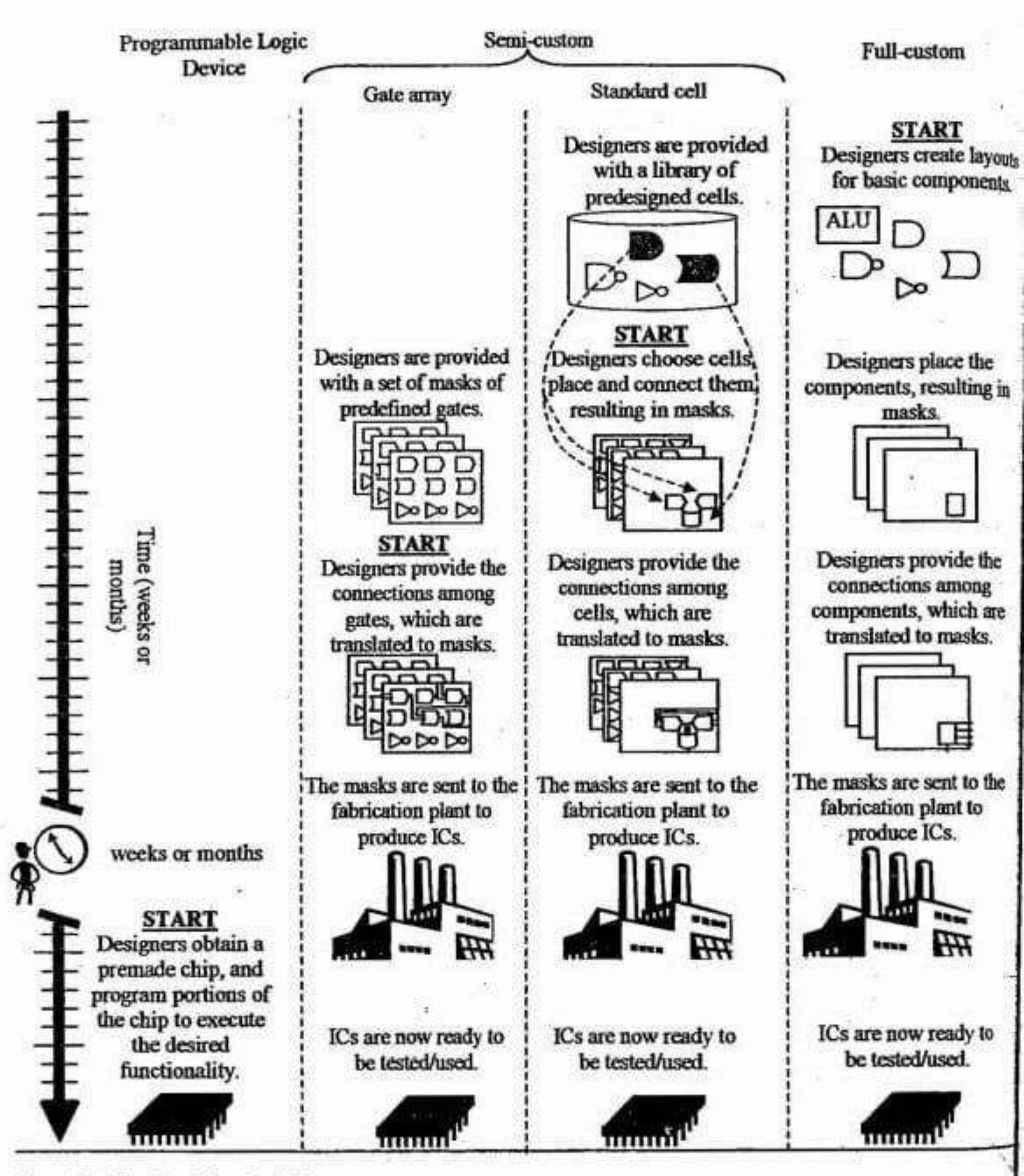


Figure 10.4: The three IC technologies.

AUTOMATION: SYSTHESIS

We now provide a brief overview of the various levels of synthesis. A standard definition for *synthesize* is "forming a complex whole by combining parts." In the context of digital hardware design, however, the term has taken on the meaning of "automatically converting a system's behavioral description into a structural implementation," where that implementation is a complex whole formed by parts. The structural implementation must optimize some set of design metrics, such as performance, size, and power.

To better understand the meaning of converting from a behavioral description to a structural implementation, Gajski developed the Y-chart, shown in Figure 11.4. The chart consists of three axes, behavioral, structural, and physical, each representing a type of a description of a digital system, as follows:

- A *behavioral* description defines outputs as a function of inputs. It describes the algorithms we'll use to obtain those outputs, but does not say how we'll implement those algorithms.
- A *structural* description implements that behavior by connecting components with known behavior.

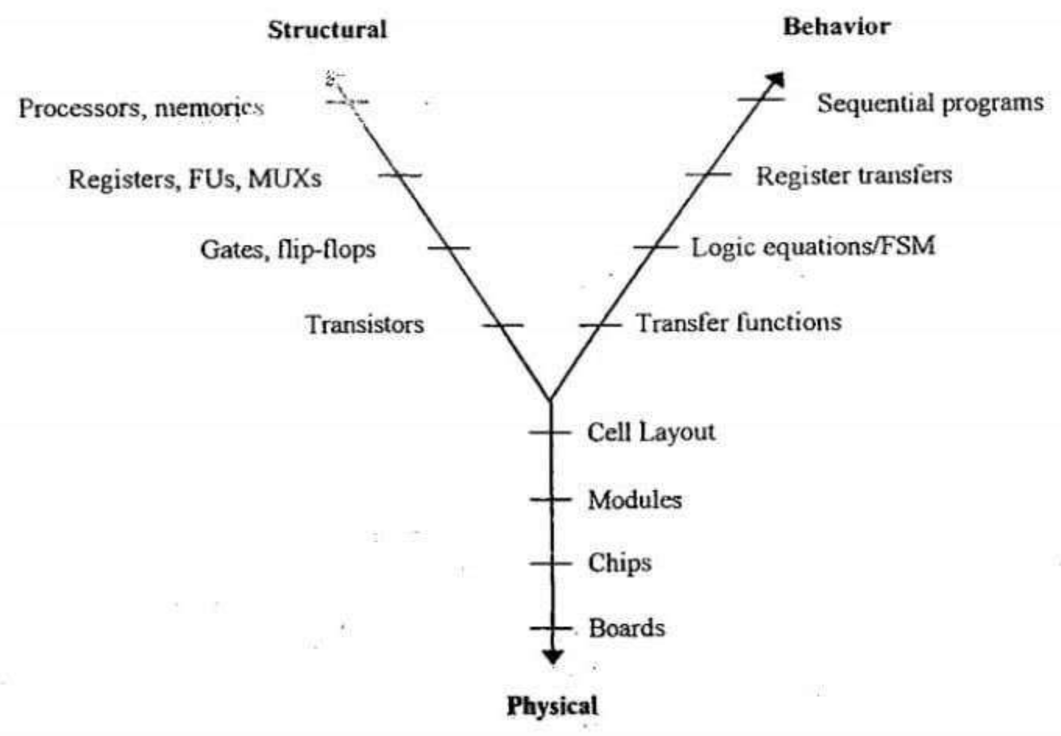


Figure 11.4: Gajski's Y-chart.

- A *physical* description tells us the sizes and locations on a chip or board of a system's components and their interconnecting wires.

For example, addition is a behavior, while a carry-ripple adder is a structure. Likewise, a sequential program that sequences through an array to find the array's largest-valued element is a behavior, while a controller and datapath implementing that algorithm is a structure.

The chart also shows that each description can exist at one of various levels of abstraction. For example, at the gate-level of abstraction, a behavioral description consists of logic equations, a structural description consists of a connection of gates, and a physical description consists of a placement of gates/cells and a routing among them. As another example, at the system level of abstraction, a behavioral description may consist of communicating sequential programs (processes), a structural description of a connection of processors and memories, and a physical description of a placement of processor/memory cores and buses on an IC or a board.

Synthesis can generally be thought of as converting a behavioral description at a particular abstraction level to a structural description. That structural description may be at the same level or a lower one, but not a higher one. We now describe synthesis techniques at several different abstraction levels.

Logic Synthesis

Logic synthesis automatically converts a logic-level behavior, consisting of logic equations and/or an FSM, into a structural implementation, consisting of connected gates. Let us divide logic synthesis into combinational-logic synthesis and FSM synthesis. Combinational logic synthesis can be further subdivided into two-level minimization and multilevel minimization.

Two-level logic minimization: We can represent any logic function as a sum of products (or a product of sums). We can implement this function directly using a level consisting of AND gates, one for each product term, and a second level consisting of a single OR gate. Thus, we have two levels, plus inverters necessary to complement some inputs to the AND gates. The longest possible path from an input signal to an output signal passes through at most two gates, not counting inverters. We cannot in general obtain faster performance. F

Multilevel Logic Minimization: The previous paragraphs dealt with minimizing the AND gates and their sizes in a two-level sum-of-products implementation. We noted that a two-level implementation has excellent performance, with the longest path being only two gates. However, perhaps we don't need such great performance. Rather, perhaps we are willing to sacrifice some performance if such a sacrifice would decrease the circuit size further than even the best two-level implementation. We can achieve such a trade-off by using multiple levels of logic.

Register–Transfer Synthesis

Logic synthesis allowed us to describe our system as boolean equations, or as an FSM. However, many systems are too complex to initially describe at this logic level of abstraction. Instead, we often describe our system using a more abstract (and hence powerful) computation model, such as an FSMD.

Recall that an FSMD allows variable declarations of complex data types, and allows arithmetic actions and conditions. Clearly, more work is necessary to convert an FSMD to gates than to convert an FSM to gates, and this extra work is performed by register–transfer synthesis. Register–transfer (RT) synthesis takes an FSMD and converts it to a custom single-purpose processor, consisting of a datapath and an FSM controller. In particular, it generates a complete datapath, consisting of register units to store variables, functional units to implement arithmetic operations, and connection units (buses and multiplexors) to connect these other units. It also generates an FSM that controls this datapath.

Creating the datapath requires solving two key subproblems: allocation and binding. Allocation is the problem of instantiating storage units, functional units, and connection units. Binding is the problem of mapping FSMD operations to specific units.

As in logic synthesis, both of these RT synthesis problems are hard to solve optimally.

Behavioral Synthesis

In RT synthesis, we describe the actions that occur on every clock cycle of the system, using an FSMD. However, for many systems, we are only interested in having the output be a correct function of the inputs, and we don't care how that function is broken down into clock cycles. Therefore, we may want to describe such a system using a sequential program.

Behavioral synthesis converts a single sequential program into a single-purpose processor structure that executes only that one program. Behavioral synthesis has also been referred to as *high-level synthesis*.

A sequential program differs from an FSMD in that it does not require us to schedule the system's actions into states when describing the behavior. Therefore, implementing a sequential program requires not only allocation and binding as in RT synthesis, but also scheduling. Scheduling is the assignment of a sequential program's operations to states.

In Chapter 2, we provided a simple technique for behavioral synthesis. First, we provided templates for converting every sequential program construct into an equivalent set of states, thus accomplishing scheduling. Second, we provided a simple allocation and binding method, namely, allocating one storage unit for every variable, one functional unit for every operation, and one connection unit for every transfer. While this approach results in a correct processor circuit, the circuit is clearly not optimized. Thus, behavioral synthesis tools use advanced techniques to carry out the tasks of scheduling, allocation, and binding in order to optimize a circuit. They also typically include standard compiler optimizations that are applied before those tasks, such as constant propagation, dead-code elimination, and loop unrolling.

Hardware-Software Co-Simulation

More generally, a variety of simulation approaches exist, varying in their simulation speed and precision/accuracy. For a given processor, whether general-purpose or single-purpose,

simulation can vary from very detailed, like a gate-level model, to very abstract, like an instruction-level model. An instruction-level model of a general-purpose processor is known as an instruction-set simulator (ISS). An instruction-level model of a single-purpose processor is simply known as a system-level model. Lower-level simulations of either type of processor is usually done by creating a behavior, RT, or gate-level model in a hardware description language (HDL) environment. Because of the past separation of software design and hardware design, the simulation tools for each domain have evolved quite independently. The emphasis in software simulation has been on ISSs. The emphasis of hardware simulation has been models in hardware description languages (HDLs).

The integration of general-purpose and single-purpose processors onto a single IC has increased the need for an integrated method for simultaneously simulating these different types of processors. Thus, there is much interest in merging previously distinct software and hardware simulation tools.

One simple but naive form of integration is to create an HDL model of the microprocessor that will run the software of a system, and then integrate that model with the HDL models of the remaining single-purpose processors. While straightforward to implement, simulating a microprocessor in an HDL has two key disadvantages. First, this approach will be much slower than an ISS, since the HDL simulator represents an extra layer of software that must be executed. Second, such an approach ignores the fact that ISSs have excellent controllability and observability features that designers have become accustomed to.

As it turns out, in many embedded systems, those processors do have frequent communication. Therefore, modern hardware-software co-simulators do more than just integrate two simulators. They also seek to minimize the communication between those simulators. Consider, for example, a system having one microprocessor, one single-purpose processor representing a coprocessor, and one memory, all connected using a single shared bus. Suppose the microprocessor's program is stored in this memory, and that the coprocessor uses the memory extensively also. We can simulate the microprocessor using an ISS and the coprocessor using an HDL. But where should the shared memory be modeled, in the ISS or the HDL? If in the HDL, then on every instruction, the ISS will need to stall in order to communicate with the HDL simulator to fetch the next instruction from memory. If in the ISS, then the HDL simulator will need to stall in order interrupt the ISS for access to the memory. However, note that most of these stalls are probably not necessary. For example, the ISS accesses of its instructions in memory are really irrelevant to the coprocessor. Likewise, the coprocessor's manipulation of data in memory is not relevant to the microprocessor, except in cases where that data is being transferred between the processors using the memory.

In order to minimize this communication, we can model the memory in both the ISS and the HDL simulator. Each simulator can use its own copy of the memory without bothering the other simulator most of the time. The co-simulator must ensure that the memories remain consistent and that shared data does get communicated properly. Co-simulators using this speedup technique exhibit much faster performance, with some reports indicating a factor of 100 or more.

Reuse: Intellectual Property Cores

Designers have always had at their disposal commercial off-the-shelf (COTS) components. which they could purchase and use in building a given system. Using such predesigned and prepackaged ICs, each implementing general-purpose or single-purpose processors. greatly reduced design and debug time, as compared to building all system components from scratch.

As discussed in Chapter 1, the trend of growing IC capacities is leading to all the components of a system being implemented on a single chip, known as a system-on-a-chip (SOC). This trend, therefore, is leading to a major change in the distribution of such off-the-shelf components. Rather than being sold as ICs, such components are increasingly

being sold in the form of intellectual property, or IP. Specifically, they are sold as behavioral, structural or physical descriptions, rather than actual ICs. A designer can integrate those descriptions with other descriptions to form one large SOC description that can then be fabricated into a new IC.

Processor-level components that are available in the form of IP are known as *cores*. Initially, the term core referred only to microprocessors, but now is used for nearly any general-purpose or single-purpose processor IP component.

Hard, soft and firm cores

Cores come in three forms:

- A *soft core* is a synthesizable behavioral description of a component, typically written in a hardware description language (HDL) like VHDL or Verilog.
- A *firm core* is a structural description of a component, again typically provided in an HDL.
- A *hard core* is physical description, provided in any of a variety of physical layout file formats.

Note that the three forms of cores, namely, soft, firm, and hard, correspond to the three axes in Gajski's Y-chart in Figure 11.4.

A hard core has the advantages of ease of use and predictability. Since the core developer has already designed and tested the core, the core can be used right away and can be expected to work correctly. Furthermore, the size, power and performance of the core can be predicted quite accurately. However, a hard core is specific to a particular IC process, and thus cannot be easily mapped to a different process. For example, a hard core *A* may be available for IC vendor *X*'s 0.25 micrometer CMOS process. If a designer wishes to use vendor *X*'s 0.18 micrometer process, or wishes to use vendor *Y*, then the hard core *A* cannot be used.

On the other hard, a soft core has the advantages of retargeting and optimization potential. A hard core must be designed using a particular IC technology, and thus can't be used in a different technology. In contrast, a soft core can be synthesized (targeted) to nearly any technology, as long as the user has access to the synthesis and physical design tools for the desired technology. Furthermore, a designer can modify the behavior to be optimized for a particular use — for example, deleting unused functions of the core — resulting in lower-power and smaller designs. But, soft cores obviously require more design effort. and may not work properly in a technology for which it has never been tested. Furthermore, a soft core will likely not be as optimized as a hard core for the same processor, since hard cores typically have been given much more design attention.

Firm cores are a compromise between soft and hard cores, providing some retargetability and some limited optimization, but also providing better predictability and ease of use.

Design Process Models

A designer must proceed through several steps when designing a system. We can think of describing behavior as one design step, converting behavior to structure as another step, and mapping structure to a physical implementation as another step. Each step will obviously consist of numerous substeps. A design process model describes the order in which these steps are taken. The term *process* here should not be confused with the notion of a process in the concurrent process model discussed in an earlier chapter, nor should it be confused with the IC manufacturing process. Here, process refers to the manner in which the embedded system designer proceeds through design steps.

One process model is the waterfall model, illustrated in Figure 11.9(a). Suppose a designer has six months to build a system. In the waterfall model, the designer first exerts extensive effort, perhaps two months, describing the behavior completely. Once fully satisfied that the behavior is correct, after extensive behavioral simulation and debugging, the designer moves on to the next step of designing structure. Again, much effort is exerted, perhaps another two months, until the designer is satisfied the structure is correct. Finally, the physical implementation step is carried out, occupying perhaps the last two months. The result is a final system implementation, hopefully a correct one. In the waterfall model, when we proceed to the next step, we never come back to the earlier steps, much like water cascading down a mountain doesn't return to higher elevations.

Unfortunately, the waterfall model is not very realistic, for several reasons. First, we will almost always find bugs in the later steps that should be fixed in an earlier step. For example, when testing the structure, we may notice that we forgot to handle a certain input combination in the behavior. Second, we often do not know the complete desired behavior of the system until we have a working prototype. For example, we may build a prototype device and show it to a customer, who then gets the idea of adding several features. Third, system specifications commonly change unexpectedly. For example, we may be halfway done designing a system when our company decides that to be competitive, the product must be smaller and consume less power than originally expected, requiring several features to be dropped. Nevertheless, many designers design their systems following the waterfall model. The accompanying unexpected iterations back through the three steps often result in missed deadlines, and hence in lost revenues or products that never make it to market.

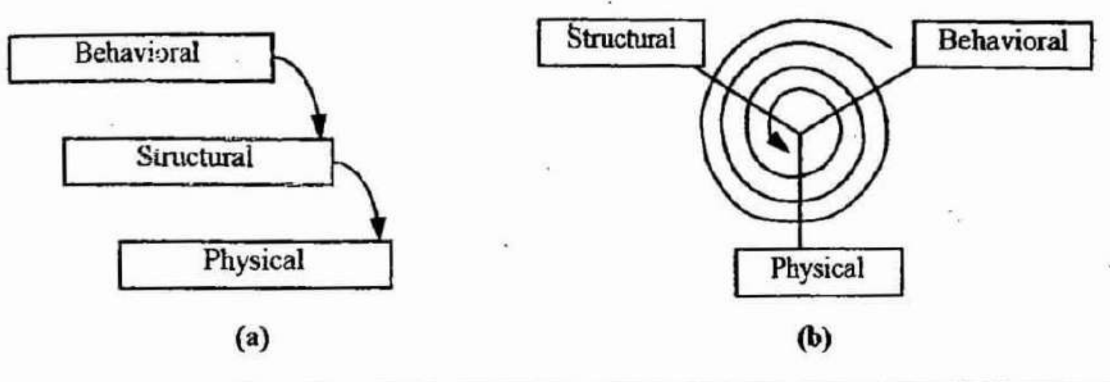


Figure 11.9: Design process models: (a) waterfall, (b) spiral.

An alternative process model is the spiral model, shown in Figure 11.9(b). Suppose again that the designer has six months to build the system. In the spiral model, the designer first exerts some effort to describe the basic behavior of the system, perhaps a few weeks. This description will be incomplete, but have the basic functions, with many functions left to be filled in later. Next, the designer moves on to designing structure, again taking maybe a few weeks. Finally, the designer creates a physical prototype of the system. This prototype is used to test out the basic functions, and to get a better idea of what functions we should add to the system. With this experience, the designer proceeds to proceed through the three steps again, expanding the original behavioral description or even starting with a new one, creating structure, and obtaining a physical implementation again. These steps may be repeated several times until the desired system is obtained.

The spiral model has its drawbacks, too. The designer must come up with ways to obtain structure and physical implementations quickly. For example, the designer may have to use FPGAs for the physical prototypes, finally generating new silicon (a task that can take months) for the final product. Thus, the designer may have to use more tools, which itself can require extra effort and costs. Also, if a system was well defined in the beginning and if we would have created a first-time correct implementation using the waterfall model, then the spiral model requires more time due to the overhead of creating numerous prototypes. Nevertheless, variations of the spiral model have become extremely popular, both in software development as well as hardware development.

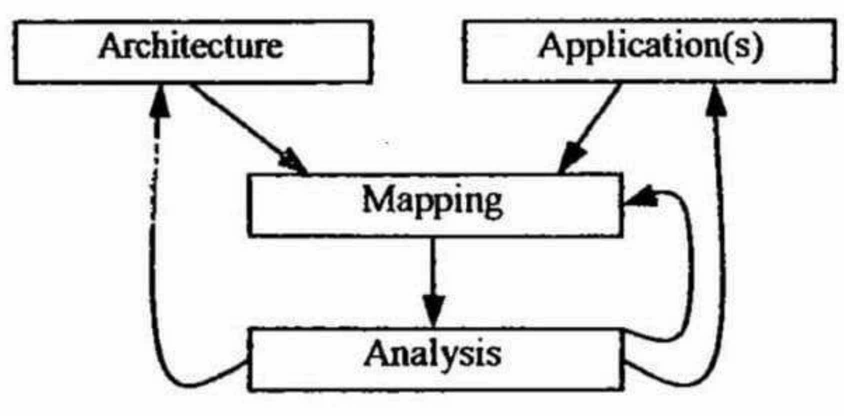
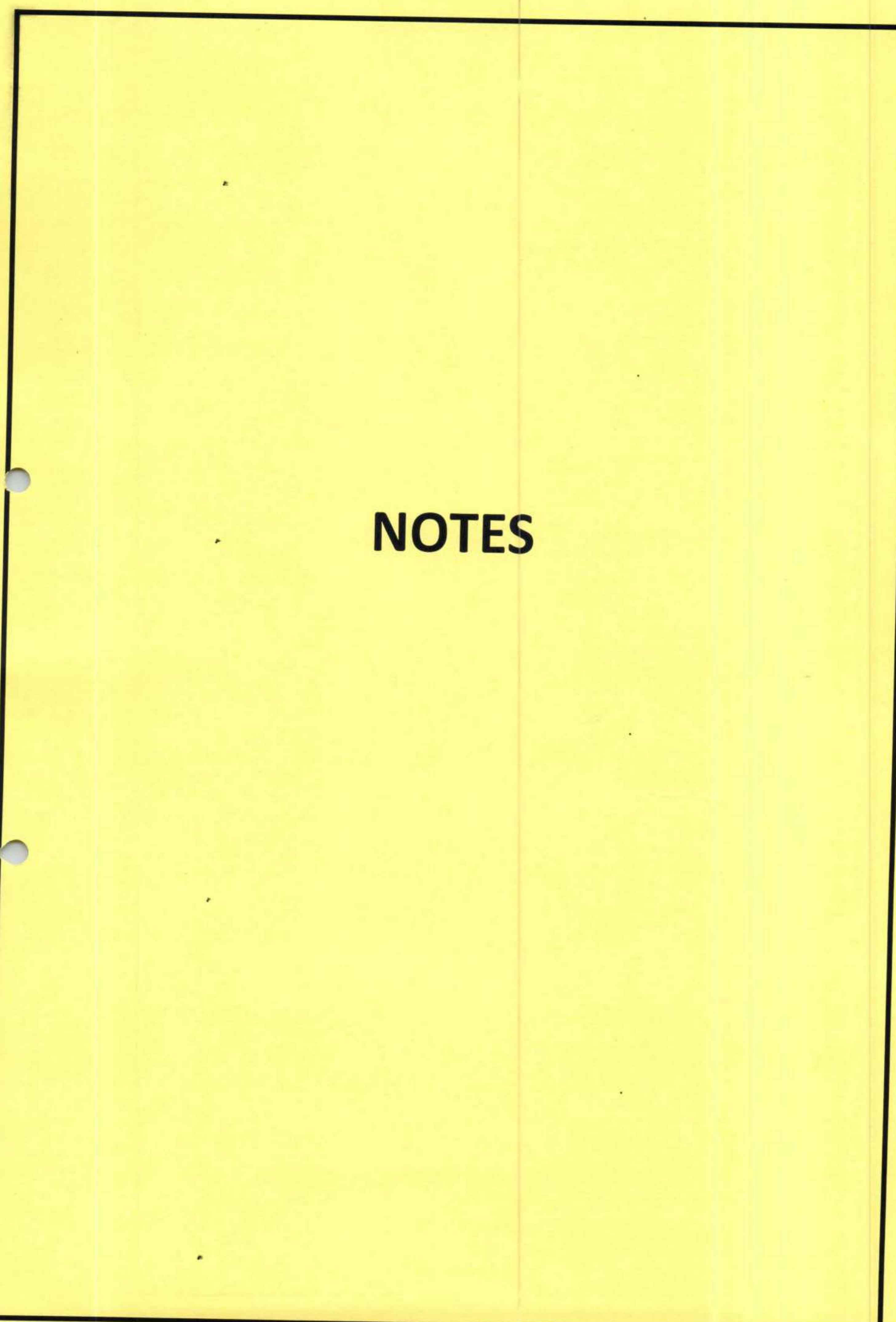


Figure 11.10: A spiral-like approach represented using another Y-chart.

However, even this widely accepted approach is beginning to change. A spiral-like process model, illustrated in Figure 11.10, is beginning to be applied by embedded system designers. In this model, the designer develops or acquires an architecture, and develops an application or set of applications. The designer then maps the application to the architecture, and analyzes the design metrics of this combination of application, architecture and mapping. The designer can then choose to (a) modify the mapping, (b) modify the application to better suit the architecture, or (c) modify the architecture to better suit the application. This last step of modifying the architecture was previously too difficult to consider. However, with the maturation of synthesis tools as well as compilers that can generate code for a variety of instruction sets, this last step is much more feasible. Furthermore, as mentioned above, designers are increasingly obtaining the microprocessor architecture in the form of intellectual property, which can thus be potentially be tuned to the application. This is in stark contrast to the past, when an obtained microprocessor IC obviously could not be modified. By coincidence, the depiction in Figure 11.10 of this process model is referred to as the *Y-chart*, but has no relation with Gajski's Y-chart described earlier.

Refining to lower abstraction levels (whether behavioral, structural, or physical models) narrows the potential implementations, as illustrated in Figure 11.3(b). Such narrowing proceeds until a particular implementation is chosen.



NOTES

30.3.21

1. INTRODUCTION

contents:

* Embedded system overview
* Design challenges
* Processor Technology
* IC Technology
* Design Technology
* Trade-offs

Embedded system overview:

System:

→ A system is a way of working or organizing one or many tasks at set of rules.

→ A system is an arrangement of many units.

Eg: Microprocessor, memory [primary memory (RAM, ROM, caches etc), Secondary memory (Magnetic memory, tapes, optical memory etc)], Input, output, Networking units.

Embedded System:

→ Embedded system is nothing but it's like a computer hardware with software program embedded in it.

→ The most important is to operate in real time. It is called real-time system.

Constraints:

→ An embedded system has hardware design with software program to keep in view the following three constraints are to be taken into consideration.

1. Available system memory
2. Available processor speed
3. power dissipation

Examples of Embedded system:

* Smart phone
* Digital camera
* Automatic washing machine
* ATM etc
* Traffic light system etc

Applications of Embedded system:

(i) Mobile computing
(ii) Banking
(iii) Defense
(iv) Automobile
(v) Networking systems
(vi) Robotics
(vii) communication.

Characteristics of Embedded/Real Time systems:

1. Single functioned Embedded system
2. Tightly constrained Embedded system
3. Reactive and Real time Embedded system

Single functioned Embedded system:

An embedded system usually executes a specific program repeatedly.

case(i): when there are applications like videogame, the embedded system operates as single functioned.

case(ii): A missile targeting the opponent and locking its range is the embedded system operated at single function.

Tightly constrained Embedded system:

Thus, the embedded system categorised in tightly constrained is a measure of implementation with design metric considering features such as cost, size, performance and power.

Eg: The design which is made for low cost, size fit into single chip, fast process and consume less power.

Reactive and Real Time Embedded System:

Embedded System continuously reacts to changes in real time and gives the result without delay.

Eg: Automobiles (car), Navigation System (RADAR)

Classification of Embedded/Real Time Systems:

1. Small Scale systems
2. Medium Scale systems
3. Sophisticated Systems

Small Scale Embedded Systems:

* The embedded system designed with a single 8 or 16 bit microcontrollers.
* They have little hardware and simple software program.
* The small scale embedded system need to limit the power dissipation and has to fit within the memory.

Medium Scale Embedded systems:

* The embedded System designed with a single or few 16 to 32 bit microcontrollers such as DSPs.
* They have both hardware and software complex.
* These systems are used to employ ASSP (Application Specific system programming) and use IP.

Sophisticated Embedded Systems:

* They have enormous hardware and software.
* These embedded systems are configured using PLAs.
* These Embedded systems use TCP/IP protocol.
* These are used in big machinaries and manufacturing factories etc.
* These should have additional speed and backup.

31-3-21

Design challenges:

The embedded system designer must construct an implementation that fulfills desired functionality.

The following are the design metrics of embedded systems.

1. NRE cost (Non recurring Engineering cost)
2. Unit cost
3. Size
4. Performance
5. Flexibility
6. Time to market
7. Maintainance
8. Safety

Mainly in the designmetrics the three things will worse. Eg: size reduction, Adopting to new technology, familiarization with new hardware and software.

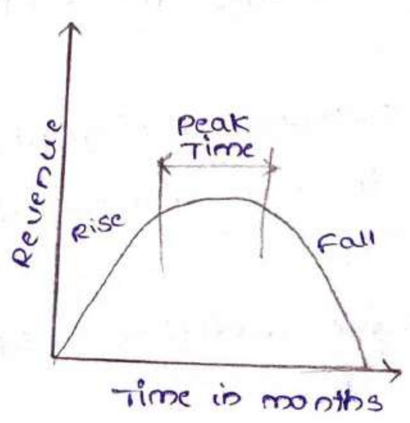
→ If we reduce the size the performance may suffer.

→ To meet the optimization challenges the designer must make all the units comfortable and reliable to hardware and software technologies.

→ the designer must also think to migrate from one technology to another technology for best implementation.

Market Scale:

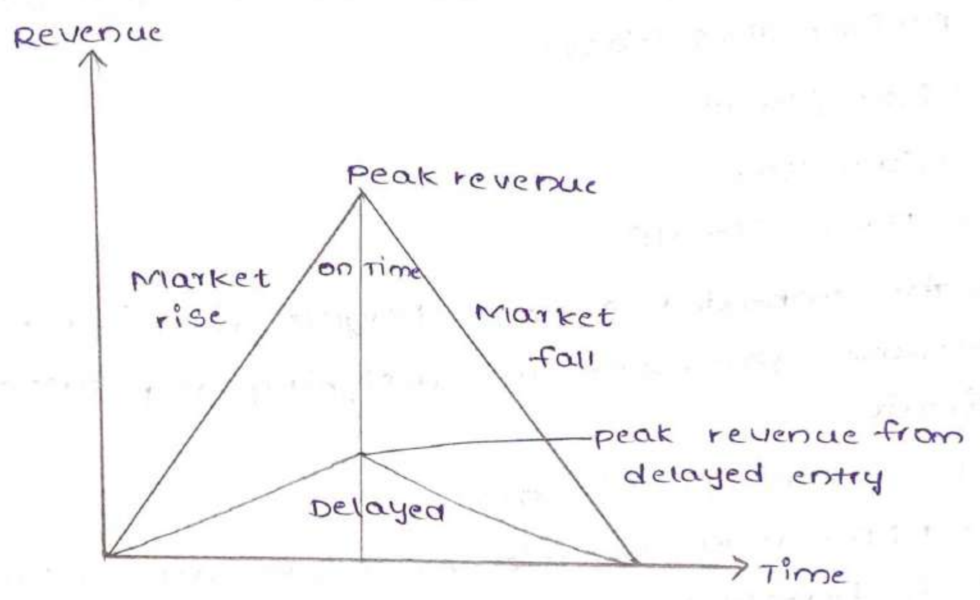
The time to market designmetric is the market scale of graph time in months V/S revenue.



Thus the above figure shows the simple product window.

→ The product would have highest sales in the peak time.

Modified market graph:



Processor Technology:

Processor technology relates to the architecture of embedded system design for desired functionality.

1. General purpose processors.
2. Single purpose processors
3. Application specific processors

General purpose processors:

→ General purpose processors are related to software that means these are programmable

→ The following specifications are considered.

(i) program Memory
(ii) Data path
(iii) ALU

→ An embedded system designer uses a general purpose processor by programming processor memory to carry out required functionality.

Single purpose processors:

→ They are meant for hardware design mainly a digital circuit designed to execute exactly one program.

→ The following are specified in single purpose processors.

(i) Performance (fast)

(ii) Size (Small)

(iii) Cost (Low)

(iv) power (Small)

→ The embedded system designer make use of single purpose processor for designing any custom digital circuit.

Application Specific processors:

→ Application specific processors are meant for both hardware and software involvement having the characteristics such as controlling, signal processing communication etc.

→ This involves

(i) Performance (Good)

(ii) Flexibility

(iii) ASIP (Microcontroller or DSP)

→ The embedded system designer using application specific processors are oriented to business class.

Differences between single purpose,

IC Technology:

In Embedded system design the following IC Technologies are used for implementation.

(i) Full Custom/VLSI

(ii) Semi Custom ASIC

(iii) PLD

5·4½'

Full custom/VLSI :

→ In this IC Technology all the layers are optimised for an embedded system for particular digital implementation.

→ The benefits of full custom / VLSI IC Technology is excellent performance, small size, low power.

→ considerations in developing Full custom/VLSI is

* Placing transistors
* Sizing transistors
* Routing wires

→ The drawbacks are

* cost
* Less life span

Semi custom :

→ Semi custom IC technology make use of gate arrays in embedded system design.

→ These are partially built or fully with low layers.

→ The benefits are ;

* Good performance
* Good size
* Less cost

→ The drawback is takes more time to develop.

PLD :

→ In this all layers exist with connections created to be implemented for desired functionality.

→ In this the very popular IC technology is FPGA (Field programmable Gate Array)

→ The advantages are

* Low cost
* can be made in less time

→ The drawbacks are

* Size is bigger
* Power hungry
* Slower in performance

Design Technology:

Design technology involves in which the concept of desired system with functionality to an implementation.

→ In this design technology the following are should be taken into consideration.

1. System specification
2. Behavioural specification
3. Register transfer specification
4. Logic specification

6-4-21

System specification:

The designer describes the desired functionality in a language preferably executable language with some specifications of design is known as system specification.

Behavioural specification:

The designer refines the specifications and distributing portions to work accordingly by means of processor and program embedded is known as Behavioural specification.

Register Transfer Specification:

converting the behavioural to assembly code for operation and communicating the components according to state machines represents the Register Transfer Specification.

Logic Specification:

The designer refines the register transfer specification to boolean equations representing the operation mode Done is defined by logic specification.

Trade-offs:

The choice of hardware and software for a particu-
-lar function is defined by Trade-off.

→ Thus Trade-off depends on various design metrics.

7.4.21

→ Mainly Trade off depends on processor.

→ A processor is a digital circuit designed to per-
-form computation task.

→ A processor consists of data path and controller.

→ Thus the following techniques are used in designing.

(i) combinational logic

(ii) Sequential logic

(iii) custom single purpose processor design technique.

(iv) Register transfer level custom single purpose proce-
-ssor design

(V) optimizing custom single purpose processor design

Combinational logic:

→ combinational logic involves transistors, logic gates
are used.

→ Here basic programming is involved in combinational
logic design.

→ Thus RT level is implemented to components.

→ Thus the transistors may be simple or belongs to
MOS Technology.

Eg: CMOS

→ Thus logic gates involved are represented in digital.

→ In addition to this the required components are
multiplexer, decoder, adder, comparator, ALU.

Sequential logic:

→ The components involved in Sequential logic are
flipflops.

→ Here also RT level execution is involved.

→ A intermediate level of programming is used.

→ The additional components used are registers, counters
and shift register.

8·4·21

Custom single purpose processor design:

→ A basic processor can be built with controller and data path.

→ A controller defines the configuration to data path.

→ In this there will be registers, multiplexers, signals, functional with connection units.

→ Registers are used to load the data.

→ Multiplexer is used to select the signals.

→ Signals are meant for carrier data.

→ Functional units are the basic building blocks used for operation.

→ Connection units are defined for working the process.

→ Here high level programming is used for implementation.

→ We can apply combinational as well as sequential logic design techniques to built a controller and data path.

RT level custom single purpose processor design:

→ The basic technique converting the sequential program into a custom single purpose processor is defined by register transfer (RT) level custom single purpose processor design technique.

→ Here FSM controlling is used.

→ FSM stands for Finite State Machine.

9·4·21

→ Thus here ~~opti~~ RT level custom single purpose processor design uses sequential technique design consists of registers, multiplexers and flip flops and also used the custom single purpose process design technique.

→ Thus here high level programming is used for implementation.

optimizing custom single purpose processor design:

→ optimization is the task of making design metric values the best possible.

→ optimization involves the following.

(i) Original code (Suitable programming is used for implementation)

(ii) FSMD (Data relevant finite state Machine)

(iii) optimized data path and controller

→ Thus in this technique all the algorithms are more efficient.

→ These algorithms reduce the time complexity.

→ Thus here the FSMD used merges the relevant and reduces the algorithm.

→ The optimizing the data path and controller involves in three main tasks such as

* Scheduling
* Allocation
* Binding

→ Scheduling refers to make the program by ready.

→ Allocation refers to RT components use.

→ Binding refers to FSMD operations.

→ Thus FSMD operations involves two optimization techniques related to FSM. They are

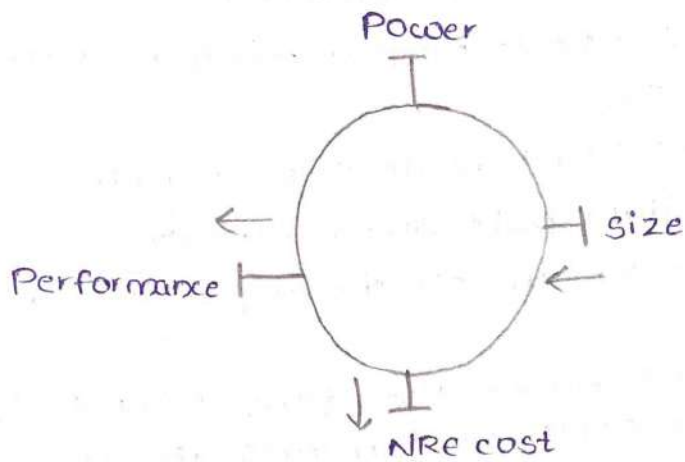
1. State encoding
2. State minimization

→ State encoding is the task of assigning a unit to each state in FSM.

→ State minimization is the task of merging relevant States into a single state.

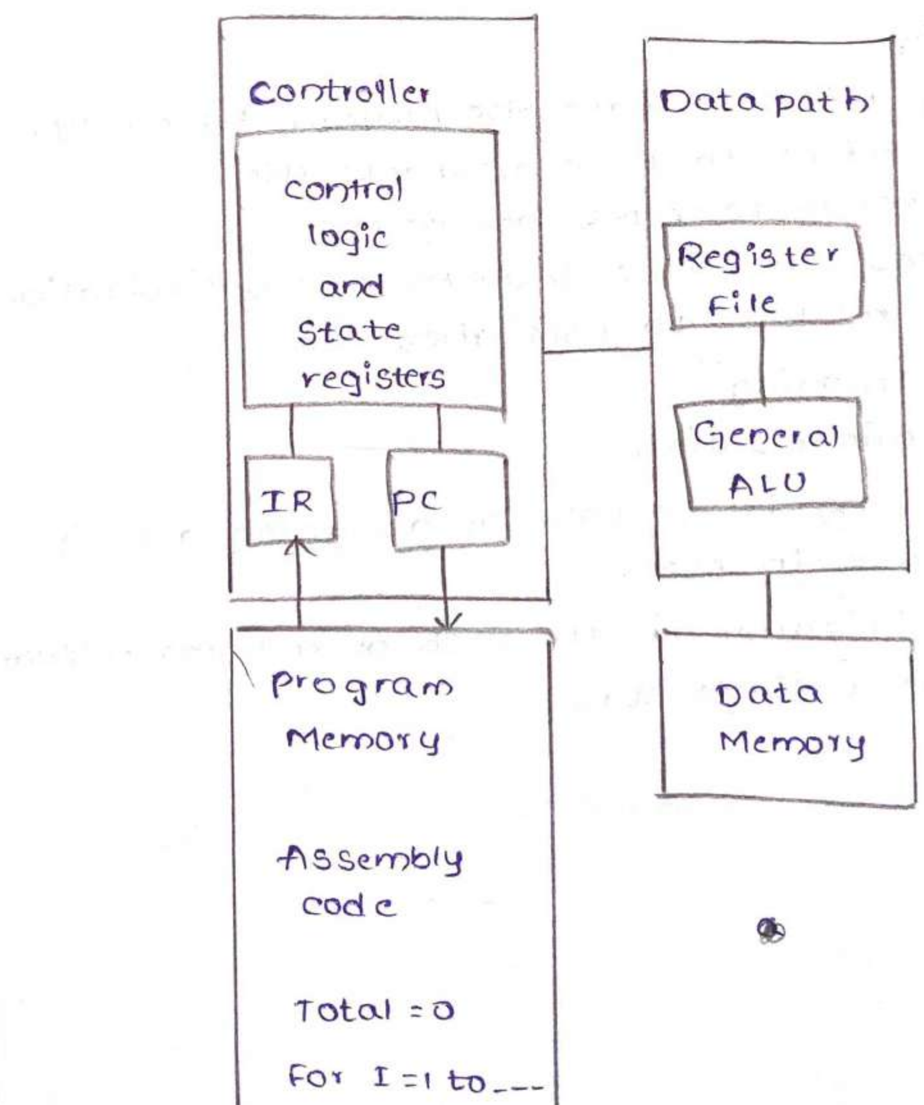
Diagrams:

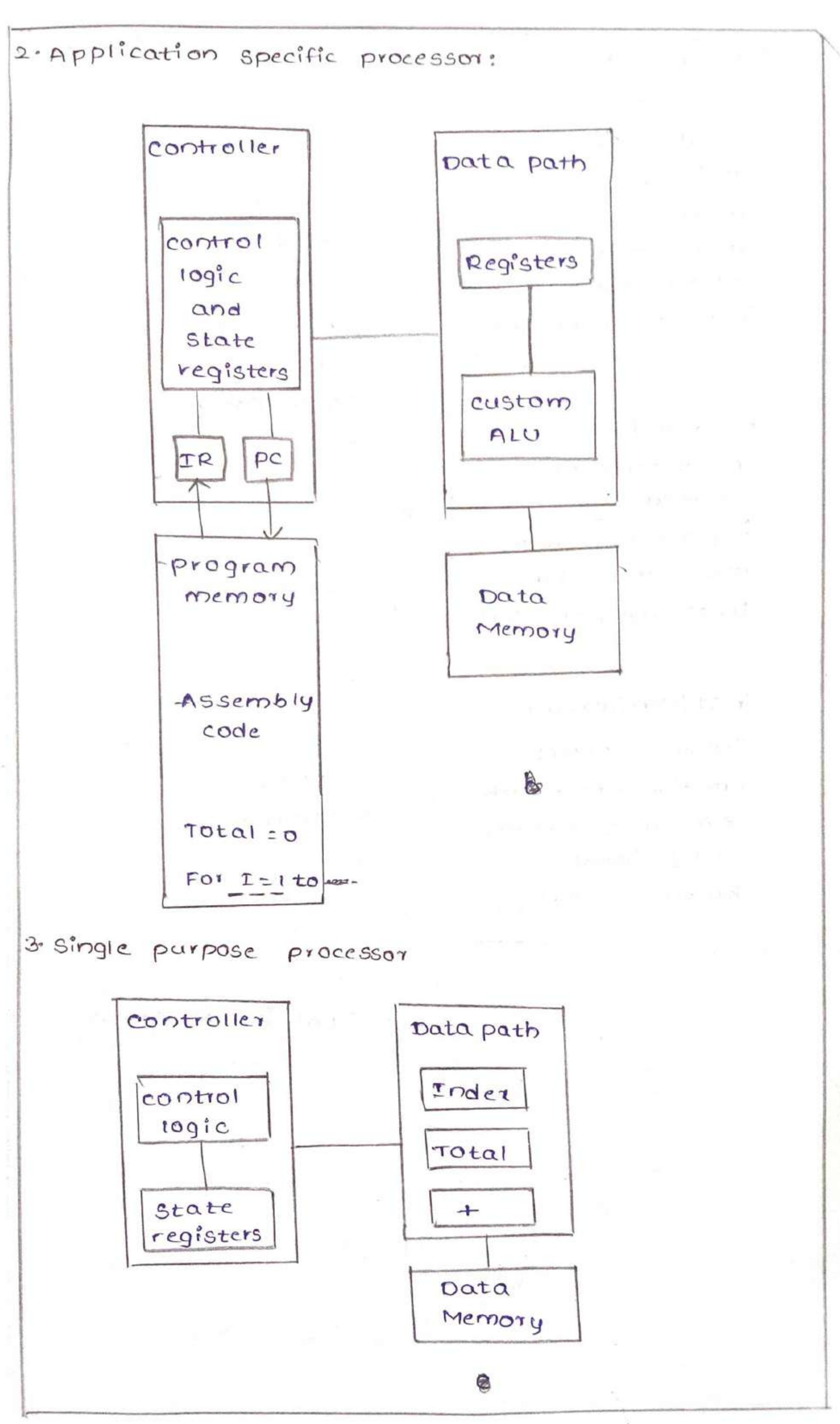
Design challenges:



Processor Technology:

1. General purpose processor:

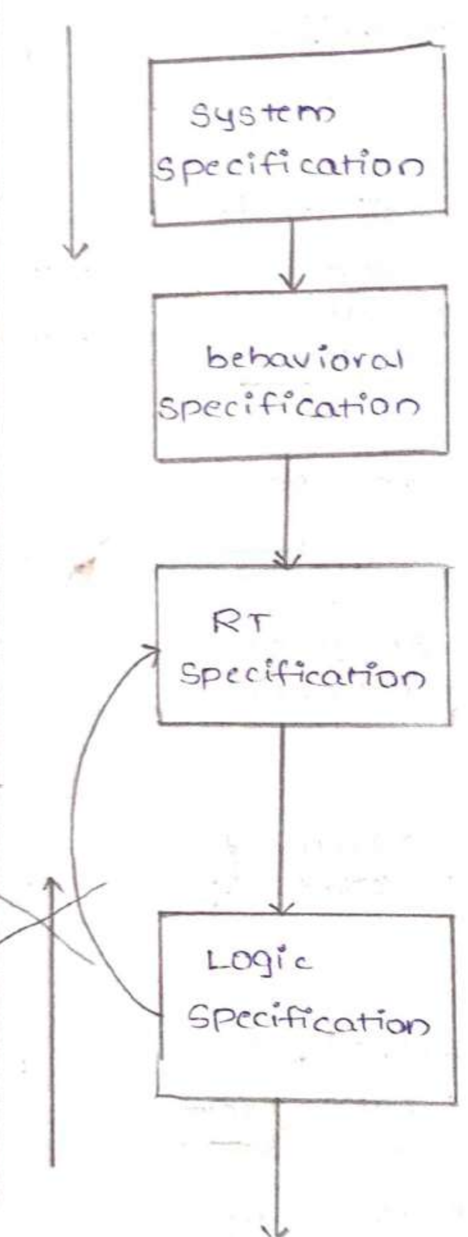



2. Application specific processor:
Controller
control logic and State registers
IR
PC
Data path
Registers
custom ALU
program memory
Assembly code
Total = 0
For I = 1 to
Data Memory
3. Single purpose processor
Controller
control logic
State registers
Data path
Index
Total
+
Data Memory

10-4-21

Design Technology:

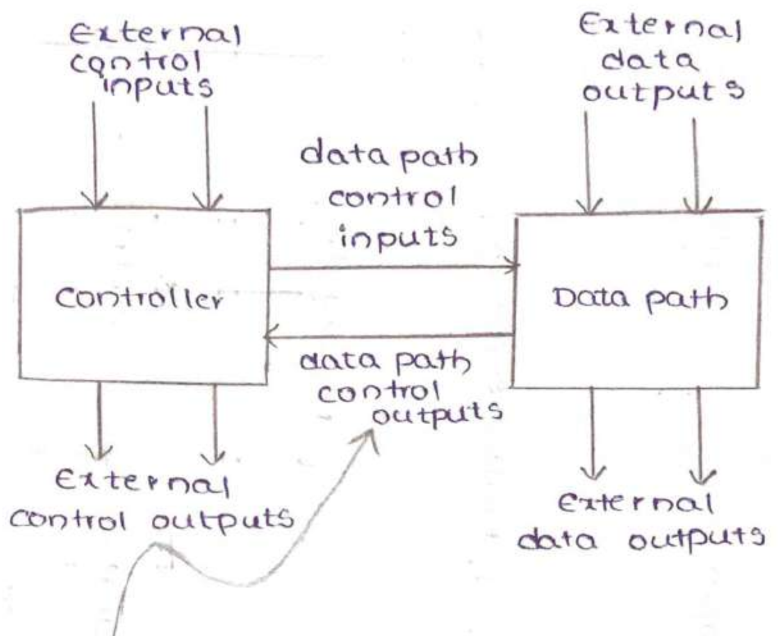
compilation/synthesis: Automates exploration of implementation details for lower level
Libraries/IP: incorporates pre designed implementation from lower abstraction level into higher level
Test/verification: Ensures correct functionality at each level thus reducing costly iterations between levels.



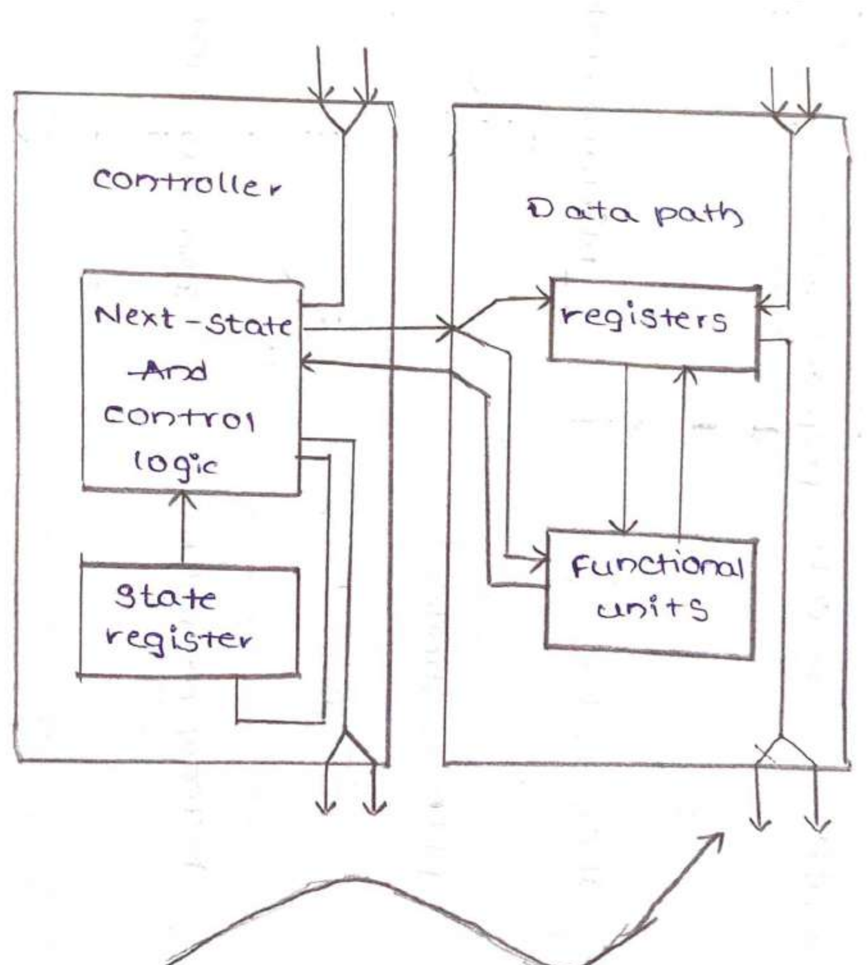
To final implementation

Trade - offs :

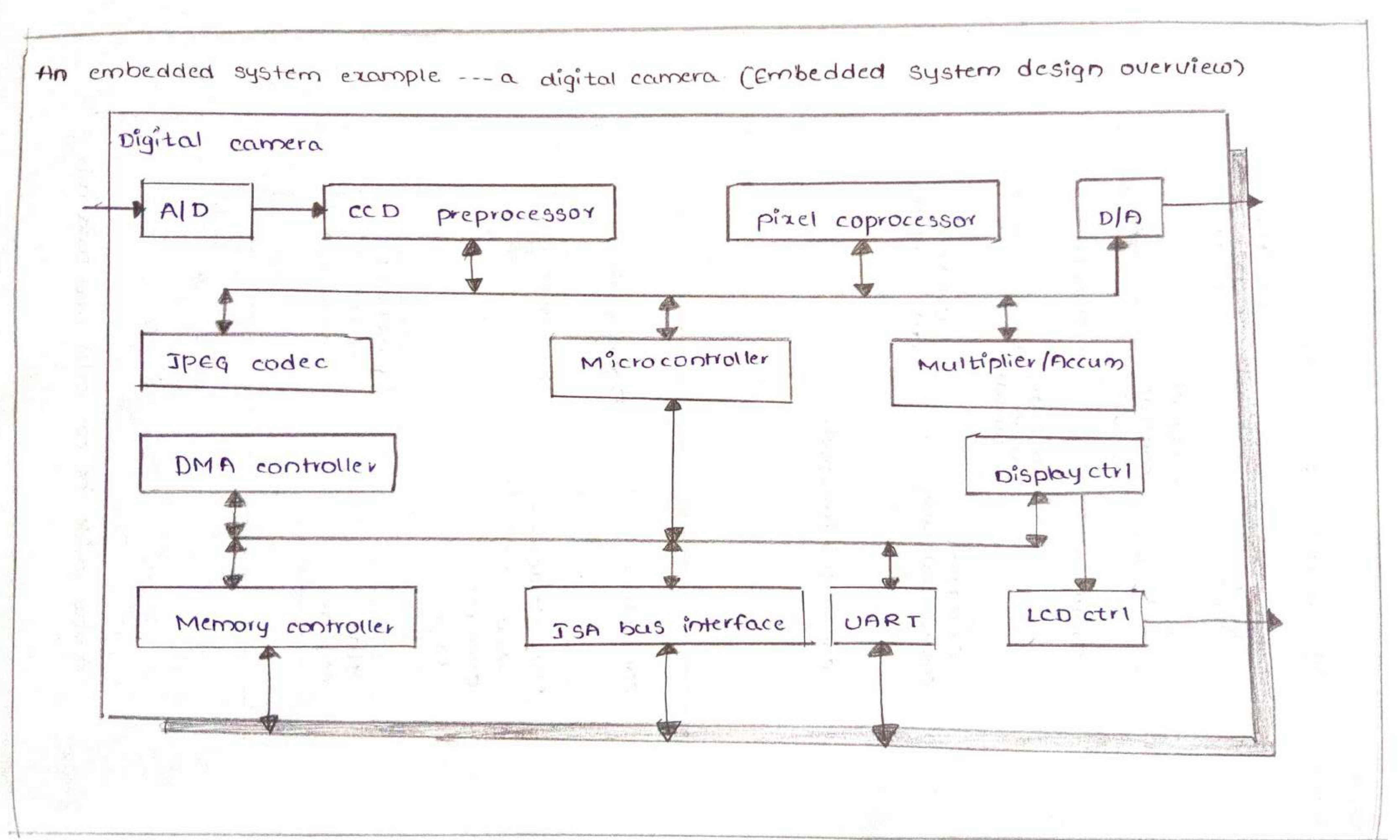
Custom single purpose processor :



controller & Data path.



A view inside the controller and Data path


An embedded system example --- a digital camera (Embedded system design overview)
Digital camera
A/D
CCD preprocessor
pixel coprocessor
D/A
JPEG codec
Microcontroller
Multiplier/Accum
DMA controller
Display ctrl
Memory controller
ISA bus interface
UART
LCD ctrl

15.4.21

Trade-offs:

→ Thus hardware and software design technologies were very different in embedded system design.

→ There will be different variations in view of hardware and software.

→ The particular hardware and the correct software used for embedded system design is represented by Trade-off.

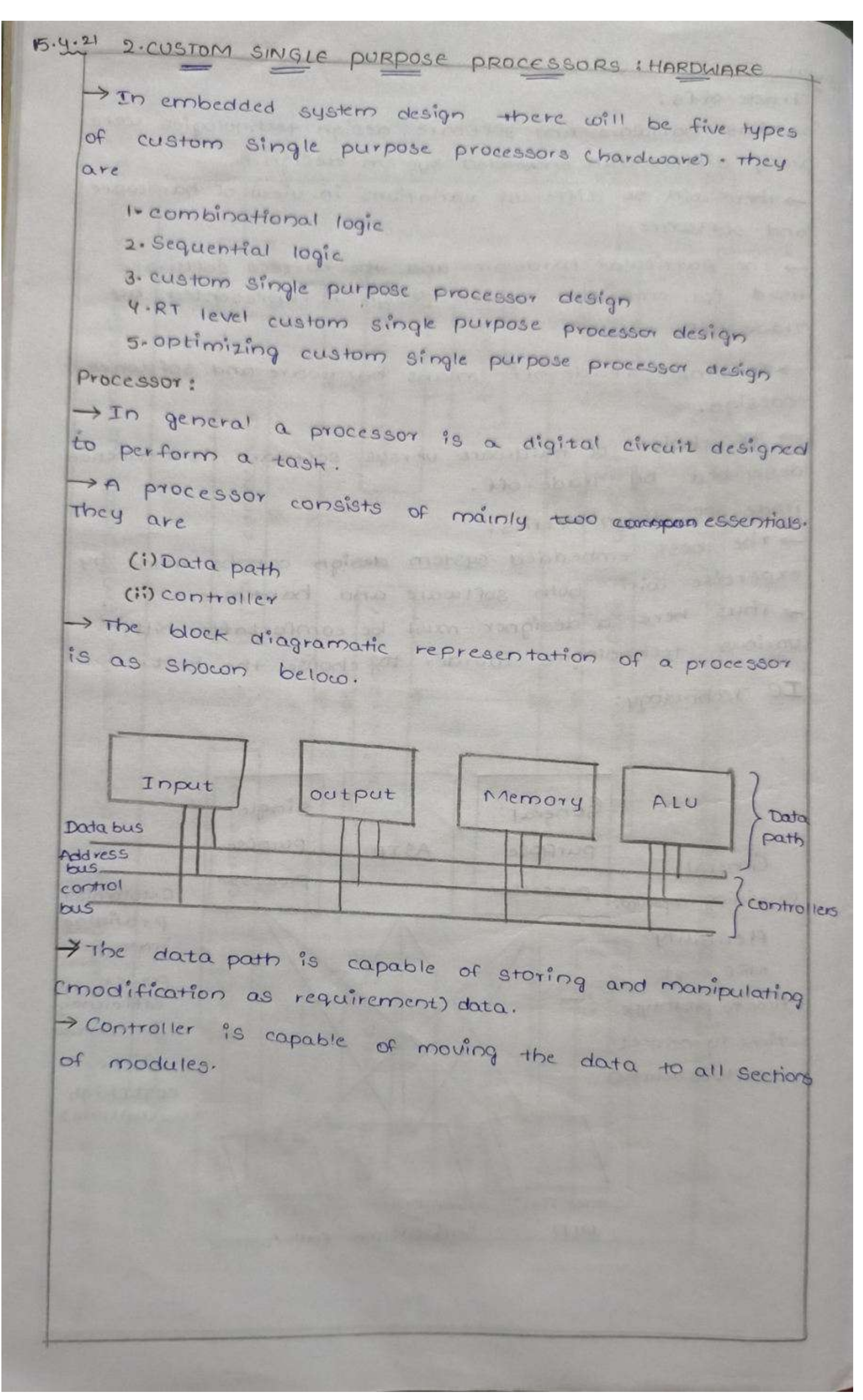
→ In simple, Trade-off means hardware and software codesign.

→ The choice of hardware versus software is a function described by Trade-off.

Trade-off design Metrics:

→ The best embedded system design can be made by expertise with both software and hardware.

→ Thus here a designer must be comfortable with various technologies inorder to choose the best.



15.4.21 **2. CUSTOM SINGLE PURPOSE PROCESSORS : HARDWARE**

→ In embedded system design there will be five types of custom single purpose processors (hardware). They are

1. combinational logic
2. Sequential logic
3. custom single purpose processor design
4. RT level custom single purpose processor design
5. optimizing custom single purpose processor design

Processor:

→ In general a processor is a digital circuit designed to perform a task.

→ A processor consists of mainly two ~~compon~~ essentials. They are

(i) Data path
(ii) controller

→ The block diagramatic representation of a processor is as shown below.

→ The data path is capable of storing and manipulating (modification as requirement) data.

→ Controller is capable of moving the data to all sections of modules.

Single purpose processor:

→ A single purpose processor is designed specially to carry out a particular task.

→ An embedded system designer may choose a custom (accordingly) to single purpose processor for implementation of task.

→ The hardware requirement will be as per the standard specifications in any of these following types.

(i) combinational logic

(ii) Sequential logic

(iii) custom single purpose processor design logic

(iv) RT level custom single purpose processor design logic

(v) optimizing custom single purpose processor design logic

Combinational logic:

→ In combinational logic design, the basic hardware is implemented using transistors and logic gates.

(i) Transistors:

→ The transistor is a basic electrical component in a digital system with a simple operation of ON or OFF switch.

→ In the combinational logic design the transistors mainly used are CMOS.

→ CMOS stands for crystalline Metal oxide semiconductor.

Logic gates:

→ In digital system logic gates are the basic building blocks for any function.

→ In combinational logic design the following types of gates are used.

* Inverter
* AND gate
* NAND gate
* OR gate
* NOR gate
* XOR gate
* XNOR gate

→ In addition to transistors and logic gates, multiplexer, decoder, adder, comparator, ALU are used in rare cases.

Sequential logic design:

→ In the sequential logic design implementation the main components used are flipflops.

Flipflop:

→ A flip flop is a digital circuit where the output depends on present as well as previous input.

16/4/21

→ Thus the flip flop stores the bit and reflect their behaviour as output.

→ In sequential logic design for the implementation of embedded systems the D-type flipflop and SR type Flipflop are used for implementation.

→ These flip flops are used to prevent unexpected behaviour. from signal glitches.

→ The other additional components used in sequential logic design are registers, shift registers and counters.

Shift register:

→ A shift register has 1 bit data and 2 control pulses (clock and shift).

→ Shift register is used to store n bits data that to be shifted into register serially.

Counter:

→ A counter is a register with input signals such as count, clock and clear.

→ Thus counter make use of increment (Add 1)

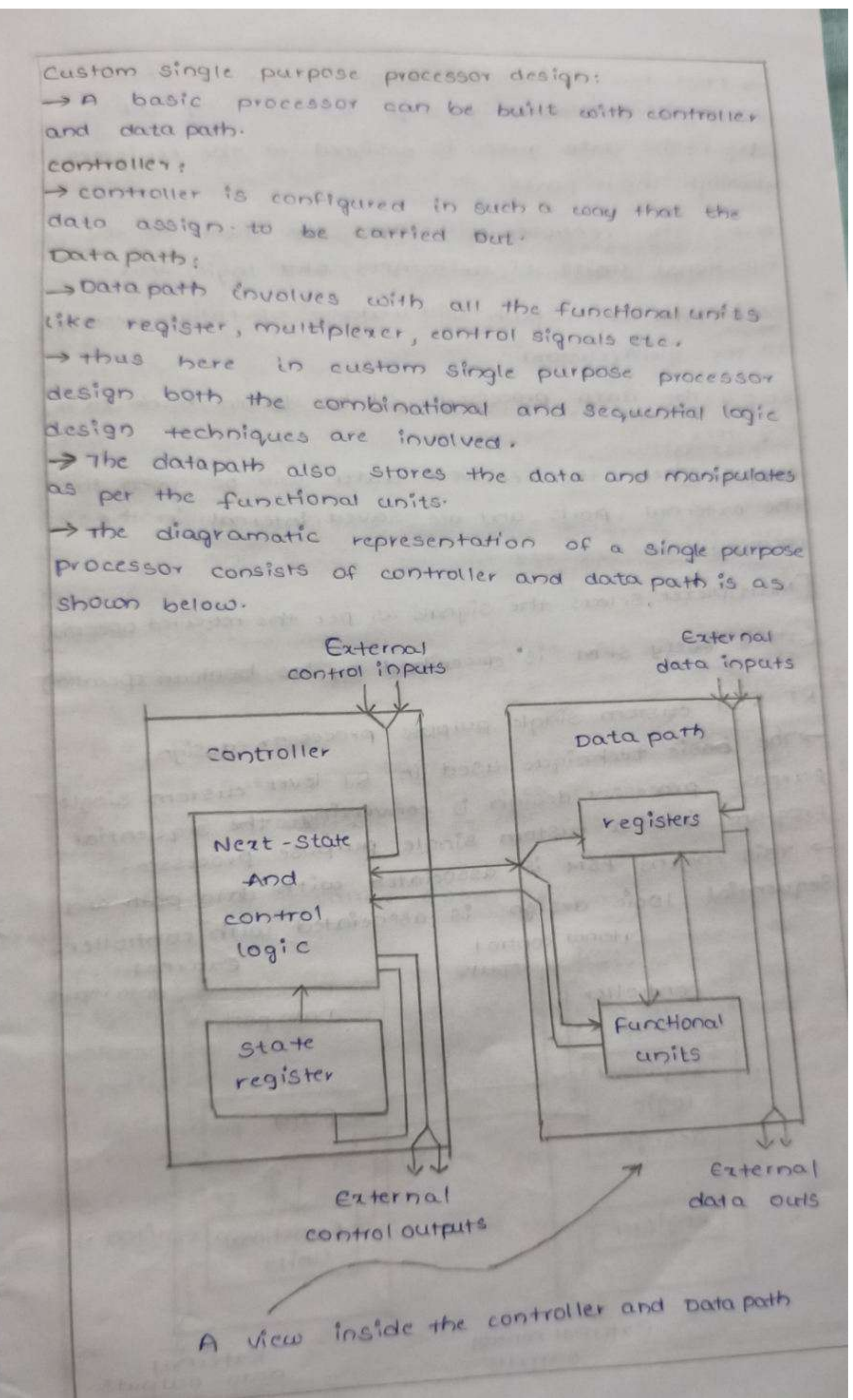
→ There are two types of counters according to the options. They are

(i) Upcount

(ii) Downcount

→ Upcount, add 1; whereas downcount decrement to 0

→ Thus sequential logic design involves in state diagram described by finite state machine (FSM).



Custom single purpose processor design:

→ A basic processor can be built with controller and data path.

controller:

→ controller is configured in such a way that the data assign to be carried out.

Datapath:

→ Data path involves with all the functional units like register, multiplexer, control signals etc.

→ thus here in custom single purpose processor design both the combinational and sequential logic design techniques are involved.

→ The datapath also stores the data and manipulates as per the functional units.

→ the diagramatic representation of a single purpose processor consists of controller and data path is as shown below.

A view inside the controller and Data path

→ Thus here the custom single purpose processor design is constructed through the following steps.

Step 1: The data given is assigned to the registers through input ports.

Step 2: The controller will takes the data to the functional units of arithmetic and logic unit.

Step 3: The data path will undergo operation according to the given (custom)

Step 4: The data processed will be taken back to the registers.

Step 5: There the received data will be given to the external ports and are saved internal in the memory.

[Multiplexer selects the signals as per the required operation]

(Thus every SFM is described by its boolean operation)

-4-21

RT level custom single purpose processor design:

→ The basic technique used in RT level custom single purpose processor design is converting the sequential program into a custom single purpose processor.

→ This means FSM is associated with data path and Sequential logic design is associated with controller.

→ Thus in the RT level custom single purpose processor design the following steps of implementation are involved. They are

Step 1: Input (8-bit) is given to the register by means of controller.
In this 8-bit first four bits represents lower order and next 4-bits represent higher order.
Thus here the complete 8-bit is given by data-in.
Step 2: The division of bits associated with lower order and higher order are given by rdy_in.
Step 3: The data given will be connected to the sequential logic design consists of registers, multiplexers and flip-flops.
Step 4: According to the sequential program they get operated and moves towards FSM approach through data path.
Step 5: Thus here multiplexer selects the signal in the form of pulse and distributes data to all functional units.
Step 6: The received data is get processed and given as output by rdy_out signal.
Here each transition in FSM is done by means of clock.
This method is called FSMD approach.

27-4-21

Optimizing custom single purpose processor design:
→ Optimization is the task of making design metric values the best possible.
→ optimization involves the following.
1. optimizing the original code
2. Optimizing the FSMD (Finite State Machine Data/Design)
3. optimizing the datapath
4. optimizing FSM (Finite state machine)

Optimizing the original code:

→ optimizing the original code can be analyzed and get reduced.

→ Here analyzation should be done in terms of algorithm.

→ By this complexity will be reduced that leads to some amount of time executed fast represents the embedded system more efficient.

28-4-21

Optimizing the FSMD:

→ One program was decided the convertes program into FSMD.

→ Here there will be many states which will be same, this likely to be merged into few states.

→ Here optimizing state FSMD merging can be done by process of Scheduling.

→ Scheduling is the task of assigning operation particularly

Optimizing the data path:

→ Thus this is a unique function where operations are categorized and distributed accordingly.

→ In this optimizing the data path the main tasks are scheduling, allocation and binding.

→ Scheduling refers to ready.

→ Allocation refers to allow and view.

→ Binding refers to complete operations allocated.

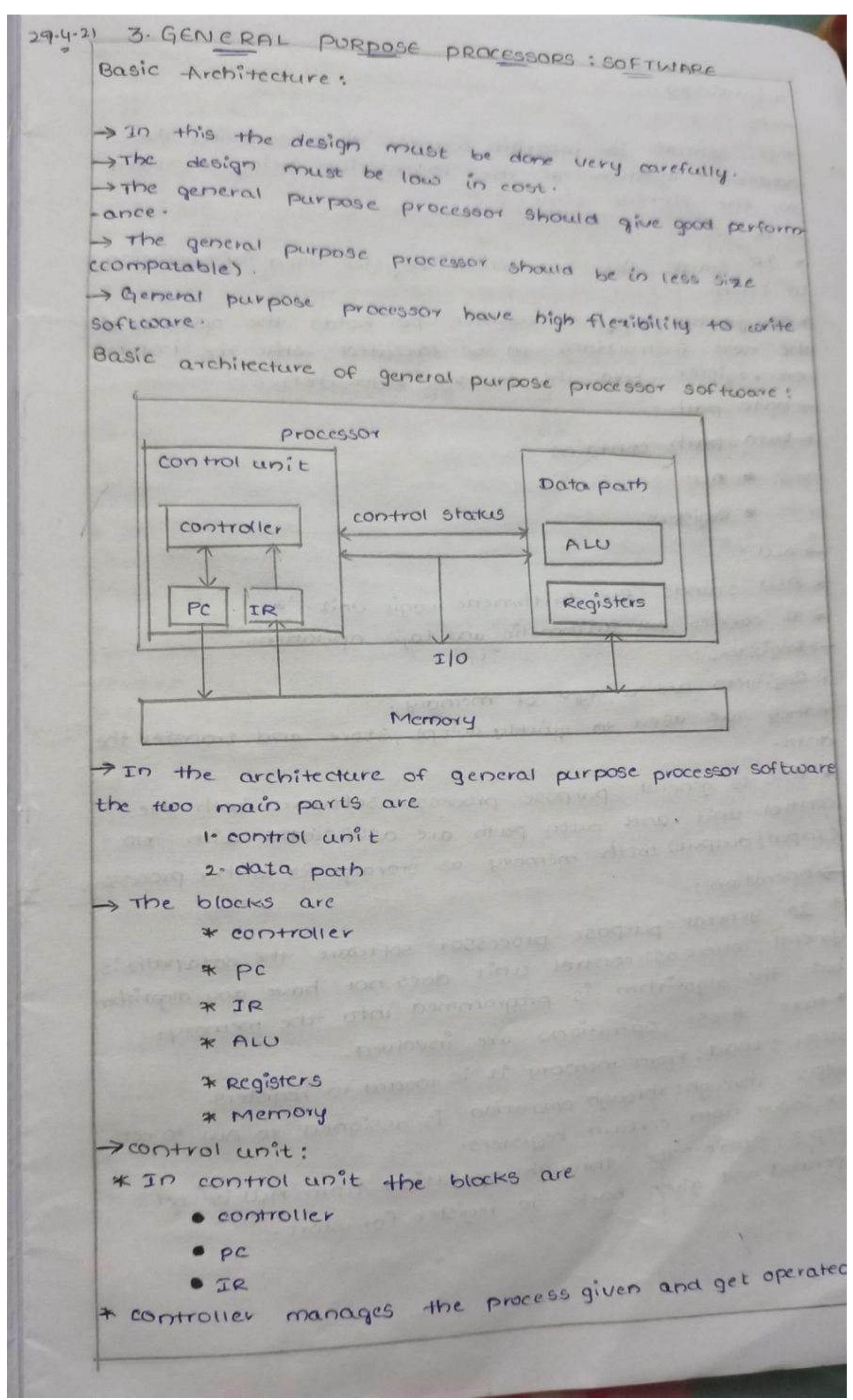
Optimizing the FSM:

→ In optimizing the FSM there will be two opportun-ities for optimization. They are

1. State encoding
2. State minimization

→ State encoding is the task of assigning bit pattern to each state in FSM.

→ State minimization is the task of merging equiva-lent state into a single state.



29-4-21 3. GENERAL PURPOSE PROCESSORS : SOFTWARE

Basic Architecture :

→ In this the design must be done very carefully.
→ The design must be low in cost.
→ The general purpose processor should give good performance.
→ The general purpose processor should be in less size (compatable).
→ General purpose processor have high flexibility to write software.

Basic architecture of general purpose processor software :

→ In the architecture of general purpose processor software the two main parts are

1. control unit
2. data path

→ The blocks are

* controller
* PC
* IR
* ALU
* Registers
* Memory

→ control unit :

* In control unit the blocks are
  - controller
  - PC
  - IR

* controller manages the process given and get operated

accordingly.

→ PC :

* PC stands for program counter that contains the address location of the instruction being executed at the current time.

→ IR :

* IR stands for instruction register used for operation to fetch.

* In general, program counter PC holds the address of the next instructions to be executed, while the instruction register hold the encoded instruction.

→ Data path :

* Data path contains
  - ALU
  - Registers

→ ALU :

* ALU stands for Arithmetic Logic Unit.
* It carries out arithmetic and logic operations.

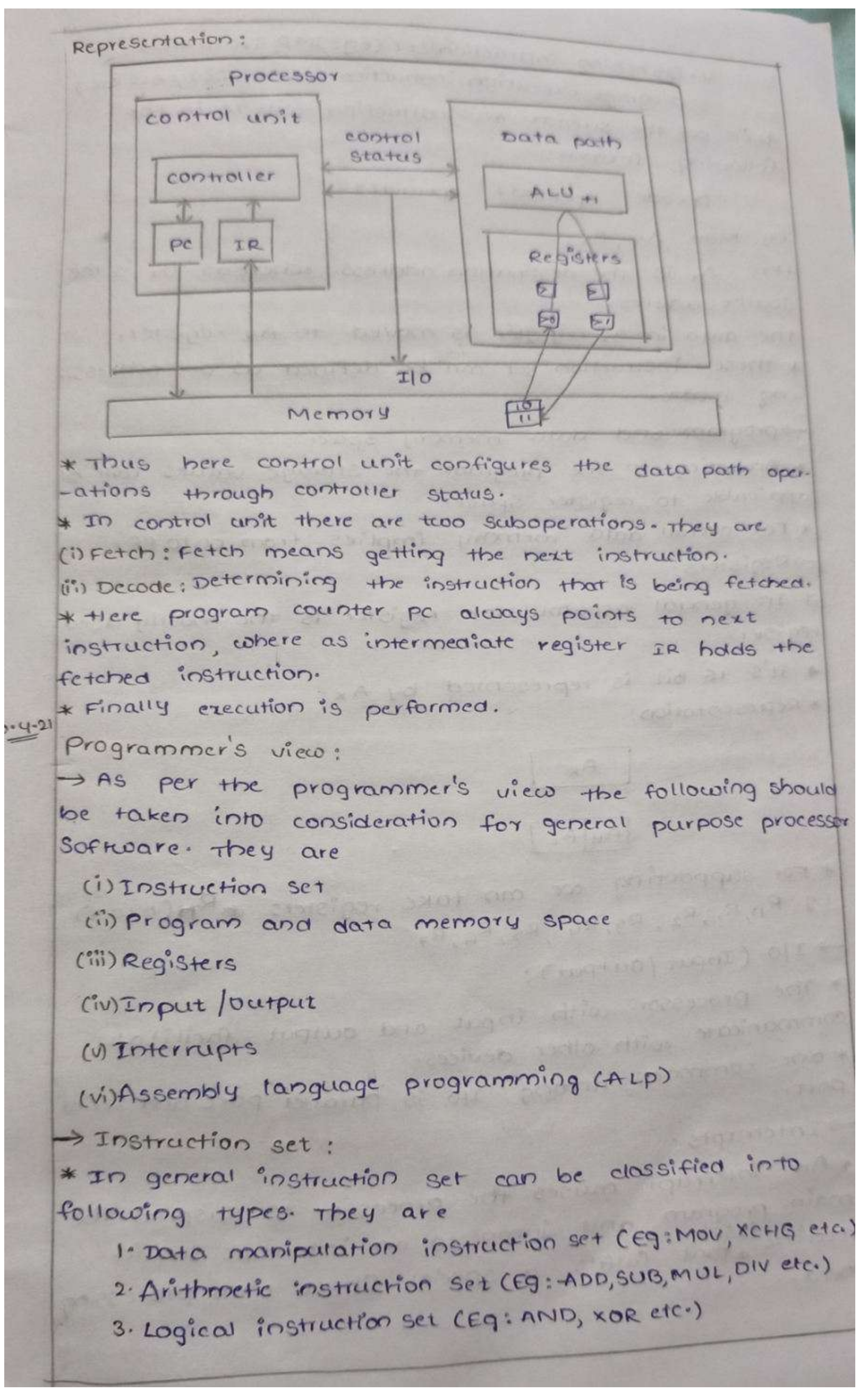
→ Registers :

* Registers are a type of memory.
* They are used to quickly accept, store and transfer the data.
* This is general purpose processor software both the control unit and data path are associated with I/O (Input/output) with memory as storage for further processing.

→ Operation :

* In general purpose processor software the datapath is general, where as control unit does not have any algorithm, but the algorithm is programmed into the memory.
* Here these operations are involved.

Step 1 : Load : From memory it is loaded to registers.
Step 2 : Through : Through operation is assigned to ALU to get the input from certain registers.
Step 3 : Store : The through operation from ALU is get operated and given back to register for store.



Representation :

* Thus here control unit configures the data path operations through controller status.
* In control unit there are two suboperations. They are
(i) Fetch : Fetch means getting the next instruction.
(ii) Decode : Determining the instruction that is being fetched.
* Here program counter PC always points to next instruction, where as intermediate register IR holds the fetched instruction.
* Finally execution is performed.

-4-21

Programmer's view :

→ As per the programmer's view the following should be taken into consideration for general purpose processor Software. They are

(i) Instruction set
(ii) Program and data memory space
(iii) Registers
(iv) Input / output
(v) Interrupts
(vi) Assembly language programming (ALP)

→ Instruction set :

* In general instruction set can be classified into following types. They are
1. Data manipulation instruction set (Eg: MOV, XCHG etc.)
2. Arithmetic instruction set (Eg: ADD, SUB, MUL, DIV etc.)
3. Logical instruction set (Eq: AND, XOR etc.)

4. Branching instruction set (Eg: JMP, SJMP etc.)

5. Program execution instruction set (Eg: RET, CALL etc.)

* As per the syntax an instruction will be in the following format.

Opcode operand1 operand2

eg: MOV A_x, B_x;

Here A_x is the destination address whereas B_x is the source address.

The data in B_x register is moved to A_x register.

* These instruction set can be defined as per addressing modes.

→ Program and data memory space:

* According to the program the storage values will be applicable to register space.

* For overall data memory implies from 00 to FF.

→ Registers:

* In general accumulator register is the mostly used register.

* It's 16 bit is represented by Ax.

* Representation:

Ax

A_H	A_L

* For supporting we can take registers R_N (0 to 7)

Eg: $R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7$

→ I/O (Input/output):

* The processor with input and output facilitate communicate with other devices.

* One common among I/O is parallel port and serial port.

→ Interrupts:

* An interrupt causes the processor to suspend the main program and jump to an interrupt service routine (ISR) that fulfils a special short term processing need.

* There will be hardware interrupts and software interrupts
* Hardware interrupts occurs due to mismatch in hardware parts.
* Software interrupts occurs due to bugs or errors in the program.

→ Assembly language programming (ALP):

* Assembly language programming involves the following steps as shown below.

C File
C File
asm File
compiler
Assembler
Binary File
Binary File
Binary File
Library
Linker
Debugger
exec. File
Profiler
Verification phase

Implementation phase

01-05-21

* For embedded system design the program will be either in .asm or .hex.
* Thus this undergo three steps.
  1. .asm is converted into .obj.
  2. .obj is converted into .exe
  3. Finally .exe is converted to .hex
* Finally the hex file is dumped into system or sometime .exe is enough for operation.
* Thus here the main operation can be done by the following.

(i) Assembler: Assembler is used to make use of program in .asm (written program is saved.

(ii) Linker: The assembler file is converted to objectable file and here the values are linked.

(iii) Debugger: converts the object file to executable file.

Development environment:

→ In the development environment of an embedded system the following components are used as tools.

1. Assemblers
2. compilers
3. Linker

→ Assembler: Assembler is used to convert assembly instructions to binary machine instructions.

* Assembler is used to translate the assembly instructi-ons (user written program) to binary machine instructions (system understandable program).

→ compiler:

* compiler is used to convert structured program into machine program.

* It translates high level to simple programming.

→ Linker:

* Linker is used to allow assembled files.

* A linker is designed to reduce the program execution and also used to eliminate default procedures and functions.

→ In the development environment the design flow of environment system should include support of programming to general purpose processor.

→ The designer has to identify the development Processor or target processor.

→ The design flow tools must write and debug the Program.

→ Target processor:

* Target processor is a processor to which we will Send our program and it becomes the part of the embedded system.

*** Application specific Instruction set processors:

→ Instruction: A line of command to execute a operation.

→ Instruction set: A set of lines scheduled to operate a Program.

→ Instruction set processor: Processor running a program according to the instructions given.

→Application specific instruction set processor:

* An application specific instruction set processor (ASIP) can serve as desired output for a particular task.
* This will have common characteristics.
* will have characteristics like digital signal processor.
* A designer has to think about the complete functional units in operations and their units.
* Using ASIP the system can have good flexibility.
* They will have good performance, neat utilizing power and simple size.

3·5·21

* this processor may undercome NRE Rule.
* the very important thing in ASIP is software.
* thus a designer has to write assembly language particularly.
* Thus Application specific instruction set processor (ASIP) will have special demand when compared to Digital Signal Processor (DSP).
* A DSP is a processor designed to perform common operations on digital signals like filtering, transform, combination.
* whereas ASIP are customized for desired functionality will all unique features.

Controller
control logic
IR
PC
Data path
Registers
custom ALU
Program Memory
Assembly code for
total = 0
for i=1 to---
Data Memory

Fig: Application specific instruction set processor (ASIP)

* Thus here in the ASIP the software must be compatab-le to all the functional units associated in it.
* The assembly programming written in should have access to all the units.
* The program embedded will have specific task as application which gives a desired output.
* Thus here ASIP the software is main and it depends on operating system.
* An operating system is a layer of software that provide application to the network.
* Thus operating system is a set of one or more progr-am executed on cpu consuming input data and proce-ssing output.
* Here operating system main task is managing and it involves the following operations like loading and executing the program and also involves sharing and allocating.
* The process involves the loading, executing, sharing and allocating.
* Thus with the involvement of software the ASIPs are performed well.

5-21

Selecting a processor (or) processor selection for an embedded system:

→ while selecting a processor it should be compatable to software and flexible to the hardware of an embed-ded system.

→ Thus in general there will be numerous kinds of processors with various design.

→ In among this the following considerations need to be factored during processor selection for an embedded system.

→ The considerations are

(i) performance

(ii) power

(iii) Peripheral set

(iv) operating voltage

(v) Specialized processing units

→ performance considerations:

* The first and foremost consideration in selecting the processor is its performance.
* thus performance depends on various factors such as primarily on architecture and design.
* Secondarily on fabrication of transistors in it.

→ power:

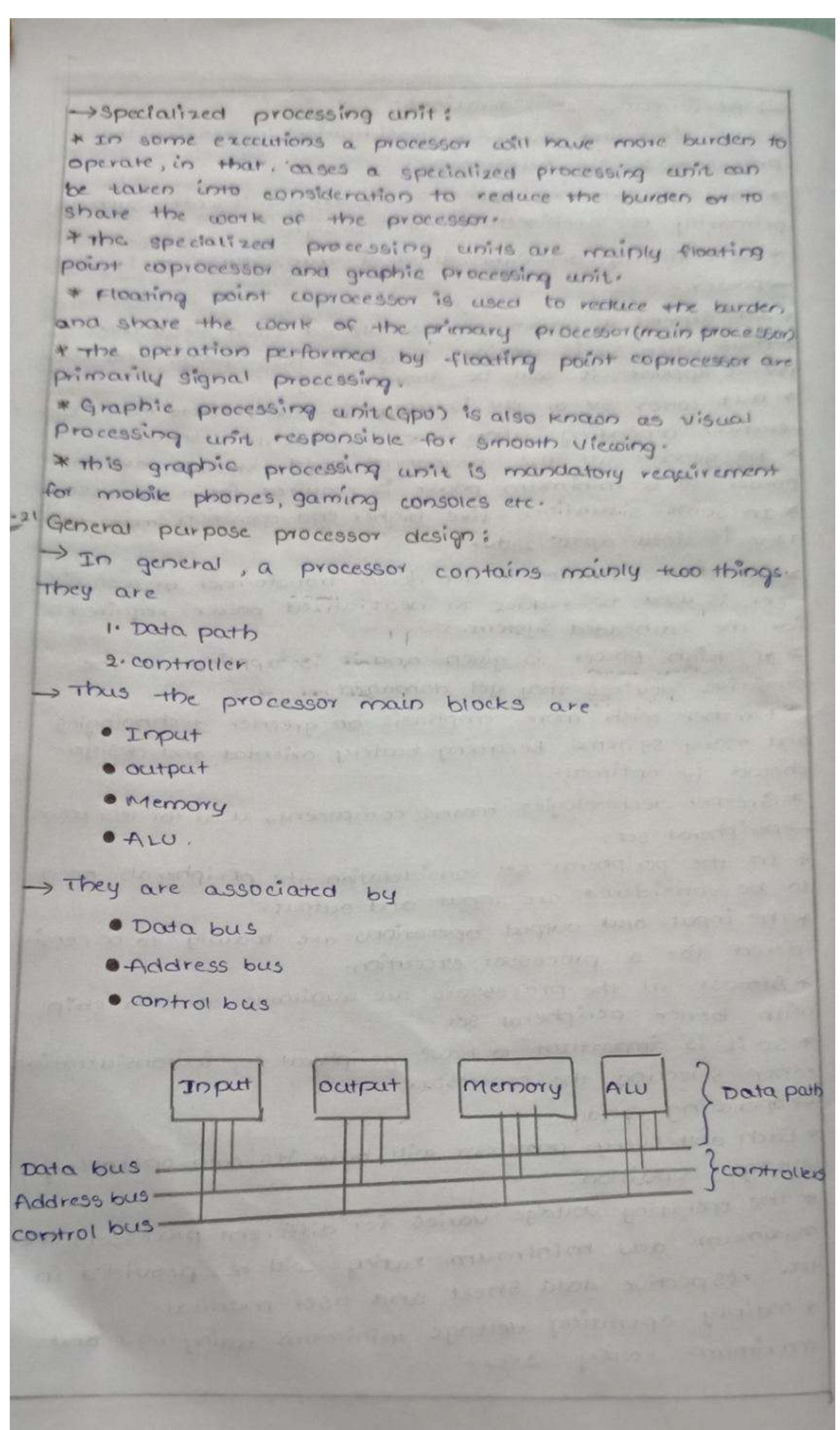
* In general a power can be allowed from 5v to 12v.
* thus when 5v is applied it will be less charge and when 12v is applied it will be faster charge.
* But when 5V or 12V is used is taken into the consideration depending upon the components used.
* In general all the component used should take less power so a minimum power is required for a processor.
* In some situations like higher end operation more than 12v is used upto 230v.
* In that situation a step down transformer or a stabilizer is used to reduce or neutralized power required for the embedded System.
* If high power is given and it is applied to the sensitive devices they get damaged.
* Further with more emphasis on greener technologies and many systems becoming battery oriented and design should be optimal.
* Greener technologies means components used for less power.

→ Peripheral set:

* In the peripheral set consideration the peripherals need to be considered are input and output.
* The input and output operations are mainly to be considered for a processor execution.
* Almost all the processors are available with a chip with better peripheral set.
* so it is important to have peripheral set in consideration when selecting the processor.

→ operating voltage:

* Each and every processor will have its own operating voltage condition.
* the operating voltage varies for different processors, the maximum and minimum rating will be provided in the respective data sheet and user manual.
* mainly operating voltage minimum rating 1.8v and maximum rating 3.3v.



→ Specialized processing unit:

* In some executions a processor will have more burden to operate, in that cases a specialized processing unit can be taken into consideration to reduce the burden or to share the work of the processor.
* The specialized processing units are mainly floating point coprocessor and graphic processing unit.
* Floating point coprocessor is used to reduce the burden and share the work of the primary processor (main processor)
* The operation performed by floating point coprocessor are primarily signal processing.
* Graphic processing unit (GPU) is also known as visual processing unit responsible for smooth viewing.
* This graphic processing unit is mandatory requirement for mobile phones, gaming consoles etc.

2.21 General purpose processor design:

→ In general, a processor contains mainly two things. They are

1. Data path
2. controller

→ Thus the processor main blocks are

- Input
- output
- Memory
- ALU.

→ They are associated by

- Data bus
- Address bus
- control bus

→ The designer of a general purpose processor builts a device suitable for a variety of applications.
→ One feature of general purpose processor is its program memory.
→ The designer has to write the program according to the requirement.
→ In general purpose processor there are large number of registers and one or more Arithmetic and logic Units (ALUs).
→ Here the data path handle the variety of computations
→ Many embedded systems are designed using general purpose processor.
→ The general purpose processor design is as shown below

controller
control logic
IR
PC
Program memory
Assembly code for:
total = 0
for i=1 to ---
Data path
Register file
General ALU
Data memory

→ In general purpose processor embedded system may result in several design metric benefits.
→ Here the designer must write only the program according to the requirement.
→ The general purpose processor have high flexibility, it means changes the functionality simply by changing the program.
→ The general purpose processor performance is also fast.

→ when compared to small the cost is low but in some cases high is required at that movement cost is also more.

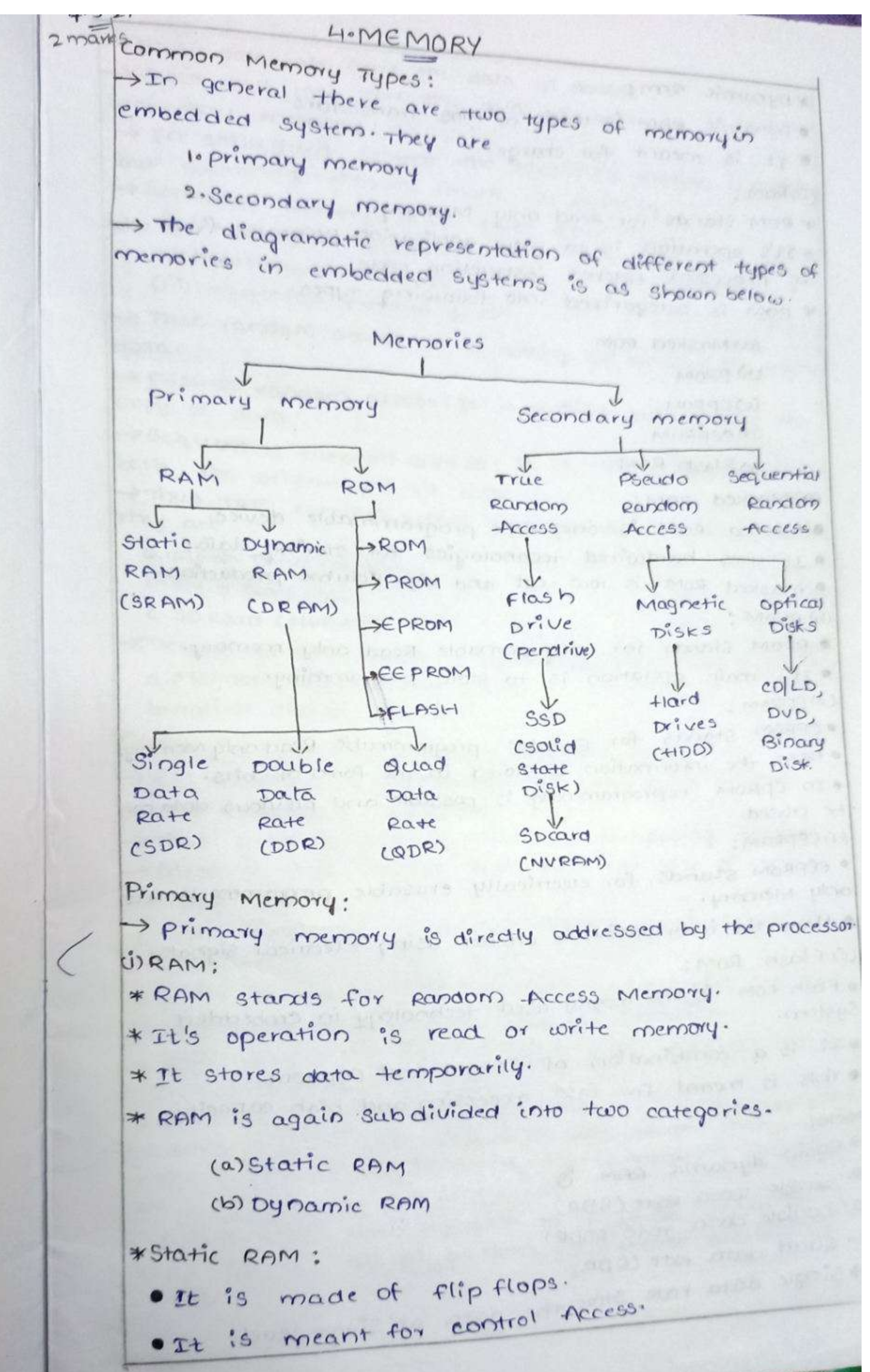
→ So, in that cases unnecessary hardware is to be reduced to get low cost in general purpose processor design.

→ General purpose processor carry out different functionalities by means of controller. as shown in the diagram.

→ The functionality is stored in a program memory.

→ Operation:

* The functionality is stored in program memory.
* the controller fetches the current instruction as indicated by the program counter (PC) into the instruction register (IR).
* Then the data path executes as per the instruction.
* this procedure continues until the last instruction.
* Finally the achieved output is to be done.
* Finally determines the approximate next instruction address, sets the PC to this address and fetches again.



2 marks

4. MEMORY

Common Memory Types:

→ In general there are two types of memory in embedded system. They are

1. primary memory
2. Secondary memory.

→ The diagramatic representation of different types of memories in embedded systems is as shown below.

Primary Memory:

→ primary memory is directly addressed by the processor.

(i) RAM:

* RAM stands for Random Access Memory.
* It's operation is read or write memory.
* It stores data temporarily.
* RAM is again subdivided into two categories.

(a) Static RAM

(b) Dynamic RAM

*Static RAM:

- It is made of flip flops.
- It is meant for control Access.

* Dynamic RAM:

- Dynamic RAM is made of MOS transistors.
- It is meant for charge.

(ii) ROM:

* ROM stands for Read Only Memory.
* It's operation is to store application programs from where processor fetches instruction code.
* ROM is categorized into following types.
  - (a) Masked ROM
  - (b) PROM
  - (c) EPROM
  - (d) EEPROM
  - (e) Flash ROM

(a) Masked ROM:

- Masked ROM is one time programmable device.
- It uses hardwired technologies for storing data.
- Masked ROM is low cost and high volume production.

(b) PROM:

- PROM stands for programmable Read only memory.
- It's main operation is to store programming.

(c) EPROM:

- EPROM stands for Erasable programmable Read only Memory.
- Here the information is stored in the form of bits.
- In EPROM reprogramming is possible and previous data can be erased.

(d) EEPROM:

- EEPROM stands for electrically erasable programmable Read only Memory.
- Here the information is altered using electrical signals.

(e) Flash ROM:

- Flash ROM is the mostly used technology in Embedded System.
- It is a combination of EPROM and EEPROM.
- This is meant for fast accessing and high capacity.

08.05.21 notes:

→ Again dynamic RAM is

a. Single Data Rate (SDR)

b. Double data rate (DDR)

c. Quad data rate (QDR)

→ Single data rate store the data at single level.

→ Double data rate store the data at double
→ Quad data rate store the data at Quad.

Secondary memory:

→ In embedded system the secondary memory is nothing but supporting memory meant for accessing.

→ Secondary memory is of three types.

(i) True random access

(ii) Pseudo random access

(iii) Sequential random access

→ True random access: It is nothing but accessing the original data.

→ Pseudo random access: It is nothing but accessing the copy of data.

→ Sequential random acceso: It is nothing but accessing both the original and it's copy.

→ Thus true random access is categorized into three types. They are

a. Flash Drive (pendrive)

b. Solid state (SSD)

c. SD card (NVRAM)

→ Pseudo random access is again categorized into two types

a. Magnetic Disks (Hard drives)

b. optical disks (CD/DVD)

Composing memory: (dumping the program)

→ For composing memory in an embedded system, memory size is required.

→ Thus memory size is main for composing memory.

→ Memory size differs from different size of readily memory size.

→ when available memory is larger simply ignore the unneeded.

→ while composing memory the address bits and data lines are should be taken into consideration.

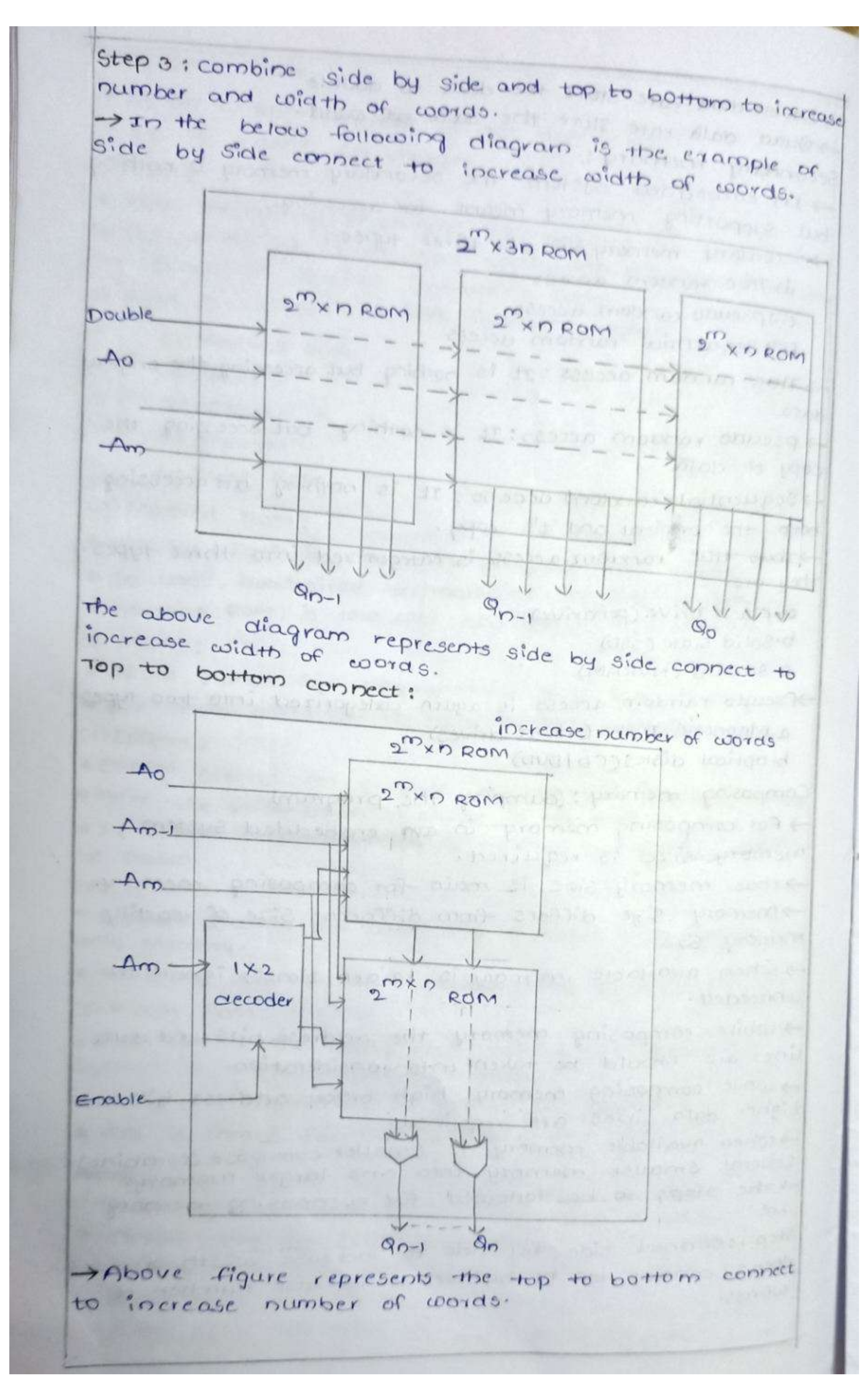
→ while composing memory high order address bits and higher data lines are required.

→ when available memory is smaller compose (combine) several smaller memory into one larger memory.

→ The steps to be followed for composing memory are

Step 1: connect side by side to increase width of words.

Step 2: connect top to bottom to increase number of words.



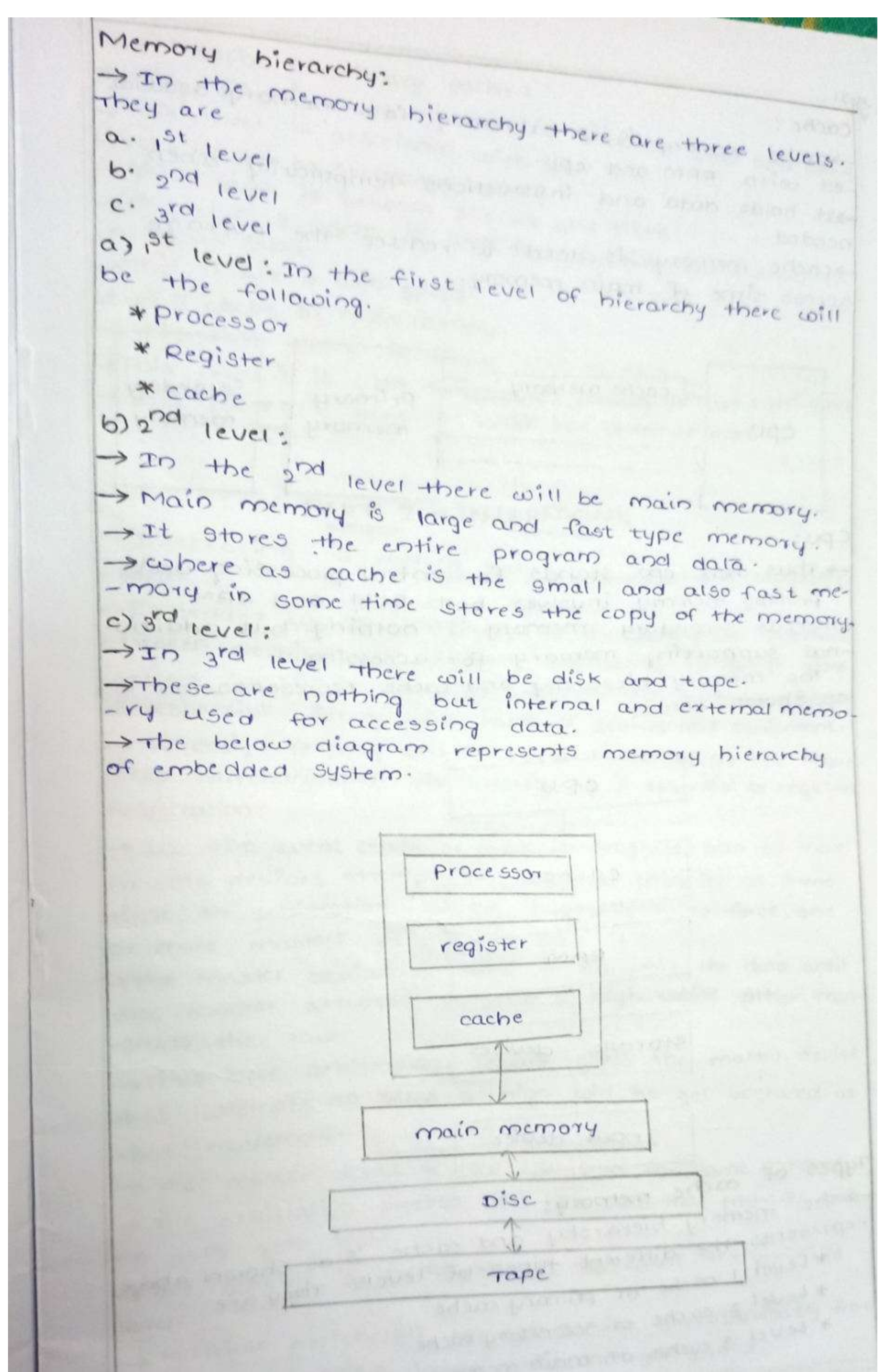
Step 3 : combine side by side and top to bottom to increase number and width of words.

→ In the below following diagram is the example of side by side connect to increase width of words.

The above diagram represents side by side connect to increase width of words.

Top to bottom connect :

→ Above figure represents the top to bottom connect to increase number of words.



Memory hierarchy:

→ In the memory hierarchy there are three levels. They are

a. 1st level

b. 2nd level

c. 3rd level

a) 1st level: In the first level of hierarchy there will be the following.

* Processor
* Register
* cache

b) 2nd level:

→ In the 2nd level there will be main memory.

→ Main memory is large and fast type memory.

→ It stores the entire program and data.

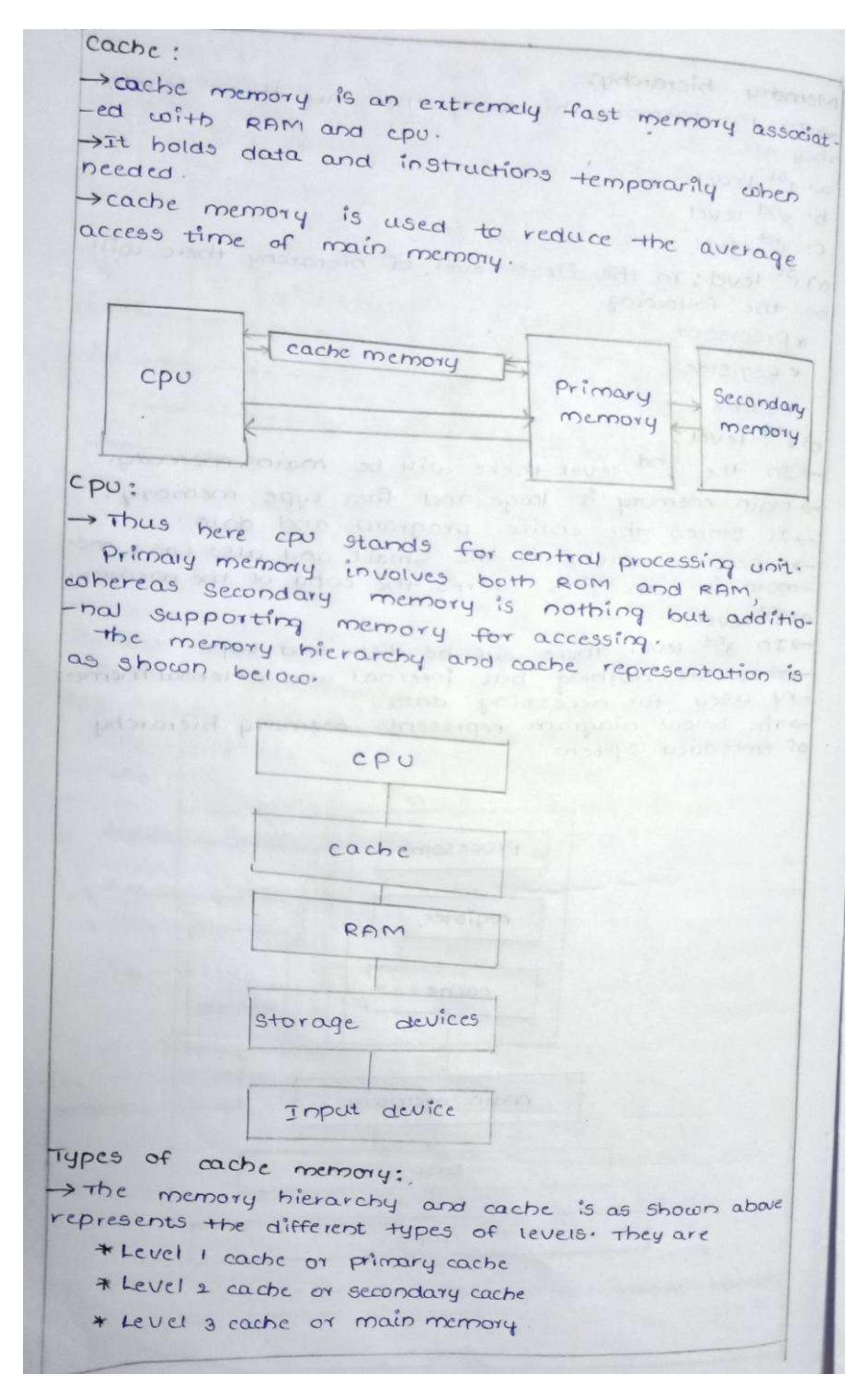
→ where as cache is the small and also fast memory in some time stores the copy of the memory.

c) 3rd level:

→ In 3rd level there will be disk and tape.

→ These are nothing but internal and external memory used for accessing data.

→ The below diagram represents memory hierarchy of embedded system.



Cache :

→ cache memory is an extremely fast memory associat-
-ed with RAM and cpu.

→ It holds data and instructions temporarily when needed.

→ cache memory is used to reduce the average access time of main memory.

CPU

cache memory

Primary memory

Secondary memory

cpu:

→ Thus here cpu stands for central processing unit.
Primary memory involves both ROM and RAM, whereas Secondary memory is nothing but additio-
-nal supporting memory for accessing.
the memory hierarchy and cache representation is as shown below.

CPU

cache

RAM

Storage devices

Input device

Types of cache memory:

→ The memory hierarchy and cache is as shown above represents the different types of levels. They are

* Level 1 cache or primary cache
* Level 2 cache or secondary cache
* Level 3 cache or main memory

Level 1 cache or primary cache:

→ This type of cache memory is between 2KB and 64KB.

→ This level is associated with CPU.

Level 2 cache or secondary cache:

→ This level is between 256 KB and 512KB.

→ This level 2 cache is used for searching the instructions in level 1 cache.

→ This level 2 is high speed.

Level 3 cache or main memory:

→ This level cache is about 1MB to 8MB.

→ This level 3 is used for common sharing of level 1 and level 2.

→ This level 3 is used to double the speed of RAM.

PART 2 – INTERFACING

→ Interfacing is nothing but get connected and communicated.

→ Embedded system interfacing is the conceptual interface between electrical and computer engineering to design good.

→ Here interfacing is important because embedded system is responsible for a wide range of devices and equipment.

→ To design properly and to communicate as per the required communication the interfacing is essential or required.

Arbitration:

→ In the event (task or work in progress) two or more master devices attempt to begin a transfer at same time, an arbitration scheme is employed to force one or more masters to give the bus.

→ The master devices continue to transmit the data until one master attempts to send a high while other transmits the low.

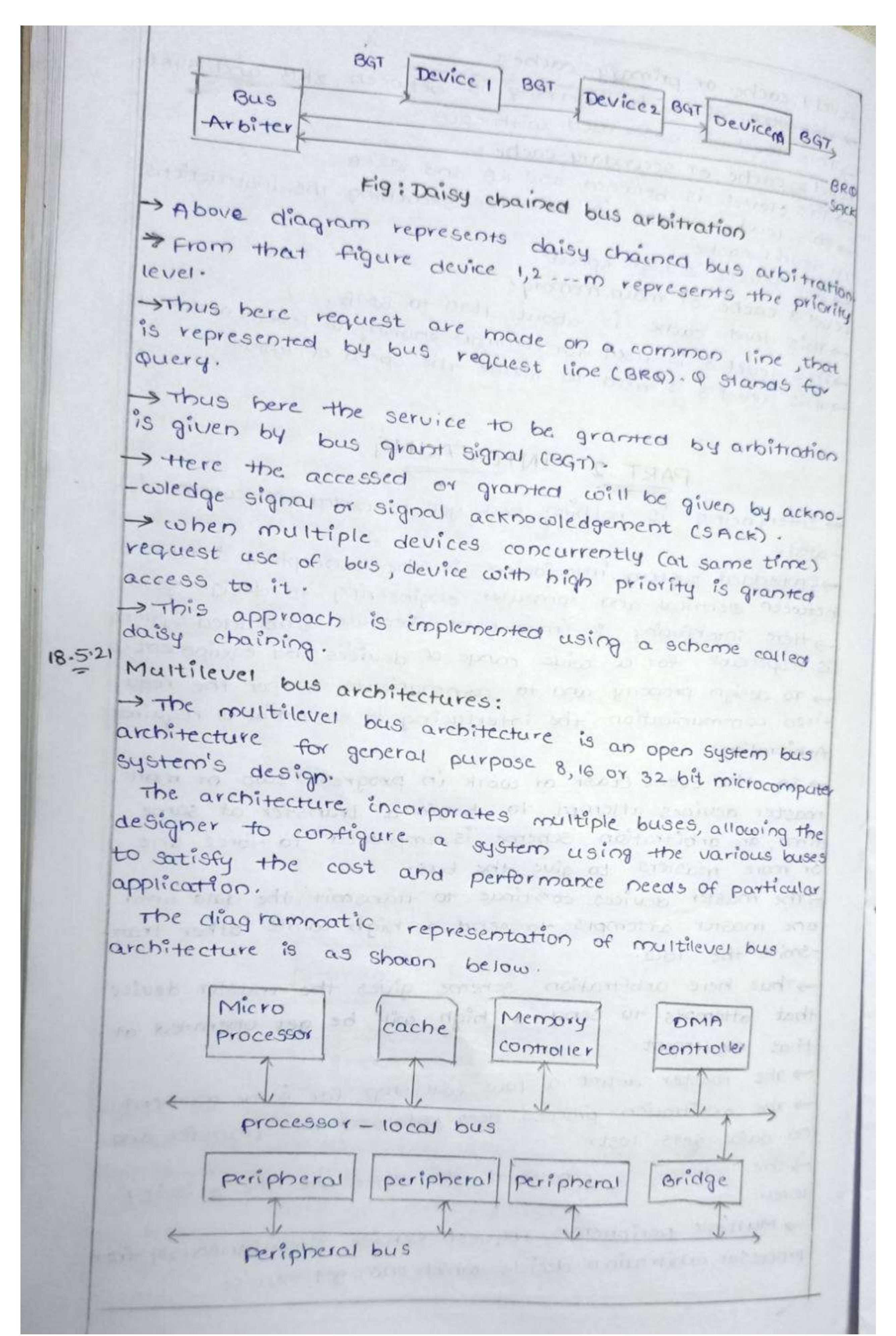
→ Thus here arbitration scheme gives the master device that attempts to send a high will be get operated at that movement.

→ The master detect a low will stop for some moment.

→ The arbitration process does not slow the transfer and no data gets lost.

→ The scheme is employed to represent the priority level.

→ Multiple peripherals request service simultaneously from processor, arbitration decide which one get service.



BGT Device 1 BGT Device 2 BGT Device m BGT

Bus Arbiter

BRQ

SACK

Fig: Daisy chained bus arbitration

→ Above diagram represents daisy chained bus arbitration

→ From that figure device 1,2 --- m represents the priority level.

→ Thus here request are made on a common line, that is represented by bus request line (BRQ). Q stands for Query.

→ Thus here the service to be granted by arbitration is given by bus grant signal (BGT).

→ Here the accessed or granted will be given by acknowledge signal or signal acknowledgement (SACK).

→ when multiple devices concurrently (at same time) request use of bus, device with high priority is granted access to it.

→ This approach is implemented using a scheme called daisy chaining.

18.5.21

Multilevel bus architectures:

→ The multilevel bus architecture is an open system bus architecture for general purpose 8, 16 or 32 bit microcomputer system's design.

The architecture incorporates multiple buses, allowing the designer to configure a system using the various buses to satisfy the cost and performance needs of particular application.

The diagrammatic representation of multilevel bus architecture is as shown below.

Micro Processor | cache | Memory controller | DMA controller

processor - local bus

peripheral | peripheral | peripheral | Bridge

peripheral bus

→ Thus multilevel bus architecture will have the following.

* Microprocessor
* Cache
* Memory controller
* DMA controller
* Bridge
* Peripherals

→ Microprocessor is nothing but representation of input, output, memory and ALU.

→ cache is a temporary storage memory associated with cpu and primary memory.

→ Memory controller is a digital circuit that manages the flow of data going to and from main memory.

→ DMA controller : DMA stands for Direct Memory-Access. DMA controller is a hardware device used for direct memory access.

→ The microprocessor, cache, memory controller and DMA controller are linked to processor local bus.

→ processor local bus is a high speed, wide, most frequent communication used.

→ Bridge is a single purpose processor converts communication between buses.

→ peripheral is nothing but connection for bus interface.

→ The bridge and peripherals are linked to peripheral bus.

→ The peripheral bus is a low speed, narrow, less frequent communication used.

19.5.21

Advanced communication principles:

→ In advanced communication principles the term layering is mostly used for representation.

→ Layering is defined by it's break complexity of communication protocol into pieces easier to design and understand.

→ In layering lower levels provide services to higher level.

→ Thus here lower level might work with bits while higher level might work with packets of data.

→ Layering is mainly implemented to physical layer.

→ physical layer is lower level in hierarchy.

→ Thus layering in physical layer is used for and or represented as medium to carry data from one node to another.

→ In advanced communication principles the following communication are represented. They are

a) Parallel communication

b) Serial communication

c) wireless communication

Parallel communication:

→ Parallel communication make use of advanced communication principles implemented to physical layer which are capable of transporting multiple bits of data.

→ That transport include high data throughput with short distances.

→ This parallel communication is higher cost with bulky design.

→ Parallel communication is nothing but long parallel wires results in high capacitance value which requires more time discharge or discharge.

Serial communication:

→ Serial communication follows advanced communication principles with single data wire possible to control and power.

→ In serial communication transport of one bit at a time is possible i.e words transmitted one bit at a time.

→ In serial communication high data throughput with long distances.

→ Thus serial communication is low cost with less bulky.

→ In serial communication less average capacitance so more bits unit of time are transmitted.

→ In serial communication complex interfacing logic is implemented.

→ complex interfacing logic means sender needs to decompose words into bits whereas receiver needs recompose bits into words.

.5.21 Wireless communication:

→ Wireless communication also follows advanced communication principles.

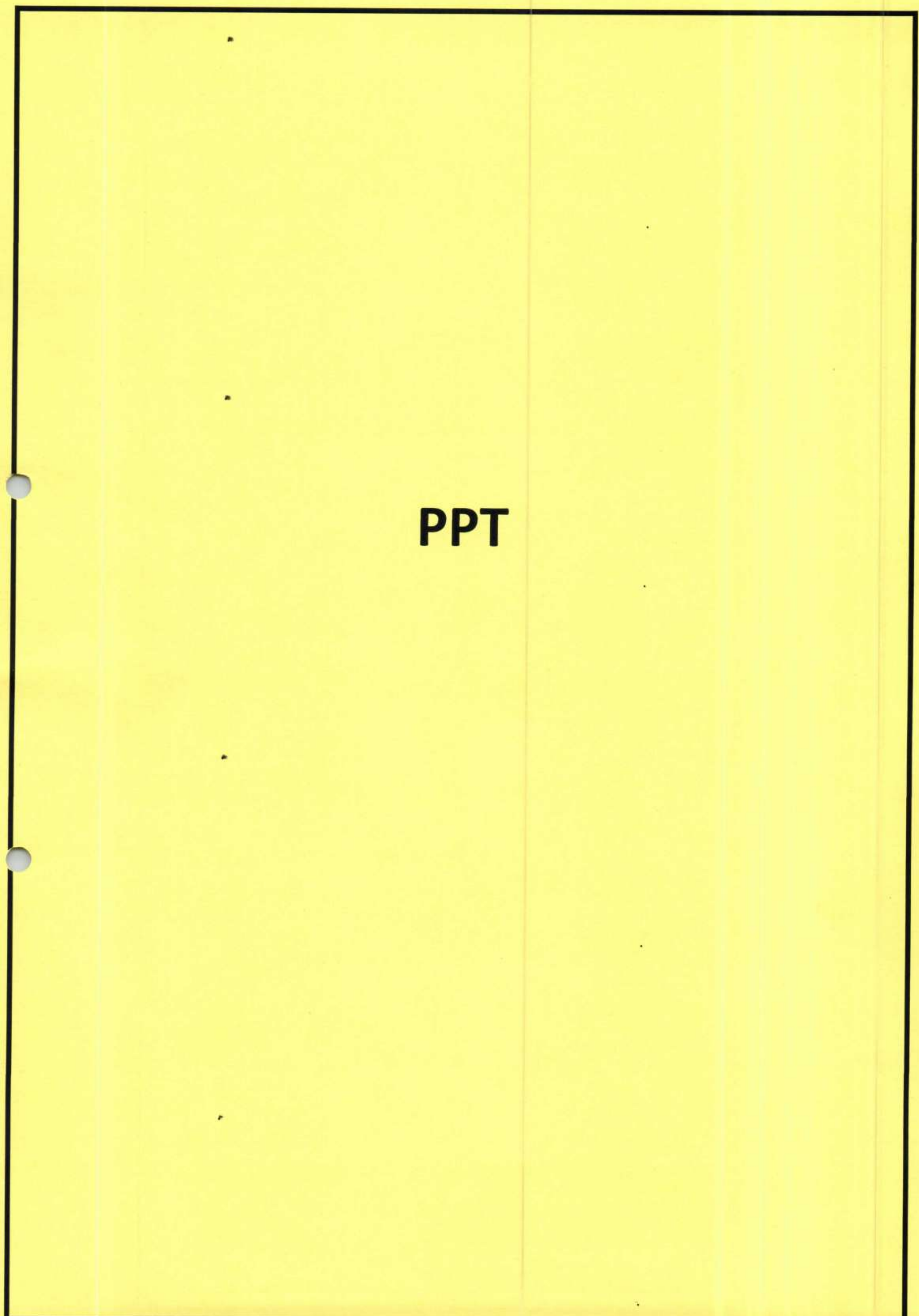
→ Wireless communication in embedded system are nothing but infrared (IR) and Radio Frequency(RF).

Infrared (IR):

* Infrared (IR) is a electronic wave frequency just below visible light spectrum.
* Infrared detects signal and conducts when exposed.
* It is cheap to build.
* Infrared is limited range.

Radiofrequency (RF):

* Radio frequency is an electromagnetic wave frequency.
* For radio frequency there we need analog circuitry and antenna.
* Radio frequency is having intermediate range.


PPT

Embedded Systems Design: A Unified Hardware/Software Introduction

Introduction

Outline

- Embedded systems overview
  - What are they?
- Design challenge – optimizing design metrics
- Technologies
  - Processor technologies
  - IC technologies
  - Design technologies

Embedded systems overview

- Computing systems are everywhere
- Most of us think of "desktop" computers
  - PC's
  - Laptops
  - Mainframes
  - Servers
- But there's another type of computing system
  - Far more common...

Embedded systems overview

- Embedded computing systems
  - Computing systems embedded within electronic devices
  - Hard to define. Nearly any computing system other than a desktop computer
  - Billions of units produced yearly, versus millions of desktop units
  - Perhaps 50 per household and per automobile

Computers are in here...

and here...

and even here...

Lots more of these, though they cost a lot less each.

A "short list" of embedded systems

Anti-lock brakes
Auto-focus cameras
Automatic teller machines
Automatic toll systems
Automatic transmission
Avionic systems
Battery chargers
Camcorders
Cell phones
Cell-phone base stations
Cordless phones
Cruise control
Curbside check-in systems
Digital cameras
Disk drives
Electronic card readers
Electronic instruments
Electronic toys/games
Factory control
Fax machines
Fingerprint identifiers
Home security systems
Life-support systems
Medical testing systems

Modems
MPEG decoders
Network cards
Network switches/routers
On-board navigation
Pagers
Photocopiers
Point-of-sale systems
Portable video games
Printers
Satellite phones
Scanners
Smart ovens/dishwashers
Speech recognizers
Stereo systems
Teleconferencing systems
Televisions
Temperature controllers
Theft tracking systems
TV set-top boxes
VCR's, DVD players
Video game consoles
Video phones
Washers and dryers

And the list goes on and on

Some common characteristics of embedded systems

- Single-functioned
  - Executes a single program, repeatedly
- Tightly-constrained
  - Low cost, low power, small, fast, etc.
- Reactive and real-time
  - Continually reacts to changes in the system's environment
  - Must compute certain results in real-time without delay

An embedded system example -- a digital camera

Digital camera chip: CCD, lens, A2D, CCD preprocessor, Pixel coprocessor, D2A, JPEG codec, Microcontroller, Multiplier/Accum, DMA controller, Display ctrl, Memory controller, ISA bus interface, UART, LCD ctrl

- Single-functioned -- always a digital camera
- Tightly-constrained -- Low cost, low power, small, fast
- Reactive and real-time -- only to a small extent

Design challenge – optimizing design metrics

- Obvious design goal:
  - Construct an implementation with desired functionality
- Key design challenge:
  - Simultaneously optimize numerous design metrics
- Design metric
  - A measurable feature of a system's implementation
  - Optimizing design metrics is a key challenge

Design challenge – optimizing design metrics

- Common metrics
  - Unit cost: the monetary cost of manufacturing each copy of the system, excluding NRE cost
  - NRE cost (Non-Recurring Engineering cost): The one-time monetary cost of designing the system
  - Size: the physical space required by the system
  - Performance: the execution time or throughput of the system
  - Power: the amount of power consumed by the system
  - Flexibility: the ability to change the functionality of the system without incurring heavy NRE cost

Design challenge – optimizing design metrics

- Common metrics (continued)
  - Time-to-prototype: the time needed to build a working version of the system
  - Time-to-market: the time required to develop a system to the point that it can be released and sold to customers
  - Maintainability: the ability to modify the system after its initial release
  - Correctness, safety, many more

Design metric competition -- improving one may worsen others

Power, Performance, Size, NRE cost

Hardware, Software

- Expertise with both **software and hardware** is needed to optimize design metrics
  - Not just a hardware or software expert, as is common
  - A designer must be comfortable with various technologies in order to choose the best for a given application and constraints

Time-to-market: a demanding design metric

Revenues ($), Time (months)

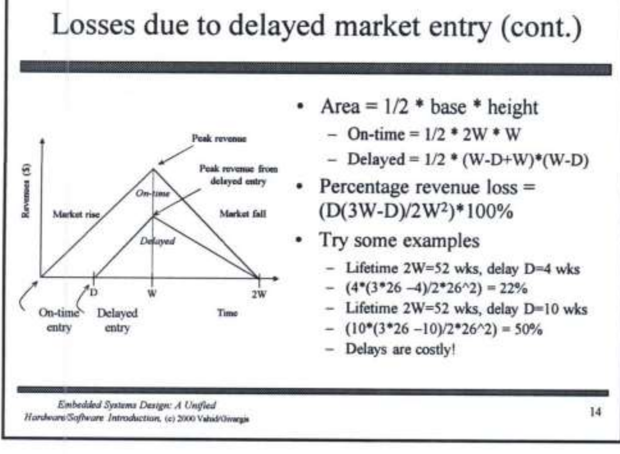
- Time required to develop a product to the point it can be sold to customers
- Market window
  - Period during which the product would have highest sales
- Average time-to-market constraint is about 8 months
- Delays can be costly

Losses due to delayed market entry

Revenues ($) — Peak revenue — Peak revenue from delayed entry — On-time — Market rise — Market fall — Delayed — D — W — 2W — Time — On-time entry — Delayed entry

- Simplified revenue model
  - Product life = 2W, peak at W
  - Time of market entry defines a triangle, representing market penetration
  - Triangle area equals revenue
- Loss
  - The difference between the on-time and delayed triangle areas

Embedded Systems Design: A Unified Hardware/Software Introduction, (c) 2000 Vahid/Givargis — 13

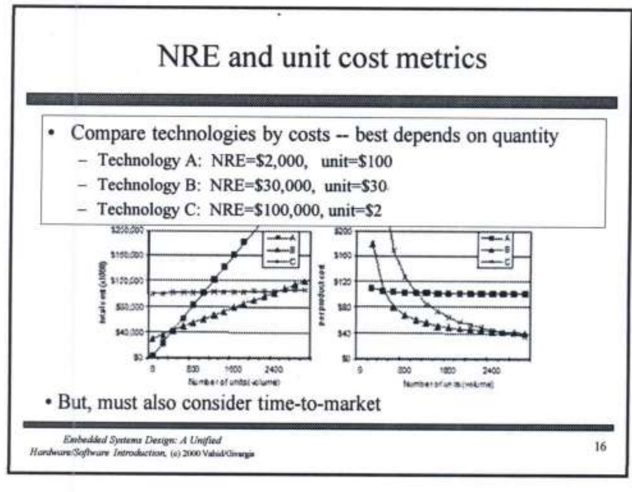


NRE and unit cost metrics

- Costs:
  - Unit cost: the monetary cost of manufacturing each copy of the system, excluding NRE cost
  - NRE cost (Non-Recurring Engineering cost): The one-time monetary cost of designing the system
  - *total cost = NRE cost + unit cost * # of units*
  - *per-product cost = total cost / # of units*
    = (NRE cost / # of units) + unit cost
- Example
  - NRE=$2000, unit=$100
  - For 10 units
    - total cost = $2000 + 10*$100 = $3000
    - per-product cost = $2000/10 + $100 = $300

Amortizing NRE cost over the units results in an additional $200 per unit

Embedded Systems Design: A Unified Hardware/Software Introduction, (c) 2000 Vahid/Givargis — 15



The performance design metric

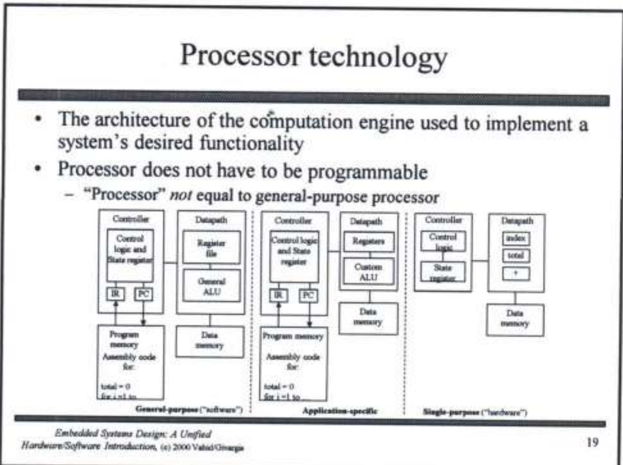
- Widely-used measure of system, widely-abused
  - Clock frequency, instructions per second – not good measures
  - Digital camera example – a user cares about how fast it processes images, not clock speed or instructions per second
- Latency (response time)
  - Time between task start and end
  - e.g., Camera's A and B process images in 0.25 seconds
- Throughput
  - Tasks per second, e.g. Camera A processes 4 images per second
  - Throughput can be more than latency seems to imply due to concurrency, e.g. Camera B may process 8 images per second (by capturing a new image while previous image is being stored).
- *Speedup* of B over S = B's performance / A's performance
  - Throughput speedup = 8/4 = 2

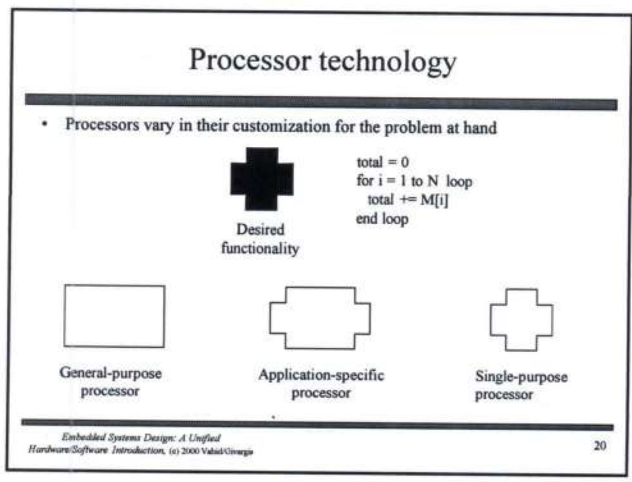
Embedded Systems Design: A Unified Hardware/Software Introduction, (c) 2000 Vahid/Givargis — 17

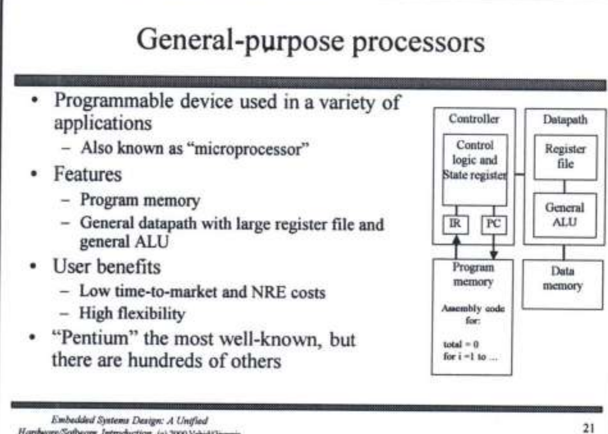
Three key embedded system technologies

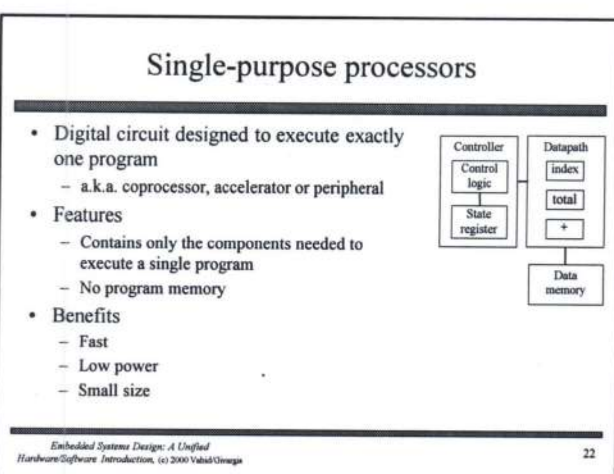
- Technology
  - A manner of accomplishing a task, especially using technical processes, methods, or knowledge
- Three key technologies for embedded systems
  - Processor technology
  - IC technology
  - Design technology

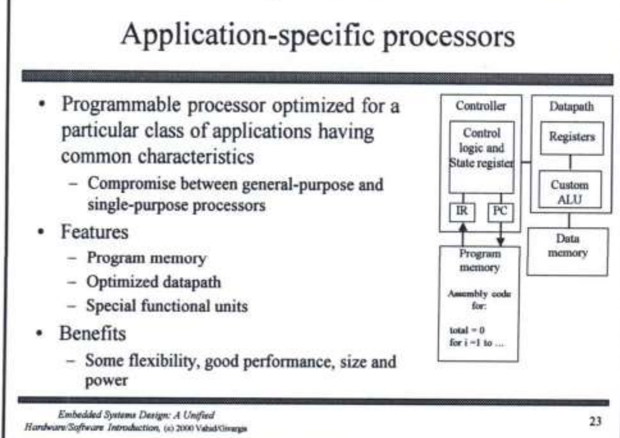
Embedded Systems Design: A Unified Hardware/Software Introduction, (c) 2000 Vahid/Givargis — 18

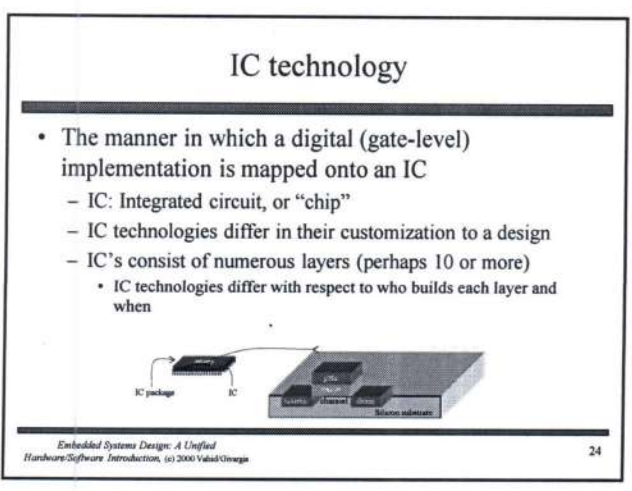

Processor technology
• The architecture of the computation engine used to implement a system's desired functionality
• Processor does not have to be programmable
– "Processor" not equal to general-purpose processor
Controller
Datapath
Control logic and State register
Register file
General ALU
IR
PC
Program memory
Assembly code for:
total = 0
for i =1 to ...
Data memory
Controller
Datapath
Control logic and State register
Registers
Custom ALU
Program memory
Assembly code for:
total = 0
for i =1 to ...
Data memory
Controller
Datapath
Control logic
State register
index
total
+
Data memory
General-purpose ("software")
Application-specific
Single-purpose ("hardware")
Embedded Systems Design: A Unified Hardware/Software Introduction, (c) 2000 Vahid/Givargis
19


Processor technology
• Processors vary in their customization for the problem at hand
total = 0
for i = 1 to N loop
total += M[i]
end loop
Desired functionality
General-purpose processor
Application-specific processor
Single-purpose processor
Embedded Systems Design: A Unified Hardware/Software Introduction, (c) 2000 Vahid/Givargis
20


General-purpose processors
• Programmable device used in a variety of applications
– Also known as "microprocessor"
• Features
– Program memory
– General datapath with large register file and general ALU
• User benefits
– Low time-to-market and NRE costs
– High flexibility
• "Pentium" the most well-known, but there are hundreds of others
Controller
Datapath
Control logic and State register
Register file
General ALU
IR
PC
Program memory
Assembly code for:
total = 0
for i =1 to ...
Data memory
Embedded Systems Design: A Unified Hardware/Software Introduction, (c) 2000 Vahid/Givargis
21


Single-purpose processors
• Digital circuit designed to execute exactly one program
– a.k.a. coprocessor, accelerator or peripheral
• Features
– Contains only the components needed to execute a single program
– No program memory
• Benefits
– Fast
– Low power
– Small size
Controller
Datapath
Control logic
State register
index
total
+
Data memory
Embedded Systems Design: A Unified Hardware/Software Introduction, (c) 2000 Vahid/Givargis
22


Application-specific processors
• Programmable processor optimized for a particular class of applications having common characteristics
– Compromise between general-purpose and single-purpose processors
• Features
– Program memory
– Optimized datapath
– Special functional units
• Benefits
– Some flexibility, good performance, size and power
Controller
Datapath
Control logic and State register
Registers
Custom ALU
IR
PC
Program memory
Assembly code for:
total = 0
for i =1 to ...
Data memory
Embedded Systems Design: A Unified Hardware/Software Introduction, (c) 2000 Vahid/Givargis
23


IC technology
• The manner in which a digital (gate-level) implementation is mapped onto an IC
– IC: Integrated circuit, or "chip"
– IC technologies differ in their customization to a design
– IC's consist of numerous layers (perhaps 10 or more)
• IC technologies differ with respect to who builds each layer and when
IC package
IC
Silicon substrate
Embedded Systems Design: A Unified Hardware/Software Introduction, (c) 2000 Vahid/Givargis
24

IC technology

- Three types of IC technologies
  - Full-custom/VLSI
  - Semi-custom ASIC (gate array and standard cell)
  - PLD (Programmable Logic Device)

Full-custom/VLSI

- All layers are optimized for an embedded system's particular digital implementation
  - Placing transistors
  - Sizing transistors
  - Routing wires
- Benefits
  - Excellent performance, small size, low power
- Drawbacks
  - High NRE cost (e.g., $300k), long time-to-market

Semi-custom

- Lower layers are fully or partially built
  - Designers are left with routing of wires and maybe placing some blocks
- Benefits
  - Good performance, good size, less NRE cost than a full-custom implementation (perhaps $10k to $100k)
- Drawbacks
  - Still require weeks to months to develop

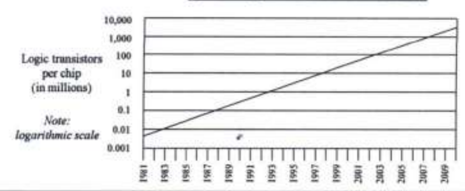
PLD (Programmable Logic Device)

- All layers already exist
  - Designers can purchase an IC
  - Connections on the IC are either created or destroyed to implement desired functionality
  - Field-Programmable Gate Array (FPGA) very popular
- Benefits
  - Low NRE costs, almost instant IC availability
- Drawbacks
  - Bigger, expensive (perhaps $30 per unit), power hungry, slower

Moore's law

- The most important trend in embedded systems
  - Predicted in 1965 by Intel co-founder Gordon Moore

IC transistor capacity has doubled roughly every 18 months for the past several decades



Moore's law

- Wow
  - This growth rate is hard to imagine, most people underestimate
  - How many ancestors do you have from 20 generations ago
    - i.e., roughly how many people alive in the 1500's did it take to make you?
    - 2^{20} = more than *1 million people*
  - *(This underestimation is the key to pyramid schemes!)*

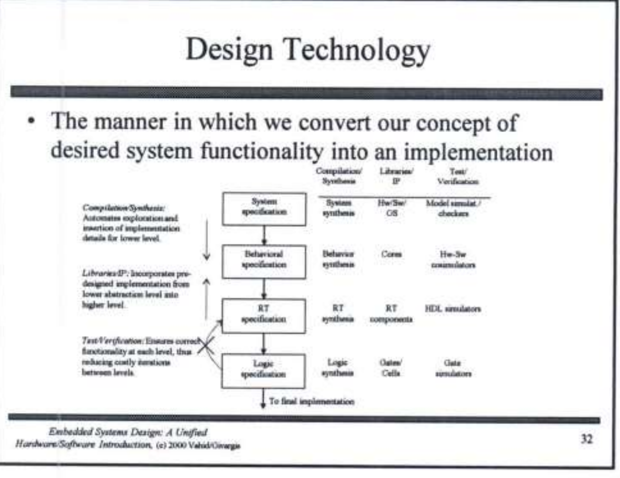
Graphical illustration of Moore's law

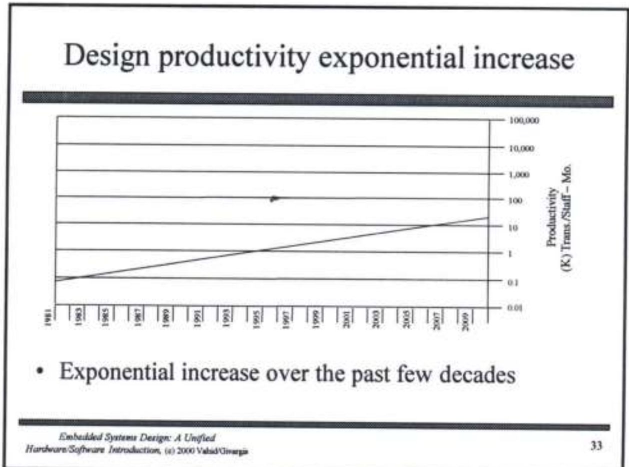
1981 1984 1987 1990 1993 1996 1999 2002

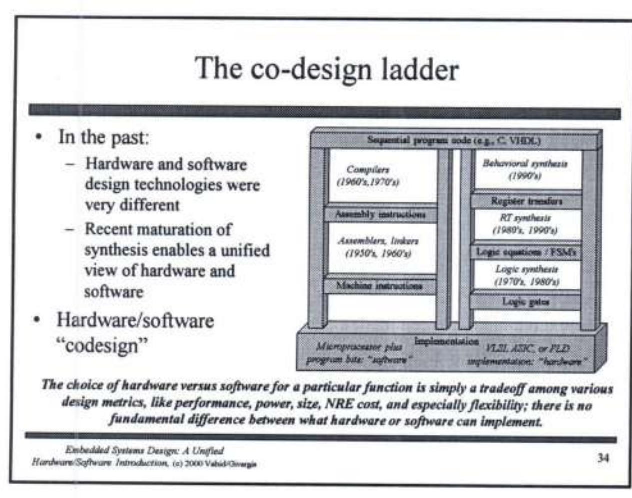
10,000 transistors — *Leading edge chip in 1981*

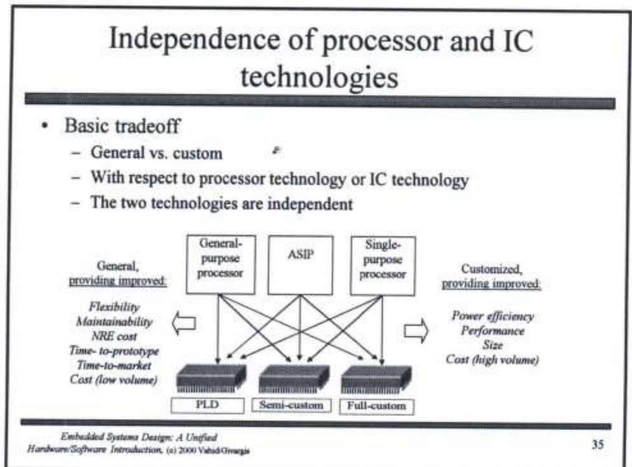
150,000,000 transistors — *Leading edge chip in 2002*

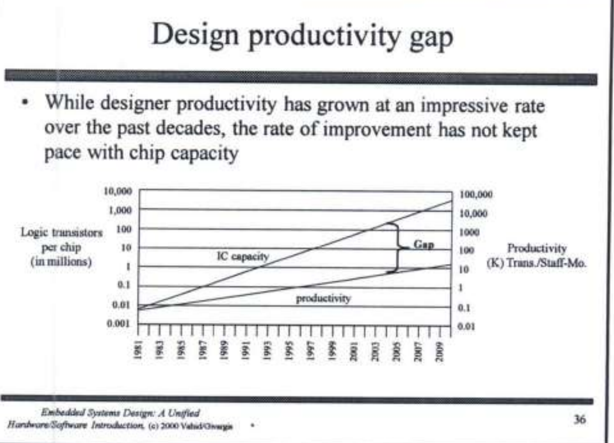
- Something that doubles frequently grows more quickly than most people realize!
  - A 2002 chip can hold about 15,000 1981 chips inside itself





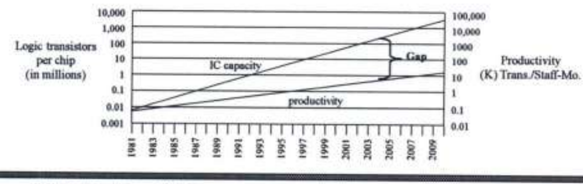






Design productivity gap

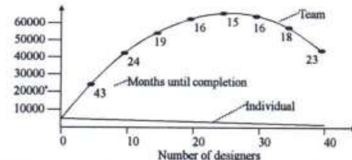
- 1981 leading edge chip required 100 designer months
  - 10,000 transistors / 100 transistors/month
- 2002 leading edge chip requires 30,000 designer months
  - 150,000,000 / 5000 transistors/month
- Designer cost increase from $1M to $300M



The mythical man-month

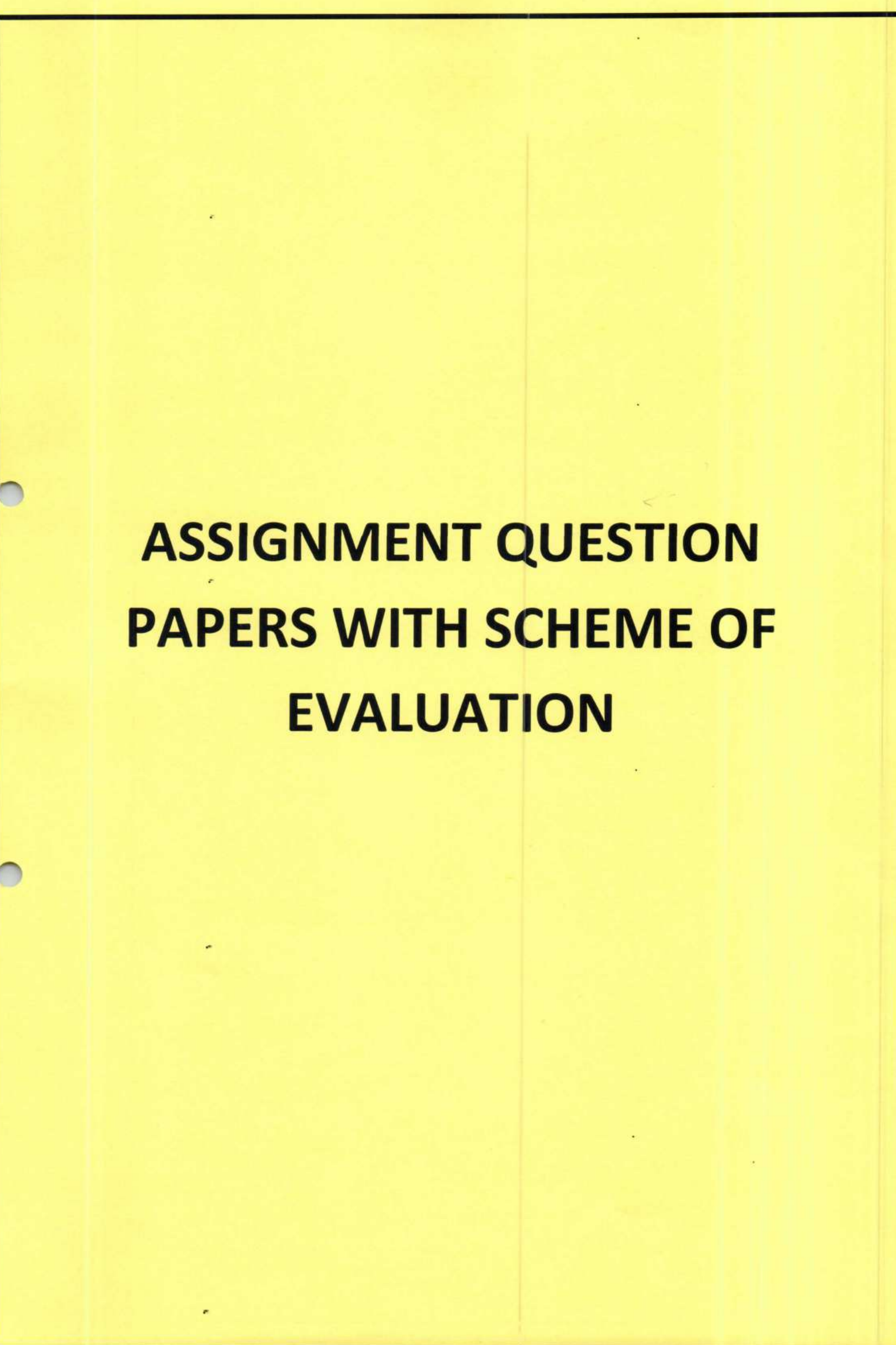
- The situation is even worse than the productivity gap indicates
- In theory, adding designers to team reduces project completion time
- In reality, productivity per designer decreases due to complexities of team management and communication
- In the software community, known as "the mythical man-month" (Brooks 1975)
- At some point, can actually lengthen project completion time! ("Too many cooks")

- 1M transistors, 1 designer=5000 trans/month
- Each additional designer reduces for 100 trans/month
- So 2 designers produce 4900 trans/month each



Summary

- Embedded systems are everywhere
- Key challenge: optimization of design metrics
  - Design metrics compete with one another
- A unified view of hardware and software is necessary to improve productivity
- Three key technologies
  - Processor: general-purpose, application-specific, single-purpose
  - IC: Full-custom, semi-custom, PLD
  - Design: Compilation/synthesis, libraries/IP, test/verification



ASSIGNMENT QUESTION PAPERS WITH SCHEME OF EVALUATION

NARASARAOPETA ENGINEERING COLLEGE (AUTONOMOUS): NARASARAOPET
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
IV B.TECH - II SEMESTER, ASSIGNMENT TEST-I, APRIL-2021

SUBJECT: EMBEDDED SYSTEM DESIGN	**DATE:** 22-04-2021
DURATION: 30 MIN	**MAX MARKS:** 10

Q.No.	Questions	Course Outcome (CO)	Knowledge Level as Per Bloom's Taxonomy	Marks
1	List out the challenges in building an embedded system.	1	Remembering (K1)	10
2	Explain the possible steps involved in build process of embedded systems.	1	Understanding (K2)	10
3	Explain about the structural units in embedded processor selected for an application.	1	Understanding (K2)	10
4	Explain the typical Embedded system architecture of digital camera.	1	Understanding (K2)	10
5	Draw the independence of processor and IC technologies with explanation	1	Understanding (K2)	10
6	List the characteristics, applications and trade-offs of embedded system	1	Remembering (K1)	10

NARASARAOPETA ENGINEERING COLLEGE (AUTONOMOUS): NARASARAOPET
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
IV B.TECH - II SEMESTER, ASSIGNMENT TEST-I, APRIL-2021

SUBJECT: EMBEDDED SYSTEM DESIGN	**DATE:** 22-04-2021
DURATION: 30 MIN	**MAX MARKS:** 10

Scheme of Evaluation

Q.No.	Questions	Course Outcome (CO)	Knowledge Level as Per Bloom's Taxonomy	Marks
1	List out the challenges in building an embedded system. Embedded System – 2 Marks List (Points min. 10) – 8 Marks	1	Remembering (K1)	10
2	Explain the possible steps involved in build process of embedded systems. Embedded System – 2 Marks Diagram Relevant – 4 Marks Steps Explanation – 4 Marks	1	Understanding (K2)	10
3	Explain about the structural units in embedded processor selected for an application. Relevant Diagram – 5 Marks Explanation Each – 5 Marks	1	Understanding (K2)	10
4	Explain the typical Embedded system architecture of digital camera. Architecture Diagram – 5 Marks Explanation Each – 5 Marks	1	Understanding (K2)	10
5	Draw the independence of processor and IC technologies with explanation Diagrams Relevant – 5 Marks Explanation – 5 Marks	1	Understanding (K2)	10
6	List the characteristics, applications and trade-offs of embedded system Embedded System – 2 Marks Characteristics – 3 Marks Applications & trade-offs – 3 + 2	1	Remembering (K1)	10

NARASARAOPETA ENGINEERING COLLEGE (AUTONOMOUS): NARASARAOPET
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
IV B.TECH - II SEMESTER, ASSIGNMENT TEST-II, MAY-2021

SUBJECT: EMBEDDED SYSTEM DESIGN	**DATE:** 13-05-2021
DURATION: 30 MIN	**MAX MARKS:** 10

Q.No.	Questions	Course Outcome (CO)	Knowledge Level as Per Bloom's Taxonomy	Marks
1	Explain the Combinational Logic with transistor and logic Gates?	1	Remembering (K1)	10
2	Construct the Combinational Logic Design?	1	Understanding (K2)	10
3	Explain about the RT-level Combinational Components?	1	Understanding (K2)	10
4	Construct the Sequential logic Design?	1	Understanding (K2)	10
5	Develop Custom-single purpose processor Design?	1	Understanding (K2)	10
6	Explain an Optimising Custom-single purpose Processor?	1	Remembering (K1)	10

NARASARAOPETA ENGINEERING COLLEGE (AUTONOMOUS): NARASARAOPET
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
IV B.TECH - II SEMESTER, ASSIGNMENT TEST-II, MAY-2021

SUBJECT: EMBEDDED SYSTEM DESIGN	**DATE:** 13-05-2021
DURATION: 30 MIN	**MAX MARKS:** 10

Scheme of Evaluation

Q.No.	Questions	Course Outcome (CO)	Knowledge Level as Per Bloom's Taxonomy	Marks
1	Explain the Combinational Logic with transistor and logic Gates? Diagrams Relevant – 5 Marks Explanation – 5 Marks	1	Remembering (K1)	10
2	Construct the Combinational Logic Design? Diagram – 5 Marks Explanation – 5 Marks	1	Understanding (K2)	10
3	Explain about the RT-level Combinational Components? RT – level Combinational – 4 Marks Explanation each – 6 Marks	1	Understanding (K2)	10
4	Construct the Sequential logic Design? Sequential Logic Design – 3 Marks Construction & Explanation – 7 Marks	1	Understanding (K2)	10
5	Develop Custom-single purpose processor Design? Custom - Single purpose processor – 4 Marks Design, Development & Explanation – 6 Marks (2+2+2)	1	Understanding (K2)	10
6	Explain an Optimising Custom-single purpose Processor? Optimising Custom-Single purpose processor – 5 Marks Explanation in depth – 5 Marks	1	Remembering (K1)	10

NARASARAOPETA ENGINEERING COLLEGE (AUTONOMOUS): NARASARAOPET
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
IV B.TECH - II SEMESTER, ASSIGNMENT TEST-III, JUNE-2021

SUBJECT: EMBEDDED SYSTEM DESIGN	**DATE:** 02-06-2021
DURATION: 30 MIN	**MAX MARKS:** 10

Q.No.	Questions	Course Outcome (CO)	Knowledge Level as Per Bloom's Taxonomy	Marks
1	List out the common memory types.	4	Remembering (K1)	10
2	Explain the INTERFACING arbitration process in embedded systems.	4	Understanding (K2)	10
3	Explain about the memory hierarchy and cache.	4	Understanding (K2)	10
4	Explain the typical multilevel bus architecture.	4	Understanding (K2)	10
5	Draw the side by side and top to bottom connect of composing memory	4	Understanding (K2)	10
6	List the advanced communication principles of embedded system	4	Remembering (K1)	10

NARASARAOPETA ENGINEERING COLLEGE (AUTONOMOUS): NARASARAOPET
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
IV B.TECH - II SEMESTER, ASSIGNMENT TEST-III, JUNE-2021

SUBJECT: EMBEDDED SYSTEM DESIGN	**DATE:** 02-06-2021
DURATION: 30 MIN	**MAX MARKS:** 10

Scheme of Evaluation

Q.No.	Questions	Course Outcome (CO)	Knowledge Level as Per Bloom's Taxonomy	Marks
1	List out the common memory types. Common memory types – 3 Marks List (all) – 3 Marks Explanation (Each) – 4 Marks	4	Remembering (K1)	10
2	Explain the INTERFACING arbitration process in embedded systems. Embedded Systems – 2 Marks Interfacing – 3 Marks Explanation – 5 Marks	4	Understanding (K2)	10
3	Explain about the memory hierarchy and cache. Memory hierarchy – 3 Marks cache – 3 Marks Explanation – 4 Marks	4	Understanding (K2)	10
4	Explain the typical multilevel bus architecture. Multilevel bus architecture – 5 Marks Explanation indepth – 5 Marks	4	Understanding (K2)	10
5	Draw the side by side and top to bottom connect of composing memory Side-by-Side Connect – 3 Marks top-to-bottom connect – 3 Marks Explanation both – 4 Marks	4	Understanding (K2)	10
6	List the advanced communication principles of embedded system Advanced Communication – 2 Marks Principles (about) – 3 Marks List and description – 5 Marks	4	Remembering (K1)	10

NARASARAOPETA ENGINEERING COLLEGE (AUTONOMOUS): NARASARAOPET
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
IV B.TECH - II SEMESTER, ASSIGNMENT TEST-IV, JUNE-2021

SUBJECT: EMBEDDED SYSTEM DESIGN	**DATE:** 23-06-2021
DURATION: 30 MIN	**MAX MARKS:** 10

Q.No.	Questions	Course Outcome (CO)	Knowledge Level as Per Bloom's Taxonomy	Marks
1	Classify about models vs languages.	5	Understanding (K2)	10
2	Illustrate the finite state machine model (FSM) with a neat representation.	5	Understanding (K2)	10
3	Relate the HCFSM and state charts	5	Understanding (K2)	10
4	Interpret the program state machine model (PSM) with neat representation.	5	Understanding (K2)	10
5	Explain about the communication and synchronization among processes.	5	Understanding (K2)	10
6	Outline the concurrent process model.	5	Understanding (K2)	10

NARASARAOPETA ENGINEERING COLLEGE (AUTONOMOUS): NARASARAOPET
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
IV B.TECH - II SEMESTER, ASSIGNMENT TEST-IV, JUNE-2021

SUBJECT: EMBEDDED SYSTEM DESIGN	**DATE:** 23-06-2021
DURATION: 30 MIN	**MAX MARKS:** 10

Scheme of Evaluation

Q.No.	Questions	Course Outcome (CO)	Knowledge Level as Per Bloom's Taxonomy	Marks
1	Classify about models vs languages. Models – 3 Marks Languages – 3 Marks Classification – 4 Marks	5	Understanding (K2)	10
2	Illustrate the finite state machine model (FSM) with a neat representation. FSM – 2 Marks Representation – 4 Marks Illustration – 4 Marks	5	Understanding (K2)	10
3	Relate the HCFSM and state charts HCFSM – 2 Marks State charts – 2 Marks Relation & Explanation – 6 Marks	5	Understanding (K2)	10
4	Interpret the program state machine model (PSM) with neat representation. PSM – 2 Marks Representation – 3 Marks Interpretation – 5 Marks	5	Understanding (K2)	10
5	Explain about the communication and synchronization among processes. Communication among processes – 2 Synchronization among processes – 2 Explanation both – 6 (3+3)	5	Understanding (K2)	10
6	Outline the concurrent process model. Concurrent process model – 5 Marks Outline & Explanation – 5 Marks	5	Understanding (K2)	10

NARASARAOPETA ENGINEERING COLLEGE, NARASARAOPET.

(AUTONOMOUS)

(Approved by AICTE New Delhi & Permanently Affiliated to J.N.T.U.K., Kakinada)

NEC

ASSIGNMENT TEST ANSWER BOOK

2021 (A) 55159

B.Tech. / M.Tech. / M.B.A. / M.C.A. / (Br.) ECE

Year: IV Semester: II Sec: C Test No.: II

Sub: ESD Date: 22-04-21

Name: V.Tulasiram

HALL TICKET NO.									
1	7	4	7	1	A	0	4	D	4

MARKS: Tens 0 Ones 7 — Marks in words: Zero Seven

Signature of the Principal   Signature of the Examiner - I   Signature of the Examiner - II

4) Embedded System:

* Embedded System is a combination of Both hardware & Software.
* Embedded system has hardware Design with Software Program to keep in following constrains:-
  * Available Memory space
  * Available Process speed
  * Power Dispation.

* Available Memory Space:

In Device maintain Available Memory space to store the Data.

Available Process speed:

In Device contain contain Process speed. Process speed is used to execute the Program fastly.

* Power Dispation:

In Embedded System Device maintain Low Power Dispation.

* Embedded system is used to Real time Applications.

Applications of Embedded System:

* Smart home
* Digital camera
* Hi Train Applications
* Military Applications

Characteristics of Embedded system:

There are three types of Embedded system characteristics.

* Single functional System
* Tightly functional System
* Reactive and Real time

Single functional system:

* In these system devices execute separate software program Repeatedly.

Case1: when the missile targetting the opponent to measure the distance, this time Embedded system act as single functional system.

Tightly functional system:

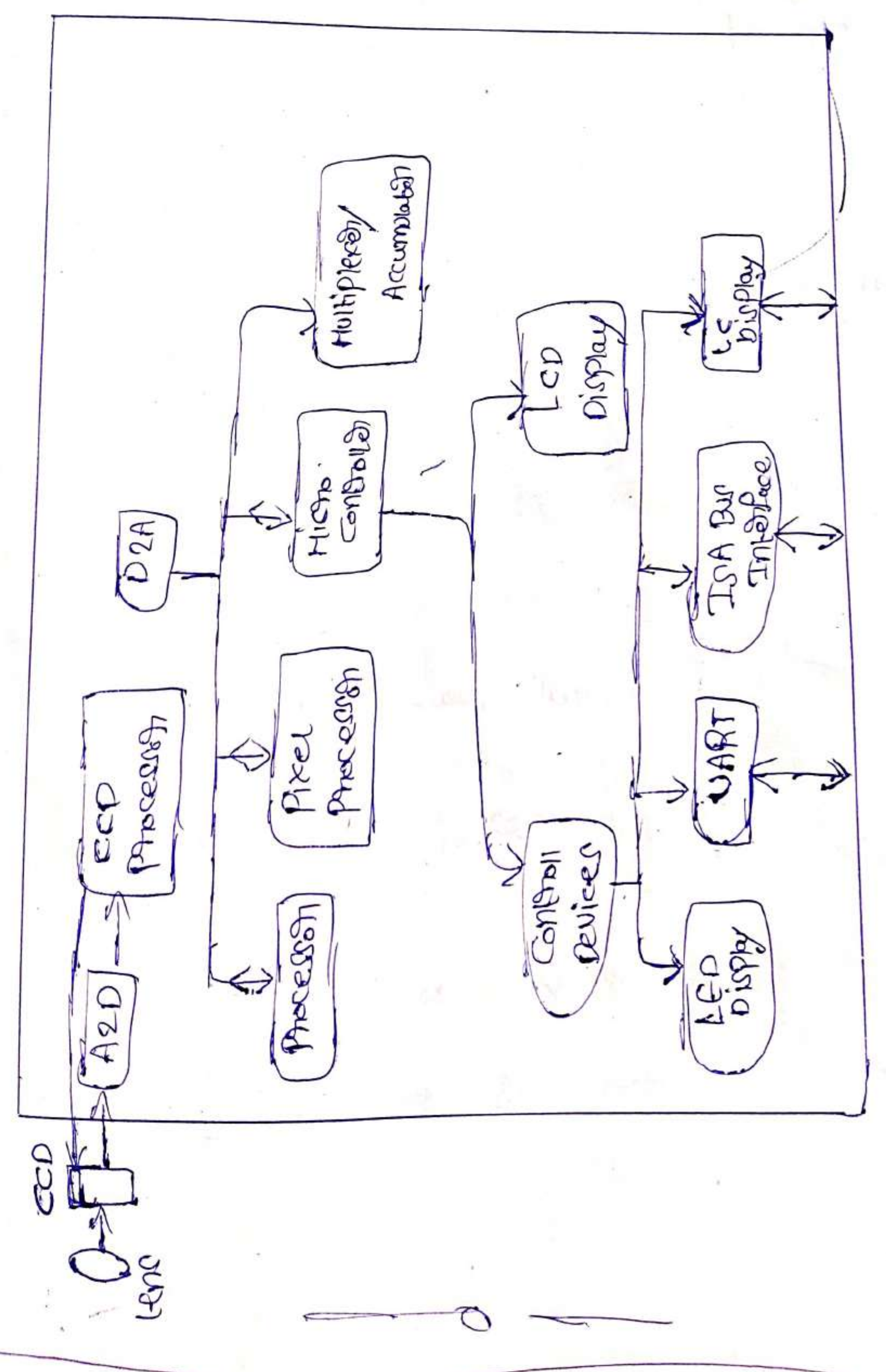
These system the hardware device follow the main constraint are cost, size, performance power dispation.

4	Explain the typical Embedded system architecture of digital camera.	1	Understanding (K2)	10

Reactive and Real time:

The Embedded system to React the changes in the hardware Device to get Results in Real time.

ii) Architecture of Digital camera:



NARASARAOPETA ENGINEERING COLLEGE, NARASARAOPET.

(AUTONOMOUS)

NEC

(Approved by AICTE New Delhi & Permanently Affiliated to J.N.T.U.K., Kakinada)

ASSIGNMENT TEST ANSWER BOOK

2021 (A) 55068

B.Tech. / M.Tech. / M.B.A. / M.C.A. / B.Tech (Br.) ECE

Year: IV Semester: II Sec: C Test No.: 01

Sub: ESD Date: 22-04-2021

Name: G. Jagadeesh chandra Bose

HALL TICKET NO. | 1 | 7 | 4 | 7 | 1 | A | 0 | 4 | G | 5 |

MARKS | 0 | 0 | Marks in words Zero Zero

Signature of the Principal Signature of the Examiner - I Signature of the Examiner - II

2. Ans.

Design specifications involved to build an embedded systems:

0

-> Design specifications are nothing but the conditions or consideration during the construction in of an embedded system

-> There are certain specifications involved in embedded systems

they are:

-> system specifications

-> Behaviourial specifications

-> Design transfer specification

-> Logical specification,

-> system

NARASARAOPETA ENGINEERING COLLEGE, NARASARAOPET.
(AUTONOMOUS)
(Approved by AICTE New Delhi & Permanently Affiliated to J.N.T.U.K., Kakinada)
NEC
ASSIGNMENT TEST ANSWER BOOK

2021 (A) 55098

B.Tech. / M.Tech. / M.B.A. / M.C.A. / B.Tech (Br.) E.C.E

Year: IV Semester: II Sec: C Test No.: 1

Sub: Embedded System Design Date: 22.04.2021

Name: M.Sri Bharathi

HALL TICKET NO.									
1	7	4	7	1	A	0	4	E	5

MARKS 07 Marks in words Tens: Zero Ones: Seven

Signature of the Principal Signature of the Examiner - I Signature of the Examiner - II

3. System:

A system is a way of working or organizing tasks at a set of rules.

Embedded System:

Embedded system is nothing but a computer hardware with a software program embedded in it.

Processor technology:

→ A processor is a combination of a controller and a data path.

→ A processor technology refers to the architecture of the embedded system.

→ we have three types in processor technology. They are

1. General purpose processor
2. Single purpose processor
3. Application specific processor.

General purpose processor:

→ The designer uses a general purpose processor for designing an embedded system that fulfils the required functionality.

→ General purpose processor mainly refers to the hardware.

→ The considerations that are to be taken are

1. placing transistors
2. Sizing transistors
3. Routing wires

Single purpose processor:

→ single purpose processor is referred to the software.

→ the designer uses a single purpose process for designing embedded system of digital circuit.

Application specific processor:

→ Application specific processor is a combination of both hardware and software.

→ The embedded system design made for application specific processor are used for business class.

General purpose processor:

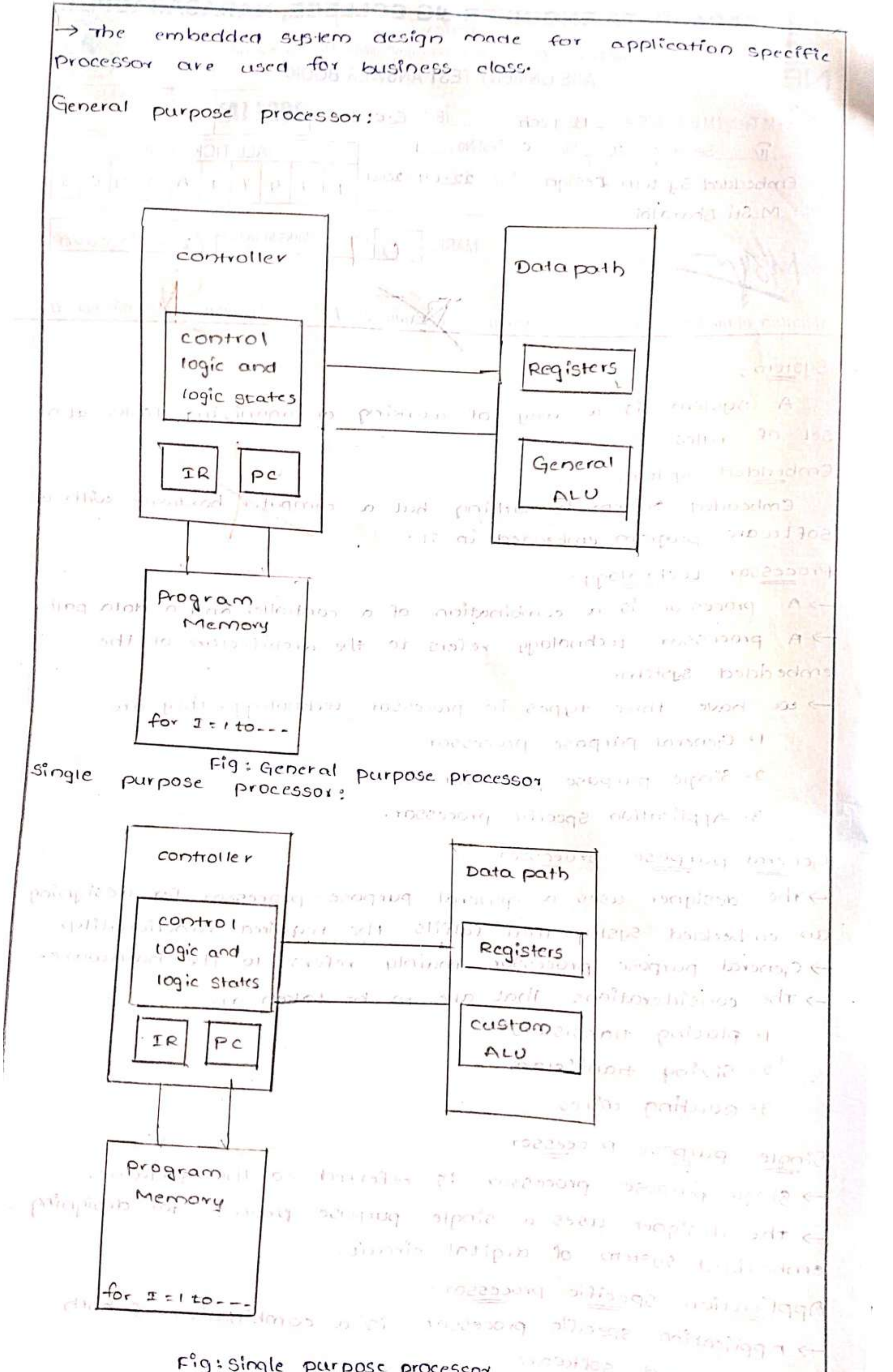


Fig: General purpose processor

Single purpose processor:

Fig: Single purpose processor

3	Explain about the Processor technology for an embedded system.	1	Understanding (K2)	10

Application Specific processor:

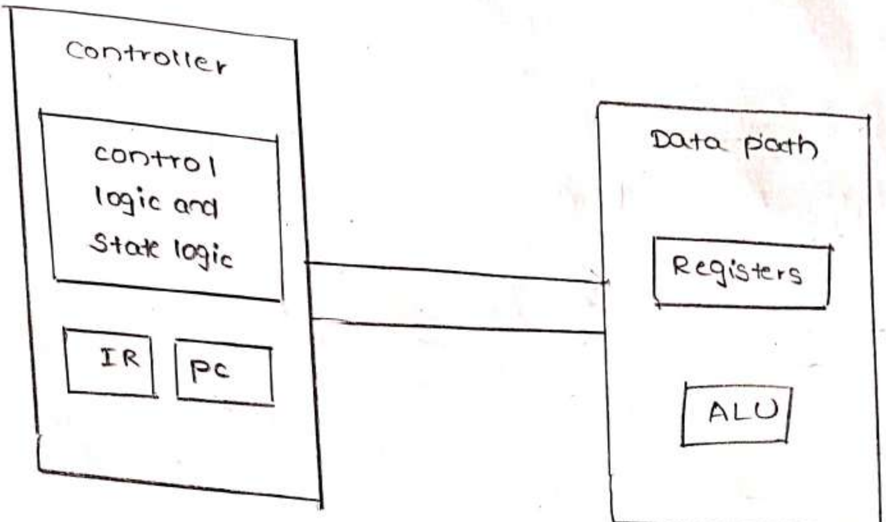


Fig: Application specific Processor

MID EXAM QUESTION PAPERS WITH SCHEME OF EVALUATION

NARASARAOPETA ENGINEERING COLLEGE (AUTONOMOUS): NARASARAOPET
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
IV B.TECH II SEMESTER, MID – I EXAMINATIONS, JULY – 2021

SUBJECT: EMBEDDED SYSTEM DESIGN DATE: 25-02-2021
DURATION: 90 MIN MAX MARKS: 30

ANSWER ALL THE QUESTIONS

Q.No.	Questions	Course Outcome (CO)	Knowledge Level as Per Bloom's Taxonomy	Marks
1	Classify the processor technologies. What are the benefits of using each of the different processor technologies?	1	Analyze (K4)	10
2	a. What is a single purpose processor? What are the benefits of choosing a custom single purpose processor? b. Distinguish between combinational and sequential logic.	2	Analyze (K4)	10
3	a. Illustrate how program and data memory fetches can be overlapped in a Harvard architecture b. Discuss about the following i. Pipelining ii. Interrupts	3	Analyze (K4)	10

NARASARAOPETA ENGINEERING COLLEGE (AUTONOMOUS): NARASARAOPET
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
IV B.TECH II SEMESTER, MID – I EXAMINATIONS, JULY – 2021

SUBJECT: EMBEDDED SYSTEM DESIGN — DATE: 25-02-2021
DURATION: 90 MIN — MAX MARKS: 30

Scheme of Evaluation

Q.No.	Questions	Course Outcome (CO)	Knowledge Level as Per Bloom's Taxonomy	Marks
1	Classify the processor technologies. What are the benefits of using each of the different processor technologies? Processor Technologies – 3 Marks Benefits – 2 Marks Different processor technologies & classification – 5 Marks	1	Analyze (K4)	10
2	a. What is a single purpose processor? What are the benefits of choosing a custom single purpose processor? b. Distinguish between combinational and sequential logic. a. Single purpose processor – 2 Benefits – 3 b. Difference between the combination & sequential – 5 (min. 6)	2	Analyze (K4)	10
3	a. Illustrate how program and data memory fetches can be overlapped in a Harvard architecture b. Discuss about the following i. Pipelining ii. Interrupts a. Illustration – 2 Marks Steps/procedure – 3 Marks b. Pipelining – 2 Marks Interrupts – 3 Marks	3	Analyze (K4)	10

NARASARAOPETA ENGINEERING COLLEGE (AUTONOMOUS): NARASARAOPET
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
IV B.TECH II SEMESTER, MID –II EXAMINATIONS, JULY – 2021

SUBJECT: EMBEDDED SYSTEM DESIGN DATE:10-07- 2021
DURATION: 90 MIN MAX MARKS: 30M

ANSWER ALL THE QUESTIONS

Q.No.	Questions	Course Outcome (CO)	Knowledge Level as Per Bloom's Taxonomy	Marks
1	Construct Memory Hierarchy? How does Cache Operates? Discuss the Cache mapping techniques? List its Merits and Demerits?	4	Apply (K3)	10
2	Describe Concurrent process model? and Develop Hierarchical State machine with Elevator's control unit?	5	Apply (K3)	10
3	What is Hardware Software Co-simulation? and Distinguish various IC technologies, and discuss the benefits of using them?	6	Apply (K4)	10

NARASARAOPETA ENGINEERING COLLEGE (AUTONOMOUS): NARASARAOPET
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
IV B.TECH II SEMESTER, MID –II EXAMINATIONS, JULY – 2021

SUBJECT: EMBEDDED SYSTEM DESIGN | DATE:10-07- 2021
DURATION: 90 MIN | MAX MARKS: 30M

Scheme of Evaluation

Q.No.	Questions	Course Outcome (CO)	Knowledge Level as Per Bloom's Taxonomy	Marks
1	Construct Memory Hierarchy? How does Cache Operates? Discuss the Cache mapping techniques? List its Merits and Demerits? Memory Hierarchy – 2 Marks Cache operation – 3 Marks Mapping techniques – 2 Marks Merits and Demerits – 3 Marks	4	Apply (K3)	10
2	Describe Concurrent process model? and Develop Hierarchical State machine with Elevator's control unit? Concurrent process model – 2 Description – 3 Hierarchical State machine – 2 Elevator's control unit – 3	5	Apply (K3)	10
3	What is Hardware Software Co-simulation? and Distinguish various IC technologies, and discuss the benefits of using them? Hardware Software Co-simulation – 3 Marks Various IC technologies – 3 Marks Discussion – 4 Marks	6	Apply (K4)	10

NARASARAOPETA ENGINEERING COLLEGE, NARASARAOPET.
(AUTONOMOUS)
(Approved by AICTE New Delhi & Permanently Affiliated to J.N.T.U.K., Kakinada)
NEC
MAIN ANSWER BOOK

B.Tech. / M.Tech. / M.B.A. / M.C.A. / B.tech (Br.) ECE 2021 (A) 13095
Year: IV Semester: II Sec: D Mid No.: I HALL TICKET NO.
Sub: EMBEDDED SYSTEM DESIGN Date: 07/7/2021 | 1 | 6 | 4 | 7 | 5 | A | 0 | 4 | 1 | 5 |
Name: P. Siva Tens Ones

MARKS 1 8 Marks in words one eight

Signature of the Principal Signature of the Examiner - I Signature of the Examiner - II

2. a categorize single purpose processor? what are the benefits of choosing a custom single purpose processor?
b. Distinguish between combinational and sequential logic

6+6+6 = 18

A. A custome singe purpose processor is the performed single program single data, enution of the proces is the two types of mainer are involued in the custome single purpose processor

i) custome single purpose processor
ii) sequential logic gates
iii) combinational logic gates

i) custome single purpose processor:

→ the custome single purpose processor is designed by basic electronics components of transistores and logic gates
ii) the basic transistors are used to the circuit combination of or pin as the ON and Off switch
iii) the transistor used of CMOS transistor used this process

iv) The metal metallic oxide semiconductor is used this circuit design process

sequential logic process :- The sequential logic processes designed by the electronic components used to combinational logic & gates

ii) The combination logic used to basic logic components

i) Inverter

ii) AND gate

iii) NAND gate

iv) XOR gate

v) OR gate

vi) counters

iii) This combinational logic gates used to another electronic components of registers and counters

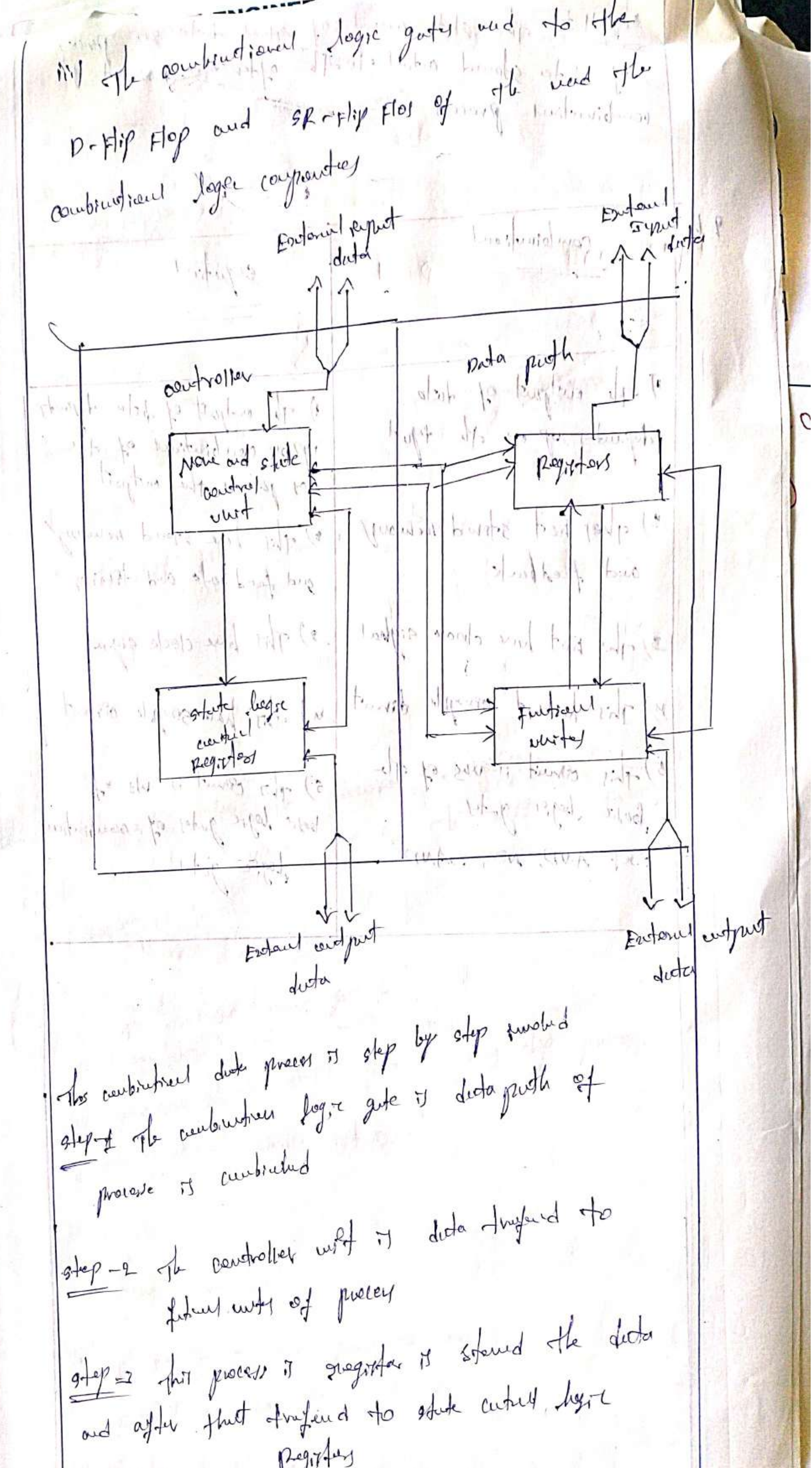
iv) The register of shift register process used to sequential logic process of data shifted to counters

v) The counters are stored the data after transferred to the controller process.

combinational logic gates :-

i) The combinational logic gates are mostly used of electronic or digital circuit design of Flip Flop

ii) The Flip Flop used of the output data is the depend the upon combination of input data process



iii) The combinational logic gates used to the D-Flip Flop and SR-Flip Flops of the used the combinational logic components

This combinational data process is step by step involved

step-1 The combination logic gate is data path of process is combinated

step-2 The controller unit is data transfered to functional units of process

step-3 this process is register is stored the data and after that transfered to state control logic Registers

step-4 :- The state control of actual data on present of data stored actual to the after to maintain the combinational process

2 b

combinational	sequential
1) The output of data depends up on the input	1) The output of data depends upon combination of the as per as the output
2) This not stored memory and feedback	2) This have stored memory and feedback and timing
3) This not have clock signal	3) This have clock signal
4) This have not simple circuit	4) This have simple circuit
5) This circuit is uses of the basic logic gates Ex :- AND, NOR, NAND	5) This circuit is uses of basic logic gates of combinational logic gates

3 Explain how program and data memory of they can be overlapped in Harvard architecture

b) Inter the following

i) Pipelining ii) Interrupts

A. Harvard interrupts : Interrupts causes the processor to stopped continue of the main program and instead Jump to an interrupts the service rotation of the full fill of the interrupts

4 List and define the processor technologies what are the benefits of using each of the different processor technologies :

A. List the processor technologies :

The processor of the single program, single chip process of the program entries list of the

i) single

ii) Tightly combined

iii) Realtime time and Real time

iv) Digital camera

v) Visible of Ad hoc of architecture

i) single embedded system :

The single embedded system is the single program and single chip. single process of the program system of single purpose of embedded system

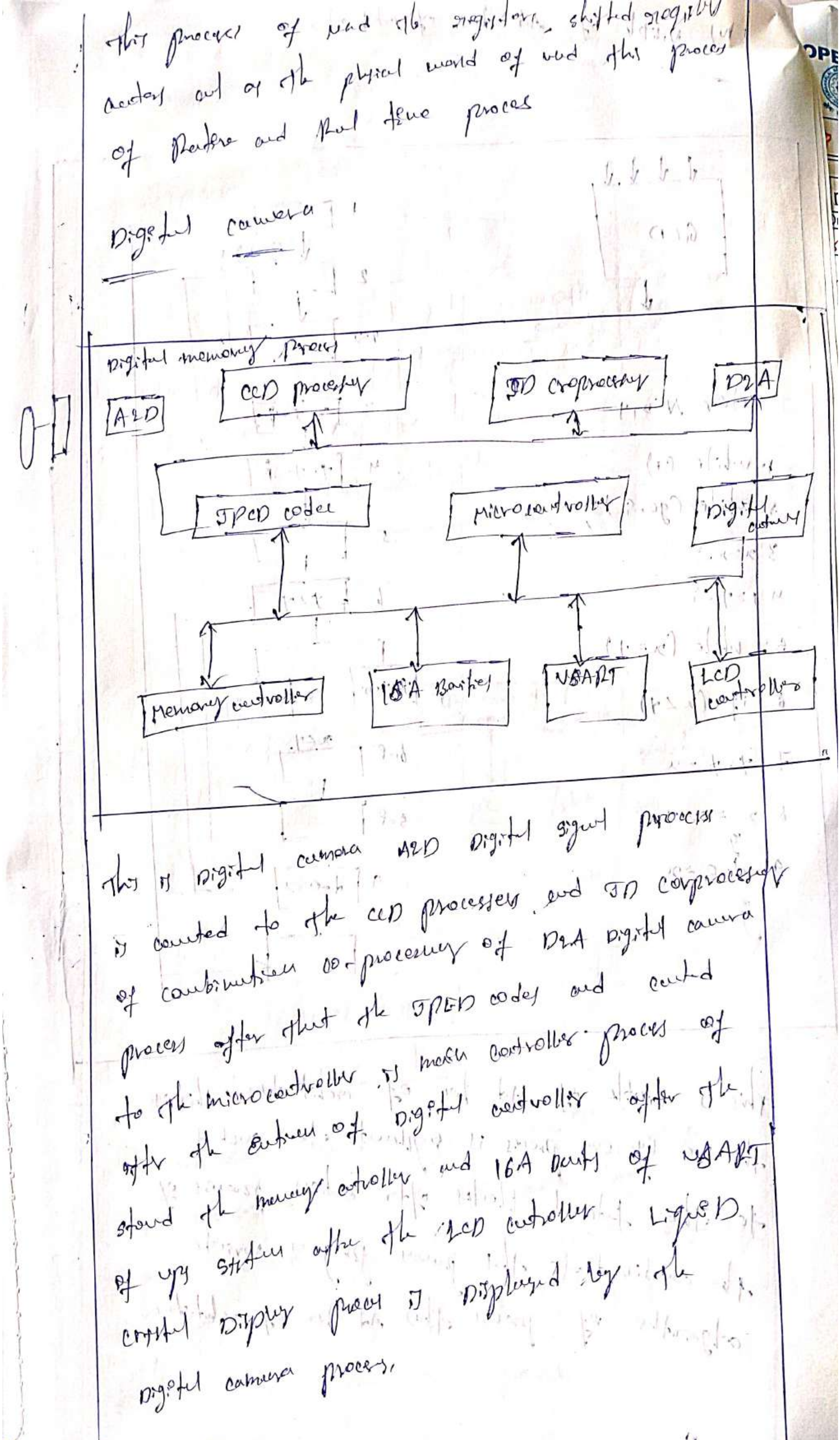
this is the simple design of digital circuit and easy program, limited process and simple logic gates of the performed of this process

ii) Tightly combined :

The tightly combined process of the digital Embedded system this process of the various of the program and many process of the program design and controller process of digital data and the after transfer to the controller, and stored the data and remaining of the entire process of used the register and counters and shifted register the register are used to the stored the data and after entire process of controller. the controller is manipulated to the entire process

Reactive time and real time :

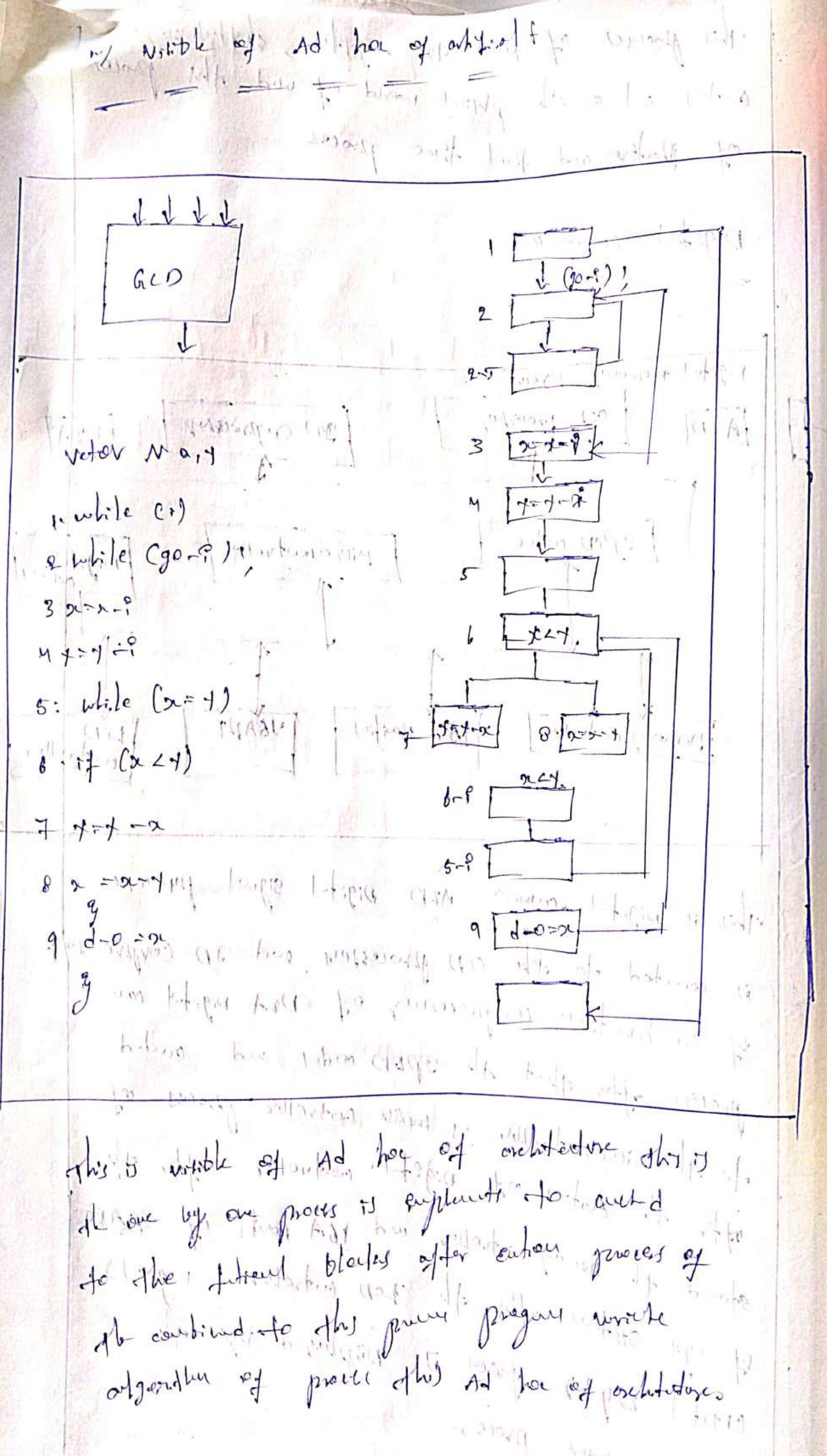
The reactive and real time process is the combinatory logic gates of the various digital program process of the SR gates and D-Flip Flop are used to the reactive and real time process



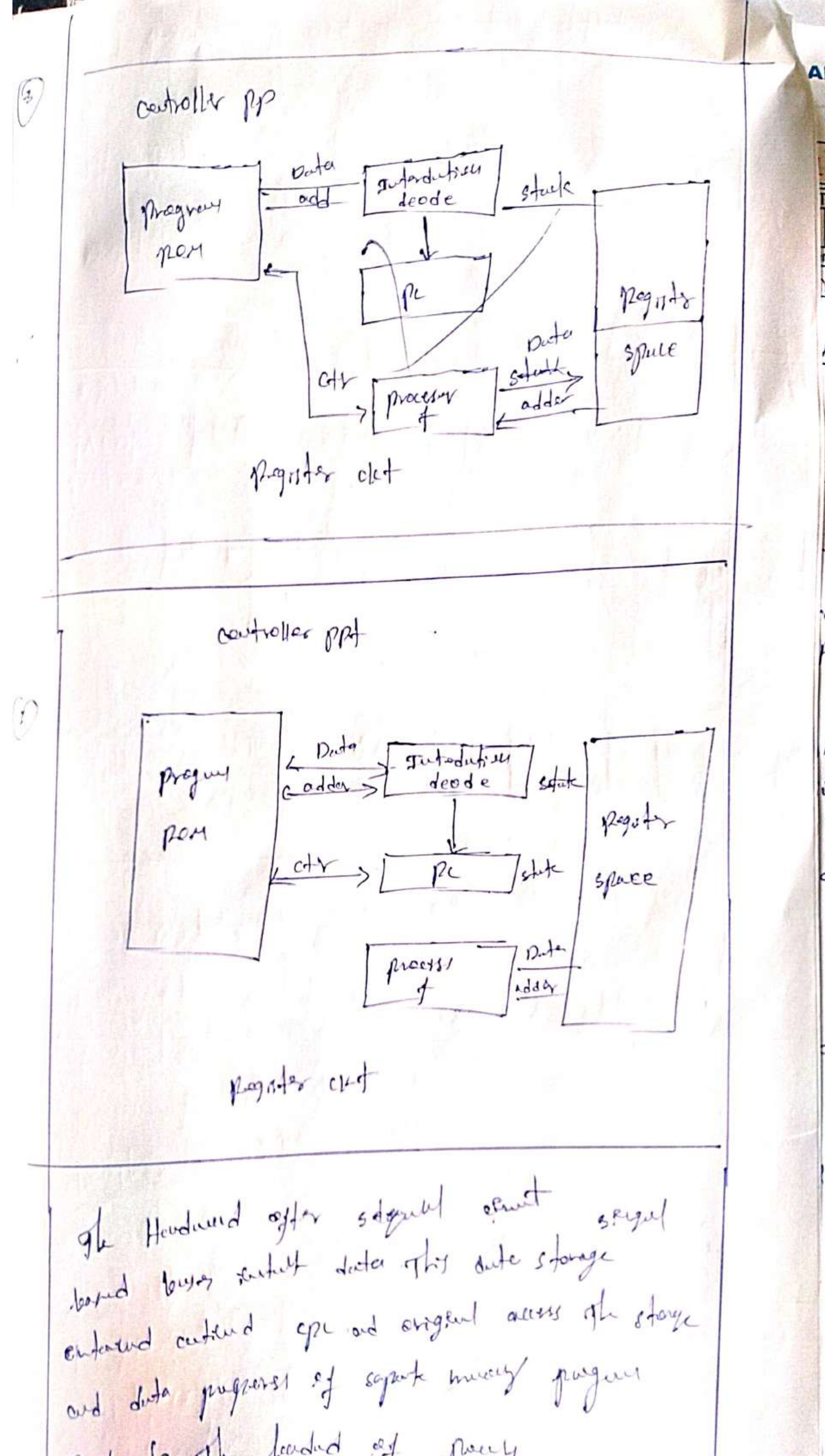
this process of read the registers, shifted register readers out of the physical world of read this process of Readers and Real time process

Digital camera :

This is digital camera A2D digital signal processor is connected to the CCD processor and JD coprocessor of combination co-processing of D2A digital camera of combination co-processing of D2A digital camera of JPED codes and connected process after that the JPED codes and connected process after that the microcontroller is main controller process of to the microcontroller is main controller process of after the entrance of digital controller after the stored the memory controller and ISA ports of USART. of up status after the LCD controller. Liquid crystal Display process is Displayed by the digital camera process.



this is variable of Ad hoc of architecture this is the one by one process is implements to attend to the different blocks after each process of the combined to this process program write algorithm of process this Ad hoc of architecture


controller pp
Data
add
decode
state
Program
ROM
PC
Register
space
Data
adder
ctr
processor
Register clet
controller ppt
Data
adder
decode
state
Program
ROM
Register
space
ctr
PC
state
process
Data
adder
Register clet

NARASARAOPETA ENGINEERING COLLEGE, NARASARAOPET.
(AUTONOMOUS)
(Approved by AICTE New Delhi & Permanently Affiliated to J.N.T.U.K., Kakinada)
NEC
MAIN ANSWER BOOK

B.Tech. / M.Tech. / M.B.A. / M.C.A. / B-Tech (Br.) ECE 2021 [A] 13070

Year: 4th Semester: 2nd Sec: D Mid No.: 01

HALL TICKET NO. 1 7 4 7 1 A 0 4 I 7

Sub: Embedded system design Date:

Name: K. Rakesh

MARKS 2 0 Marks in words Two zero

Signature of the Principal | Signature of the Examiner - I | Signature of the Examiner - II

8+6+6=20

3(b):- pipelining :-

* pipelining is a common way to increase the instruction throughhoutput of a microprocessor. we first make a simple analog of two people approching the chore of washing and drying 8 dishes, In one approach, the first person washes all 8 dishes and then the second person dries all 8 dishes, Assuming 1 minute per dish per person this approach requires 16 min The approach is clearly in efficient since at any time only one person is working and the other is idle. Obviously a better approach is for the second person to begin drying the first dish immetiatly after it has been washed.

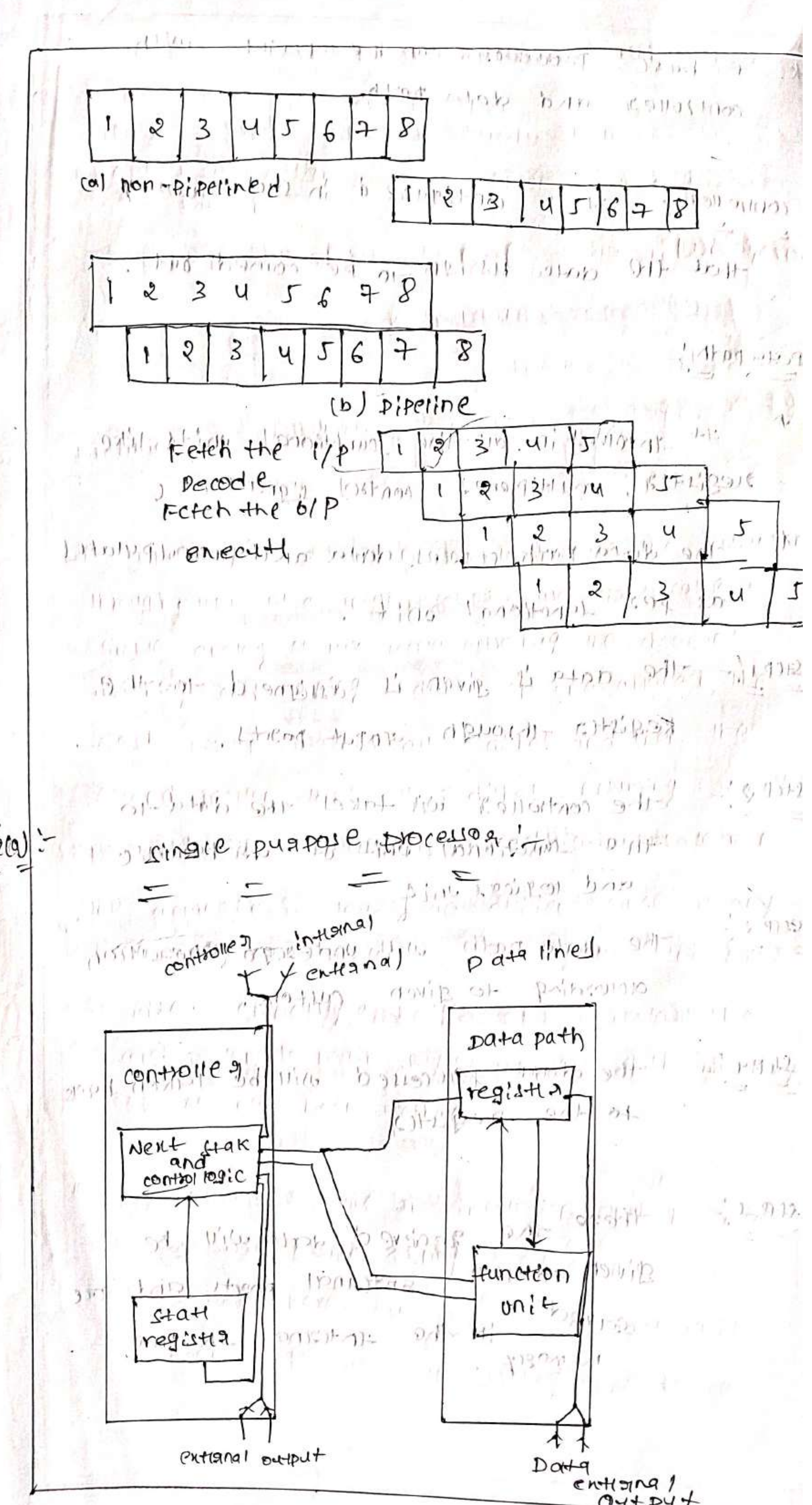
* Each dish is like an instruction and the two tasks of washing and drying are like the first stages used above. By using a separate unit for each stage we can pipeling unit decodes

it while the instruction fetch unit simultaneously fetches the next instruction.

* Branches pose a problem for pipeline since we don't know the next instruction until the current instruction has reached the execute stage.

(ii) interrupts :-

An interrupt causes the processor to suspend execution of the main programe and instead jump to an interrupt service routine that fulfills a special short term processing need. In particular the processor resume execution of the main program by restoring the PC.
the programmer should be aware of the type of interrupt supported by the processor and must write ISR's when neccessary. the assembly language programmer places each ISR at a specific address in program memory.
the structed language programmer must do also some compilers allows a programmer to force a procedure to start at a particular memory location while recognize pre-defined named for particular ISR's.


1 2 3 4 5 6 7 8
(a) non-pipelined
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
(b) pipeline
Fetch the i/p
Decode
Fetch the o/P
execute
2(a) :-
single purpose processor :-
controller
internal
external
Data lines
Data path
controller
register
Next state and control logic
function unit
state register
external output
Data
external Output

* A basic processor can be built with controller and data path.

Controller :- It is configured in such a way that the data ussign to be comed out.

Data path :-

* It involves in all the functional units like register, multiplexer, control signals etc.
* The data path stores data and manupulates as per functional unit.

Step 1 :- The data is given is assigned to the Register through input ports.

Step 2 :- The controller will takes the data to the functional unit of arthametic and logical unit

Step 3 :- The data path will undergo operation according to given custom.

Step 4 :- The data processed will be taken back to the register

Step 5 :- There the recieved data will be given to the external ports and are saved in the internal memory, in the

Sol:-

processor technology :-

There are 3-types of processor technology

They are .

1) General purpose processor technology

* Defines about software technology used in embedded system design

This includes factors such as

(a) program memory
(b) Data memory
(c) ALU

(2) single purpose processor technology

Defines about hardware tech used in ESD

This includes the factors such as

(a) performance (high)
(b) size (small)
(c) cost (Low)
(d) power (low)

(3)

Application processor technology

* It defines about how the h/w & s/w used in ESD.

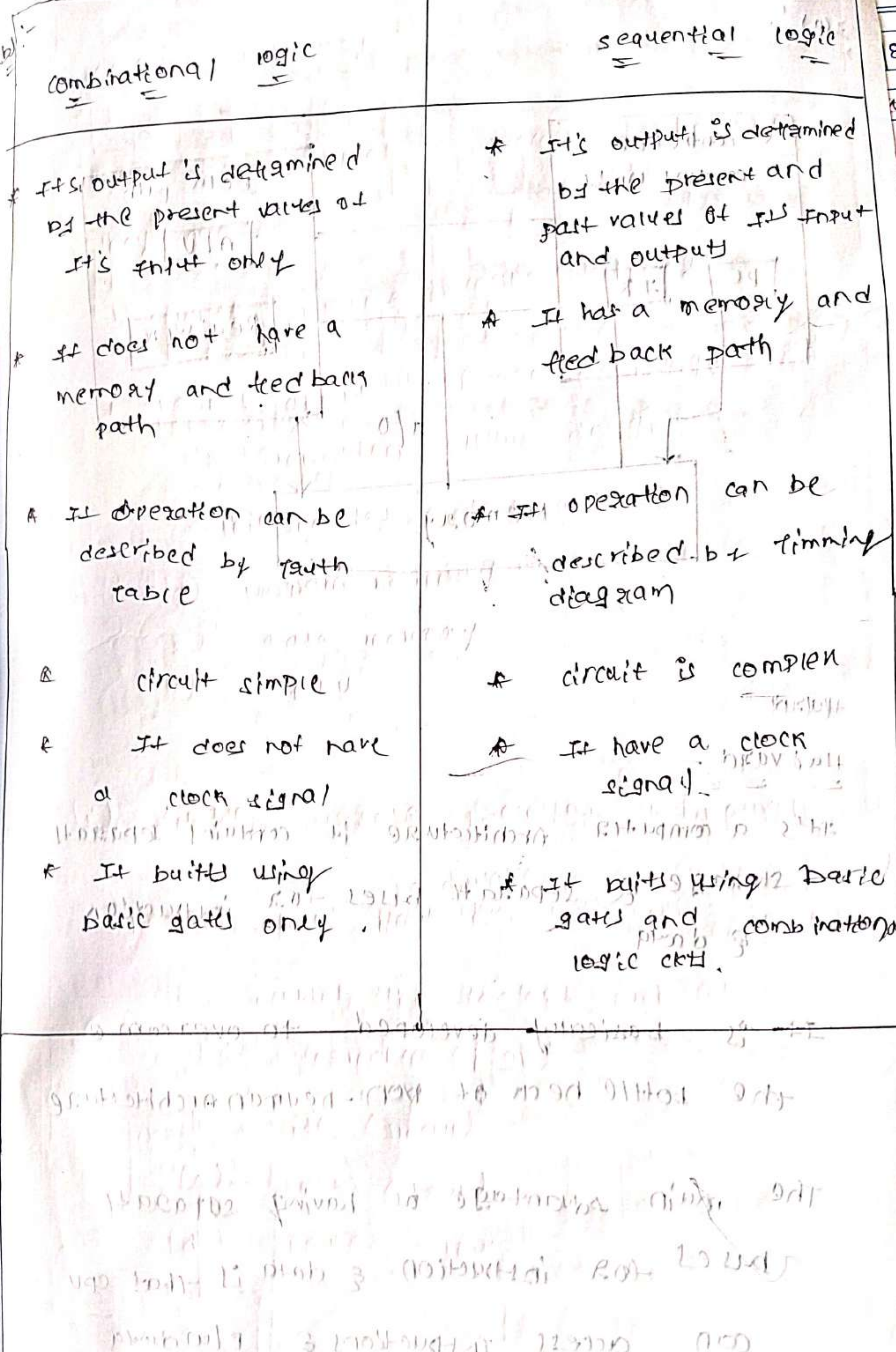
This includes factors such as

(a) performance

(b) flexibility

(c) ASIP (micro controller)

(b):-



combinational logic	sequential logic
* It's output is determined by the present values of it's input only	* It's output is determined by the present and past values of its input and outputs
* It does not have a memory and feedback path	* It has a memory and feed back path
* It operation can be described by truth table	* It operation can be described by timming diagram
* circuit simple	* circuit is complex
* It does not have a clock signal	* It have a clock signal.
* It builts using basic gates only.	* It builts using basic gates and combinational logic ckt.

3(a) :-

Controller
control Unit

PC | IR

Data path

ALU

Register

I/O

Memory

~~Harard~~

Hardvard :-

It's a computer Architecture it contains separate storage & separate buses for instruction & data

It is basically developed to overcome the bottle neck of Von-Neuman Architecture

The main Advantage of having separate buses for instruction & data is that CPU can access Instructions & r/w data at same time.

* In harvard architecture the program memory space is distinct from data memory space. such architecture requires two data connections.

* It is a simple to imagine that in above case if all the above states are executed one after another the execution time of instruction will be longer than when it is pipeline.

NARASARAOPETA ENGINEERING COLLEGE, NARASARAOPET.
(AUTONOMOUS)
(Approved by AICTE New Delhi & Permanently Affiliated to J.N.T.U.K., Kakinada)
NEC
MAIN ANSWER BOOK

B.Tech. / M.Tech. / M.B.A. / M.C.A. / B.Tech (Br.) ECE 2021 (A) 13074
Year : IV Semester : 2 Sec : D Mid No.: 01
HALL TICKET NO. 1 7 4 7 1 A 0 4 I 8
Date : 07/07/2021
Sub : ESD
Name : K. Priyanka.
Tens Ones
MARKS 3 0 Marks in words Three zero

Signature of the Principal Signature of the Examiner - I Signature of the Examiner - II

1. Processor Technology :-

The processor technology defines the hardware and Software systems used in a processor.

*They are of three types. They are:

a) General purpose processor.
b) Single purpose processor.
c) Application specific processor.

10 + 6 + 4 + 6 + 4

a) General purpose processor :-

This processor defines the hardware components used in the embedded system.

*This defines about the following factors

(i) Maintaince
(ii) Time to market

= 30

(i) Program memory
(ii) Data memory
(iii) ALU.

2.a) Single purpose processor :-

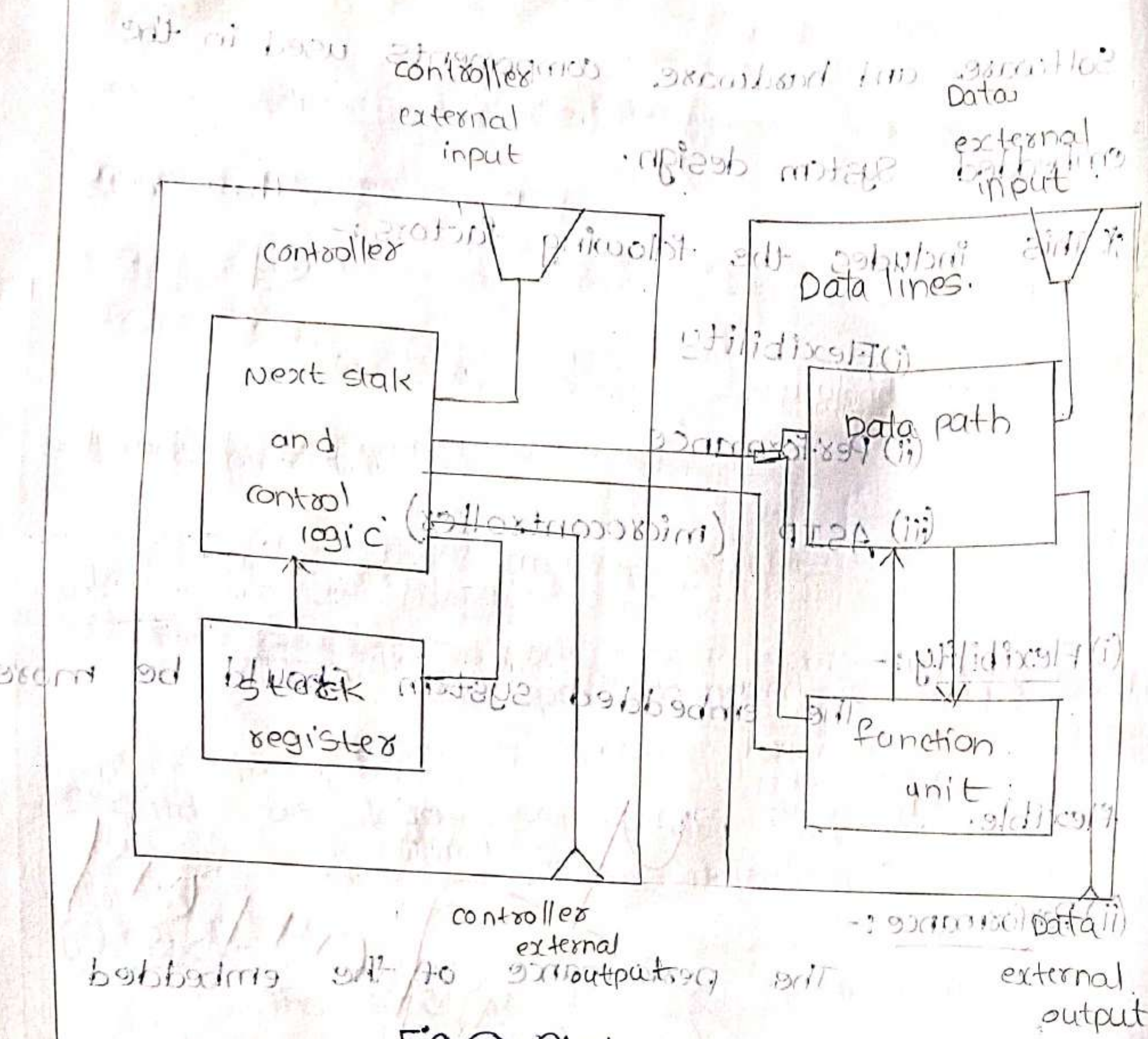


Fig @ Single purpose processor.

* The basic processor can be built with Data path and processor.

* The custom single purpose can be used using controller and Data path.

Controller :-

The controller is a device which controlls the inputs and outputs.

Data path :-

The data path is the process of sending data to the ALU and controller.

Register :-

The registers are used to store data.

Step 1 :-

The input is given to the controller.

Step 2 :-

The controller takes the input and Sends to the ALU.

Step 3 :-

The operations are done in ALU.

Step 4 :-

The output is stored in the register.

Step 5 :-

The output is send to controller external output.

2) b) Combinational Logic :-

The combination logic is a device which are having transistors and Logic gates in it.

Transistor :-

The transistor is a device which performs basic operations.

Logic Gates :-

The Logic Gates used in combinational Logic design is

(i) AND
(ii) OR
(iii) NOR
(iv) XOR
(v) XNOR
(vi) NAND

* They are used to perform logical operations like Addition, subtraction etc.

Sequential Logic :-

The flipflops are used in the sequential logic design.

* The D flipflop & SR flipflop are used in basic

* Countors :-

The counters are the devices which is used to shift the value.

* They are of two types:- They are

(i) upward counter

(ii) Downward counter

(i) Upward counter :-

Increases the value by "1".

(ii) Downward counter :-

Decrement the value by "1".

* The upward counter increments by "1" the downward counter decrements by "0".

3) a) Harvard architecture :-

* It is a computer Architecture contains separate storage & separate buses for instruction and data.

* It is basically developed to overcome the bottle neck of non-neumann Architecture.

* The main advantage of having separate buses for instruction & data is that CPU can access instructions & Read/write data at same time.

* It is a simple to imagine that in above case all the above states are executed "one after another".

ALU

Instruction memory

control unit

Data memory

I/O

Fig: Architecture of Harvard.

b) (i) Pipelining:-

Pipelining is method of performing the operation quickly without having wasted time.

* It performs (i) fetch
(ii) Decode
(iii) execute.

* For example if we take dish washing as a example

* There are eight plates which are to be washed and dried by two persons. The eight plates takes 8 minutes to wash.

* The eight plates are dried by another person in another eight minutes.

* The total time consumed is 16 minutes.

* But if we use pipelining method after washing the first plate the second plate is dried.

* So the time consumed is reduced to 9 minutes

* As the same way the program execution as reduced.

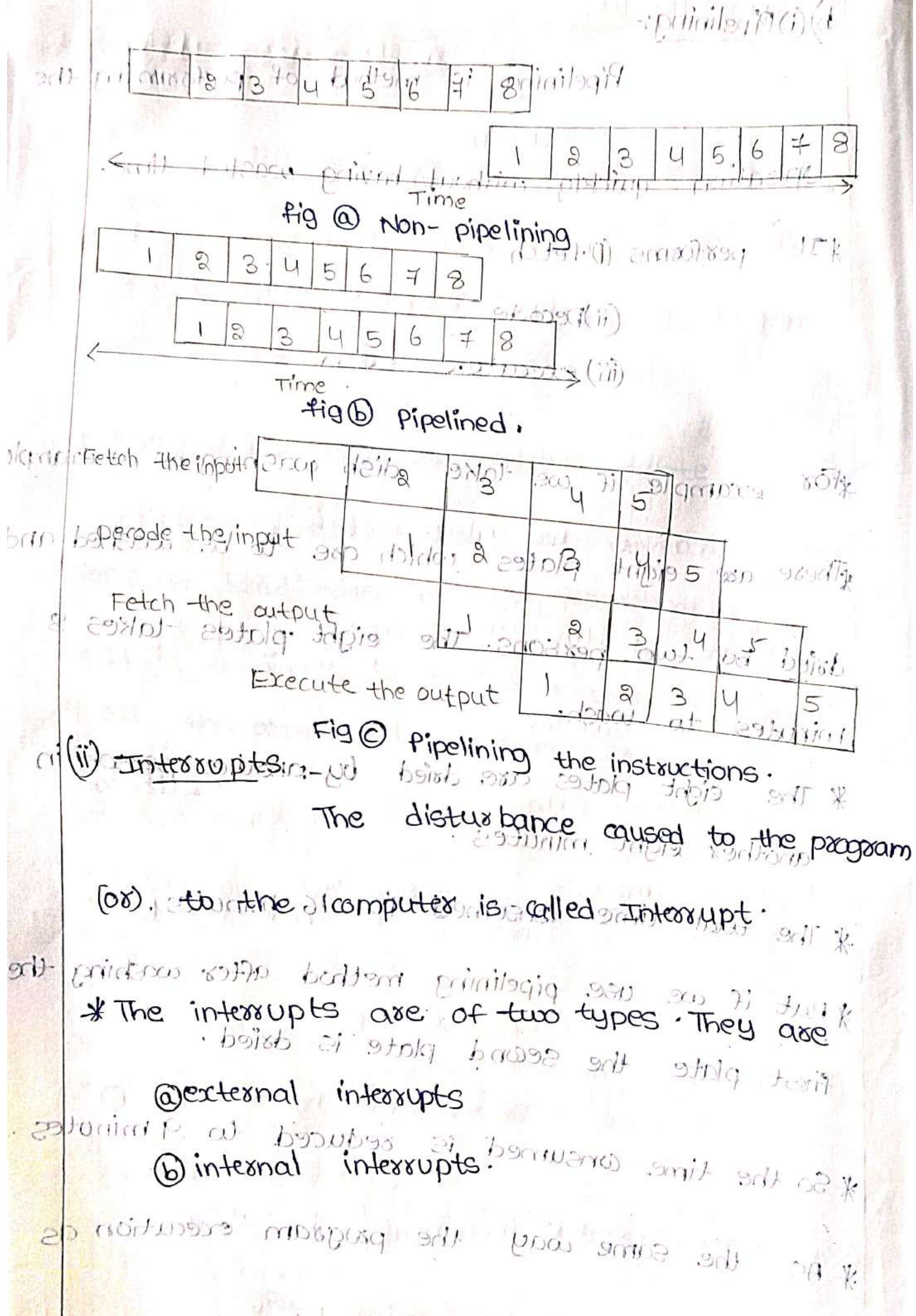


fig ⓐ Non- pipelining

fig ⓑ Pipelined.

Fig ⓒ Pipelining the instructions.

(ii) Interrupts:-

The disturbance caused to the program (or) to the computer is called Interrupt.

* The interrupts are of two types. They are

ⓐ external interrupts

ⓑ internal interrupts.

(a) external interrupts:- The disturbance caused by the external source.

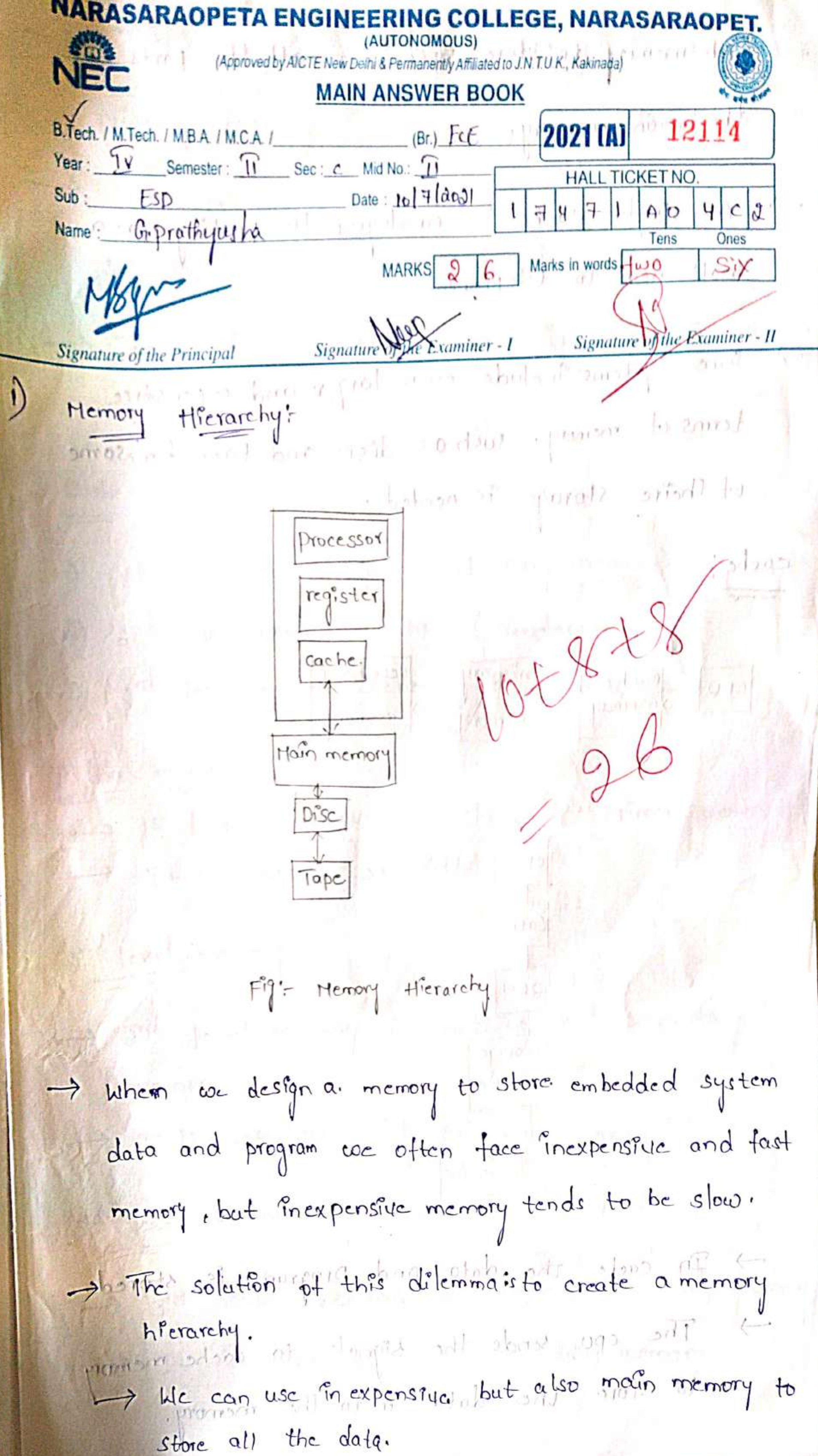
(b) internal interrupts:- The disturbance caused by the computer internally.

* The interrupts are caused internally by running multiple programs.

* The interrupt is the disturbance caused by program to the computer.

* The example for interrupts is the ALU.

* It can be stopped using ISR (Interrupt service Routine).



NARASARAOPETA ENGINEERING COLLEGE, NARASARAOPET.
(AUTONOMOUS)
(Approved by AICTE New Delhi & Permanently Affiliated to J.N.T.U.K., Kakinada)
MAIN ANSWER BOOK

2021 (A) 12114

B.Tech. / M.Tech. / M.B.A. / M.C.A. / (Br.) ECE
Year: IV Semester: II Sec: C Mid No.: II
Sub: ESD Date: 10/7/2021
Name: G. Prathyusha

HALL TICKET NO.

1	7	4	7	1	A	0	4	C	2

MARKS 2 6 Marks in words: Tens two, Ones Six

Signature of the Principal   Signature of the Examiner - I   Signature of the Examiner - II

1) Memory Hierarchy:

Fig:- Memory Hierarchy

10+8+8 = 26

→ When we design a memory to store embedded system data and program we often face inexpensive and fast memory, but inexpensive memory tends to be slow.

→ The solution of this dilemma is to create a memory hierarchy.

→ We can use inexpensive but also main memory to store all the data.

→ Each memory to store copies like all the parts of the main memory.

→ All single cache is analogous to posting on a wall near to telephone.

→ Some systems include even larger and expensive terms of memory. such as disk and tape for some of theire storage is needed.

cache:-

CPU ⇄ cache memory ⇄ primary memory ⇄ secondary memory.

CPU
|
Cache
|
RAM
|
Storage device
|
I/o device

→ In cache the data and program is stored.

→ The cpu sends the signals to cache memory to store the data in to the memory.

→ Then the cache memory sends it to the primary memory.

→ The primary memory is temperory but the secondary memory is permanent.

→ The final data and program is stored in secondary memory.

Cache mapping techniques:

1) 1st level memory cache (primary memory)
2) 2nd level memory cache (secondary memory)
3) 3rd level memory cache (main memory)

1st level memory:

→ 1st level memory is nothing but primary memory.
→ It is associated with cpu

2nd level memory:

→ 2nd level memory is nothing but secondary memory.
→ It is associated with primary memory.

3rd level memory:

→ 3rd level memory is nothing but main memory.
→ It is associated with secondary memory.

2) concurrent process model:-

pseudo code:-

```
X=Readx();
Y = Ready();
call concurrently;
    Print Helloworld (x); and
    print HOW ARE YOU (Y);

print Hello World(x).
while(1) {
 print ("Hello World";
 delay(x);
}

Print HOWARE YOU(x)
    while(1) {
        print "HOW ARE YOU".
        delay(Y);
```

sample i/p & o/p:-

Enter x: 1
Enter Y: 2.

Hello World (time =1s)
Hello World (time =2s)
HOW ARE YOU (time=3s)
Hello World (time =4s)
HOW ARE YOU (time=5s)

→ In concurrent process model the pseudo code is compiled for sample output.

→ Here 'x' can reads the data in HelloWorld and 'y' can reads the data in HOW ARE YOU.

→ The while (1) is true statement it means that the loop is continuously repeated.

→ It is an Infinite loop and endless process.

→ In concurrent process model the taken to print one statement is one second.

→ And second can take 2 seconds, third statment takes 3 seconds.

→ At the output if you enter x then it print first statement i.e, Hello World.

→ At the output if you enter y then it print second statement i.e, HOW ARE YOU.

3) Ic technologies:-

There are various types of Ic technologies. They are:-

(i) Full custom (VLSI) Ic technology:-

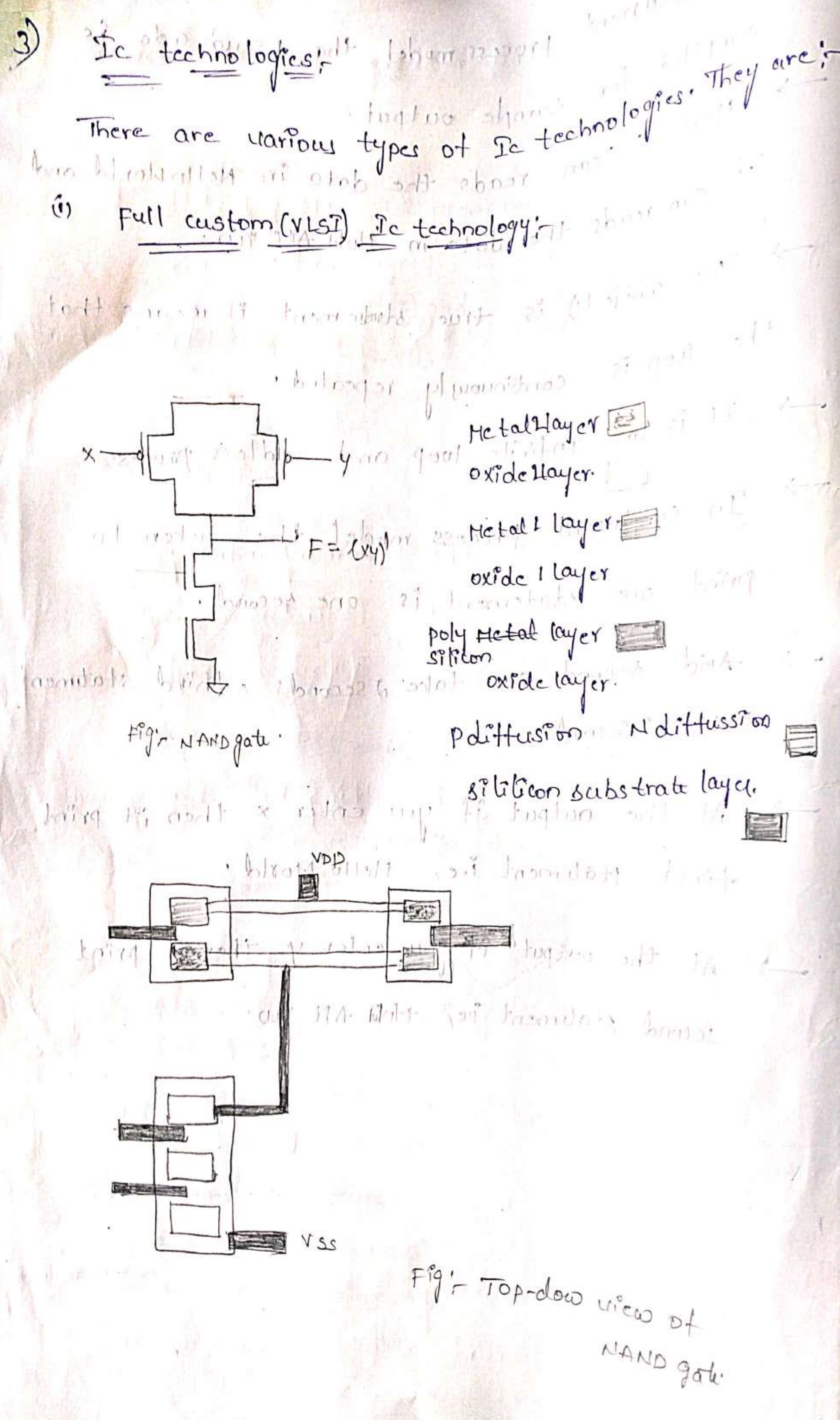


Fig:- NAND gate.

Fig:- Top-dow view of NAND gate.

semi cqustom (ASCI) Ic technologies:-

(ii) → standard cell semi custom:

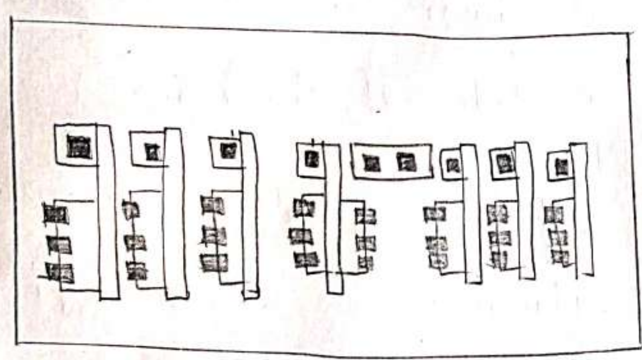
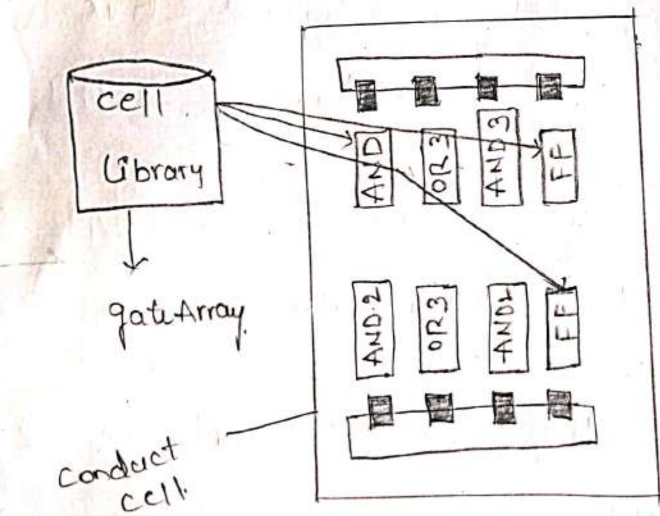


Fig:- semicustom Ic technology

→ Gate Array semi custom Ic technology:-



NARASARAOPETA ENGINEERING COLLEGE, NARASARAOPET.
(AUTONOMOUS)
(Approved by AICTE New Delhi & Permanently Affiliated to J.N.T.U.K., Kakinada)

NEC

MAIN ANSWER BOOK

2021 (A) 12017

B.Tech. / M.Tech. / M.B.A. / M.C.A. / B.tech (Br.) E.C.E

Year : IV Semester : II Sec : C Mid No.: II

HALL TICKET NO. 1 7 4 7 1 A 0 4 C 5

Sub : E.S.D Date : 10-02-21

Name : G venkat rao

MARKS 1 0 Marks in words one zero

Signature of the Principal

Signature of the Examiner - I

Signature of the Examiner - II

1.

Memory hierachy:

In memory hierachy there are three levels

The three levels in the memory hierachy are

1) 1st level

2) 2nd level

3) 3rd level

1st level:

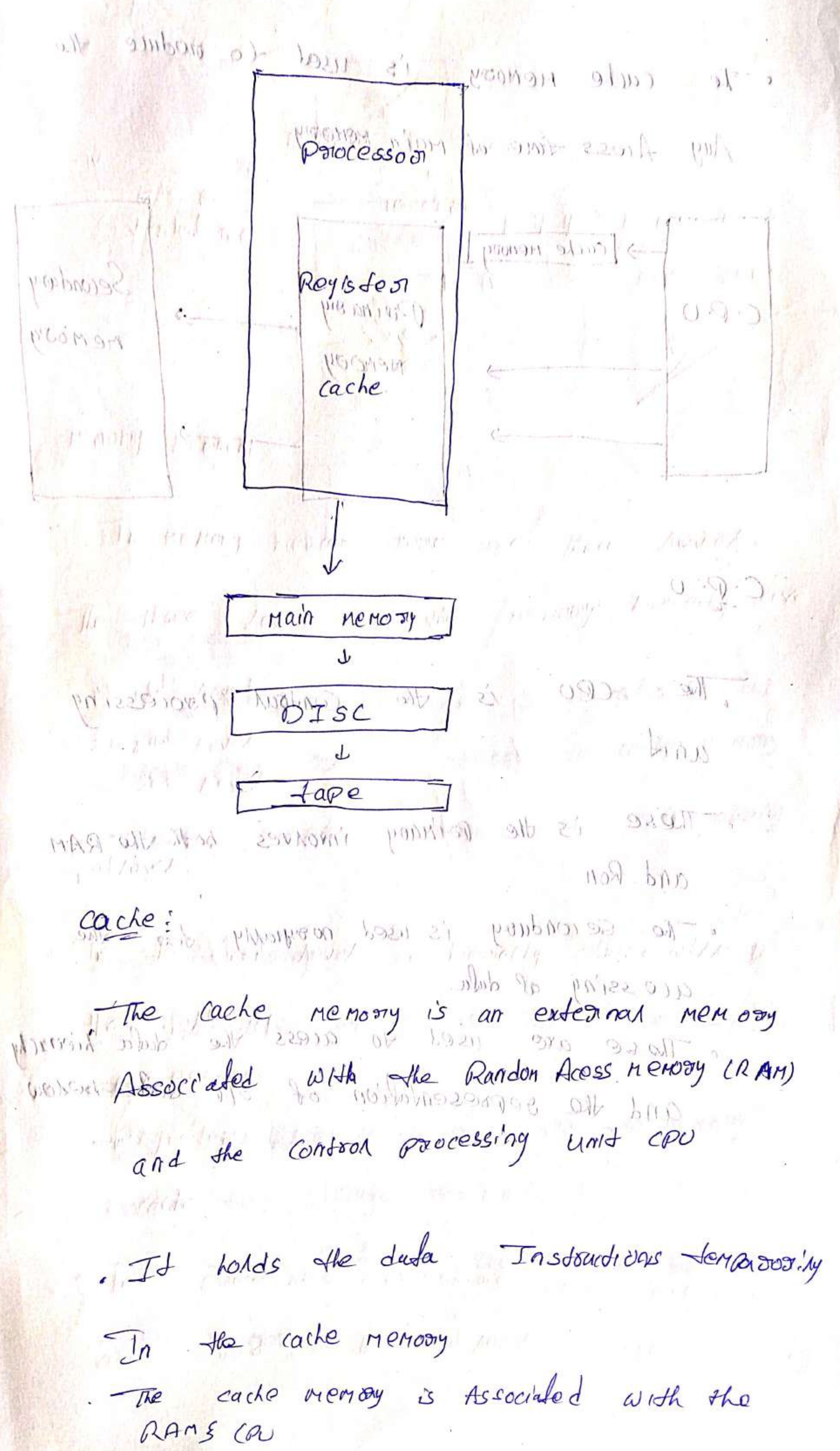
- In the first level of hierachy there will be the following
  - processors
  - Registers
  - cache
- The processors, Registers and cache are used in the 1st level of memory hierachy

2nd level.

- In the second level there will be the main memory
- The main memory is large and fast type memory
- It stores the data from the memory and the entire program
- Where the cache is small and fast memory
- In sometimes stores the copy of the memory

3rd level

- In the 3rd level there will be disk and tape
- These are nothing but Internal & External memory used for acessing the data
- In the 3rd level of the memory hierachy the disc and tape are these
- These are used for acessing the data which are the Internal & External memory

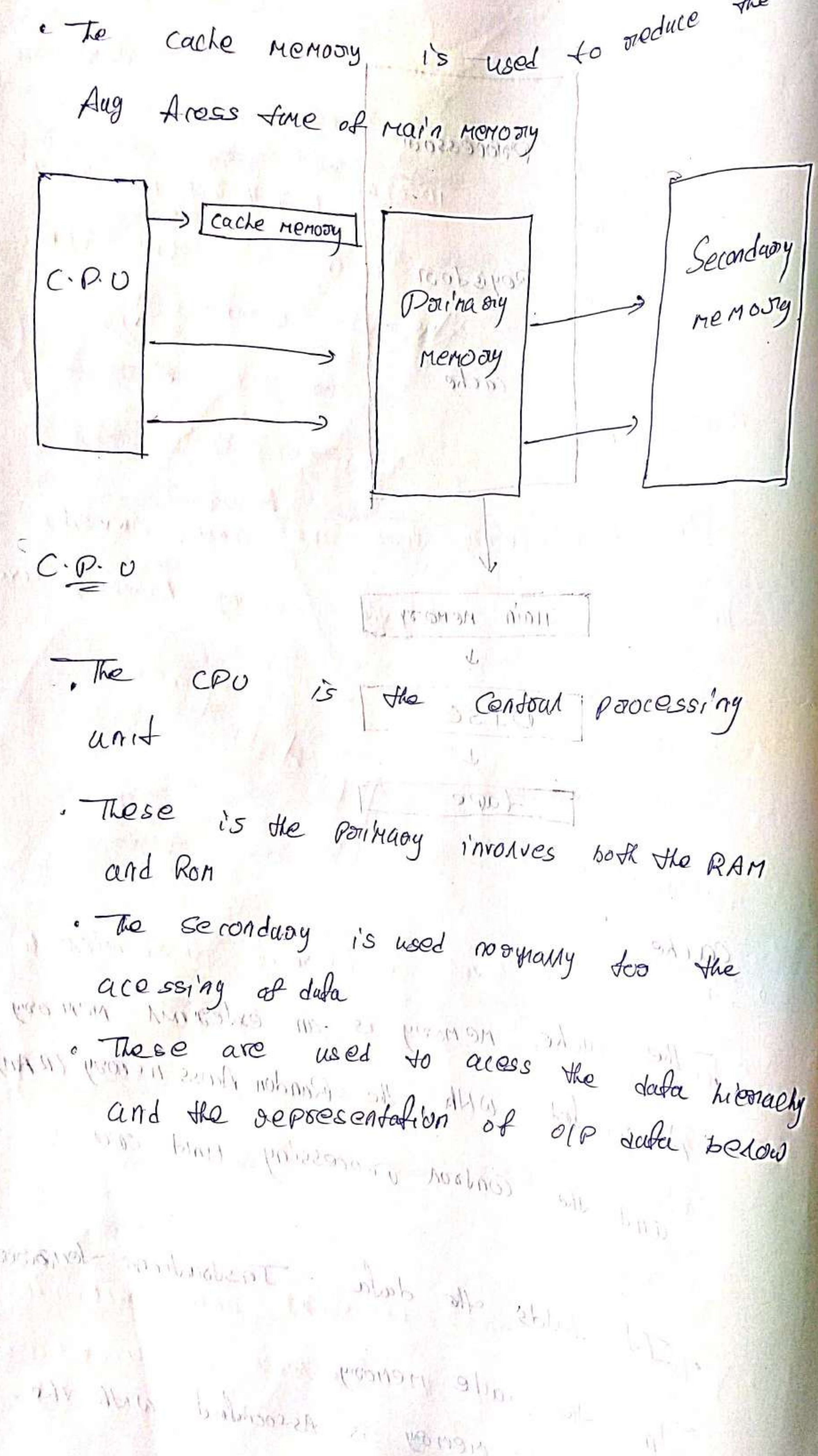


Cache:

The cache memory is an external memory Associated with the Randon Acess memory (RAM) and the Control processing unit CPU

. It holds the data Instructions temporarily In the cache memory

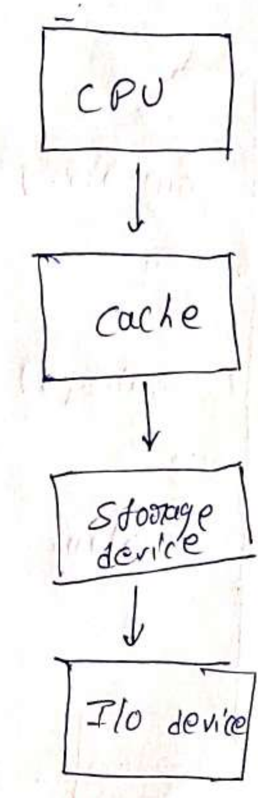
. The cache memory is Associated with the RAM & CPU



- The cache memory is used to reduce the Avg Acess time of main memory

C.P.U

- The CPU is the Central processing unit
- These is the primary involves both the RAM and Rom
- The seconday is used normally for the acessing of data
- These are used to acess the data hierachy and the representation of O/P data below



cache mapping

The cache memory data is transferred as block from primary memory to cache memory These process is known as cache mapping

Associate Mapping:

In Associate mapping both Acess the data of the memory

- This is flexible on the Acess the memory from the storage device
- In these both can acess the data from the memory

NARASARAOPETA ENGINEERING COLLEGE, NARASARAOPET.

(AUTONOMOUS)

(Approved by AICTE New Delhi & Permanently Affiliated to J.N.T.U.K., Kakinada)

NEC

MAIN ANSWER BOOK

B.Tech. / M.Tech. / M.B.A. / M.C.A. / B.Tech (Br.) ECE 2021 (A) 12018

Year: IV Semester: II Sec: C Mid No.: II

Sub: ESD Date: 10/07/2021

Name: P. Sripracanna.

HALL TICKET NO. | 1 | 7 | 4 | 7 | 1 | A | 0 | 4 | C | 6 |

MARKS 1 5 Marks in words one five

Signature of the Principal Signature of the Examiner - I Signature of the Examiner - II

1. Memory Hierachy:-

→ Basically hierachy means a set of predefined positions Arranged to the system. Here in the memory hierachy-

→ The different types of memories placed in a hierachial way as show in Fig.

Process

Registers

cache.

Main Memory.

DISK

Tape.

Fig:- Memory heirachy.

10 or 10

→ In Memory hierachy of a Embedded system design It consist's of Two categories to store data
→ InExpensive with slow, and Expensive and Fast
→ In Memory hierachy, Actually Inexpensive with slow That is easy to store, but There is need of expensive.

→ We use small amount of Expensive and Fast with InExpensive and slower one
→ In order to balance or Manage in a hierachial way.

→ In Memory hierachy Model It consists of Process, registers, cachememory
→ And The Memory (or) data is stored in Main Memory, Tape and Disk.

Cache Memory :-

→ In cacheMemory, we use Expensive and Fast which is difficult But its efficiency is More.

→ In Cache Memory, We use Both types of Expensive and In Expensive

→ In cache, It stores the data in Disk type or Tape. Memory.

→ Cache operates at Two operations Expensive with Fast and In Expensive With slow.

Cache Mapping :-

→ cache Mapping is used to Address the data Stored in a Memory.

→ And This Mappings are Three cateogires
→ Direct Mapping.

→ Full Associate Managing technique

→ Set-of Mapping technique.

→ Any type of Mapping techniques of Main Memory Addresses to cache memory

Direct Mapping :-

→ In Direct Mapping, we use (or) based on the size of the cache.

→ Data that Mapping is Based on size of cache.

→ Data = log (cache size).

→ In Direct Mapping we use only one, we consists of Tag, offset, Index.

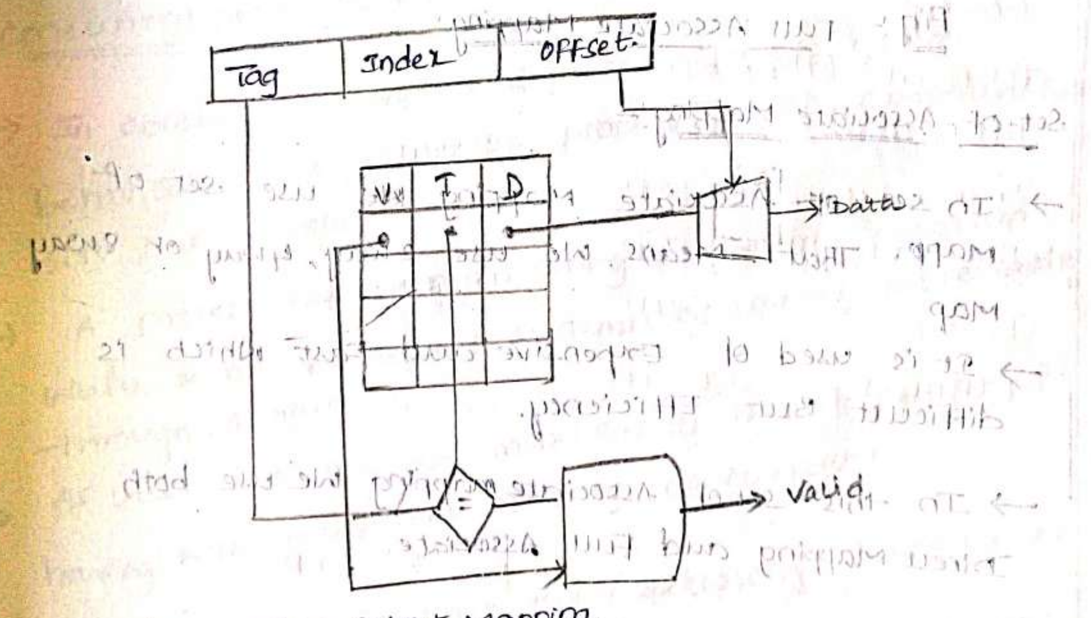


Fig:- Direct Mapping.

Full Associate Mapping :-

→ In Full Associate Mapping we use both memory of cache and Total Memory in cache.

→ In Full Associate Mapping, Mapping is done with Tag and off set.

→ It does not map with Index. Because It is Based on the size of Cache.

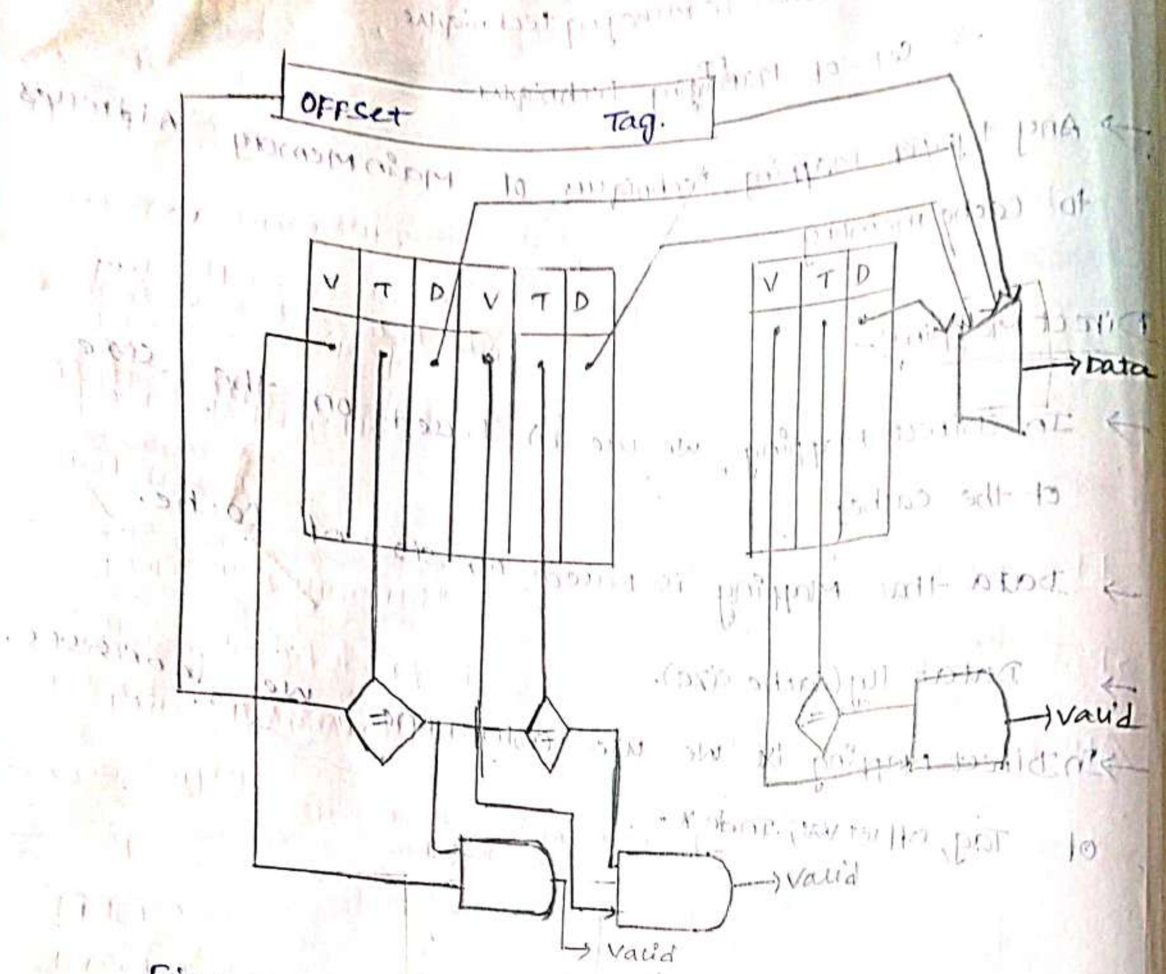


Fig :- Full Associate Mapping

Set-of-Associate Mapping :-

→ In set of Associate Mapping we use set of Mapp. That Means we use 2way, 4way or 8way Map

→ It is used of Expensive and Fast which is difficult But Efficiency.

→ In this set of Associate mapping we use both Direct Mapping and Full Associate

→ It comprises the both Direct Mapping and Full Associate Mapping.

→ In this It consider a set of Mapping 2way, 4way or 8way

→ For Example of 2way Mapping is below Fig.

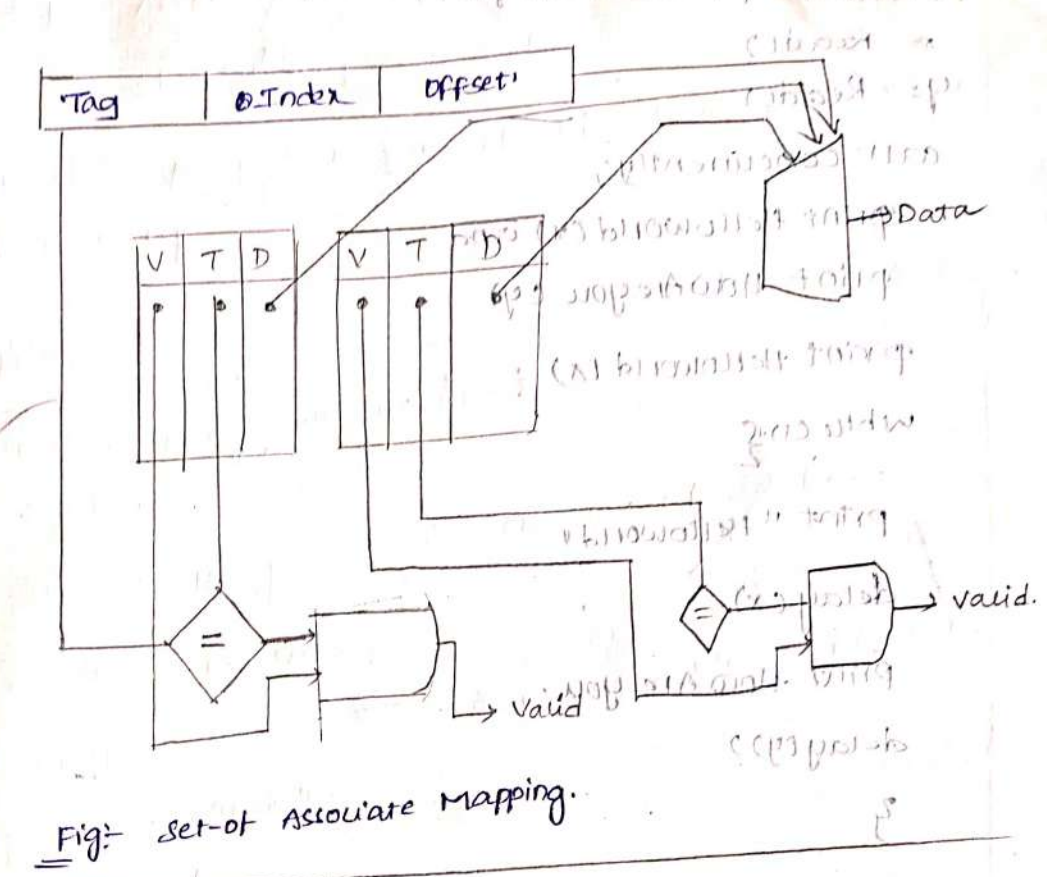


Fig:- set-of Associate Mapping.

concurrent process Model :-

→ In concurrent process model we describe system behaviour as a set of process which communicate with one Another.

→ A process refers to repeating sequential program while many embedded systems are mostly more easily through output.

→ As other systems are more easily through output of having multiple process running concurrent.

→ For example consider the following made up of The system Allows a user to provide number x and y.

To write helloworld program.

→ describing some system using a sequential program model.

concurrent process example()

x= Read()

y= Read()

call concurrently;

print Helloworld (x) and

print How Are you (y)

print Helloworld (x)

while(1){

print "Helloworld"

delay (x).

print How Are you.

delay (y)?

}

Enter x: 1

Enter y: 1

Helloworld (Time = 1s).

Helloworld (Time=2s)

Howe Areyou(Time=2s).

How Are you (T= 3s)

How Are you (T=4s)

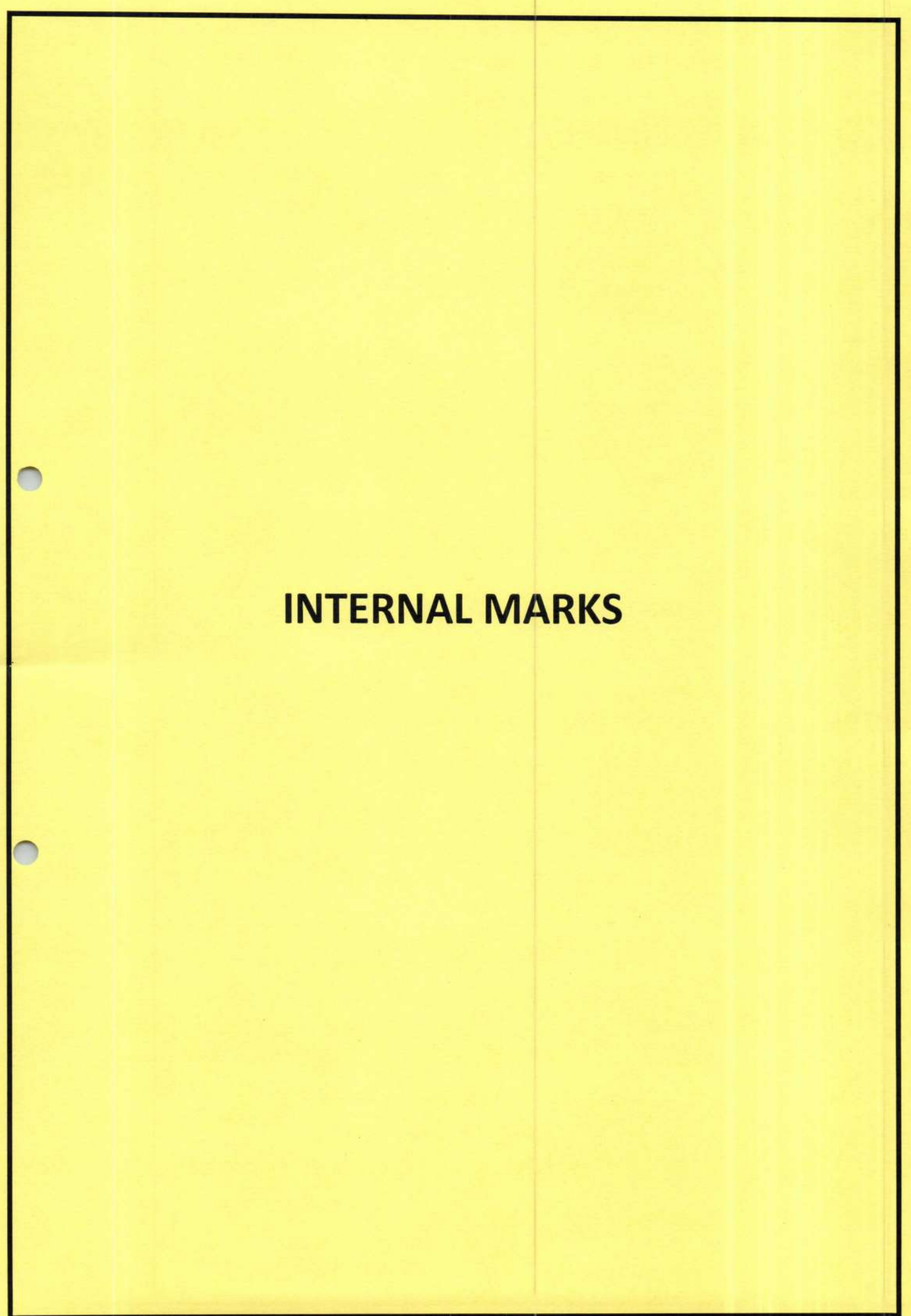
Helloworld (T= 4s)

!

→ Read → Realay → print helloworld

print howe areyou.

→ Time.



INTERNAL MARKS

NARASARAOPETA ENGINEERING COLLEGE::NARASARAOPET

(AUTONOMOUS)

17 BATCH IV B.TECH II SEMESTER (R16) FINAL INTERNAL MARKS-2021

BRANCH/SEC - ECE/A			SUBJECT NAME & CODE :EMBEDDED SYSTEM DESIGN (R16EC4211)										
SL.NO.	H.T.NO.	STUDENT NAME	A1	A2	D1	O1	CYCLE-1	A3	A4	D2	O2	CYCLE-2	TOTAL
1	16471A0405	NELAKUDITI VENKATA SIVA SAI PRUDHVI KUMAR	10	9	20	10	40	10	10	11	7	28	37
2	17471A0401	KOLLA CHAKRI SAI VIJAYACHANDRA	10	9	17	0	27	10	A	5	10	25	27
3	17471A0403	MANAM YASWANTH CHOWDARY	8	9	18	10	37	10	A	6	8	24	34
4	17471A0404	CHINTAGUNTLA KALYAN KUMAR	8	7	15	10	33	8	7	6	9	23	31
5	17471A0405	YERUVA SUDHEER KUMAR REDDY	9	A	16	10	35	8	7	A	A	8	29
6	17471A0406	KOLISETTY BABA SRI RAM KUMAR	A	9	14	10	33	10	A	0	4	14	29
7	17471A0407	BATTULA CHANDAN	7	7	14	9	30	A	10	2	8	20	28
8	17471A0408	KOTABHATTAR V V S PRATHYUSHA	10	6	18	9	37	10	A	0	8	18	33
9	17471A0409	CHERUKULA KASI MAITHRI	10	8	18	10	38	10	10	20	7	37	38
10	17471A0410	KANCHETI VINAY	8	A	16	10	34	A	9	4	4	17	30
11	17471A0411	YAKKALA NAGA MADAN DATHA KUMAR	10	9	14	10	34	10	10	10	9	29	33
12	17471A0412	SANAMPUDI VENKATA NARASIMHA REDDY	10	10	16	10	36	9	A	0	8	17	32
13	17471A0413	GADAM RAM BHUPAL REDDY	8	9	14	9	32	10	A	6	6	22	30
14	17471A0414	YANDAPALLI SAI VAMSIKRISHNA	7	10	14	9	33	9	A	15	6	30	33
15	17471A0415	MAMILLAPALLI SAI RAM	9	9	15	8	32	9	A	14	9	32	32
16	17471A0416	NEMALIDINNE VENKATAJAHNAVI	7	6	16	10	33	10	A	6	8	24	31
17	17471A0417	NEMALIDINNE VENKATA YASHASWINI	10	9	19	10	39	10	10	6	9	25	36
18	17471A0418	MANDALA SAI BHARGAV REDDY	9	9	14	10	33	10	A	0	8	18	30
19	17471A0419	MAMIDIPAKA SAI SRIDHAR	9	8	14	9	32	9	A	10	9	28	31
20	17471A0420	POTHURI YASWANTH GUPTHA	10	10	14	10	34	10	A	8	8	26	32
21	17471A0421	POPURI VENU	8	10	12	10	32	9	A	8	5	22	30
22	17471A0422	KALANGI KRISHNA AKHIL	9	7	16	10	35	10	A	10	9	29	34
23	17471A0423	DESABOINA PRUDHVISAI	10	9	A	A	10	8	A	7	9	24	21
24	17471A0424	CHINNI ESWAR RAO	7	7	16	10	33	9	A	10	8	27	32
25	17471A0425	YAKKALA PRATHAP	8	10	14	9	33	9	A	6	4	19	30
26	17471A0426	PATHURI SAIPAVAN	7	10	16	10	36	10	6	6	7	23	33
27	17471A0427	KOPPURAVURI JEEVAN JITHENDRA	8	9	17	10	36	10	A	6	7	23	33
28	17471A0428	SANKARAPU SEKHAR BABU	A	9	15	10	34	A	10	2	8	20	31
29	17471A0429	NUTHALAPATI DURGA PRASAD	A	9	16	10	35	9	A	2	8	19	31
30	17471A0430	BOGGAVARAPU YASWANTH AMARESH	9	8	14	10	33	10	8	5	8	23	31
31	17471A0432	CHANDRAGIRI SAI PRAGNA	A	9	16	8	33	10	A	15	3	28	32
32	17471A0433	SYED MANISHA	A	10	A	A	10	10	A	A	A	10	10
33	17471A0434	GADDAM VAMSI	9	A	7	6	22	A	9	3	8	20	22

34	17471A0435	MINDYALA NAGASAI	A	7	15	8	30	A	10	12	6	28	30
35	17471A0436	IRUVANTI SATYA SITA RAMA SASTRY	8	9	12	8	29	10	A	10	5	25	28
36	17471A0437	PANGA SRINIVASA RAO	A	9	8	10	27	10	A	7	7	24	27
37	17471A0438	POTHRALA RAMANJI	7	A	6	9	22	8	A	14	10	32	30
38	17471A0439	PASUPULATI SURESH	9	9	6	8	23	9	A	12	9	30	29
39	17471A0440	GUDIPATI CHARITHA	7	9	12	10	31	10	A	8	8	26	30
40	17471A0441	SHAIK SALMAN	8	10	11	3	24	10	9	8	4	22	24
41	17471A0442	MANDALAPU AKHIL SURYA	8	9	12	6	27	10	9	8	5	23	26
42	17471A0443	PABBA VENKATESH NAIDU	9	9	10	10	29	9	A	16	9	34	33
43	17471A0444	NANDHYALA LINGA REDDY	10	8	12	10	32	10	A	12	8	30	32
44	17471A0445	GANGAVARAPU TEJESWAR REDDY	7	9	7	8	24	9	9	6	8	23	24
45	17471A0446	TALLAPANENI VYSHNAVI	10	10	14	8	32	10	A	12	4	26	31
46	17471A0448	YELURI NAVYA	10	10	14	6	30	10	A	12	10	32	32
47	17471A0449	DORAGACHARLA PAVAN KUMAR REDDY	10	9	20	10	40	10	10	15	10	35	39
48	17471A0450	GOPALAM NAVYASRI	A	9	6	8	23	10	A	8	6	24	24
49	17471A0451	SHAIK ABTHAB	10	10	20	9	39	9	A	16	6	31	37
50	17471A0452	TADIKAMALLA SURESH BABU	A	10	7	6	23	9	A	4	6	19	22
51	17471A0453	THUMATI MUKESH CHOWDARY	9	9	20	10	39	10	10	17	9	36	39
52	17471A0454	ANEKALLA LAKSHMAN REDDY	8	9	16	7	32	10	9	4	9	23	30
53	17471A0455	RAGHUVU VENKAT SIVA RAMA NAGENDRA	5	10	15	6	31	10	10	8	6	24	30
54	17471A0456	NANDIKONDA ANJI REDDY	10	9	10	10	30	9	A	9	9	27	30
55	17471A0457	KAKUMANU GANESH KRISHNA SAI	10	9	3	10	23	9	A	11	10	30	29
56	17471A0459	RAMIDEVI SUMANTH	8	10	10	4	24	9	A	4	4	17	23

NARASARAOPETA ENGINEERING COLLEGE::NARASARAOPET
(AUTONOMOUS)
17 BATCH IV B.TECH II SEMESTER (R16) FINAL INTERNAL MARKS-2021

BRANCH/SEC - ECE/B			SUBJECT NAME & CODE :EMBEDDED SYSTEM DESIGN (R16EC4211)										
SL.NO.	H.T.NO.	STUDENT NAME	A1	A2	D1	O1	CYCLE-1	A3	A4	D2	O2	CYCLE-2	TOTAL
1	17471A0461	PONUGOTI RAMESH	10	9	10	8	28	10	A	4	7	21	27
2	17471A0462	MATTRAM VISHNU BABU	10	A	10	7	27	A	7	6	6	19	25
3	17471A0463	RAMISETTY RAMCHARAN	10	9	14	8	32	10	A	12	8	30	32
4	17471A0464	ANANTHA DURGA	A	9	12	7	28	9	10	9	9	28	28
5	17471A0465	KAMMA NAGA SAI RITHVIK	A	9	6	10	25	10	8	8	9	27	27
6	17471A0467	DANDE NAGALAKSHMI	10	9	14	8	32	10	10	14	9	33	33
7	17471A0468	JANAPATI YASASWINI JAYA BHARATHI SAHITHI	10	9	18	10	38	10	10	15	9	34	37
8	17471A0469	JANAPATI SAILAKSHMI SRAVANI	10	7	16	10	36	10	A	17	9	36	36
9	17471A0470	KUNISETTY GOPINADH	A	A	16	9	25	A	9	16	4	29	28
10	17471A0471	SHAIK MD YASIN	10	9	16	9	35	9	9	11	8	28	34
11	17471A0472	RAMISETTI LAKSHMISAITEJA	A	9	14	9	32	8	9	12	9	30	32
12	17471A0473	B. MANI DEEPAK	A	A	7	7	14	A	A	10	8	18	17
13	17471A0474	BOKKA JOHN VICTOR	A	7	19	9	35	9	A	11	3	23	32
14	17471A0475	GODUGUNURI VIJAYA SAI DILEEP KUMAR REDDY	10	7	10	9	29	10	A	17	8	35	34
15	17471A0476	GANAPATHI JYOTHI PRAKASH	10	10	18	9	37	10	A	17	8	35	37
16	17471A0477	MUNAGAPATI MANOJKUMAR	A	9	20	10	39	10	10	15	6	31	37
17	17471A0478	SYED MD GOUSE	10	9	16	9	35	10	A	13	9	32	35
18	17471A0479	GOLI SRINIVASARAO	10	A	20	9	39	10	A	16	9	35	38
19	17471A0480	PATHI VENKATESWARI	10	A	19	8	37	9	10	14	9	33	36
20	17471A0481	VANUKURI HARIVARDHAN VEERA REDDY	10	A	16	9	35	8	10	13	9	32	35
21	17471A0482	PANCHUMARTHI DILEEP KUMAR	7	9	19	9	37	10	10	16	8	34	37
22	17471A0483	KOLLURU KRISHNA MOHAN	9	9	19	9	37	9	10	14	6	30	36
23	17471A0484	RACHUMALLU SASIDHAR	A	9	16	10	35	8	10	17	8	35	35
24	17471A0485	KARNATI HEMANTH SAI	9	9	18	9	36	9	8	16	10	35	36
25	17471A0486	THUMATI VENKATA SUNIL	8	A	9	8	25	10	A	10	10	30	29
26	17471A0487	KOPPURAVURI AKHILA	10	9	16	10	36	10	A	16	9	35	36
27	17471A0488	NAIDU RACHANA	10	9	17	10	37	10	10	13	9	32	36
28	17471A0489	TELAPROLU PAVAN KALYAN	10	10	20	10	40	10	A	10	7	27	37
29	17471A0490	BODEMPUDI SRI HARSHA	10	9	20	10	40	10	A	16	9	35	39
30	17471A0491	SHAIK RUKSANA	7	9	19	10	38	10	A	11	9	30	36
31	17471A0492	KATTAMURI SATYANARAYANA	10	9	20	10	40	10	10	12	8	30	38
32	17471A0493	KOPPULA GANESH REDDY	10	9	15	10	35	9	10	14	7	31	34
33	17471A0494	SYED MAHABOOB JANI BASHA	10	9	20	9	39	10	10	12	8	30	37

34	17471A0495	PERUMALLA PREETHI KOUMIKA	10	9	20	9	39	10	10	14	8	32	38
35	17471A0496	SHAIK JANI BASHA	A	10	5	8	23	10	A	10	9	29	28
36	17471A0497	SHAIK MOHAMMED ALTHAF	A	9	14	8	31	10	A	12	10	32	32
37	17471A0498	JANGALA KIRAN BABU	7	8	13	10	31	A	10	10	6	26	30
38	17471A0499	RAMA CHANDRULA KAVYASRI	10	10	18	8	36	10	A	15	7	32	35
39	17471A04A0	MUVVA MANOJ KUMAR	10	9	19	10	39	10	9	14	7	31	37
40	17471A04A1	KAKUMANU SUMANTH	9	9	18	8	35	10	A	10	9	29	34
41	17471A04A2	YANDAPALLI N V S L MALLIKA BRAMARAMBIKA	10	9	18	6	34	10	A	18	5	33	34
42	17471A04A3	TUMMALACHERUVU SAITEJA	7	10	18	10	38	10	10	12	9	31	37
43	17471A04A4	JAKKIREDDY KEERTHI	10	A	14	6	30	10	A	9	10	29	30
44	17471A04A5	SHAIK AFRID	10	8	19	7	36	10	A	16	8	34	36
45	17471A04A6	YERRAMSETTY SAI PAVAN	7	10	13	10	33	10	9	11	6	27	32
46	17471A04A7	DESABOYINA HEMARAMACHANDRA VASU	7	9	16	8	33	10	10	17	4	31	33
47	17471A04A8	SHAIK TANGEDA CHINA BAJI	7	A	18	4	29	10	A	12	5	27	29
48	17471A04A9	KOLLA SIVA HEMANTH	10	8	20	10	40	10	A	15	10	35	39
49	17471A04B0	AKULA ASHOK KUMAR	10	10	18	10	38	10	A	15	10	35	38
50	17471A04B1	MOHAMMED ZAKIR HUSSAIN KHAN	10	9	18	7	35	10	A	17	6	33	35
51	17471A04B2	BALUPUNURI KASU VASU DEVA VENKATA REDDY	A	A	14	10	24	A	8	5	9	22	24
52	17471A04B3	GORANTLA SRAVAN KRISHNA	10	9	16	8	34	9	A	6	9	24	32
53	17471A04B4	JAMMULA CHANDRIKA	10	9	16	10	36	10	A	16	9	35	36
54	17471A04B5	BONDE RAJENDRA	10	A	15	8	33	A	9	6	4	19	30
55	17471A04B6	NANNEM VEENA VATSALYA	10	9	16	5	31	A	10	8	8	26	30
56	17471A04B7	GOGULA NAVEEN KUMAR	A	9	15	4	28	10	A	A	A	10	24
57	17471A04B8	KURANGI MUKUNDA SAI	10	8	15	10	35	A	9	8	8	25	33
58	17471A04B9	GOUSE MOMITH BAIG	10	9	20	8	38	10	10	12	5	27	36
59	17471A04C0	BUSSI JOSEPH BALA YASWANTH BABU	10	9	19	10	39	10	A	19	9	38	39

34	17471A04F4	CHEVALA PRITHVI RAJ	A	4	10	10	24	6	7	14	6	27	27
35	17471A04F5	DEVARAPALLI NAGA POOJA SAI SRI	A	8	20	10	38	10	A	16	10	36	38
36	17471A04F6	MEKAPOTHULA GOPI KRISHNA	0	9	A	A	9	7	8	17	9	34	28
37	17471A04F7	REPALLE PRATHYUSHA	A	8	17	10	35	10	A	16	6	32	35
38	17471A04F8	KOMMANABOYINA NAGA ANIL	0	6	0	7	13	5	6	13	9	28	25
39	17471A04F9	BATCHU DURGA BHAVANI	0	7	19	10	36	9	9	15	10	34	36
40	17471A04G0	DIRISALA SRAVANI	0	7	14	6	27	9	A	19	4	32	31
41	17471A04G1	KOMIREDDY MANJU BHARGAVI	0	4	14	10	28	7	A	14	8	29	29
42	17471A04G2	POLA VENKATA MALLIKHARJUNA RAO	6	6	20	10	36	6	7	16	9	32	35
43	17471A04G3	KOLIPAKULA DEVI CHAMUNDESWARI	A	7	20	10	37	8	A	9	9	26	35
44	17471A04G4	BOBBA PRASANTH	A	8	7	9	24	6	A	8	9	23	24
45	17471A04G5	G JAGADEESH CHANDRA BOSE	0	8	15	10	33	7	6	11	9	27	32
46	17471A04G6	GUNTU NAVEEN CHOWDARY	1	6	12	10	28	8	A	11	9	28	28
47	17471A04G8	YELLANURU JAHNAVI	A	6	7	6	19	7	A	7	6	20	20
48	17471A04G9	MAMIDIPAKA NAGASUSHMA	A	7	20	10	37	10	A	16	10	36	37
49	17471A04H0	DUGGARAJU GOWTHAMY	6	8	20	9	37	10	7	14	7	31	36
50	17471A04H1	P RAMA KRISHNA	7	6	18	8	33	6	6	12	9	27	32
51	17471A04H2	NALAGANGULA KOTIREDDY	2	6	11	10	27	7	A	8	7	22	26
52	17471A04H3	GUNTUPALLI THIRUMALA PRASANNA SANKAR	A	A	13	8	21	6	7	7	8	22	22
53	17471A04H4	RAJARAPU SRILAKSHMI TIRUMALESWARI	A	6	20	10	36	7	10	18	4	32	35
54	17471A04H5	PAMURU DIVYA	A	7	10	10	27	7	A	11	9	27	27
55	17471A04H6	SHAIK SAMEER	2	8	18	10	36	10	A	10	8	28	34
56	17471A04H7	KUNCHALA GOPI KRISHNA	0	6	16	9	31	6	7	12	10	29	31
57	17471A04H8	PUSALA MADHU KUMAR	0	7	14	8	29	8	A	13	8	29	29
58	17471A04H9	GAJJALA MARUTHI VENKATA KRISHNA REDDY	6	6	16	10	32	8	A	8	6	22	30
59	17471A04I0	MEDA RAVI TEJA	4	3	12	9	25	6	7	14	10	31	30
60	18475A0401	MARELLA VAMSI	0	6	20	4	30	A	7	14	5	26	29
61	18475A0402	RACHAKONDA SHYAM PREMKUMAR	0	5	12	10	27	9	9	20	9	38	36
62	18475A0403	SURISETTI ARCHANA	A	7	12	9	28	6	6	11	6	23	27
63	18475A0404	PALADUGU GANESH	A	6	20	10	36	5	9	20	9	38	38
64	18475A0405	VUYYURU SRILAKSHMI	0	8	10	10	28	8	8	11	6	25	28
65	18475A0406	NUNNA VENKATA SIVASAI	8	7	20	10	38	9	10	20	8	38	38
66	18475A0407	PARITALA HARITHA	8	6	20	10	38	10	8	20	7	37	38
67	18475A0408	MADDINA VENKATA SANDEEP	4	5	16	6	27	7	A	16	5	28	28
68	18475A0409	CHERUKURI BRAHMA VENKATESWARLU	0	7	15	10	32	5	8	6	5	19	29

Unit wise important questions

UNIT - I

1. What is an embedded system? What are the components of embedded system?
2. What are the various classifications of embedded systems?
3. Classify the processors in embedded system?
4. Draw the simple view of organization of processor and memory in a system.

UNIT-II

5. Draw and compare von-Neumann and Harvard architecture.
6. Define interrupt latency? How to avoid it
7. What are the design metrics?
8. What are the challenges of embedded systems?
9. Give the steps in embedded system design?
10. Explain about the custom single-purpose processor design

UNIT-III

11. How to select the processor based upon its architecture and applications?
12. Explain about the general purpose processor design
13. Explain about the Application specific Instruction set processors

UNIT-IV

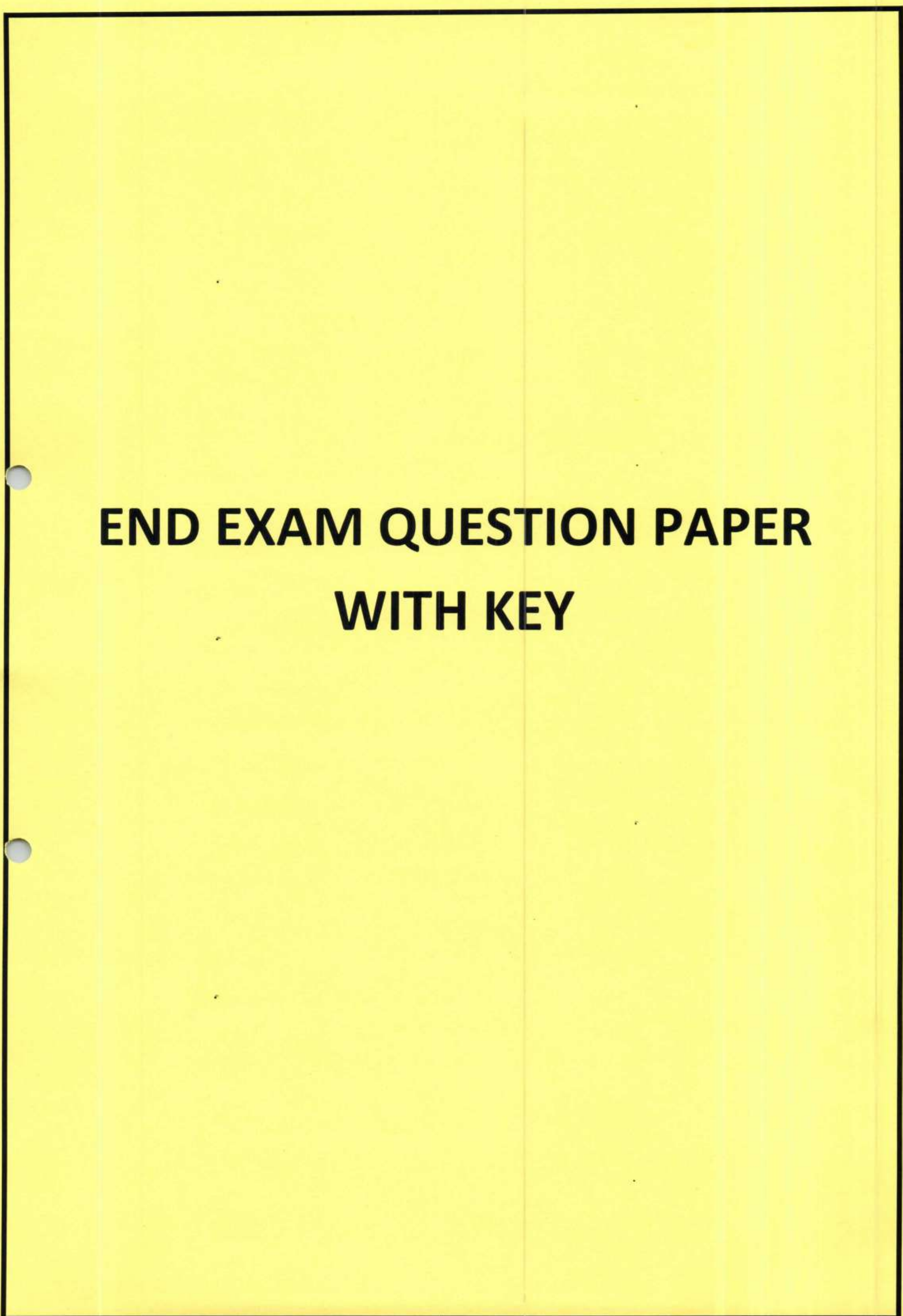
14. What is the difference between SRAM and DRAM
15. What are the functions of memory?
16. Explain in brief about various memories used in embedded systems

UNIT-V

17. What is the difference between Model and languages?
18. Define State Machine Model, sequential Programming Model and concurrent process Model?
19. What are the steps involved in describing a system's behaviour as a state Machine?
20. What is FSMD?
21. How the processes communicate through message passing?
22. Explain the shared memory concept in inter process communication
23. State the difference between FSM and FSMD models.
24. Explain the design concept of an Elevator control mechanism using a sequential model.
25. Explain in brief about the HCFSM and state charts?

UNIT-VI

26. Explain about ASICs and PLDs.
27. Explain about the Hardware/Software co-simulation



END EXAM QUESTION PAPER
WITH KEY

Narasaraopeta Engineering College (Autonomous)
Kotappakonda Road, Yellamanda (P.O), Narasaraopet- 522601, Guntur District, AP.

Subject Code: R16EC4211

IV B.Tech II Semester Regular & Supple Examinations, July-2021

EMBEDDED SYSTEM DESIGN

(ECE)

Time: 3 hours **Max Marks: 60**

Question Paper Consists of **Part-A** and **Part-B.**

Answering the question in **Part-A** is Compulsory & Four Questions should be answered from Part-B

All questions carry equal marks of 12.

PART-A

1. (a) Give few examples of embedded systems
(b) List out the different combinational components used in embedded system design
(c) Differentiate between general purpose processor and application specific instruction processor.
(d) Define Flash Memory and explain its importance
(e) Explain the Basic state machine model in detail
(f) writs short notes on Full custom IC technology

[2+2+2+2+2+2]

PART-B

4 X 12 = 48

2. (a) Explain the classification of embedded systems based on different criteria in detail and give an example for each
(b) Explain the following terms (i) ASIC (ii)PLD

3. (a) Explain the concept of single-purpose processor Design along with one example
(b) write short notes on Optimizing the FSM in detail

4. (a) Draw the architecture of VLIW processor and explain its operation
(b) Explain the concept of Testing and Debugging

5. List out different Cache mapping Techniques and explain each one in detail

6. (a) Explain the concept of concurrent process model along with one example
(b) Write short notes on program state machine model

7. (a) Explain the concept of Standard Cell Semi-Custom IC Technology along with one example
(b) Write short notes on Hardware/Software co-simulation in detail

***

EMBEDDED SYSTEM DESIGN

PART-A

1.

(a). Examples of Embedded systems are microwave ovens, answering machines, thermostat, Home security, Washing machines, and Automatic lighting systems, printers, and scanners and etc.

(b). Combinational components used in embedded system design are Transistors and Logic Gates, N-bit Multiplexers, decoders, adders, Comparators, ALU (arithmetic-logic unit),Registers, Shift registers, Counters.

(c). Differences between general purpose processor and application specific instruction processor

General purpose processor	Application specific instruction processor
General purpose processor is a Programmable device.	Application specific processor have Programmable memory
Low NRE cost.	High NRE Cost
Less Flexibility	Good Flexibility
Performance is not high	good performance, size, and power
Design cost and time of general-purpose processor is low.	low cost and low power consumption.

(d).

Flash Memory

Flash memory is an extension of EEPROM that was developed in the late 1980s. While also using the floating-gate principle of EEPROM, flash memory is designed such that large blocks of memory can be erased all at once, rather than just one word at a time as in

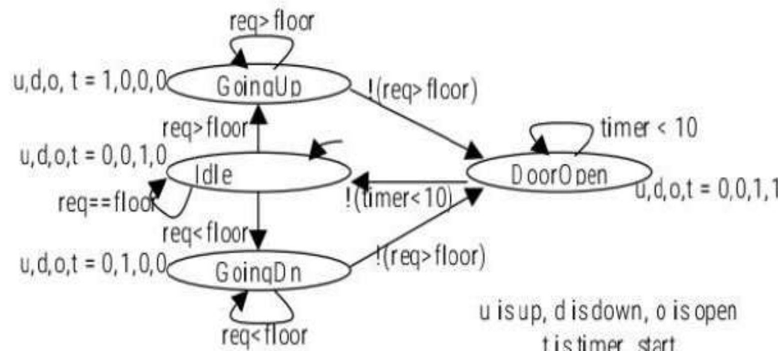
traditional EEPROM. A block is typically several thousand bytes large. This fast erase ability can vastly improve the performance of embedded systems where large data items must be stored in nonvolatile memory, systems like digital cameras, TV set-top boxes, cell phones, and medical monitoring equipment. It can also speed manufacturing throughput, since programming the complete contents of flash may be faster than programming a similar-sized EEPROM.

Like EEPROM, each block in a flash memory can typically be erased and reprogrammed tens of thousands of times before the block loses its ability to store data, and can store its data for 10 years or more.

A drawback of flash memory is that writing to a single word in flash may be slower than writing to a single word in EEPROM, since an entire block will need to be read, the word within it updated. and than the block written back.

(e). Basic State machine model: In a state machine model, we describe system behaviour as a set of possible states; the system can only be in one of these states at a given time. We also describe the possible transitions from one state to another depending on input values. Finally, we describe the actions that occur when in a state or when transitioning between states.

Figure 8.2: The elevator's UnitControl process described using a state machine.



(f). Full custom IC technology: In a full-custom IC technology, we optimize all layers for our particular embedded system's digital implementation. Such optimization includes placing the transistors to minimize interconnection lengths, sizing the transistors to optimize signal transmissions and routing wires among the transistors. Once we complete all the masks, we send the mask specifications to a fabrication plant that builds the actual ICs. Full-custom IC design, often referred to as VLSI (Very Large-Scale Integration) design, has very high NRE cost and long turnaround times (typically months) before the IC becomes available, but can yield excellent performance with small size and power. It is usually used only in high-volume or extremely performance-critical applications.

<u>**PART-B**</u>

2. (a). Classification of Embedded Systems:

Embedded systems are classified based on the applications they used. They are

(a) Consumer electronics --cell phones, pagers, digital cameras, camcorders, videocassette recorders, portable video games, calculators, and personal digital assistants

(b) Home appliances -- microwave ovens, answering machines, thermostat, home security, washing machines, and lighting systems

(c) Office automation -- fax machines, copiers, printers, and scanners

(d) Business equipment -- cash registers, curb side check-in, alarm systems, card readers, product scanners, and automated teller machines

(e) Automobiles – transmission control, cruise control, fuel injection, anti-lock brakes, and active suspension.

(f) Embedded Systems in Smart Cards, Missiles and Satellites-- Security systems, Telephone and banking, Defence and aerospace, Communication.

(g) Embedded Systems in Peripherals & Computer Networking-- Displays and Monitors, Networking Systems, Image Processing, Network cards and printers.

(h) Embedded Systems in Consumer Electronics--Digital Cameras, Set top Boxes, High-Definition TVs, DVDs.

(i) Environment &agriculture: smart water management, smart irrigation etc.

(j) Military: Intelligence Gathering Operations. Military commanders need correct information to make the best decisions, Surveillance and Reconnaissance UAVs, Communication, Computing, Cyber Security, Vehicle Electronics

(b). i. ASIC: In an ASIC (Application-Specific IC) technology, the lower layers are fully or partially built, leaving us to finish the upper layers. In a gate array technology, the masks

for the transistor and gate levels are already built (i.e., the IC already consists of arrays of gates). The remaining task is to connect these gates to achieve our particular implementation. In a standard cell technology, logic-level cells (such as an AND gate or an AND-OR-INVERT combination) have their mask portions pre-designed, usually by hand. Thus, the remaining task is to arrange these portions into complete masks for the gate level, and then to connect the cells. ASICs are by far the most popular IC technology, as they provide for good performance and size, with much less NRE cost than full-custom IC's.

ii. PLD: In a PLD (Programmable Logic Device) technology, all layers already exist, so we can purchase the actual IC. The layers implement a programmable circuit, where programming has a lower-level meaning than a software program. The programming that takes place may consist of creating or destroying connections between wires that connect gates, either by blowing a fuse, or setting a bit in a programmable switch. Small devices, called programmers, connected to a desktop computer can typically perform such programming. We can divide PLD's into two types, simple and complex. One type of simple PLD is a PLA (Programmable Logic Array), which consists of a programmable array of AND gates and a programmable array of OR gates. Another type is a PAL (Programmable Array Logic), which uses just one programmable array to reduce the number of expensive programmable components. One type of complex PLD, growing very rapidly in popularity over the past decade, is the FPGA (Field Programmable Gate Array), which offers more general connectivity among blocks of logic, rather than just arrays of logic as with PLAs and PALs, and are thus able to implement far more complex designs. PLDs offer very low NRE cost and almost instant IC availability.

3.

(a). We now have the knowledge needed to build basic processor. A basic processor consists of a controller and data path shown in below figure. The data path stores and manipulates a

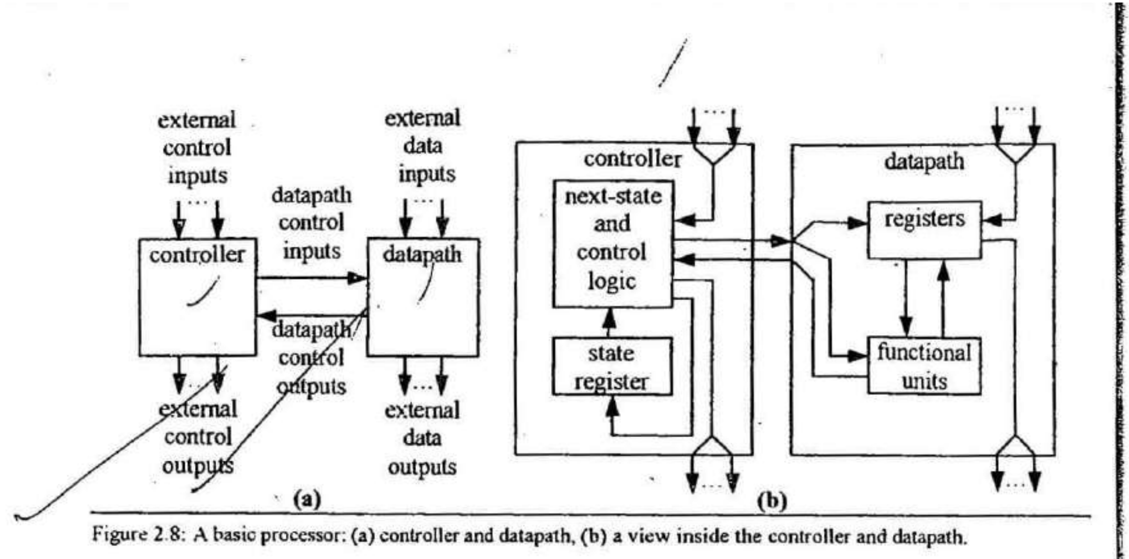


Figure 2.8: A basic processor: (a) controller and datapath, (b) a view inside the controller and datapath.

system's data. Examples of data in an embedded system include binary numbers representing external conditions like temperature or speed, characters to be displayed on a screen, or a digitized photographic image to be stored and compressed. The datapath contains register units, functional units, and connection units like wires and multiplexors. The datapath can be configured to read data from particular registers, feed that data through functional units configured to carry out particular operations like add or shift, and store the operation results back into particular registers. A *controller* carries out such configuration of the datapath. It sets the datapath control inputs, like register load and multiplexor select signals, of the register units, functional units, and connection units to obtain the desired configuration at a particular time. It monitors external control inputs as well as datapath control outputs, known as status signals, coming from functional units, and it sets external control outputs as well.

Example: GCD

Figure 4.3 provides a example based on computing a greatest common divisor (GCD). Figure 4.3(a) shows a black-box diagram of the desired system, having x_i and y_i data inputs and a data output d_i. The system's functionality is straightforward: the output should represent the GCD of the inputs. Thus, if the inputs are 12 and 8, the output should be 4. If the inputs are 13 and 5, the output should be 1. Figure 4.3(b) provides a simple program with this functionality. The reader might trace this program's execution on the above examples to verify that the program does indeed compute the GCD.

To begin building our single-purpose processor implementing the GCD program, we first convert our program into a complex state diagram, in which states and arcs may include arithmetics expressions, and these expressions may use external inputs and outputs or variables. In contrast, our earlier state diagrams only included boolean expressions, and these expressions could only use external inputs and outputs, not variables. Thus, these more complex state diagram looks like a sequential program in which statements have been scheduled into states.

We are now well on our way to designing a custom single-purpose processor that executes the GCD program. Our next step is to divide the functionality into a data path part and a controller part, as shown in Figure 4.4. The data path part should consist of an

interconnection of combinational and sequential components. The controller part should consist of a basic state diagram

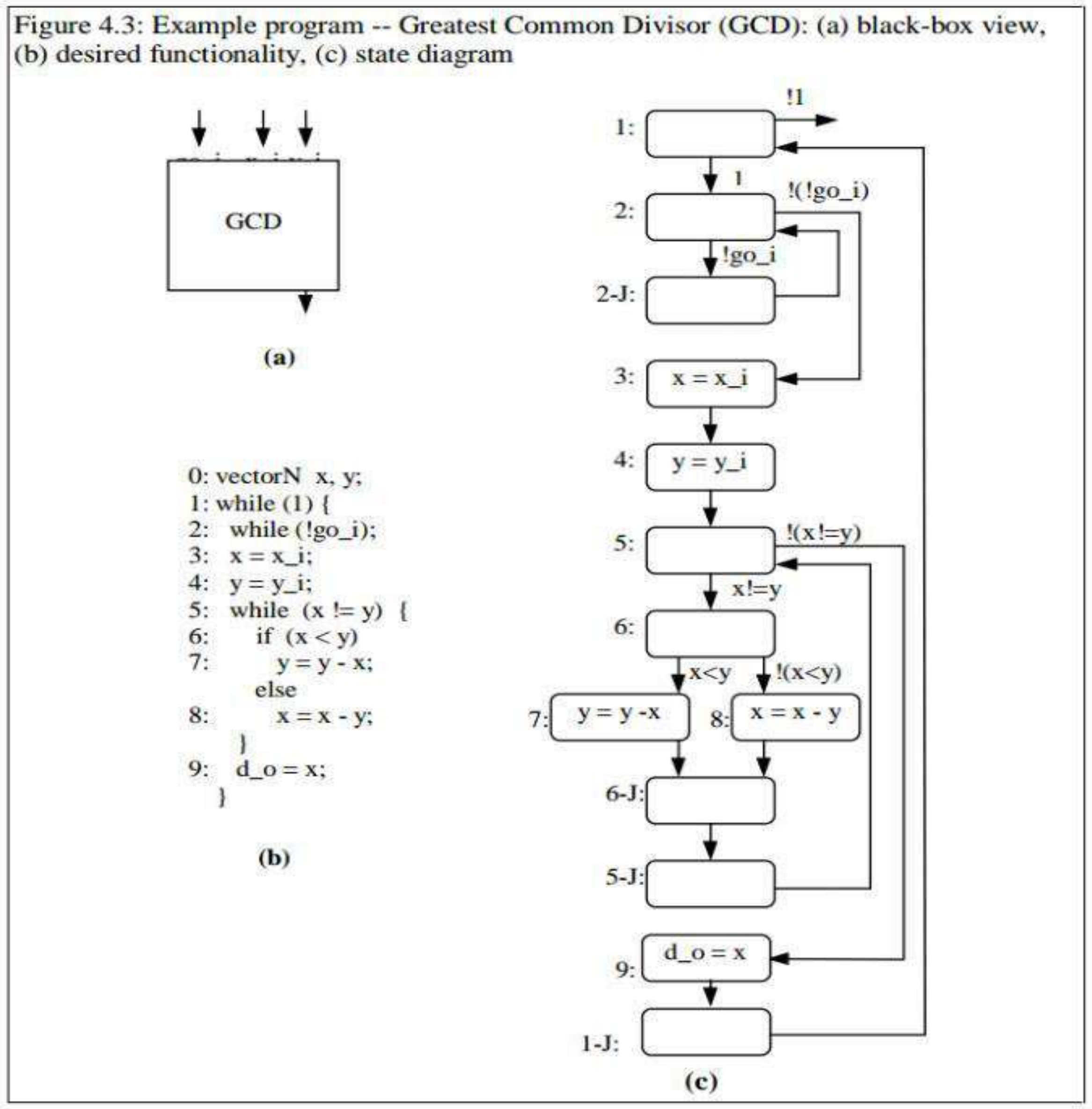


Figure 4.3: Example program -- Greatest Common Divisor (GCD): (a) black-box view, (b) desired functionality, (c) state diagram

(b).

Optimizing the FSM

Designing a sequential circuit to implement an FSM also provides some opportunities for optimization, namely, state encoding and state minimization.

State encoding is the task of assigning a unique bit pattern to each state in an FSM. Any assignment in which the encodings are unique will work properly, but the size of the state register as well as the size of the combinational logic may differ for different encodings. For example, four states *A*, *B*, *C*, and *D* can be encoded as 00, 01, 10, and 11, respectively. Alternatively, those states can be encoded as 11, 10, 00, and 01, respectively. In fact, for an FSM with *n* states where *n* is a power of 2, there are *n*! possible encodings. We can see this easily if we treat encoding as an ordering problem — we order the states and assign a straightforward binary encoding, starting with 00...00 for the first state, 00...01 for the second state, and so on. There are *n*! possible orderings of *n* items, and thus *n*! possible encodings. *n*! is a very large number for large *n*, and thus checking each encoding to determine which yields the most efficient controller is a hard problem. Even more encodings are possible, since we can use more than $\log_2(n)$ bits to encode *n* states, up to *n* bits to achieve a one-hot encoding. CAD tools are therefore a great aid in searching for the best encoding.

State minimization is the task of merging equivalent states into a single state. Two states are equivalent if, for all possible input combinations, those two states generate the same outputs and transition to the same next state. Such states are clearly equivalent, since merging them will yield exactly the same output behavior.

The state merging that we did when optimizing our FSMD was not the same as state minimization as defined here. The reason is that our state merging in the FSMD actually changed the output behavior, in particular the output timing, of the FSMD. Typically, by the time we arrive at an FSM, we assume output timing cannot be changed. State minimization does not change the output behavior in any way.

4

(a). VLIW ARCHITECHTURE:

The limitations of the Superscalar processor are prominent as the difficulty of scheduling instruction becomes complex. The intrinsic parallelism in the instruction stream, complexity, cost, and the branch instruction issue get resolved by a higher instruction set architecture called the **Very Long Instruction Word (VLIW)** or **VLIW Machines**. VLIW uses Instruction Level Parallelism, i.e. it has programs to control the parallel execution of the instructions. In other architectures, the performance of the processor is improved by using either of the following methods: pipelining (break the instruction into subparts), superscalar processor (independently execute the instructions in different parts of the

processor), out-of-order-execution (execute orders differently to the program) but each of these methods add to the complexity of the hardware very much. VLIW Architecture deals with it by depending on the compiler. The programs decide the parallel flow of the instructions and to resolve conflicts. This increases compiler complexity but decreases hardware complexity by a lot.

Features of VLIW: The processors in this architecture have multiple functional units, fetch from the Instruction cache that have the Very Long Instruction Word.

- Multiple independent operations are grouped together in a single VLIW Instruction. They are initialized in the same clock cycle.
- Each operation is assigned an independent functional unit.
- All the functional units share a common register file.
- Instruction words are typically of the length 64-1024 bits depending on the number of execution unit and the code length required to control each unit.
- Instruction scheduling and parallel dispatch of the word is done statically by the compiler.
- The compiler checks for dependencies before scheduling parallel execution of the instructions.

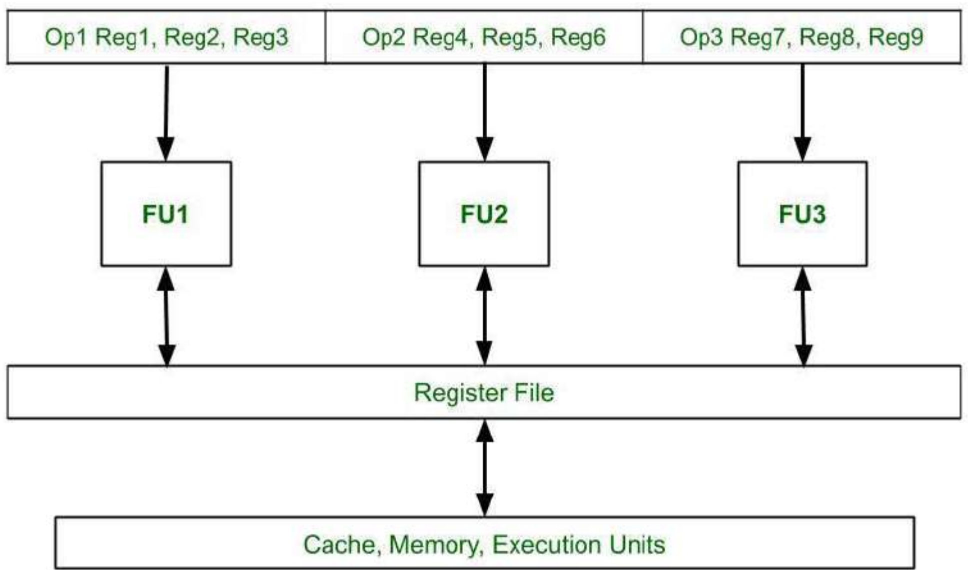


Fig. Block Diagram of VLIW Architecture

(b).

Testing and debugging: Test/Verification involves ensuring that functionality is correct. Such assurance can prevent time-consuming debugging at low abstraction levels and iterating back to high abstraction levels.

Debuggers help programmers evaluate and correct their programs. They run on the development processor and support stepwise program execution, executing one instruction and then stopping, proceeding to the next instruction when instructed by the user. They permit execution up to user-specified breakpoints, which are instructions that when encountered cause the program to stop executing. Whenever the program stops, the user can examine values of various memory and register locations. A source-level debugger enables step-by-step execution in the source program language, whether assembly language or a structured language. A good debugging capability is crucial, as today's programs can be quite complex and hard to write correctly. Device programmers download a binary machine program from the development processor's memory into the target processor's memory.

Emulator's support debugging of the program while it executes on the target processor. An emulator typically consists of a debugger coupled with a board connected to the desktop processor via a cable. The board consists of the target processor plus some support circuitry (often another processor). The board may have another cable with a device having the same pin configuration as the target processor, allowing one to plug this device into a real embedded system. Such an in-circuit emulator enables one to control and monitor the program's execution in the actual embedded system circuit. Incircuit emulators are available for nearly any processor intended for embedded use, though they can be quite expensive if they are to run at real speeds. The availability of low-cost or high-quality development environments for a processor often heavily influences the choice of a processor.

5. Cache mapping Techniques:

Cache is usually designed using static RAM rather than dynamic RAM, which is one reason that cache is more expensive but faster than main memory. Because cache usually appears on the same chip as a processor, where space is very limited, cache size is typically only a fraction of the size main memory. Cache access time may be as low as just one clock cycle, whereas main memory access time is typically several cycles.

A cache operates as follows. When we want the processor to access (read or write) a main memory address, we first check for a copy of that location in cache. If the copy is in the

cache, called a cache hit, then we can access it quickly. If the copy is not there, called a cache miss, then we must first read the address (and perhaps some of its neighbors) into the cache. This description of cache operation leads to several cache design choices: cache mapping, cache replacement policy, and cache write techniques. These design choices can have significant impact on system cost, performance, as well as power, and thus should be evaluated carefully for a given application.

Cache mapping techniques: Cache mapping is the method for assigning main memory addresses to the far fewer number of available cache addresses, and for determining whether a particular main memory address' contents are in the cache. Cache mapping can be accomplished using one of three basic techniques:

1. Direct mapping: In this technique, the main memory address is divided into two fields, the index and the tag. The index represents the cache address, and thus the number of index bits is determined by the cache size, i.e., index size = log2(cache size). Note that many different main memory addresses will map to the same cache address. When we store a main memory address' content in the cache, we also store the tag. To determine if a desired main memory address is in the cache, we go to the cache address indicated by the index, and we then compare the tag there with the desired tag.

2. Fully-associative mapping: In this technique, each cache address contains not only a main memory address' content, but also the complete main memory address. To determine if a desired main memory address is in the cache, we simultaneously (associatively) compare all the addresses stored in the cache with the desired address.

3. Set-associative mapping: This technique is a compromise between direct and fullyassociative mapping. As in direct-mapping, an index maps each main memory address to a cache address, but now each cache address contains the content and tags of two or more memory locations, called a set or a line. To determine if a desired main memory address is in the cache, we go to the cache address indicated by the index, and we then simultaneously (associatively) compare all the tags at that location (i.e., of that set) with the desired tag. A cache with a set of size N is called an N-way set-associative cache. 2-way, 4- way and 8-way set associative caches are common.

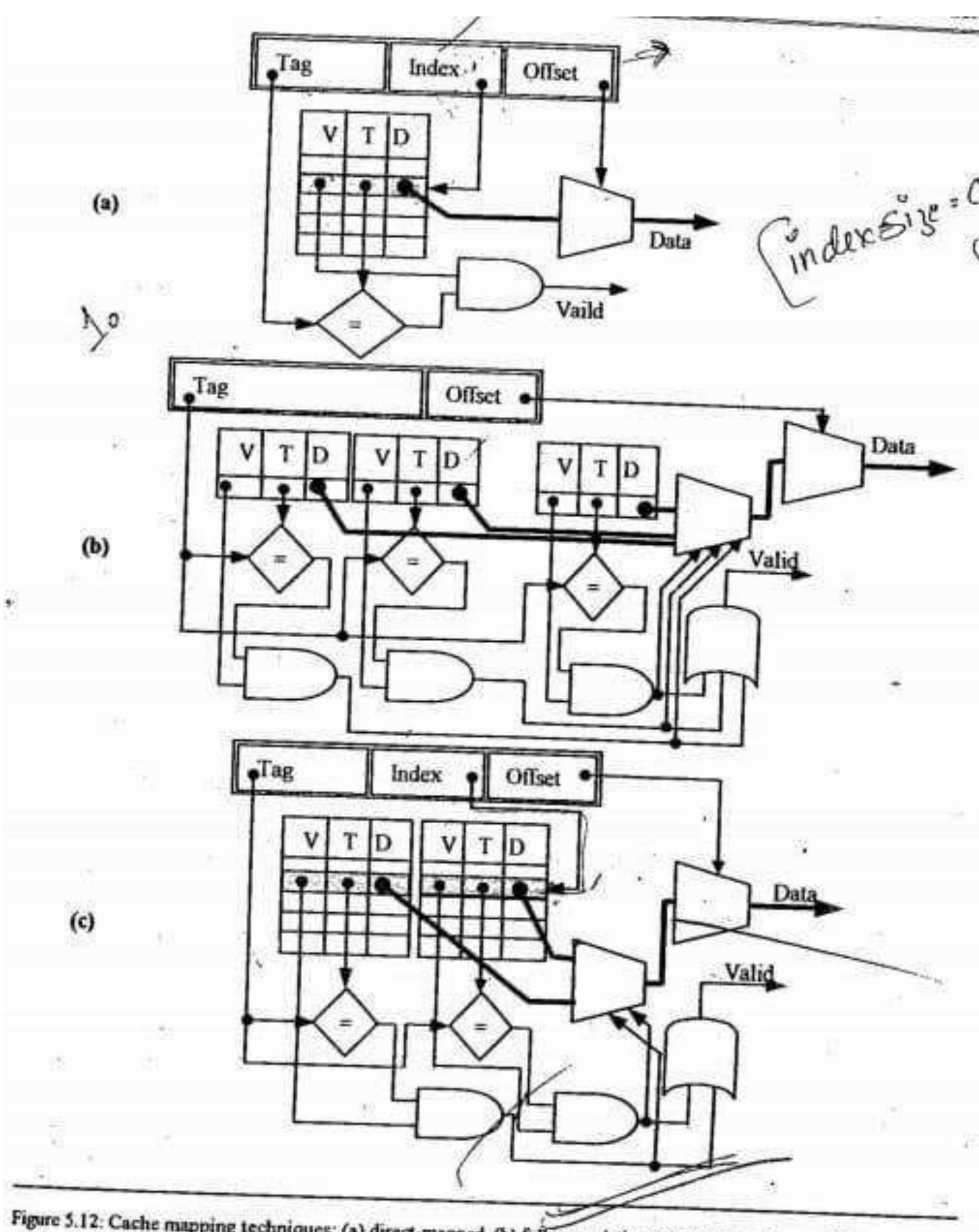


Figure 5.12: Cache mapping techniques: (a) direct-mapped, (b) fully associative, (c) two-way set associative.

6.

(a). Concurrent process model:

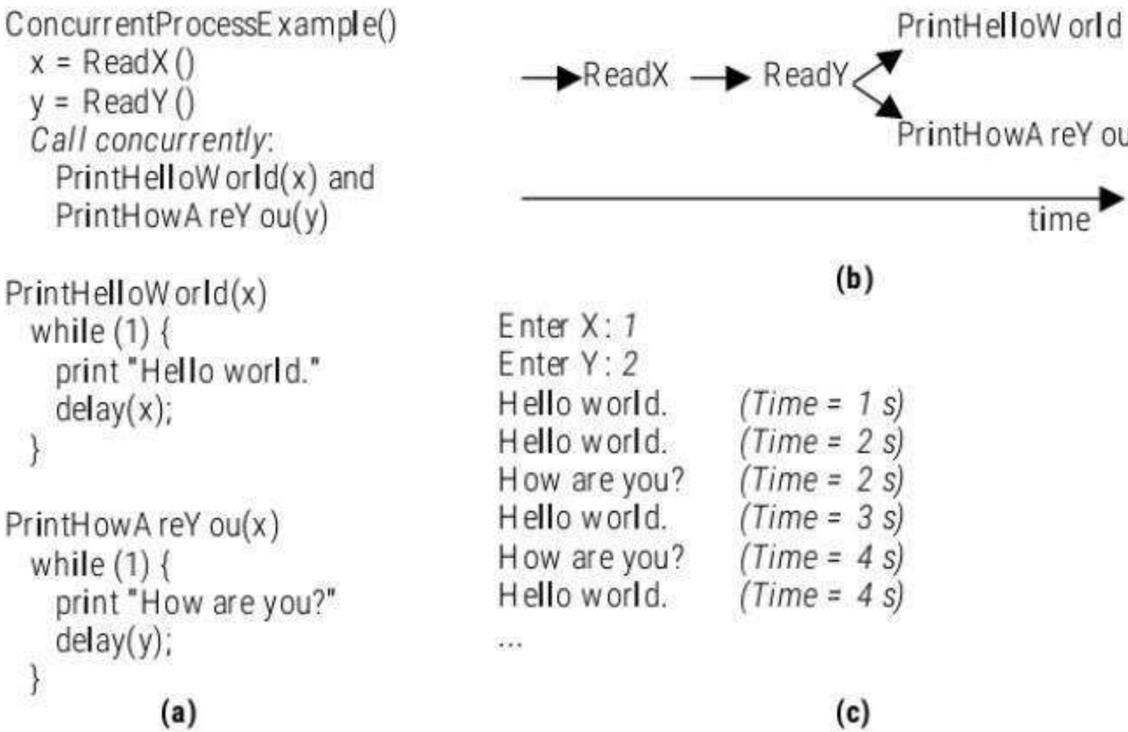
In a concurrent process model, we describe system behaviour as a set of processes, which communicate with one another. A process refers to a repeating sequential program. While many embedded systems are most easily thought of as one process, other systems are more easily thought of as having multiple processes running concurrently.

For example, consider the following made-up system. The system allows a user to provide two numbers X and Y. We then want to write "Hello World" to a display every X seconds,

and "How are you" to the display every Y seconds. A very simple way to describe this system using concurrent processes is shown in Figure 8.9(a). After reading in X and Y, we call two subroutines concurrently. One subroutine print's "Hello World" every X seconds, the other prints "How are you" every Y seconds. (Note that you can't call two subroutines concurrently in a pure sequential program model, such as the model supported by the basic version of the C language). As shown in Figure 8.9(b), these two subroutines execute simultaneously. Sample output for X=1 and Y=2 is shown in Figure 8.9(c).

To see why concurrent processes were helpful, try describing the same system using a sequential program model (i.e., one process). You'll find yourself exerting effort figuring out how to schedule the two subroutines into one sequential program. Since this example is a trivial one, this extra effort is not a serious problem, but for a complex system, this extra effort can be significant and can detract from the time you have to focus on the desired system behaviour. Recall that we described our elevator controller using two "blocks." Each block is really a process. The controller was simply easier to comprehend if we thought of the two blocks independently.

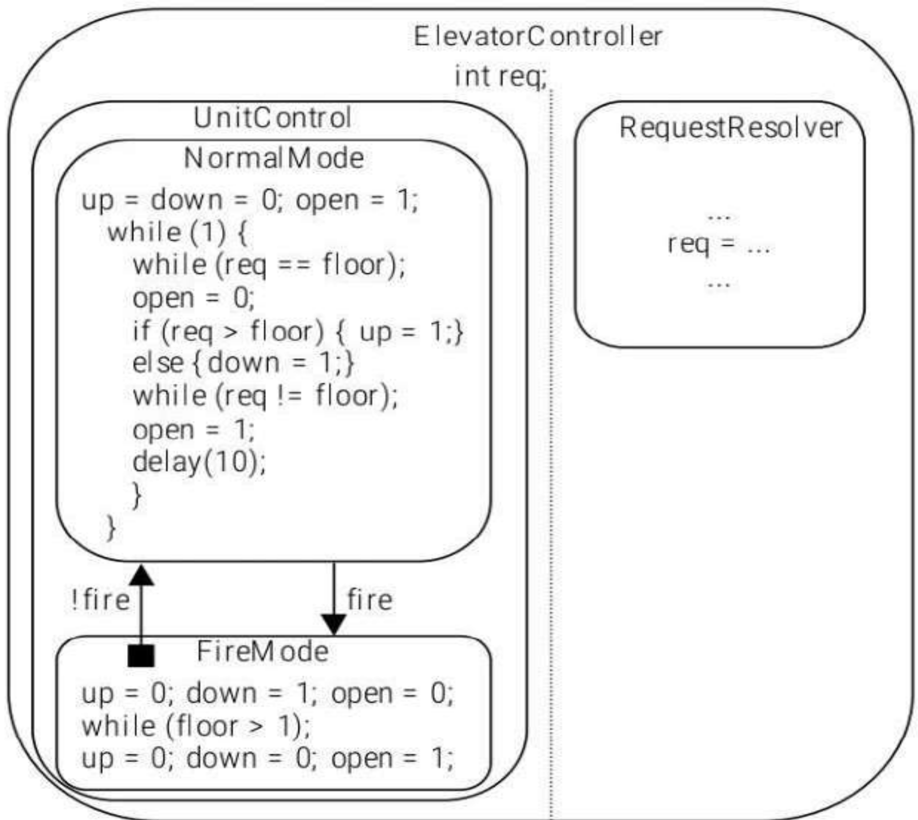
Figure 8.9: A simple concurrent process example: (a) pseudo-code, (b) subroutine execution over time, (c) sample input and output.



(b). Program state machine model: The program-state machine (PSM) model extends state machines to allow use of sequential program code to define a state's actions (including extensions for complex data types and variables), as well as including the hierarchy and concurrency extensions of HCFSM. Thus, PSM is a merger of the HCFSM and sequential program models, subsuming both models. A PSM having only one state (called a program-state in PSM terminology), where that state's actions are defined using a sequential program, is the same as a sequential program. A PSM having many states, whose actions are all just

assignment statements, is the same as an HCFSM. Lying between these two extremes are various combinations of the two models. For example, Figure 8.8 shows a PSM description of the Elevator Controller behaviour, which we AND-decompose into two concurrent program-states Unit Control and Request Resolver, as in the earlier HCFSM example. PSM enforces a stricter hierarchy than the HCFSM model used in State charts. In State charts, transitions may point to or from a substate within a state, such as the transition pointing from the substate of the state to the Normal Mode state. As in the sequential programming model, but unlike the HCFSM model, PSM includes the notion of a program-state completing. If the program-state is a sequential program, then reaching the end of the code means the program-state is complete. If the program-state is OR-decomposed into substates, then a special complete substate may be added. Transitions may occur from a substate to the complete substate (but no transitions may leave the complete substate), which when entered means that the program-state is complete. Consequently, PSM introduces two types of transitions. A transitionimmediately (TI) transition is taken immediately if its condition becomes true, regardless of the status of the source program-state -- this is the same as the transition type in an HCFSM. A second, new type of transition, transition-on-completion (TOC), is taken only if the condition is true AND the source program-state is complete.

Figure 8.8: Using PSM to describe the ElevatorController.



7.

(a).

Standard Cell Semi-Custom IC Technology

In standard cell IC technology, common logic functions, or *cells*, have already been compactly layed out. Examples of cells include a NAND gate, a NOR gate, a 2×1 multiplexor, and a combination of AND-OR-INVERT gates. The transistors within a cell are already layed out, but the placement of cells has not been determined. A designer thus must decide which cells to use, where to place them, and how to route among them. A standard cell layout is shown in Figure 10.6(b).

Standard cell therefore requires more NRE cost and longer time-to-market than gate array, since there is more layout remaining to be performed and all masks must still be made. However, NRE and time-to-market is still much less than full-custom, since the intricate layout within each cell is already completed. In addition, effieiency is very good compared to gate array, since only those cells needed are actually used, and their placement can be made so as to reduce interconnect. Furthermore, each cell may implement more complex functions than in gate arrays, leading to more compact designs.

A compromise between gate array and standard cell semi-custom ICs is known as a *cell array*, or cell-based array. A cell array is pretty much what we'd expect it to be based on its name. Cells, which you'll remember can be more complex than gates, have already been layed out, and have also already been placed. Thus, the designer need only connect the cells together.

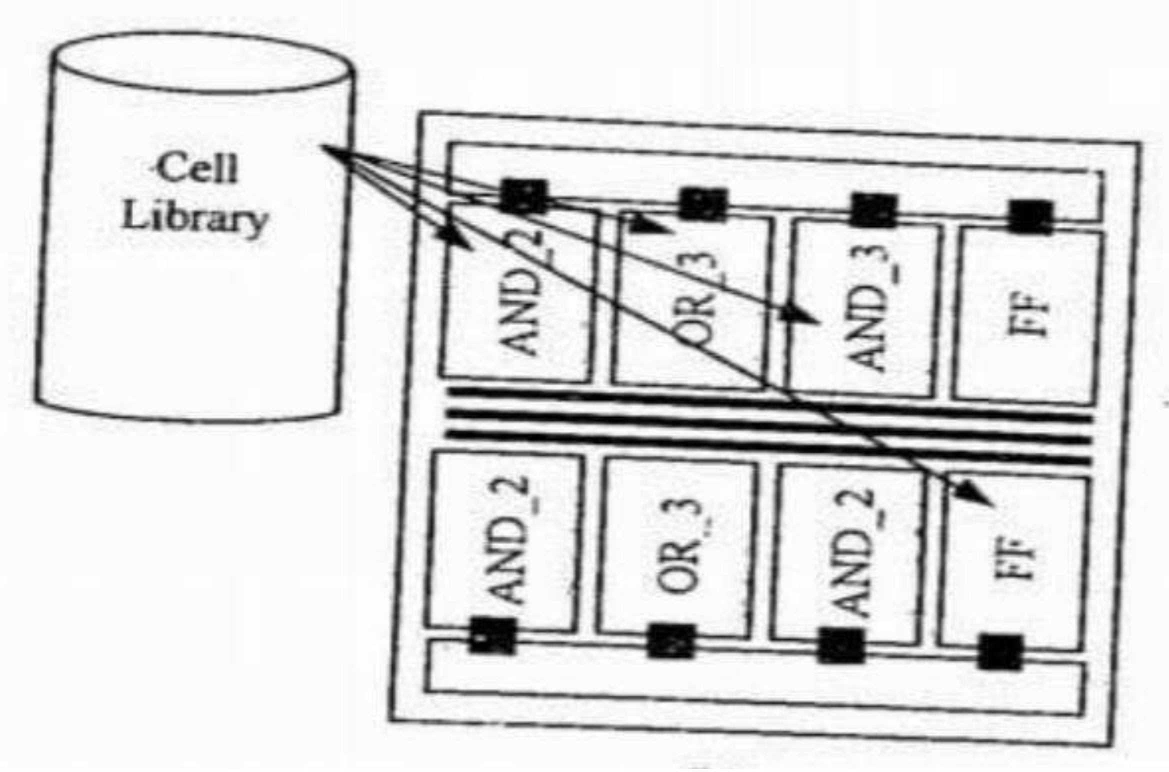


Fig. Standard cell

(b).

Hardware-Software Co-Simulation

More generally, a variety of simulation approaches exist, varying in their simulation speed and precision/accuracy. For a given processor, whether general-purpose or single-purpose, simulation can vary from very detailed, like a gate-level model, to very abstract, like an instruction-level model. An instruction-level model of a general-purpose processor is known as an instruction-set simulator (ISS). An instruction-level model of a single-purpose processor is simply known as a system-level model. Lower-level simulations of either type of processor is usually done by creating a behavior, RT, or gate-level model in a hardware description language (HDL) environment. Because of the past separation of software design and hardware design, the simulation tools for each domain have evolved quite independently. The emphasis in software simulation has been on ISSs. The emphasis of hardware simulation has been models in hardware description languages (HDLs).

The integration of general-purpose and single-purpose processors onto a single IC has increased the need for an integrated method for simultaneously simulating these different types of processors. Thus, there is much interest in merging previously distinct software and hardware simulation tools.

One simple but naive form of integration is to create an HDL model of the microprocessor that will run the software of a system, and then integrate that model with the HDL models of the remaining single-purpose processors. While straightforward to implement, simulating a microprocessor in an HDL has two key disadvantages. First, this approach will be much slower than an ISS, since the HDL simulator represents an extra layer of software that must be executed. Second, such an approach ignores the fact that ISSs have excellent controllability and observability features that designers have become accustomed to.

As it turns out, in many embedded systems, those processors do have frequent communication. Therefore, modern hardware-software co-simulators do more than just integrate two simulators. They also seek to minimize the communication between those simulators. Consider, for example, a system having one microprocessor, one single-purpose processor representing a coprocessor, and one memory, all connected using a single shared bus. Suppose the microprocessor's program is stored in this memory, and that the coprocessor uses the memory extensively also. We can simulate the microprocessor using an ISS and the coprocessor using an HDL. But where should the shared memory be modeled, in the ISS or the HDL? If in the HDL, then on every instruction, the ISS will need to stall in order to communicate with the HDL simulator to fetch the next instruction from memory. If in the ISS, then the HDL simulator will need to stall in order interrupt the ISS for access to the memory. However, note that most of these stalls are probably not necessary. For example, the ISS accesses of its instructions in memory are really irrelevant to the coprocessor. Likewise, the coprocessor's manipulation of data in memory is not relevant to the microprocessor, except in cases where that data is being transferred between the processors using the memory.

Narasaraopeta Engineering College (Autonomous)

Kotappakonda Road, Yellamanda (P.O), Narasaraopet- 522601, Guntur District, AP.

Subject Code: R16EC4211

IV B.Tech II Semester Adv. Supple Examinations, December-2020

EMBEDDED SYSTEM DESIGN

(ECE)

Time: 3 hours **Max Marks: 60**

Question Paper Consists of **Part-A** and **Part-B.**
Answering the question in **Part-A** is Compulsory & Four Questions should be answered from Part-B
All questions carry equal marks of 12.

PART-A **[2+2+2+2+2+2]**

1.	(a)	Define "Time-to-market".	2M
	(b)	What is single-purpose processor? What are the benefits of choosing a single purpose processor over a general purpose processor	2M
	(c)	How to evaluate a embedded processor's speed?	2M
	(d)	What is the key feature of the PCI bus?	2M
	(e)	Explain the important aspect of a real time system.	2M
	(f)	Why Antifuses are implemented in a PLD?	2M

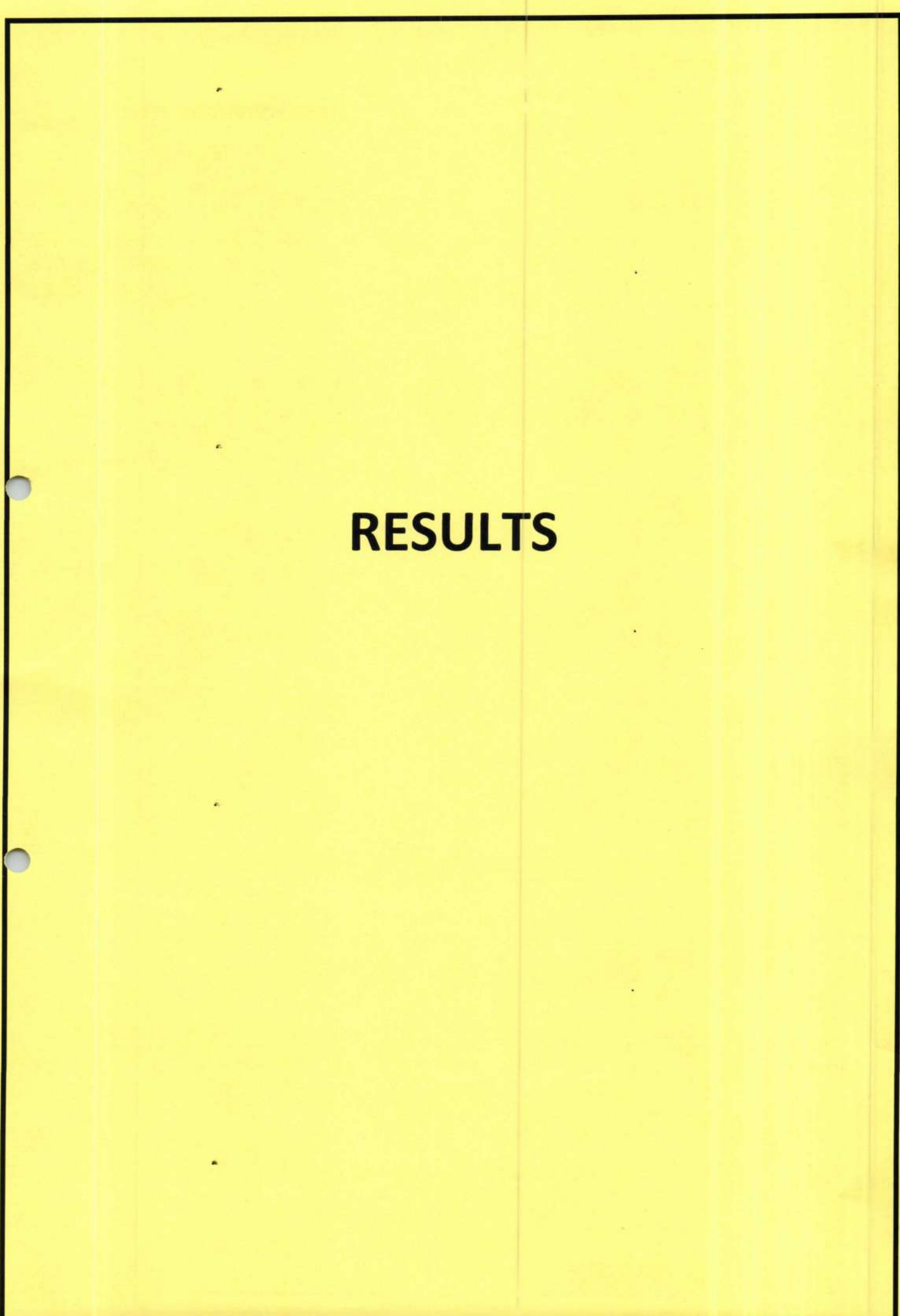
PART-B

4 X 12 = 48

2.	(a)	Describe the characteristics of an embedded system in detail.	6M
	(b)	Explain the various purposes of embedded systems in detail with illustrative examples.	6M
3.	(a)	Describe the procedure of designing a general-purpose processor.	6M
	(b)	Design greatest common divisor based on custom single-purpose processor basic model.	6M
4.	(a)	Illustrate how program and data memory fetches can be overlapped in a Harvard architecture.	6M
	(b)	Explain about main software utility tool	6M
5.		Explain the Dynamic RAM,Pseudo static RAM and Static RAM	12M
6.	(a)	Briefly describe three computation models commonly used to describe embedded systems and their peripherals.	6M
	(b)	Show why, in addition to ordered locking is necessary to avoid deadlocks.	6M
7.	(a)	Define various IC technologies and discuss the benefits of using them.	6M
	(b)	What are general-purpose processor design models? And explain briefly any one.	6M

Missing Topics(Course gaps) and Topics beyond Syllabus

- DSP used in embedded systems
- Watch dog timer
- RS232
- Bluetooth
- I2C
- Multitasking
- RTOS
- Priority inversion
- Message Queue
- Mailbox and Pipe
- Kernel
- Thread
- Semaphore
- Internet of Things (IoT)
- Machine-to-Machine (M2M)



RESULTS

NARASARAOPETA ENGINEERING COLLEGE::NARASARAOPET
(AUTONOMOUS)
17 BATCH IV B.TECH II SEMESTER (R16) FINAL INTERNAL MARKS-2021

BRANCH/SEC - ECE/D			SUBJECT NAME & CODE :EMBEDDED SYSTEM DESIGN (R16EC4211)									
SL.NO.	H.T.NO.	STUDENT NAME	A1	A2	D1	O1	CYCLE-1	A3	A4	D2	O2	CYCLE-2	TOTAL
1	16475A0415	POLLA SIVA	A	A	12	5	17	7	A	18	8	33	29
2	16471A0424	GOLLA VENKATESWARI	6	6	2	A	8	9	A	10	6	25	21
3	16471A04D2	THALLAPALLI SIVANAGARAJU	A	7	7	6	20	9	A	15	10	34	31
4	16471A04G1	SRIRAM NAVEEN KUMAR	A	7	4	6	17	A	A	15	6	21	20
5	17471A04I1	CHERUKURI RAVI KUMAR	6	5	12	9	27	8	8	14	6	28	28
6	17471A04I2	KOLA PAVAN KALYAN	A	5	12	10	27	7	A	16	7	30	30
7	17471A04I3	TEMPALLI PRABHU KUMAR	8	6	19	10	37	7	9	15	9	33	36
8	17471A04I4	SHAIK MASTANVALI	8	6	20	10	38	8	A	15	8	31	37
9	17471A04I5	TANNIRU AVINASH BABU	6	A	17	10	33	A	7	15	10	32	33
10	17471A04I6	TELLAGORLA MANIKANTA GOPALA KRISHNA	A	7	20	10	37	8	A	14	9	31	36
11	17471A04I7	KOLA RAKESH	5	8	14	8	30	5	A	14	7	26	29
12	17471A04I8	KESANUPALLI PRIYANKA	6	8	20	6	34	10	9	14	3	27	33
13	17471A04I9	NARISETTI UMAMAHESWARI	7	8	20	10	38	9	A	19	7	35	38
14	17471A04J0	MANYAM UDAY BHASKAR	6	0	12	10	28	0	10	14	8	32	31
15	17471A04J1	Y SUPRAJA	7	7	18	10	35	8	9	16	9	34	35
16	17471A04J2	GOSULA THIRUPATHI RAO	6	6	15	10	31	10	A	15	7	32	32
17	17471A04J3	NARE TEJASWI	8	6	20	10	38	7	8	14	9	31	37
18	17471A04J4	VAKA GOPI CHAND	A	7	15	8	30	6	8	14	9	31	31
19	17471A04J5	DOPPALAPUDI NELSON RAJU	7	7	14	8	29	8	A	14	6	28	29
20	17471A04J6	SHAIK AFRIN	6	8	14	10	32	10	7	15	10	35	35
21	17471A04J7	MOLAMANTI SAIKALYAN	5	6	12	10	28	6	A	15	10	31	31
22	17471A04J8	SHAIK KANDIPATI MOULALI	7	A	20	10	37	7	A	14	7	28	35
23	17471A04K0	PALLEPOGU SHARONU	6	8	20	10	38	9	8	17	8	34	37
24	17471A04K1	CHILAKA VAMSI KRISHNA	5	5	16	8	29	7	8	16	8	32	32
25	17471A04K2	GOCHIPATHALA RAJ KAMAL	A	5	15	10	30	5	8	16	9	33	33
26	17471A04K3	KESENAPALLI MARIYA BABU	6	7	14	9	30	8	9	15	10	34	33
27	17471A04K4	ANGALAKURTHI SUMA PRIYA	6	5	16	9	31	10	A	15	4	29	31
28	17471A04K5	PERUMALLA VINAY KUMAR	A	7	10	9	26	8	A	15	8	31	30
29	17471A04K6	SHAIK SAJID HASAN	A	7	4	10	21	A	A	0	A	0	16
30	17471A04K7	USAA PAVAN KALYAN	6	A	11	6	23	5	A	14	9	28	27
31	17471A04K8	GORANTLA ASHOK	7	6	16	10	33	6	7	13	7	27	32
32	17471A04K9	NALLAMOLU KUSUMA	6	5	19	10	35	8	7	15	9	32	35
33	17471A04L0	JUPUDI RAJU	5	5	20	9	34	6	A	15	9	30	33
34	17471A04L1	MOGAL IRFAN	7	A	16	10	33	6	7	14	9	30	33
35	17471A04L2	VELISALA MANISH PREETHAM	A	7	18	10	35	7	A	15	9	31	34
36	17471A04L4	V PREETHI MANISHA	6	6	2	9	17	8	A	16	4	28	26
37	17471A04L5	JAMPANI KRISHNAVAMSI	A	5	4	4	13	9	A	18	9	36	31
38	17471A04L6	MEDATATI DIVYA	6	8	20	10	38	8	8	18	8	34	37
39	17471A04L8	SADHU KOTESWA RAO	5	7	16	5	28	7	8	14	9	31	31

SL.NO.	H.T.NO.	STUDENT NAME	A1	A2	D1	O1	CYCLE-1	A3	A4	D2	O2	CYCLE-2	TOTAL
40	17471A04M0	BAPATLA VIJAYA LAKSHMI	5	8	20	10	38	9	7	20	4	33	37
41	17471A04M1	PATIBANDLA NARESH	A	5	2	9	16	5	A	16	9	30	27
42	17471A04M2	KANDULA GURU KIRAN	A	7	6	9	22	8	A	14	10	32	30
43	17471A04M3	DARSI ANIL KUMAR	A	A	0	A	0	A	8	14	7	29	22
44	17471A04M4	BATRAJU NAGA UMAMAHESH	4	6	2	2	10	7	7	16	7	30	25
45	17471A04M5	PALLEMPATI DURGA PRASAD	5	6	20	10	36	6	A	18	10	34	36
46	17471A04M6	PUTTA SRIKANTH	6	5	14	10	30	6	7	16	10	33	33
47	17471A04M7	MEKALA YASHWANTH KUMAR	5	6	16	10	32	8	7	12	10	30	32
48	17471A04M8	SHAIK IMRAN	7	6	6	9	22	A	A	11	9	20	22
49	17471A04N0	SHAIK MAHAMOOD SHAREEF	A	6	19	10	35	7	A	14	8	29	34
50	17471A04N1	MADHIREDDY ANIL KUMAR REDDY	7	A	20	10	37	8	A	19	10	37	37
51	17471A04N2	KADIYAM SUDHAKAR	A	5	14	6	25	A	A	15	4	19	24
52	17471A04N3	VEMULURI YASASWI	6	6	20	7	33	9	A	18	4	31	33
53	18475A0410	THOTA BHARATH	6	7	15	7	29	7	A	10	8	25	28
54	18475A0411	SIDDEALA DILEEP SAGAR	7	7	14	8	29	A	10	18	8	36	35
55	18475A0412	DOPPALAPUDI SARATH CHANDRA	A	7	12	10	29	A	8	10	10	28	29
56	18475A0413	JANGA MAHENDRA	A	6	12	6	24	7	A	7	4	18	23
57	18475A0414	AINAOLU GOPIKRISHNA	A	8	18	8	34	7	7	16	7	30	33
58	18475A0415	THUNGALA PRAMOD	A	8	12	6	26	8	A	16	8	32	31
59	18475A0416	SURUBULA LEELA PAVANKUMAR	A	6	11	5	22	6	A	10	5	21	22
60	18475A0417	SARIKONDA RAMA KRISHNAM RAJU	6	5	14	5	25	7	A	10	7	24	25
61	18475A0418	ATTULURI SYAM PRASAD	4	7	12	6	25	5	A	10	10	25	25
62	18475A0419	KUNDA JASHUVA	7	7	18	10	35	8	8	16	9	33	35
63	18475A0420	YALAVARTHI MADHU BABU	A	8	11	9	28	7	A	16	9	32	31
64	18475A0421	M SUBRAHMANYAM	A	6	15	10	31	7	10	14	9	33	33
65	18475A0422	GUDISE VENKATESH	7	6	15	5	27	7	8	10	5	23	26
66	18475A0423	VARIKUTI KRISHNANJANEYULU	6	7	16	6	29	7	A	15	8	30	30

NARASARAOPETA ENGINEERING COLLEGE::NARASARAOPET

(AUTONOMOUS)

17 BATCH IV B.TECH II SEMESTER (R16) FINAL INTERNAL MARKS-2021

BRANCH/SEC - ECE/B			SUBJECT NAME & CODE :EMBEDDED SYSTEM DESIGN (R16EC4211)										
SL.NO.	H.T.NO.	STUDENT NAME	A1	A2	D1	O1	CYCLE-1	A3	A4	D2	O2	CYCLE-2	TOTAL
1	17471A0461	PONUGOTI RAMESH	10	9	10	8	28	10	A	4	7	21	27
2	17471A0462	MATTRAM VISHNU BABU	10	A	10	7	27	A	7	6	6	19	25
3	17471A0463	RAMISETTY RAMCHARAN	10	9	14	8	32	10	A	12	8	30	32
4	17471A0464	ANANTHA DURGA	A	9	12	7	28	9	10	9	9	28	28
5	17471A0465	KAMMA NAGA SAI RITHVIK	A	9	6	10	25	10	8	8	9	27	27
6	17471A0467	DANDE NAGALAKSHMI	10	9	14	8	32	10	10	14	9	33	33
7	17471A0468	JANAPATI YASASWINI JAYA BHARATHI SAHITHI	10	9	18	10	38	10	10	15	9	34	37
8	17471A0469	JANAPATI SAILAKSHMI SRAVANI	10	7	16	10	36	10	A	17	9	36	36
9	17471A0470	KUNISETTY GOPINADH	A	A	16	9	25	A	9	16	4	29	28
10	17471A0471	SHAIK MD YASIN	10	9	16	9	35	9	9	11	8	28	34
11	17471A0472	RAMISETTI LAKSHMISAITEJA	A	9	14	9	32	8	9	12	9	30	32
12	17471A0473	B. MANI DEEPAK	A	A	7	7	14	A	A	10	8	18	17
13	17471A0474	BOKKA JOHN VICTOR	A	7	19	9	35	9	A	11	3	23	32
14	17471A0475	GODUGUNURI VIJAYA SAI DILEEP KUMAR REDDY	10	7	10	9	29	10	A	17	8	35	34
15	17471A0476	GANAPATHI JYOTHI PRAKASH	10	10	18	9	37	10	A	17	8	35	37
16	17471A0477	MUNAGAPATI MANOJKUMAR	A	9	20	10	39	10	10	15	6	31	37
17	17471A0478	SYED MD GOUSE	10	9	16	9	35	10	A	13	9	32	35
18	17471A0479	GOLI SRINIVASARAO	10	A	20	9	39	10	A	16	9	35	38
19	17471A0480	PATHI VENKATESWARI	10	A	19	8	37	9	10	14	9	33	36
20	17471A0481	VANUKURI HARIVARDHAN VEERA REDDY	10	A	16	9	35	8	10	13	9	32	35
21	17471A0482	PANCHUMARTHI DILEEP KUMAR	7	9	19	9	37	10	10	16	8	34	37
22	17471A0483	KOLLURU KRISHNA MOHAN	9	9	19	9	37	9	10	14	6	30	36
23	17471A0484	RACHUMALLU SASIDHAR	A	9	16	10	35	8	10	17	8	35	35
24	17471A0485	KARNATI HEMANTH SAI	9	9	18	9	36	9	8	16	10	35	36
25	17471A0486	THUMATI VENKATA SUNIL	8	A	9	8	25	10	A	10	10	30	29
26	17471A0487	KOPPURAVURI AKHILA	10	9	16	10	36	10	A	16	9	35	36
27	17471A0488	NAIDU RACHANA	10	9	17	10	37	10	10	13	9	32	36
28	17471A0489	TELAPROLU PAVAN KALYAN	10	10	20	10	40	10	A	10	7	27	37
29	17471A0490	BODEMPUDI SRI HARSHA	10	9	20	10	40	10	A	16	9	35	39
30	17471A0491	SHAIK RUKSANA	7	9	19	10	38	10	A	11	9	30	36
31	17471A0492	KATTAMURI SATYANARAYANA	10	9	20	10	40	10	10	12	8	30	38
32	17471A0493	KOPPULA GANESH REDDY	10	9	15	10	35	9	10	14	7	31	34
33	17471A0494	SYED MAHABOOB JANI BASHA	10	9	20	9	39	10	10	12	8	30	37

34	17471A0495	PERUMALLA PREETHI KOUMIKA	10	9	20	9	39	10	10	14	8	32	38
35	17471A0496	SHAIK JANI BASHA	A	10	5	8	23	10	A	10	9	29	28
36	17471A0497	SHAIK MOHAMMED ALTHAF	A	9	14	8	31	10	A	12	10	32	32
37	17471A0498	JANGALA KIRAN BABU	7	8	13	10	31	A	10	10	6	26	30
38	17471A0499	RAMA CHANDRULA KAVYASRI	10	10	18	8	36	10	A	15	7	32	35
39	17471A04A0	MUVVA MANOJ KUMAR	10	9	19	10	39	10	9	14	7	31	37
40	17471A04A1	KAKUMANU SUMANTH	9	9	18	8	35	10	A	10	9	29	34
41	17471A04A2	YANDAPALLI N V S L MALLIKA BRAMARAMBIKA	10	9	18	6	34	10	A	18	5	33	34
42	17471A04A3	TUMMALACHERUVU SAITEJA	7	10	18	10	38	10	10	12	9	31	37
43	17471A04A4	JAKKIREDDY KEERTHI	10	A	14	6	30	10	A	9	10	29	30
44	17471A04A5	SHAIK AFRID	10	8	19	7	36	10	A	16	8	34	36
45	17471A04A6	YERRAMSETTY SAI PAVAN	7	10	13	10	33	10	9	11	6	27	32
46	17471A04A7	DESABOYINA HEMARAMACHANDRA VASU	7	9	16	8	33	10	10	17	4	31	33
47	17471A04A8	SHAIK TANGEDA CHINA BAJI	7	A	18	4	29	10	A	12	5	27	29
48	17471A04A9	KOLLA SIVA HEMANTH	10	8	20	10	40	10	A	15	10	35	39
49	17471A04B0	AKULA ASHOK KUMAR	10	10	18	10	38	10	A	15	10	35	38
50	17471A04B1	MOHAMMED ZAKIR HUSSAIN KHAN	10	9	18	7	35	10	A	17	6	33	35
51	17471A04B2	BALUPUNURI KASU VASU DEVA VENKATA REDDY	A	A	14	10	24	A	8	5	9	22	24
52	17471A04B3	GORANTLA SRAVAN KRISHNA	10	9	16	8	34	9	A	6	9	24	32
53	17471A04B4	JAMMULA CHANDRIKA	10	9	16	10	36	10	A	16	9	35	36
54	17471A04B5	BONDE RAJENDRA	10	A	15	8	33	A	9	6	4	19	30
55	17471A04B6	NANNEM VEENA VATSALYA	10	9	16	5	31	A	10	8	8	26	30
56	17471A04B7	GOGULA NAVEEN KUMAR	A	9	15	4	28	10	A	A	A	10	24
57	17471A04B8	KURANGI MUKUNDA SAI	10	8	15	10	35	A	9	8	8	25	33
58	17471A04B9	GOUSE MOMITH BAIG	10	9	20	8	38	10	10	12	5	27	36
59	17471A04C0	BUSSI JOSEPH BALA YASWANTH BABU	10	9	19	10	39	10	A	19	9	38	39

NARASARAOPETA ENGINEERING COLLEGE::NARASARAOPET

(AUTONOMOUS)

17 BATCH IV B.TECH II SEMESTER (R16) FINAL INTERNAL MARKS-2021

BRANCH/SEC - ECE/C			SUBJECT NAME & CODE :EMBEDDED SYSTEM DESIGN (R16EC4211)									
SL.NO.	H.T.NO.	STUDENT NAME	A1	A2	D1	O1	CYCLE-1	A3	A4	D2	O2	CYCLE-2	TOTAL
1	17471A04C1	VATTIKONDA SIVA RAMAKRISHNA	A	5	9	10	24	7	8	4	9	21	24
2	17471A04C2	GUNDA PRATHYUSHA	7	6	18	10	35	8	A	18	9	35	35
3	17471A04C3	ALLA SARATH SAI	0	6	19	8	33	5	A	16	7	28	32
4	17471A04C4	GARIKAPATI PAVAN KALYAN	6	6	19	10	35	5	7	18	9	34	35
5	17471A04C5	GUTHA VENKAT RAO	0	6	10	8	24	6	8	7	8	23	24
6	17471A04C6	PULUKURI SRI PRASANNA	A	5	18	10	33	7	5	10	10	27	32
7	17471A04C7	BANDI CHINNAPA REDDY	6	6	20	10	36	9	A	10	10	29	35
8	17471A04C8	PINNIKA SRIVANI	8	6	20	10	38	10	A	13	10	33	37
9	17471A04C9	MADDI KOTI KIRAN KUMAR	4	7	16	10	33	10	A	16	5	31	33
10	17471A04D0	DODDAKULA PRASANTH	1	6	14	8	28	5	A	7	10	22	27
11	17471A04D1	SHAIK ARSHAD	2	5	14	10	29	6	8	14	8	30	30
12	17471A04D2	PALADUGU SRINIVASULU	2	7	11	9	27	9	A	12	8	29	29
13	17471A04D3	KONETI SUNEEL	5	6	16	10	32	5	8	14	10	32	32
14	17471A04D4	VUTUKURI TULASIRAM	7	7	16	10	33	10	A	16	9	35	35
15	17471A04D5	KOMMI VENKATA SUBRAMANYAM	0	6	13	10	29	7	A	8	9	24	28
16	17471A04D6	SHAIK SHAMEEM	A	6	20	10	36	8	10	15	10	35	36
17	17471A04D7	BANDARU MAHESH	A	6	0	10	16	6	6	7	10	23	22
18	17471A04D8	GANGASANI ASHOK REDDY	A	6	9	9	24	6	A	13	10	29	28
19	17471A04D9	KOSANA PRATAP	3	5	6	10	21	7	8	14	9	31	29
20	17471A04E0	G RAGA VENKATA DEEPTHI	A	7	10	10	27	8	A	10	8	26	27
21	17471A04E1	POTHURI MANIKANTA	A	5	14	9	28	6	7	13	8	28	28
22	17471A04E2	SHAIK NANNU SHAIDA	6	8	20	10	38	9	A	18	7	34	37
23	17471A04E3	POLU DIVYA	A	8	20	10	38	8	A	15	8	31	37
24	17471A04E4	VIBHARAMPATTAPU MAMATHA	A	7	20	10	37	9	10	14	10	34	37
25	17471A04E5	MANDAVA SRI BHARATHI	7	9	20	9	38	10	10	18	9	37	38
26	17471A04E6	BHOJANAPALLI TEJASWINI	1	8	20	10	38	10	A	20	10	40	40
27	17471A04E7	SHAIK JAVEED	6	7	5	10	22	9	A	13	8	30	28
28	17471A04E8	KOLLIKONDA GANGABHAVANI	6	9	20	10	39	9	7	18	8	35	38
29	17471A04E9	GUNTA ROHITHA REDDY	A	8	13	10	31	5	8	16	10	34	34
30	17471A04F0	LAKSHMISETTY VENKATA SAI VYSHNAVI	A	8	17	9	34	9	A	14	5	28	33
31	17471A04F1	RAVULAPALLI SRINU	0	5	8	10	23	6	7	16	9	32	30
32	17471A04F2	SADINENI SOWJANYA	3	6	18	10	34	7	8	15	5	28	33
33	17471A04F3	GANJI KRANTHI	5	8	A	A	8	8	A	0	9	17	15

34	17471A04F4	CHEVALA PRITHVI RAJ	A	4	10	10	24	6	7	14	6	27	27
35	17471A04F5	DEVARAPALLI NAGA POOJA SAI SRI	A	8	20	10	38	10	A	16	10	36	38
36	17471A04F6	MEKAPOTHULA GOPI KRISHNA	0	9	A	A	9	7	8	17	9	34	28
37	17471A04F7	REPALLE PRATHYUSHA	A	8	17	10	35	10	A	16	6	32	35
38	17471A04F8	KOMMANABOYINA NAGA ANIL	0	6	0	7	13	5	6	13	9	28	25
39	17471A04F9	BATCHU DURGA BHAVANI	0	7	19	10	36	9	9	15	10	34	36
40	17471A04G0	DIRISALA SRAVANI	0	7	14	6	27	9	A	19	4	32	31
41	17471A04G1	KOMIREDDY MANJU BHARGAVI	0	4	14	10	28	7	A	14	8	29	29
42	17471A04G2	POLA VENKATA MALLIKHARJUNA RAO	6	6	20	10	36	6	7	16	9	32	35
43	17471A04G3	KOLIPAKULA DEVI CHAMUNDESWARI	A	7	20	10	37	8	A	9	9	26	35
44	17471A04G4	BOBBA PRASANTH	A	8	7	9	24	6	A	8	9	23	24
45	17471A04G5	G JAGADEESH CHANDRA BOSE	0	8	15	10	33	7	6	11	9	27	32
46	17471A04G6	GUNTU NAVEEN CHOWDARY	1	6	12	10	28	8	A	11	9	28	28
47	17471A04G8	YELLANURU JAHNAVI	A	6	7	6	19	7	A	7	6	20	20
48	17471A04G9	MAMIDIPAKA NAGASUSHMA	A	7	20	10	37	10	A	16	10	36	37
49	17471A04H0	DUGGARAJU GOWTHAMY	6	8	20	9	37	10	7	14	7	31	36
50	17471A04H1	P RAMA KRISHNA	7	6	18	8	33	6	6	12	9	27	32
51	17471A04H2	NALAGANGULA KOTIREDDY	2	6	11	10	27	7	A	8	7	22	26
52	17471A04H3	GUNTUPALLI THIRUMALA PRASANNA SANKAR	A	A	13	8	21	6	7	7	8	22	22
53	17471A04H4	RAJARAPU SRILAKSHMI TIRUMALESWARI	A	6	20	10	36	7	10	18	4	32	35
54	17471A04H5	PAMURU DIVYA	A	7	10	10	27	7	A	11	9	27	27
55	17471A04H6	SHAIK SAMEER	2	8	18	10	36	10	A	10	8	28	34
56	17471A04H7	KUNCHALA GOPI KRISHNA	0	6	16	9	31	6	7	12	10	29	31
57	17471A04H8	PUSALA MADHU KUMAR	0	7	14	8	29	8	A	13	8	29	29
58	17471A04H9	GAJJALA MARUTHI VENKATA KRISHNA REDDY	6	6	16	10	32	8	A	8	6	22	30
59	17471A04I0	MEDA RAVI TEJA	4	3	12	9	25	6	7	14	10	31	30
60	18475A0401	MARELLA VAMSI	0	6	20	4	30	A	7	14	5	26	29
61	18475A0402	RACHAKONDA SHYAM PREMKUMAR	0	5	12	10	27	9	9	20	9	38	36
62	18475A0403	SURISETTI ARCHANA	A	7	12	9	28	6	6	11	6	23	27
63	18475A0404	PALADUGU GANESH	A	6	20	10	36	5	9	20	9	38	38
64	18475A0405	VUYYURU SRILAKSHMI	0	8	10	10	28	8	8	11	6	25	28
65	18475A0406	NUNNA VENKATA SIVASAI	8	7	20	10	38	9	10	20	8	38	38
66	18475A0407	PARITALA HARITHA	8	6	20	10	38	10	8	20	7	37	38
67	18475A0408	MADDINA VENKATA SANDEEP	4	5	16	6	27	7	A	16	5	28	28
68	18475A0409	CHERUKURI BRAHMA VENKATESWARLU	0	7	15	10	32	5	8	6	5	19	29

NARASARAOPETA ENGINEERING COLLEGE::NARASARAOPET
(AUTONOMOUS)
17 BATCH IV B.TECH II SEMESTER (R16) FINAL INTERNAL MARKS-2021

BRANCH/SEC - ECE/D			SUBJECT NAME & CODE :EMBEDDED SYSTEM DESIGN (R16EC4211)										
SL.NO.	H.T.NO.	STUDENT NAME	A1	A2	D1	O1	CYCLE-1	A3	A4	D2	O2	CYCLE-2	TOTAL
1	16475A0415	POLLA SIVA	A	A	12	5	17	7	A	18	8	33	29
2	16471A0424	GOLLA VENKATESWARI	6	6	2	A	8	9	A	10	6	25	21
3	16471A04D2	THALLAPALLI SIVANAGARAJU	A	7	7	6	20	9	A	15	10	34	31
4	16471A04G1	SRIRAM NAVEEN KUMAR	A	7	4	6	17	A	A	15	6	21	20
5	17471A04I1	CHERUKURI RAVI KUMAR	6	5	12	9	27	8	8	14	6	28	28
6	17471A04I2	KOLA PAVAN KALYAN	A	5	12	10	27	7	A	16	7	30	30
7	17471A04I3	TEMPALLI PRABHU KUMAR	8	6	19	10	37	7	9	15	9	33	36
8	17471A04I4	SHAIK MASTANVALI	8	6	20	10	38	8	A	15	8	31	37
9	17471A04I5	TANNIRU AVINASH BABU	6	A	17	10	33	A	7	15	10	32	33
10	17471A04I6	TELLAGORLA MANIKANTA GOPALA KRISHNA	A	7	20	10	37	8	A	14	9	31	36
11	17471A04I7	KOLA RAKESH	5	8	14	8	30	5	A	14	7	26	29
12	17471A04I8	KESANUPALLI PRIYANKA	6	8	20	6	34	10	9	14	3	27	33
13	17471A04I9	NARISETTI UMAMAHESWARI	7	8	20	10	38	9	A	19	7	35	38
14	17471A04J0	MANYAM UDAY BHASKAR	6	0	12	10	28	0	10	14	8	32	31
15	17471A04J1	Y SUPRAJA	7	7	18	10	35	8	9	16	9	34	35
16	17471A04J2	GOSULA THIRUPATHI RAO	6	6	15	10	31	10	A	15	7	32	32
17	17471A04J3	NARE TEJASWI	8	6	20	10	38	7	8	14	9	31	37
18	17471A04J4	VAKA GOPI CHAND	A	7	15	8	30	6	8	14	9	31	31
19	17471A04J5	DOPPALAPUDI NELSON RAJU	7	7	14	8	29	8	A	14	6	28	29
20	17471A04J6	SHAIK AFRIN	6	8	14	10	32	10	7	15	10	35	35
21	17471A04J7	MOLAMANTI SAIKALYAN	5	6	12	10	28	6	A	15	10	31	31
22	17471A04J8	SHAIK KANDIPATI MOULALI	7	A	20	10	37	7	A	14	7	28	35
23	17471A04K0	PALLEPOGU SHARONU	6	8	20	10	38	9	8	17	8	34	37
24	17471A04K1	CHILAKA VAMSI KRISHNA	5	5	16	8	29	7	8	16	8	32	32
25	17471A04K2	GOCHIPATHALA RAJ KAMAL	A	5	15	10	30	5	8	16	9	33	33
26	17471A04K3	KESENAPALLI MARIYA BABU	6	7	14	9	30	8	9	15	10	34	33
27	17471A04K4	ANGALAKURTHI SUMA PRIYA	6	5	16	9	31	10	A	15	4	29	31
28	17471A04K5	PERUMALLA VINAY KUMAR	A	7	10	9	26	8	A	15	8	31	30
29	17471A04K6	SHAIK SAJID HASAN	A	7	4	10	21	A	A	0	A	0	16
30	17471A04K7	USAA PAVAN KALYAN	6	A	11	6	23	5	A	14	9	28	27
31	17471A04K8	GORANTLA ASHOK	7	6	16	10	33	6	7	13	7	27	32
32	17471A04K9	NALLAMOLU KUSUMA	6	5	19	10	35	8	7	15	9	32	35
33	17471A04L0	JUPUDI RAJU	5	5	20	9	34	6	A	15	9	30	33
34	17471A04L1	MOGAL IRFAN	7	A	16	10	33	6	7	14	9	30	33
35	17471A04L2	VELISALA MANISH PREETHAM	A	7	18	10	35	7	A	15	9	31	34
36	17471A04L4	V PREETHI MANISHA	6	6	2	9	17	8	A	16	4	28	26
37	17471A04L5	JAMPANI KRISHNAVAMSI	A	5	4	4	13	9	A	18	9	36	31
38	17471A04L6	MEDATATI DIVYA	6	8	20	10	38	8	8	18	8	34	37
39	17471A04L8	SADHU KOTESWA RAO	5	7	16	5	28	7	8	14	9	31	31

SL.NO.	H.T.NO.	STUDENT NAME	A1	A2	D1	O1	CYCLE-1	A3	A4	D2	O2	CYCLE-2	TOTAL
40	17471A04M0	BAPATLA VIJAYA LAKSHMI	5	8	20	10	38	9	7	20	4	33	37
41	17471A04M1	PATIBANDLA NARESH	A	5	2	9	16	5	A	16	9	30	27
42	17471A04M2	KANDULA GURU KIRAN	A	7	6	9	22	8	A	14	10	32	30
43	17471A04M3	DARSI ANIL KUMAR	A	A	0	A	0	A	8	14	7	29	22
44	17471A04M4	BATRAJU NAGA UMAMAHESH	4	6	2	2	10	7	7	16	7	30	25
45	17471A04M5	PALLEMPATI DURGA PRASAD	5	6	20	10	36	6	A	18	10	34	36
46	17471A04M6	PUTTA SRIKANTH	6	5	14	10	30	6	7	16	10	33	33
47	17471A04M7	MEKALA YASHWANTH KUMAR	5	6	16	10	32	8	7	12	10	30	32
48	17471A04M8	SHAIK IMRAN	7	6	6	9	22	A	A	11	9	20	22
49	17471A04N0	SHAIK MAHAMOOD SHAREEF	A	6	19	10	35	7	A	14	8	29	34
50	17471A04N1	MADHIREDDY ANIL KUMAR REDDY	7	A	20	10	37	8	A	19	10	37	37
51	17471A04N2	KADIYAM SUDHAKAR	A	5	14	6	25	A	A	15	4	19	24
52	17471A04N3	VEMULURI YASASWI	6	6	20	7	33	9	A	18	4	31	33
[illegible]3	18475A0410	THOTA BHARATH	6	7	15	7	29	7	A	10	8	25	28
54	18475A0411	SIDDEALA DILEEP SAGAR	7	7	14	8	29	A	10	18	8	36	35
55	18475A0412	DOPPALAPUDI SARATH CHANDRA	A	7	12	10	29	A	8	10	10	28	29
56	18475A0413	JANGA MAHENDRA	A	6	12	6	24	7	A	7	4	18	23
57	18475A0414	AINAOLU GOPIKRISHNA	A	8	18	8	34	7	7	16	7	30	33
58	18475A0415	THUNGALA PRAMOD	A	8	12	6	26	8	A	16	8	32	31
59	18475A0416	SURUBULA LEELA PAVANKUMAR	A	6	11	5	22	6	A	10	5	21	22
60	18475A0417	SARIKONDA RAMA KRISHNAM RAJU	6	5	14	5	25	7	A	10	7	24	25
61	18475A0418	ATTULURI SYAM PRASAD	4	7	12	6	25	5	A	10	10	25	25
62	18475A0419	KUNDA JASHUVA	7	7	18	10	35	8	8	16	9	33	35
63	18475A0420	YALAVARTHI MADHU BABU	A	8	11	9	28	7	A	16	9	32	31
64	18475A0421	M SUBRAHMANYAM	A	6	15	10	31	7	10	14	9	33	33
65	18475A0422	GUDISE VENKATESH	7	6	15	5	27	7	8	10	5	23	26
[illegible]6	18475A0423	VARIKUTI KRISHNANJANEYULU	6	7	16	6	29	7	A	15	8	30	30

(R16) 2017 BATCH IV B.TECH. II SEMESTER RESULT AFTER REVALUATION JULY - 2021

BRANCH : ECE

20.09.2021

SLNO	HT.NO	NAME	R16EC4201 CELLULAR AND MOBILE COMMUNICATIONS			R16EC4203 WIRELESS SENSOR NETWORKS			R16EC4211 EMBEDDED SYSTEM DESIGN			R16EC42PTI PRACTICAL TRAINING/INTERNSHIP			R16EC42PW PROJECT WORK			SGPA
			G	C	R	G	C	R	G	C	R	G	C	R	G	C	R	
1	16471A0405	NELAKUDITI VENKATA SIVA SAI PRUDHVI KUMAR	B	3	PASS	E	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.18
2	16471A0424	GOLLA VENKATESWARI	F	0	FAIL	F	0	FAIL	F	0	FAIL	A	3	PASS	A	10	PASS	NA
3	16471A04D2	THALLAPALII SIVA NAGA RAJU	P	3	PASS	P	3	PASS	C	3	PASS	A	3	PASS	A	10	PASS	6.91
4	16471A04G1	SRIRAM NAVEENKUMAR	F	0	FAIL	P	3	PASS	P	3	PASS	A	3	PASS	A	10	PASS	NA
5	16475A0415	POLLA SIVA	P	3	PASS	P	3	PASS	P	3	PASS	A	3	PASS	A	10	PASS	6.77
6	17471A0401	KOLLA CHAKRI SAI VIJAYACHANDRA	C	3	PASS	B	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
7	17471A0403	MANAM YASWANTH CHOWDARY	C	3	PASS	A	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.05
8	17471A0404	CHINTAGUNTLA KALYAN KUMAR	C	3	PASS	B	3	PASS	C	3	PASS	O	3	PASS	O	10	PASS	8.5
9	17471A0405	YERUVA SUDHEER KUMAR REDDY	C	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.77
10	17471A0406	KOLISETTY BABA SRI RAM KUMAR	P	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.64
11	17471A0407	BATTULA CHANDAN	P	3	PASS	C	3	PASS	P	3	PASS	E	3	PASS	E	10	PASS	7.5
12	17471A0408	KOTABHATTAR V V S PRATHYUSHA	B	3	PASS	A	3	PASS	C	3	PASS	O	3	PASS	O	10	PASS	8.77
13	17471A0409	CHERUKULA KASI MAITHRI	B	3	PASS	E	3	PASS	E	3	PASS	O	3	PASS	O	10	PASS	9.32
14	17471A0410	KANCHETI VINAY	C	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.77
15	17471A0411	YAKKALA NAGA MADAN DATHA KUMAR	B	3	PASS	A	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.18
16	17471A0412	SANAMPUDI VENKATA NARASIMHA REDDY	C	3	PASS	B	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
17	17471A0413	GADAM RAM BHUPAL REDDY	B	3	PASS	B	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.05
18	17471A0414	YANDAPALLI SAI VAMSIKRISHNA	A	3	PASS	A	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.05
19	17471A0415	MAMILLAPALLI SAI RAM	B	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.05
20	17471A0416	NEMALIDINNE VENKATAJAHNAVI	C	3	PASS	B	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.18
21	17471A0417	NEMALIDINNE VENKATA YASHASWINI	B	3	PASS	E	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.18
22	17471A0418	MANDALA SAI BHARGAV REDDY	F	0	FAIL	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	NA
23	17471A0419	MAMIDIPAKA SAI SRIDHAR	C	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	8.91
24	17471A0420	POTHURI YASWANTH GUPTHA	B	3	PASS	A	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	8.91
25	17471A0421	POPURI VENU	C	3	PASS	A	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.18
26	17471A0422	KALANGI KRISHNA AKHIL	B	3	PASS	E	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.18
27	17471A0423	DESABOINA PRUDHVISAI	F	0	FAIL	F	0	FAIL	F	0	FAIL	E	3	PASS	E	10	PASS	NA
28	17471A0424	CHINNI ESWAR RAO	B	3	PASS	E	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.18
29	17471A0425	YAKKALA PRATHAP	B	3	PASS	A	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.45
30	17471A0426	POTHURI SAIPAVAN	C	3	PASS	A	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.32
31	17471A0427	KOPPURAVURI JEEVAN JITHENDRA	B	3	PASS	E	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.18
32	17471A0428	SANKARAPU SEKHAR BABU	C	3	PASS	E	3	PASS	E	3	PASS	O	3	PASS	O	10	PASS	9.18
33	17471A0429	NUTHALAPATI DURGA PRASAD	C	3	PASS	C	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	7.91
34	17471A0430	BOGGAVARAPU YASWANTH AMARESH	C	3	PASS	C	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	7.91
35	17471A0432	CHANDRAGIRI SAI PRAGNA	B	3	PASS	B	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	8.91
36	17471A0434	GADDAM VAMSI	F	0	FAIL	P	3	PASS	P	3	PASS	E	3	PASS	E	10	PASS	NA
37	17471A0435	MINDYALA NAGASAI	B	3	PASS	B	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.32
38	17471A0436	IRUVANTI SATYA SITA RAMA SASTRY	B	3	PASS	C	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.05
39	17471A0437	PANGA SRINIVASA RAO	C	3	PASS	B	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
40	17471A0438	POTHRALA RAMANJI	C	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.05
41	17471A0439	PASUPULETI SURESH	C	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.77
42	17471A0440	GUDIPATI CHARITHA	B	3	PASS	C	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	8.64
43	17471A0441	SHAIK SALMAN	C	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.77
44	17471A0442	MANDALAPU AKHIL SURYA	C	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.77
45	17471A0443	PABBA VENKATESH NAIDU	B	3	PASS	A	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.45
46	17471A0444	NANDHYALA LINGA REDDY	B	3	PASS	B	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	8.91
47	17471A0445	GANGAVARAPU TEJESWAR REDDY	C	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.77
48	17471A0446	TALLAPANENI VYSHNAVI	B	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.05
49	17471A0448	YELURI NAVYA	C	3	PASS	B	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	8.64
50	17471A0449	DORAGACHARLA PAVAN KUMAR REDDY	A	3	PASS	E	3	PASS	E	3	PASS	O	3	PASS	O	10	PASS	9.45
51	17471A0450	GOPALAM NAVYASRI	F	0	FAIL	F	0	FAIL	F	0	FAIL	E	3	PASS	E	10	PASS	NA
52	17471A0451	SHAIK ABTHAB	B	3	PASS	E	3	PASS	E	3	PASS	O	3	PASS	O	10	PASS	9.32
53	17471A0452	TADIKAMALLA SURESH BABU	P	3	PASS	P	3	PASS	P	3	PASS	A	3	PASS	A	10	PASS	6.77
54	17471A0453	THUMATI MUKESH CHOWDARY	A	3	PASS	A	3	PASS	O	3	PASS	O	3	PASS	O	10	PASS	9.45
55	17471A0454	ANEKALLA LAKSHMAN REDDY	C	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.05
56	17471A0455	RAGHUVU VENKAT SIVA RAMA NAGENDRA	C	3	PASS	B	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	8.64

57	17471A0456	NANDIKONDA ANJI REDDY	B	3	PASS	C	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.05
58	17471A0457	KAKUMANU GANESH KRISHNA SAI	C	3	PASS	B	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
59	17471A0459	RAMIDEVI SUMANTH	F	0	FAIL	F	0	FAIL	F	0	FAIL	E	3	PASS	E	10	PASS	NA
60	17471A0461	PONUGOTI RAMESH	P	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.64
61	17471A0462	MATTRAM VISHNU BABU	C	3	PASS	B	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
62	17471A0463	RAMISETTY RAMCHARAN	C	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.05
63	17471A0464	ANANTHA DURGA	C	3	PASS	B	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
64	17471A0465	KAMMA NAGA SAI RITHVIK	B	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.18
65	17471A0467	DANDE NAGALAKSHMI	B	3	PASS	B	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.32
66	17471A0468	JANAPATI YASASWINI JAYA BHARATHI SAHITHI	B	3	PASS	E	3	PASS	E	3	PASS	O	3	PASS	O	10	PASS	9.32
67	17471A0469	JANAPATI SAILAKSHMI SRAVANI	B	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.05
68	17471A0470	KUNISETTY GOPINADH	C	3	PASS	A	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.18
69	17471A0471	SHAIK MOHAMMAD YASEEN	C	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.05
70	17471A0472	RAMISETTI LAKSHMISAITEJA	P	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	7.91
71	17471A0473	BANDARU MANI DEEPAK	F	0	FAIL	P	3	PASS	P	3	PASS	A	3	PASS	A	10	PASS	NA
72	17471A0474	BOKKA JOHN VICTOR	B	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.18
73	17471A0475	GODUGUNURI VIJAYA SAI DILEEP KUMAR REDDY	C	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	8.91
74	17471A0476	GANAPATHI JYOTHI PRAKASH	C	3	PASS	B	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	8.77
75	17471A0477	MUNAGAPATI MANOJKUMAR	B	3	PASS	B	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.32
76	17471A0478	SYED MOHAMMAD GOUSE	C	3	PASS	C	3	PASS	B	3	PASS	A	3	PASS	E	10	PASS	7.77
77	17471A0479	GOLI SRINIVASARAO	B	3	PASS	B	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	8.77
78	17471A0480	PATHI VENKATESWARI	B	3	PASS	E	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.59
79	17471A0481	VANUKURI HARIVARDHAN VEERA REDDY	B	3	PASS	A	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	8.91
80	17471A0482	PANCHUMARTHI DILEEP KUMAR	B	3	PASS	B	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	8.91
81	17471A0483	KOLLURU KRISHNA MOHAN	B	3	PASS	B	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.32
82	17471A0484	RACHUMALLU SASIDHAR	A	3	PASS	B	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.45
83	17471A0485	KARNATI HEMANTH SAI	B	3	PASS	A	3	PASS	E	3	PASS	O	3	PASS	O	10	PASS	9.18
84	17471A0486	THUMATI VENKATA SUNIL	F	0	FAIL	F	0	FAIL	F	0	FAIL	E	3	PASS	E	10	PASS	NA
85	17471A0487	KOPPURAVURI AKHILA	E	3	PASS	A	3	PASS	E	3	PASS	O	3	PASS	O	10	PASS	9.45
86	17471A0488	NAIDU RACHANA	E	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.32
87	17471A0489	TELAPROLU PAVAN KALYAN	A	3	PASS	B	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.45
88	17471A0490	BODEMPUDI SRI HARSHA	A	3	PASS	B	3	PASS	E	3	PASS	E	3	PASS	E	10	PASS	8.59
89	17471A0491	SHAIK RUKSANA	A	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	E	10	PASS	8.73
90	17471A0492	KATTAMURI SATYANARAYANA	A	3	PASS	A	3	PASS	A	3	PASS	E	3	PASS	O	10	PASS	9.05
91	17471A0493	KOPPULA GANESH REDDY	C	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.05
92	17471A0494	SYED MAHABOOB JANI BASHA	B	3	PASS	E	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.59
93	17471A0495	PERUMALLA PREETHI KOUMIKA	E	3	PASS	A	3	PASS	E	3	PASS	O	3	PASS	O	10	PASS	9.45
94	17471A0496	SHAIK JANIBASHA	F	0	FAIL	F	0	FAIL	F	0	FAIL	E	3	PASS	E	10	PASS	NA
95	17471A0497	SHAIK MOHAMMED ALTHAF	C	3	PASS	C	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.05
96	17471A0498	JANGALA KIRAN BABU	B	3	PASS	C	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.05
97	17471A0499	RAMA CHANDRULA KAVYASRI	A	3	PASS	A	3	PASS	E	3	PASS	O	3	PASS	O	10	PASS	9.32
98	17471A04A0	MUVVA MANOJ KUMAR	B	3	PASS	B	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	8.91
99	17471A04A1	KAKUMANU SUMANTH	C	3	PASS	B	3	PASS	E	3	PASS	E	3	PASS	E	10	PASS	8.32
100	17471A04A2	YANDAPALLI N V S L MALLIKA BRAMARAMBIKA	A	3	PASS	A	3	PASS	O	3	PASS	O	3	PASS	O	10	PASS	9.45
101	17471A04A3	TUMMALACHERUVU SAITEJA	A	3	PASS	E	3	PASS	E	3	PASS	O	3	PASS	O	10	PASS	9.45
102	17471A04A4	JAKKIREDDY KEERTHI	B	3	PASS	B	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.32
103	17471A04A5	SHAIK AFRID	B	3	PASS	B	3	PASS	E	3	PASS	O	3	PASS	O	10	PASS	9.05
104	17471A04A6	YERRAMSETTY SAI PAVAN	B	3	PASS	B	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.32
105	17471A04A7	DESABOYINA HEMARAMACHANDRA VASU	B	3	PASS	C	3	PASS	E	3	PASS	E	3	PASS	E	10	PASS	8.32
106	17471A04A8	SHAIK TANGEDA CHINA BAJI	B	3	PASS	A	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.45
107	17471A04A9	KOLLA SIVA HEMANTH	A	3	PASS	A	3	PASS	E	3	PASS	E	3	PASS	E	10	PASS	8.73
108	17471A04B0	AKULA ASHOK KUMAR	A	3	PASS	B	3	PASS	E	3	PASS	O	3	PASS	O	10	PASS	9.18
109	17471A04B1	MOHAMMED ZAKIR HUSSAIN KHAN	E	3	PASS	E	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.32
110	17471A04B2	BALUPUNURI KASU VASU DEVA VENKATA REDDY	C	3	PASS	F	0	FAIL	P	3	PASS	E	3	PASS	E	10	PASS	NA
111	17471A04B3	GORANTLA SRAVAN KRISHNA	C	3	PASS	B	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
112	17471A04B4	JAMMULA CHANDRIKA	B	3	PASS	B	3	PASS	C	3	PASS	O	3	PASS	O	10	PASS	8.64
113	17471A04B5	BONDE RAJENDRA	C	3	PASS	B	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	8.64
114	17471A04B6	NANNEM VEENA VATSALYA	A	3	PASS	E	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.18
115	17471A04B7	GOGULA NAVEEN KUMAR	B	3	PASS	B	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.05
116	17471A04B8	KURANGI MUKUNDA SAI	C	3	PASS	A	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.18
117	17471A04B9	GOUSE MOMITH BAIG	A	3	PASS	B	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.05
118	17471A04C0	BUSSI JOSEPH BALA YASWANTH BABU	B	3	PASS	B	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	8.91
119	17471A04C1	VATTIKONDA SIVA RAMAKRISHNA	C	3	PASS	C	3	PASS	P	3	PASS	E	3	PASS	E	10	PASS	7.64
120	17471A04C2	GUNDA PRATHYUSHA	B	3	PASS	E	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.18
121	17471A04C3	ALLA SARATH SAI	A	3	PASS	E	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.18
122	17471A04C4	GARIKAPATI PAVAN KALYAN	A	3	PASS	A	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.45

123	17471A04C5	GUTHA VENKAT RAO	F	0	FAIL	F	0	FAIL	F	0	FAIL	E	3	PASS	E	10	PASS	NA
124	17471A04C6	PULUKURI SRI PRASANNA	B	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.18
125	17471A04C7	BANDI CHINNAPA REDDY	A	3	PASS	B	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	8.91
126	17471A04C8	PINNIKA SRIVANI	E	3	PASS	E	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.45
127	17471A04C9	MADDI KOTI KIRAN KUMAR	A	3	PASS	C	3	PASS	B	3	PASS	E	3	PASS	O	10	PASS	8.64
128	17471A04D0	DODDAKULA PRASANTH	A	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	A	10	PASS	7.86
129	17471A04D1	SHAIK ARSHAD	E	3	PASS	A	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.18
130	17471A04D2	PALADUGU SRINIVASULU	A	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.05
131	17471A04D3	KONETI SUNEEL	A	3	PASS	B	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.18
132	17471A04D4	VUTUKURI TULASIRAM	E	3	PASS	B	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.05
133	17471A04D5	KOMMI VENKATA SUBRAMANYAM	B	3	PASS	C	3	PASS	P	3	PASS	E	3	PASS	E	10	PASS	7.77
134	17471A04D6	SHAIK SHAMEEM	E	3	PASS	B	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.05
135	17471A04D7	BANDARU MAHESH	A	3	PASS	B	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.18
136	17471A04D8	GANGASANI ASHOK REDDY	E	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	A	10	PASS	8
137	17471A04D9	KOSANA PRATAP	B	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
138	17471A04E0	GADAMSETTY RAGA VENKATA DEEPTHI	A	3	PASS	C	3	PASS	P	3	PASS	E	3	PASS	E	10	PASS	7.91
139	17471A04E1	POTHURI MANIKANTA	A	3	PASS	C	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.18
140	17471A04E2	SHAIK NANNU SHAIDA	A	3	PASS	A	3	PASS	B	3	PASS	E	3	PASS	O	10	PASS	8.91
141	17471A04E3	POLU DIVYA	A	3	PASS	C	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.32
142	17471A04E4	VIBHARAMPATTAPU MAMATHA	B	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.05
143	17471A04E5	MANDAVA SRI BHARATHI	A	3	PASS	A	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.45
144	17471A04E6	BHOJANAPALLI TEJASWINI	E	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.32
145	17471A04E7	SHAIK JAVEED	E	3	PASS	B	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.32
146	17471A04E8	KOLLIKONDA GANGABHAVANI	A	3	PASS	E	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.32
147	17471A04E9	GUNTA ROHITHA REDDY	A	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.32
148	17471A04F0	LAKSHMISETTY VENKATA SAI VYSHNAVI	A	3	PASS	C	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	8.91
149	17471A04F1	RAVULAPALLI SRINU	B	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
150	17471A04F2	SADINENI SOWJANYA	A	3	PASS	B	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	8.91
151	17471A04F3	GANJI KRANTHI	B	3	PASS	B	3	PASS	P	3	PASS	O	3	PASS	O	10	PASS	8.5
152	17471A04F4	CHEVALA PRITHVI RAJ	E	3	PASS	A	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.18
153	17471A04F5	DEVARAPALLI NAGA POOJA SAI SRI	E	3	PASS	E	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.32
154	17471A04F6	MEKAPOTHULA GOPI KRISHNA	A	3	PASS	A	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.05
155	17471A04F7	REPALLE PRATHYUSHA	E	3	PASS	E	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.86
156	17471A04F8	KOMMANABOYINA NAGA ANIL	F	0	FAIL	B	3	PASS	C	3	PASS	O	3	PASS	E	10	PASS	NA
157	17471A04F9	BATCHU DURGA BHAVANI	E	3	PASS	A	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.59
158	17471A04G0	DIRISALA SRAVANI	E	3	PASS	A	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.59
159	17471A04G1	KOMIREDDY MANJU BHARGAVI	E	3	PASS	A	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.18
160	17471A04G2	POLA VENKATA MALLIKHARJUNA RAO	E	3	PASS	A	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.18
161	17471A04G3	KOLIPAKULA DEVI CHAMUNDESWARI	B	3	PASS	A	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.32
162	17471A04G4	BOBBA PRASANTH	B	3	PASS	B	3	PASS	C	3	PASS	A	3	PASS	A	10	PASS	7.45
163	17471A04G5	GANGISETTY JAGADEESH CHANDRA BOSE	E	3	PASS	E	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.32
164	17471A04G6	GUNTU NAVEEN CHOWDARY	B	3	PASS	B	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.05
165	17471A04G8	YELLANURU JAHNAVI	F	0	FAIL	C	3	PASS	P	3	PASS	E	3	PASS	E	10	PASS	NA
166	17471A04G9	MAMIDIPAKA NAGASUSHMA	E	3	PASS	E	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.32
167	17471A04H0	DUGGARAJU GOWTHAMY	A	3	PASS	A	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.45
168	17471A04H1	PASUPULETI RAMA KRISHNA	B	3	PASS	A	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.32
169	17471A04H2	NALAGANGULA KOTIREDDY	B	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
170	17471A04H3	GUNTUPALLI THIRUMALA PRASANNA SANKAR	A	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.05
171	17471A04H4	RAJARAPU SRILAKSHMI TIRUMALESWARI	B	3	PASS	B	3	PASS	E	3	PASS	O	3	PASS	O	10	PASS	9.05
172	17471A04H5	PAMURU DIVYA	C	3	PASS	P	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.64
173	17471A04H6	SHAIK SAMEER	A	3	PASS	E	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.32
174	17471A04H7	KUNCHALA GOPI KRISHNA	B	3	PASS	A	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.32
175	17471A04H8	PUSALA MADHU KUMAR	A	3	PASS	A	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.05
176	17471A04H9	GAJJALA MARUTHI VENKATA KRISHNA REDDY	C	3	PASS	C	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	7.91
177	17471A04I0	MEDA RAVI TEJA	B	3	PASS	C	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.05
178	17471A04I1	CHERUKURI RAVI KUMAR	B	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
179	17471A04I2	KOLA PAVAN KALYAN	A	3	PASS	C	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.18
180	17471A04I3	TEMPALLI PRABHU KUMAR	B	3	PASS	B	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	8.77
181	17471A04I4	SHAIK MASTANVALI	A	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.18
182	17471A04I5	TANNIRU AVINASH BABU	A	3	PASS	A	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.45
183	17471A04I6	TELLAGORLA MANIKANTA GOPALA KRISHNA	A	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.32
184	17471A04I7	KOLA RAKESH	F	0	FAIL	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	NA
185	17471A04I8	KESANUPALLI PRIYANKA	A	3	PASS	E	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.32
186	17471A04I9	NARISETTI UMAMAHESWARI	A	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.18
187	17471A04J0	MANYAM UDAY BHASKAR	A	3	PASS	A	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.05
188	17471A04J1	YERRAVEDA SUPRAJA	A	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.18

189	17471A04J2	GOSULA THIRUPATHI RAO	B	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.18
190	17471A04J3	NARE TEJASWI	A	3	PASS	E	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.32
191	17471A04J4	VAKA GOPI CHAND	A	3	PASS	B	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.45
192	17471A04J5	DOPPALAPUDI NELSON RAJU	A	3	PASS	B	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.18
193	17471A04J6	SHAIK AFRIN	A	3	PASS	B	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.05
194	17471A04J7	MOLAMANTI SAIKALYAN	B	3	PASS	A	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.18
195	17471A04J8	SHAIK KANDIPATI MOULALI	E	3	PASS	B	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.18
196	17471A04K0	PALLEPOGU SHARONU	A	3	PASS	E	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.32
197	17471A04K1	CHILAKA VAMSI KRISHNA	E	3	PASS	B	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.05
198	17471A04K2	GOCHIPATHALA RAJ KAMAL	C	3	PASS	C	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	7.91
199	17471A04K3	KESENAPALLI MARIYA BABU	E	3	PASS	B	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.05
200	17471A04K4	ANGALAKURTHI SUMA PRIYA	B	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
201	17471A04K5	PERUMALLA VINAY KUMAR	A	3	PASS	C	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.18
202	17471A04K6	SHAIK SAJID HASAN	F	0	FAIL	F	0	FAIL	F	0	FAIL	E	3	PASS	E	10	PASS	NA
203	17471A04K7	USAA PAVAN KALYAN	B	3	PASS	B	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	8.77
204	17471A04K8	GORANTLA ASHOK	A	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.32
205	17471A04K9	NALLAMOLU KUSUMA	A	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.18
206	17471A04L0	JUPUDI RAJU	B	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.05
207	17471A04L1	MOGAL IRFAN	A	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.18
208	17471A04L2	VELISALA MANISH PREETHAM	B	3	PASS	C	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.05
209	17471A04L4	VEMPATI PREETHI MANISHA	B	3	PASS	B	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	8.77
210	17471A04L5	JAMPANI KRISHNAVAMSI	B	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.18
211	17471A04L6	MEDATATI DIVYA	A	3	PASS	B	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.05
212	17471A04L8	SADHU KOTESWA RAO	B	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.18
213	17471A04M0	BAPATLA VIJAYA LAKSHMI	E	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.32
214	17471A04M1	PATIBANDLA NARESH	B	3	PASS	A	3	PASS	B	3	PASS	E	3	PASS	A	10	PASS	7.86
215	17471A04M2	KANDULA GURU KIRAN	B	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.18
216	17471A04M3	DARSI ANIL KUMAR	F	0	FAIL	F	0	FAIL	F	0	FAIL	A	3	PASS	A	10	PASS	NA
217	17471A04M4	BATRAJU NAGA UMAMAHESH	A	3	PASS	A	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.45
218	17471A04M5	PALLEMPATI DURGA PRASAD	A	3	PASS	E	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.18
219	17471A04M6	PUTTA SRIKANTH	A	3	PASS	A	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.45
220	17471A04M7	MEKALA YASHWANTH KUMAR	B	3	PASS	A	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	8.91
221	17471A04M8	SHAIK IMRAN	C	3	PASS	B	3	PASS	P	3	PASS	E	3	PASS	E	10	PASS	7.77
222	17471A04N0	SHAIK MOHAMMADSHARIF	B	3	PASS	B	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.05
223	17471A04N1	MADHIREDDY ANIL KUMAR REDDY	A	3	PASS	E	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.18
224	17471A04N3	VEMULURI YASASWI	B	3	PASS	A	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	8.91
225	18475A0401	MARELLA VAMSI	C	3	PASS	A	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.05
226	18475A0402	RACHAKONDA SHYAM PREMKUMAR	B	3	PASS	A	3	PASS	A	3	PASS	E	3	PASS	E	10	PASS	8.45
227	18475A0403	SURISETTI ARCHANA	C	3	PASS	A	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.05
228	18475A0404	PALADUGU GANESH	E	3	PASS	E	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.45
229	18475A0405	VUYYURU SRILAKSHMI	C	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.77
230	18475A0406	NUNNA VENKATA SIVASAI	A	3	PASS	B	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.05
231	18475A0407	PARITALA HARITHA	B	3	PASS	A	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.05
232	18475A0408	MADDINA VENKATA SANDEEP	A	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.05
233	18475A0409	CHERUKURI BRAHMA VENKATESWARLU	C	3	PASS	F	0	FAIL	C	3	PASS	E	3	PASS	A	10	PASS	NA
234	18475A0410	THOTA BHARATH	B	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
235	18475A0411	SIDDEALA DILEEP SAGAR	A	3	PASS	B	3	PASS	A	3	PASS	O	3	PASS	O	10	PASS	9.05
236	18475A0412	DOPPALAPUDI SARATH CHANDRA	C	3	PASS	C	3	PASS	P	3	PASS	E	3	PASS	E	10	PASS	7.64
237	18475A0413	JANGA MAHENDRA	F	0	FAIL	P	3	PASS	P	3	PASS	E	3	PASS	E	10	PASS	NA
238	18475A0414	AINAOLU GOPIKRISHNA	A	3	PASS	A	3	PASS	B	3	PASS	O	3	PASS	O	10	PASS	9.05
239	18475A0415	THUNGALA PRAMOD	B	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
240	18475A0416	SURUBULA LEELA PAVANKUMAR	B	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
241	18475A0417	SARIKONDA RAMA KRISHNAM RAJU	C	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.77
242	18475A0418	ATTULURI SYAM PRASAD	B	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
243	18475A0419	KUNDA JASHUVA	A	3	PASS	B	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	8.18
244	18475A0420	YALAVARTHI MADHU BABU	B	3	PASS	C	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.05
245	18475A0421	M SUBRAHMANYAM	A	3	PASS	B	3	PASS	B	3	PASS	E	3	PASS	E	10	PASS	8.32
246	18475A0422	GUDISE VENKATESH	B	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91
247	18475A0423	VARIKUTI KRISHNANJANEYULU	B	3	PASS	C	3	PASS	C	3	PASS	E	3	PASS	E	10	PASS	7.91