Advanced Microcontroller (Code : 3151107)

Harish G. Narula, Khushboo Shah

Semester V – Electronics / Electronics and Communication Engineering
(Gujarat Technological University)

Copyright © by Authors. All rights reserved. No part of this publication may be reproduced, copied, or stored in a retrieval system, distributed or transmitted in any form or by any means, including photocopy, recording, or other electronic or mechanical methods, without the prior written permission of the publisher.

This book is sold subject to the condition that it shall not, by the way of trade or otherwise, be lent, resold, hired out, or otherwise circulated without the publisher's prior written consent in any form of binding or cover other than which it is published and without a similar condition including this condition being imposed on the subsequent purchaser and without limiting the rights under copyright reserved above.

First Printed in India : January 2015 (For Pune University)

First Edition : September 2020

This edition is for sale in India, Bangladesh, Bhutan, Maldives, Nepal, Pakistan, Sri Lanka and designated countries in South-East Asia. Sale and purchase of this book outside of these countries is unauthorized by the publisher.

Printed at : 37/2, Ashtavinayak Industrial Estate, Near Pari Company,
Narhe, Pune, Maharashtra State, India.
Pune – 411041

ISBN : 978-93-90428-10-6

Published by

TechKnowledge Publications

Head Office : B/5, First floor, Maniratna Complex, Taware Colony, Aranyeshwar Corner,
Pune - 411 009. Maharashtra State, India
Ph : 91-20-24221234, 91-20-24225678.
Email : info@techknowledgebooks.com,
Website : www.techknowledgebooks.com

[3151107] (FID : GO101) (Book Code : GO101A)

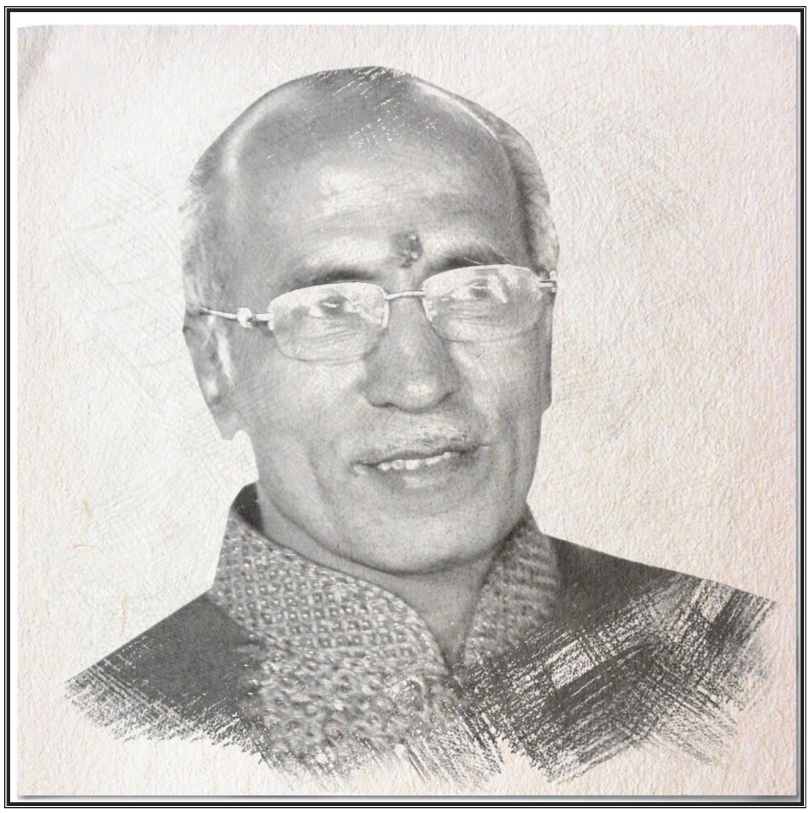
(Book Code : GO101A)

We dedicate this Publication soulfully and wholeheartedly,

in loving memory of our beloved founder director,

Late Shri. Pradeepji Lalchandji Lunawat,

who will always be an inspiration, a positive force and strong support behind us.



"My work is my prayer to God"

- Lt. Shri. Pradeepji L. Lunawat

Soulful Tribute and Gratitude for all Your Sacrifices, Hardwork and 40 years of Strong Vision…

Preface

My Dear Students,

We are extremely happy to come out with this book on **"Advanced Microcontroller"** for you. The topics within the chapters have been arranged in a proper sequence to ensure smooth flow of the subject.

We present this book in the loving memory of **Late Shri. Pradeepji Lunawat,** our source of inspiration and a strong foundation of **"TechKnowledge Publications"**. He will always be remembered in our heart and motivate us to achieve our milestone.

We are thankful to Shri. J. S. Katre, Shri. Shital Bhandari, Shri. Arunoday Kumar and Shri. Chandroday Kumar for the encouragement and support that they have extended. We are also thankful to Seema Lunawat for technology enhanced reading, E-books support and the staff members of TechKnowledge Publications for their efforts to make this book as good as it is. We have jointly made every possible efforts to eliminate all the errors in this book. However if you find any, please let us know, because that will help us to improve further.

We are also thankful to our family members and friends for their patience and encouragement. We would also like to thank Guruji Shree Swami Satyanandji Maharaj for his blessings

- Authors

(Book Code : GO101A)

Syllabus

UNIT - I : ARM Introduction **(06 Hours)**

Types of computer Architectures, ISA's and ARM, Difference between RISC and CISC, RISC Design philosophy, ARM Design Philosophy, History of ARM microprocessor, ARM processor family, Embedded System Software and Hardware.

(Refer Chapter 1)

UNIT - II : ARM Architecture and Pipeline Structure **(08 Hours)**

The Acorn RISC Machine, ARM Core data flow model, Architectural inheritance, The ARM7TDMI programmer's model: General purpose registers, CPSR, SPSR, ARM memory map, data format, load and store architecture, ARM 3 stage Pipeline, ARM 5 stage Pipeline, Pipeline Hazards, Data forwarding - a hardware solution, Stack implementation in ARM, Endianness, condition codes. Processor core Vs CPU core, ARM7TDMI Interface signals. **(Refer Chapter 2)**

UNIT - III : ARM7TDMI Assembly Language Instructions and Programming **(10 Hours)**

Different Types of Instructions, ARM Instruction set, data processing instructions. Shift Operations, shift Operations using RS lower byte, Immediate value encoding. Data processing Instructions. Addressing modes with examples. Instruction Timing, Swap Instructions, Swap Register related Instructions, Loading Constants. Program Control Flow, Control Flow Instructions, B & BL instructions, BX instruction. Interrupts and Exceptions, Exception Handlers, Reset Handling. Aborts, software Interrupt Instruction, undefined instruction exception. Interrupt Latency, Multiply Instructions,. Thumb state, Thumb Programmers model, Thumb Implementation, Thumb Applications. Thumb Instructions, Interrupt processing. Interrupt Handling schemes, Examples of Interrupt Handlers, Co-processor instructions. **(Refer Chapter 3)**

UNIT - IV : Embedded C Programming for ARM **(06 Hours)**

ARM Development Environment Embedded Software, Overview of C compiler and optimization, Basic C data types, C Looping structures, Register allocations, function calls, pointer aliasing, structure arrangement, bit-fields, unaligned data and Endianness, Division, floating point, Inline functions and inline assembly, Portability issues. C programs for General purpose I/O, general purpose timer, PWM Modulator, UART, I2C Interface, SPI Interface, ADC, DAC.

(Refer Chapter 4)

(Book Code : GO101A)

UNIT - V : Cache and Memory Management and Protection **(06 Hours)**

Memory Technologies, Need for memory Hierarchy, Hierarchical Memory Organization, Virtual Memory. Cache Memory, Mapping Functions. Cache Design, Unified or split cache, multiple level of caches, ARM cache features, coprocessor 15 for system control. Processes, Memory Map, Protected Systems, ARM systems with MPU, memory Protection Unit (MPU). Physical Vs Virtual Memory, Paging, Segmentation. MMU Advantage, virtual memory translation, Multitasking with MMU, MMU organization, Tightly coupled Memory (TCM). **(Refer Chapter 5)**

UNIT - VI : ARM Peripherals and Versions **(06 Hours)**

AMBA Overview, Typical AMBA Based Microcontroller, AHB bus features, AHB Bus transfers, APB bus transfers, APB bridge. Overview of ARM Versions: ARM v5, ARM v6, ARM v7 (Cortex family R: Real time A: Application and M: Microcontroller), ARM big. LITTLE® Technology, ARM v8. **(Refer Chapter 6)**

❑❑❑

Unit - I

Chapter 1 : ARM Introduction **1-1 to 1-16**

1.1 Types of Computer Architectures 1-1
1.1.1 Von Neumann Memory Organization 1-1
1.1.2 Harvard Memory Organization 1-2
1.2 ISA's and ARM 1-2
1.3 Difference between RISC and CISC 1-3
1.4 RISC Design Philosophy 1-4
1.4.1 Register Window 1-4
1.4.2 Miscellaneous Features of RISC Systems 1-5
1.5 ARM Design Philosophy 1-8
1.6 History of ARM Microprocessor 1-8
1.6.1 Terms Related to ARM Instruction Set 1-11
1.7 ARM Processor Family 1-12
1.8 Development of ARM Architecture 1-13
1.9 Embedded System Hardware and Software 1-15
1.9.1 Embedded System Hardware 1-15
1.9.2 Embedded System Software 1-16

Unit - II

Chapter 2 : ARM Architecture and Pipeline Structure **2-1 to 2-45**

2.1 The Acorn RISC Machine 2-1
2.2 ARM Core Data Flow Model 2-1
2.2.1 5 Stage ARM9 Core Architecture 2-1
2.2.2 Understanding the ARM Instruction Execution 2-2
2.2.3 Execution of Data Processing Instruction in ARM 2-3
2.2.4 Execution of the Branch Instructions in ARM Processor 2-4
2.3 Architectural Inheritance 2-5
2.4 The ARM7TDMI Programmer's Model: General Purpose Registers, CPSR, SPSR 2-7
2.4.1 General Purpose Registers 2-10
2.4.2 Program Status Registers (CPSR and SPSR) 2-12
2.4.3 Barrel Shifter 2-15
2.5 ARM Memory Map# 2-16
2.6 Data Format 2-17
2.7 Load and Store Architecture 2-18
2.7.1 Addressing Modes for Data Processing Operands (i. e. op1) 2-18
2.7.2 Addressing Modes for Memory Access Operands 2-19
2.8 ARM 3 and 5 Stage Pipeline 2-21
2.9 Pipeline Hazards 2-24
2.9.1 Methods to Resolve the Data Hazards and Advances in Pipelining 2-26

2.9.1(A) Pipeline Stalls 2-26
2.9.1(B) Data Forwarding: An Hardware Solution 2-26
2.9.1(C) Dynamic Instruction Scheduling (or) Out-Of-Order (OOO) Execution 2-27
2.9.2 Handling of Branch Instructions to Resolve Control Hazards 2-27
2.9.2(A) Pre-Fetch Target Instruction 2-27
2.9.2(B) Branch Target Buffer (BTB) 2-28
2.9.2(C) Loop Buffer 2-28
2.9.2(D) Branch Prediction 2-28
2.9.2(E) Pipeline Stall (Delayed Branch) 2-28
2.9.2(F) Loop Unrolling Technique 2-28
2.9.2(G) Software Scheduling or Software Pipelining 2-29
2.10 Stack Implementation in ARM 2-29
2.11 Endianess 2-29
2.12 Condition Codes 2-29
2.13 Processor Core Vs. CPU Core 2-30
2.14 ARM7TDMI 2-30
2.14.1 Features of ARM7TDMI 2-31
2.14.2 Block Diagram and ARM7TDMI Interface Signals 2-33

Unit - III

Chapter 3 : ARM7TDMI Assembly Language Instructions and Programming **3-1 to 3-44**

3.1 Different Types of Instructions 3-1
3.2 ARM Instruction Set 3-1
3.3 Data Processing Instructions 3-2
3.4 Arithmetic and Logical Instructions 3-3
3.4.1 Comparison and Test Instructions 3-5
3.4.2 Logical Instructions 3-6
3.4.3 Shift Operation using RS Lower Byte 3-9
3.5 Rotate and Barrel Shifter 3-9
3.6 Program Control Flow/ Branch Instructions 3-11
3.7 Load and Store Instructions 3-13
3.7.1 Load and Store Word or Unsigned Byte Instruction 3-13
3.7.2 Load and Store Half Word and Load Signed Byte 3-14
3.7.3 Counting Leading Zeros Instruction 3-15
3.7.4 Semaphore / Swap Register Instructions 3-18
3.8 Software Interrupt Instructions 3-19
3.8.1 Co-processor Instructions 3-19
3.9 Program Status Register Instructions 3-22
3.10 Conditional Execution 3-24
3.11 Multiple Register Load and Store Instructions 3-25
3.12 Stack Instructions 3-26
3.13 Thumb Instruction Set 3-27
3.13.1 Introduction to Thumb Instruction Set 3-27

3.13.2 Switching between ARM State and Thumb State 3-27
3.13.3 Thumb Programmer's Model 3-28
3.13.3(A) Branching Instructions of Thumb 3-29
3.13.3(B) Thumb Software Interrupt Instruction 3-30
3.13.3(C) Thumb Data Processing Instructions 3-30
3.13.4 Advantage of Thumb Instructions 3-31
3.13.5 Instruction Timing 3-31
3.14 Assembler Rules and Directives 3-32
3.15 Assembly Language Programs 3-33
3.16 Interrupt Latency 3-38
3.16.1 Interrupt Service Routine 3-38
3.16.2 Interrupt Priorities 3-39
3.16.3 Interrupt Nesting 3-39
3.16.4 Interrupt Latency 3-40
3.17 Interrupts/Exception Handling in ARM 3-40
3.17.1 Exception Process 3-41
3.17.1(A) Reset Handling 3-41
3.17.1(B) Undefined Instructions 3-42
3.17.1(C) Software Interrupt 3-42
3.17.1(D) Prefetch Abort Vector 3-42
3.17.1(E) Data Abort 3-42
3.17.1(F) Interrupt Request (IRQ) 3-42
3.17.1(G) Fast Interrupt Request (FIQ) 3-42
3.17.1(H) High Vectors 3-42
3.18 Exception Priorities 3-43
3.18.1 Interrupt Handlers and Interrupt Handling Schemes 3-43
3.18.2 Example of Interrupt Handler : NVIC (Nested Vector Interrupt Controller) in LPC17xx 3-44

Unit - IV

Chapter 4 : Embedded C Programming for ARM **4-1 to 4-72**

4.1 ARM Development Environment Embedded Software 4-1
4.2 Overview of C Compiler and Optimization 4-1
4.2.1 Disadvantages of Assembly Programming 4-2
4.2.2 C Programming 4-3
4.2.3 Using ARM GCC Tool Chain 4-3
4.3 Speed of Code or Optimization of Codes 4-4
4.4 Source Code Engineering Tools for Embedded C / C++ 4-5
4.5 Integrated Development Environment (IDE) 4-5
4.6 Optimization of Memory Needs 4-6
4.7 Basic C Data Types 4-7
4.7.1 Character Set of C 4-8
4.7.2 Keywords 4-9
4.7.3 Identifiers 4-9

4.7.4 Scalar Data Types in C 4-10
4.7.5 Constants and Variables 4-11
4.7.6 Escape Sequences 4-11
4.7.7 Unary Operators 4-12
4.7.8 Binary Operators 4-15
4.7.9 Ternary Operator 4-17
4.7.10 Assignment Operators and Statements 4-18
4.7.11 Selection Operators 4-19
4.8 C Looping Structures 4-19
4.8.1 if-else Selective Statement 4-19
4.8.2 Looping Statements 4-21
4.9 C- Control Structures for Iteration 4-21
4.9.1 for Loop 4-21
4.10 Register Allocations 4-26
4.10.1 Storage Class 4-27
4.11 Function Calls 4-29
4.12 Pointers 4-32
4.12.1 Pointer Aliasing 4-34
4.13 Structure Arrangement 4-34
4.14 Bitfields 4-35
4.15 Unaligned Data and Endianness 4-36
4.15.1 Unaligned Data 4-36
4.15.2 Endianness 4-36
4.15.3 Division 4-36
4.16 Floating Point 4-37
4.17 Inline Functions and Inline Assembly 4-37
4.17.1 Inline Assembly 4-37
4.18 Portability Issues 4-39
4.19 LPC2148# 4-39
4.20 Features of LPC 2148# 4-40
4.21 Architecture of LPC2148#(Block Diagram and its Description) 4-41
4.22 System Control Block (PLL and VPB Divider) 4-49
4.22.1 PLL 4-49
4.22.2 VPB Divider 4-49
4.23 Memory Map 4-49
4.24 Fast General Purpose Parallel I/O (GPIO) 4-50
4.24.1 Features of GPIO 4-50
4.25 Pin Connect Block 4-51
4.26 Timers 4-51
4.26.1 Features of Timers 4-51
4.27 Interfacing GPIO using Timer 4-52
4.27.1 Interfacing Light Emitting Diode (LED) and Seven Segment Display (SSD) 4-52
4.28 Interfacing DC Motor Using PWM 4-57
4.29 Interfacing GSM Module Using UART 4-59

4.29.1 GSM Modem and its Interfacing with Microcontroller 4-59
4.29.2 Interfacing GSM with LPC2148 4-59
4.30 Interfacing EEPROM using I2C Communication Protocol 4-62
4.31 Interfacing Memory Using SPI Protocol 4-63
4.31.1 Interfacing SD-MMC with LPC2148 using SPI 4-64
4.32 Interfacing Internal ADC 4-64
4.32.1 Interfacing DAC 4-66
4.32.2 Interfacing Sensors to LPC 2148 4-70

Unit - V

Chapter 5 : Cache and Memory Management and Protection **5-1 to 5-32**

5.1 Memory Technologies 5-1
5.2 Memory Hierarchy : Need and Organization 5-3
5.3 Cache Memory : Concept and Cache Consistency 5-4
5.3.1 Cache Operation 5-4
5.3.2 Principles of Locality of Reference 5-5
5.3.3 Cache Performance 5-6
5.3.4 Cache Design 5-7
5.3.4(A) Look-Through Cache Designs 5-7
5.3.4(B) Look-Aside Cache Designs 5-8
5.3.5 Cache Consistency (Also Known as Cache Coherency) 5-8
5.3.6 Write Policy 5-9
5.3.7 Replacement Algorithms 5-11
5.3.8 Multiple Levels of Cache: Cost and Performance Measurement 5-15
5.3.9 Cache Coherency with L1 and L2 Cache 5-15
5.4 Cache Mapping Functions 5-16
5.4.1 Direct Mapping Technique 5-16
5.4.2 Fully Associative Mapping 5-18
5.4.3 Set Associative Mapping 5-19
5.5 Unified and Split Cache 5-20
5.6 ARM Memory Hierarchy 5-21
5.7 ARM Cache Features 5-21
5.7.1 Coprocessor 15 for Cache System Control and Various Processes 5-21
5.8 ARM Memory Map 5-22
5.9 Protected Systems 5-22
5.9.1 Memory Protection Unit (MPU) and ARM Processor with MPU 5-22
5.10 Memory Management Unit (MMU) Advantage 5-24
5.11 Virtual Memory 5-24
5.12 Virtual Memory Translation 5-25
5.12.1 Paging Mechanism or Memory Management Unit 5-27
5.12.2 Segmentation 5-28
5.12.3 Physical Memory vs. Virtual Memory 5-28
5.13 Multitasking with MMU 5-29

5.14 MMU Organization in Virtual Memory System 5-29
5.14.1 Page Tables 5-30
5.14.2 Translation Look Aside Buffer 5-31
5.14.3 ARM MMU 5-31
5.14.4 Fast Context Switch Extension 5-31
5.15 Tightly Coupled Memory (TCM) 5-32

Unit - VI

Chapter 6 : ARM Peripherals and Versions **6-1 to 6-19**

6.1 Advanced Microprocessor Bus Architecture (AMBA) Bus System Overview 6-1
6.1.1 Advanced High-Performance Bus (AHB) 6-2
6.1.2 Advanced System Bus (ASB) 6-2
6.1.3 Comparison of AHB and APB Buses of AMBA Architecture 6-2
6.1.4 Advanced Peripheral Bus (APB) 6-3
6.1.5 A Typical AMBA-based Microcontroller 6-3
6.1.6 AHB Bus Transfer and APB Bridge 6-4
6.1.7 APB Bus Transfer 6-4
6.2 Overview of ARM Versions: ARM v5, ARM v6, ARM v7, ARM v8 6-5
6.3 ARM v7 : Cortex-A, Cortex-R and Cortex-M 6-10
6.3.1 Cortex-A Series 6-10
6.3.2 Cortex-R Series 6-11
6.3.3 Cortex-M Series 6-11
6.4 Cortex-A Processor Series Survey : Features, Versions and Applications 6-12
6.4.1 Cortex-A57 6-13
6.4.2 Cortex-A53 6-13
6.4.3 Cortex-A17 6-14
6.4.4 Cortex-A15 6-14
6.4.5 Cortex-A9 6-14
6.4.6 Cortex-A7 6-14
6.4.7 Cortex-A5 6-15
6.5 Cortex-R Processor Series Survey : Features, Versions and Applications 6-15
6.5.1 Comparison of Various Cortex-R Processors 6-15
6.5.2 Cortex-R4 Processor 6-16
6.5.3 Cortex-R5 Processor 6-16
6.5.4 Cortex-R7 Processor 6-17
6.5.5 Applications of ARM Cortex-R Processors 6-17
6.6 Cortex-M Processor Series Survey : Features, Versions and Applications 6-18
6.7 ARM Optimization Techniques 6-18

CHAPTER 1

UNIT I

ARM Introduction

Syllabus

Types of computer Architectures, ISA's and ARM, Difference between RISC and CISC, RISC Design philosophy, ARM Design Philosophy, History of ARM microprocessor, ARM processor family, Embedded System Software and Hardware

1.1 Types of Computer Architectures

GTU - S-18, W-18, S-19, W-19

Q.	Explain difference between : Harvard & Von-Neumann Architecture.	**(S-18, W-18, S-19, W-19, 2 Marks)**
Q.	Which features of RISC design philosophy are rejected in ARM architecture?	**(S-19, 3 Marks)**

- The need of speed in the processing is the main need of advanced microprocessors. Every new processor manufactured is mainly aimed to increase its speed by adding the high frequency, cache memory, faster pipelining etc.
- The applications of the processor requires not only more and more processing but also fast processing. Various digital processing applications like DSLR cameras to the processors required in rocket launcher all require high processing speed as well as volume.

1.1.1 Von Neumann Memory Organization

- Von Neumann memory organization states that "Every memory location has a unique address, while the Harvard's memory organization states that "There can be multiple memory locations with same address but different function".
- One such example of Von Neumann and Harvard's memory organization is shown in the Fig. 1.1.1. Von Neumann's memory organization has all code, data and stack in a single memory, while the Harvard's memory organization has them in separate memories.
- Most of the microprocessors use Von Neumann's memory organization while microcontrollers use Harvard's memory organization.

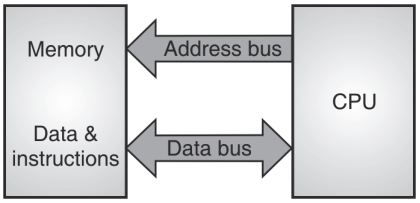


Fig. 1.1.1 : Von Neumann memory organization

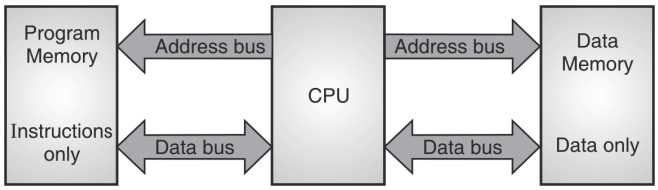


Fig. 1.1.2 : Harvard memory organization

1.1.2 Harvard Memory Organization

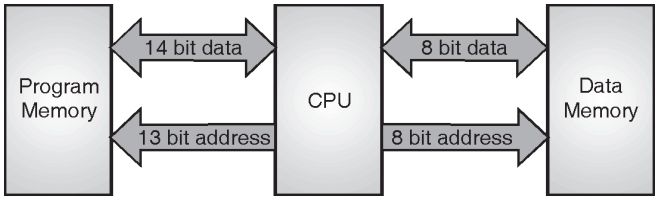


Fig. 1.1.3 : Block diagram for Harvard architecture of PIC microcontrollers

- Like 8051, PIC microcontrollers also have Harvard organization of memory. But here it is a special case wherein the number of data lines for data memory and program memory are different. Also the number of address lines for program and data are different.
- Harvard architecture is a newer concept than Von Neumann's. It rose out of the need to speed up the work of a microcontroller. In Harvard architecture, data bus and address bus are separate. Thus a faster flow of data is possible through the central processing unit, and of course, a greater speed of work. Separating a program from data memory makes it further possible for instructions not to have to be 8-bit words. Fig 1.1.3 shows block diagram for Harvard architecture in PIC controllers.
- PIC uses 14 bits for instructions which allows for all instructions to be one word instructions.
- Instructions are fetched from program memory using buses that are different from the ones that are used to access the data memory.
- 14-bit wide data bus for program memory fetches an entire instruction in a single access (as the size of all instructions in PIC is 14 bits each).

1.2 ISA's and ARM

- An ISA (Instruction Set Architecture) or simply architecture or computer architecture is an abstract model of a computer. The ISA model is implemented in all computers and is thereby called as CPU.
- The ISA model gives the details of the supported data types, instructions, registers, addressing modes, memory model, memory consistency model etc.
- The ISA can be extended by adding more addressing modes or instructions etc. to the standard abstract processor.
- ISAs are classified as RISC (Reduced Instruction Set Computing) and CISC (Complex Instruction Set Computing). The difference between these two ISAs will be studied in the next section.

- There are more types of ISAs introduced in recent times like VLIW (Very Large Instruction Width), EPIC (Explicitly Parallel Instruction Computing) etc.
- ARM is RISC architecture. The ISAs supported by ARM processor are A64, A32 and T32.
- A32 is a set of 32-bit instructions and used in 32-bit ARM architecture.
- A64 is a set of 64-bit instructions introduced to support 64-bit ARM architecture.
- T32 was earlier introduced as a 16-bit instruction set called as Thumb instructions that improved the code density of the user code. Later, the T32 was formed as a mixed set of 16-bit and 32-bit instructions. Hence the compiler can balance performance and code size trade-off in a single instruction set.

1.3 Difference between RISC and CISC

GTU - S-16, S-19

Q. Write a difference between RISC and CISC Processor. **(S-16, S-19, 7 Marks)**

- The microprocessors and the microcontrollers available are mainly of two type namely the RISC (Reduced Instruction Set Computing) and the CISC (Complex Instruction Set Computing). These processors are used according to their applications.
- Most of the CISC processors are used in computers while the RISC processors in Embedded System. Table 1.3.1 shows the difference between these two types of processors i.e. the RISC and CISC processors.

Table 1.3.1 : Difference between RISC and CISC processors

Sr. No.	Properties	RISC	CISC
1.	Number of Instructions	Less	More
2.	Addressing Modes	Less	More
3.	Instruction Formats	Less	More
4.	Instruction Size	Fixed	Variable
5.	Control Unit	Hardwired	Micro-programmed
6.	Number of Bus Cycles to execute an instruction	Single CPU cycle (for 80% Instructions)	Multiple CPU cycles
7.	Control Logic And Decoding Subsystem	Simple	Complex
8.	Pipelining	Huge no. of stages of Pipelining	Comparatively less number of pipelining stages
9.	Design time and Probability of Design Errors	Smaller time and less probable	Long time and Significant probability
10.	Complexity of Compiler	Simpler	More complex and the results of "optimization" may not be most efficient and the fastest machine language code
11.	HLL instructions	Supported	Not Supported

1.4 RISC Design Philosophy

GTU - W-16, S-17, W-18, S-19

Q.	Write a note on RISC design philosophy.	**(W-16, 4 Marks)**
Q.	List out in detail any three designed rules implemented in RISC design philosophy.	**(S-17, 3 Marks)**
Q.	List the RISC properties and explain RISC design philosophy.	**(W-18, 3 Marks)**
Q.	Explain features of RISC system.	**(S-19, 3 Marks)**

A RISC system must satisfy the following properties :

1. Single-cycle execution of all (or at least 80 percent) instructions.
2. Single-word standard length of all instructions.
3. Small number of instructions (<= 128).
4. Small number of instruction formats (<= 4).
5. Small number of addressing modes (<= 4).
6. Memory access possible by load and store instructions only.
7. All operations, except load and store, are register to register i.e. within the CPU.
8. It must have a hardwired control unit.
9. It must also have a relatively large (at least 32) general-purpose CPU register file.

1.4.1 Register Window

1. Since there are a huge number of registers in a RISC processor, it can be useful in saving the latency period during procedure call. Parameter passing in is possible by the **register window**. This policy also allows reasonable HLL support in RISC designs.
2. The register file is subdivided into groups of registers, called register **windows**.
3. A certain group of 'i' registers, suppose R_0 to $R_{(i-1)}$, are designated as **global registers**. The global registers are accessible to all procedures running on the system at all the times.
4. The advantage of register windowing is that each task is assigned a separate window that is not accessible to the other tasks.
5. A current window pointer, normally in the status register points to the current window accessible to the task.
6. If the currently running procedure is assigned the register window J, hence taking up registers K, K+1, . . . , K+W-1 (where W is the number of registers per window), the CWP contains the value J, hence pointing to the base of window J. If the next procedure to execute takes up window J+1, the value in the CWP field will be incremented accordingly to point to J+1.

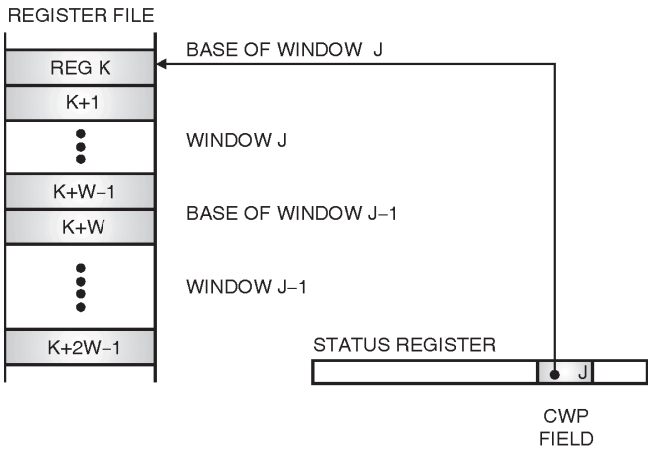


Fig. 1.4.1 : Simple non-overlapping register window

7. Register windowing can work more efficiently for parameter passing between calling and called procedures by partial overlapping of the windows.

   The last N registers of window J will be the first N registers of window J+1.

1.4.2 Miscellaneous Features of RISC Systems

GTU - W-16

Q. Define the following : Delayed Branch **(W-16, 1 Mark)**

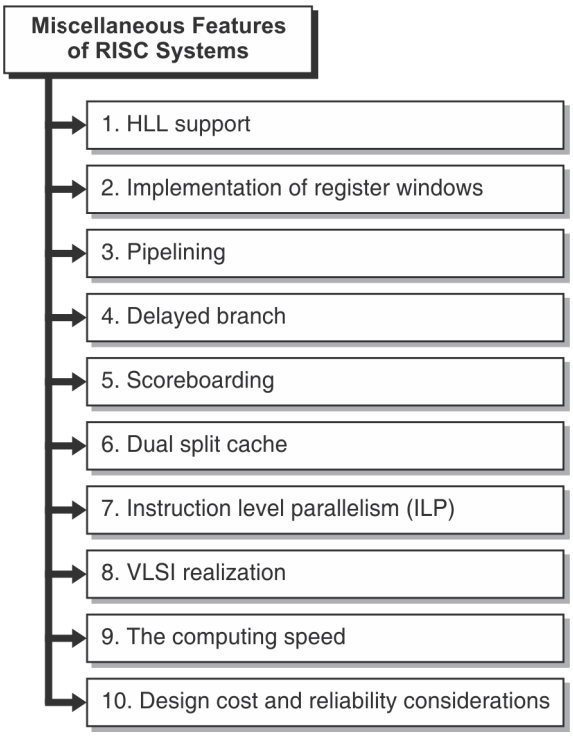


Fig. 1.4.2 : Miscellaneous features of RISC systems

1. HLL support

(a) The support for High Level Language (HLL) features is mandatory in the design of any computing system.

(b) The procedure call-return and parameters passing is the most time-consuming operation in typical HLL programs.

2. Implementation of register windows

(a) A CISC processor's control unit takes up a large percentage of the chip area, leaving very little space for other subsystems and basically not permitting a large register file, needed for an efficient implementation of windowing.

(b) A RISC processor's control unit makes up a must smaller percentage of the chip area, yielding the necessary space for a large register file.

(c) And hence implementation of register windowing (that requires huge number of registers) is possible in RISC processors

3. Pipelining

(a) Pipelining was used on various CISC systems even before the RISC approach became popular.

(b) But, a streamlined RISC can handle pipelines more efficiently.

4. Delayed branch

(a) The problem occurs in a system where instructions are prefetched, right after a branch.

(b) If the branch is conditional, and the condition is not satisfied, then the next instruction, which was prefetched, is executed, and since no branch is to be performed, no time is lost.

(c) But, if the branch condition is satisfied, or the branch is unconditional, the next prefetched instruction is to be flushed and other instruction pointed to by the branch address is to be fetched in its place. The time required to prefetch the flushed instruction is wasted.

(d) Such waste of time is solved by using the **delayed branch approach**.

(e) In this approach, the instructions are reshuffled such that the operation does not change the result.

(f) A successful branch is assumed and the execution of the branch is delayed until the already prefetched instructions are executed. Hence, no time is lost and there is no change in the intended program operation.

(g) But the compiler has to take care that the instructions followed by the branch instructions are to be executed irrespective to the branch to be taken or not.

5. Scoreboarding

(a) Another problem in instruction pipelines is that of data dependency.

(b) The data in some register put by instruction 1 may be required by instruction 2; and before the value in the register is available, the instruction 2 may be ready for execution yielding a possibly incorrect result.

(c) A method used to solve this problem is called as **scoreboarding**.

(d) A special CPU control register i.e. the **scoreboard register,** is required for this purpose.

6. Dual split cache

Another feature, implemented in not only a CISC but also RISC systems, is separated data and code caches, or split cache.

7. Instruction Level Parallelism (ILP)

Superscalar and super pipelined designs are also mostly implemented in a RISC design.

8. VLSI realization

(a) The chip area, dedicated to the realization of the control unit, is considerably less. Therefore, on a RISC VLSI chip, there is more area available for other features (cache, FPU, part of the main memory, memory management unit, I/O ports, etc).

(b) As a result of the considerable reduction of the control area, a large number of CPU registers can fit on-chip.

(c) By reducing the control area on the VLSI chip and filling the area by numerous identical registers, the **regularization factor** (which is defined as, ratio of chip area utilized by other features to the chip space required by control unit) of the chip increases. The higher the regularization factor, the lower is the VLSI design cost.

9. The computing speed

(a) A simpler and smaller control unit in RISC requires fewer gates. This results in shorter propagation paths for control unit signals, decreasing the delay time for control signals and hence yielding a faster operation.

(b) A significantly reduced number of instructions, formats, and addressing modes results in a simpler and smaller decoding system, resulting in faster decoding operation.

(c) A hardwire-controlled system can hence be implemented, with a reduced control unit that will in general be faster than a microprogrammed control unit.

(d) A relatively large CPU register file also reduces CPU-memory traffic to fetch and store data operands.

(e) A large register set can also be used to store parameters to be passed from a calling to a called procedure, to store the information of a process that was interrupted by another.

10. Design cost and reliability considerations

(a) It takes a shorter time to complete the design of a RISC control unit, because of the smaller instruction set, fixed instruction length and less instruction formats. Thus contributing to the reduction in the overall design cost.

(b) A simpler and smaller control unit will obviously have a reduced number of design errors and, therefore, a higher reliability.

11. High Code density

(a) Since the number of instructions for RISC is less, the number of bits required to select the instruction or the operation code is smaller.

(b) Hence the space required for an instruction of RISC processor is low.

(c) Therefore the density of instructions in RISC is higher compared to that of CISC as the CISC instructions are longer in size.

1.5 ARM Design Philosophy

GTU - S-16, W-17, W-18

Q.	Describe the principle features of ARM Architecture.	**(S-16, 7 Marks)**
Q.	Describe the features inherited from RISC design by ARM processor architecture.	**(W-17, W-18, 3 Marks)**
Q.	Explain ARM design philosophy.	**(W-18, 4 Marks)**

The ARM processor had the main aim of having a small silicon usage and high performance with maximum operations implemented. The implementation was aimed at RISC organization. The main objectives/principle features of the ARM processor design are as listed below:

1. Design small processor so as to have low power consumption as required for an embedded system.
2. Programs are made of highly dense code so as to cater the limited memory and physical size constraints.
3. A very important factor to use slow and low-cost memory, so as to reduce the overall system cost.
4. Implementing more peripherals on smaller die so as to reduce the cost of the chip manufacturing as well as the chip size and hence the system size.

1.6 History of ARM Microprocessor

- The first ARM processor called as ARM 1 had its design completed in 1985, with another version ARM 2 in 1986.
- The basic features of ARM 2 were:
- 32-bit data bus,
- 26-bit address bus
- 16, 32-bit registers.
- ARM 2 was the world's simplest processor with only 30000 transistors and hence requiring minimal space on the silicon wafer.
- A successor, ARM3, was produced with a 4KB cache which further improved performance.
- In the late 1980s Apple Computer started working with Acorn on newer versions of the ARM core. The work was so important that Acorn spun off the design team in 1990 into a new company called Advanced RISC Machines.
- For this reason you often see ARM lengthened to Advanced RISC Machine instead of Acorn RISC Machine. Advanced RISC Machines became ARM Limited when the company floated on the London Stock Exchange and NASDAQ in 1998.
- The most successful implementation has been the ARM7TDMI with hundreds of millions sold in cellular phones. While ARM's business has always been to sell IP cores, some of the licensees generated microcontrollers based on this core.
- The five versions of ARM are as follows :

(a) Version 1

- This version was implemented on ARM 1 processor. It has 26 bit address space.
- It has branching instructions with simple branch as well as branch and link instruction for function/sub-routine calls
- It also has software interrupt calls as required for high level language.
- It supports byte, word and multi-word load-store instructions.
- It supports the data processing instructions also.
- Version 1 nowadays has become obsolete.

(b) Version 2

- This version is a updated version of version 1. It also has 26 bit address space.
- It supports coprocessor supported instructions.
- It supports multiply and multiply-accumulate instructions.
- It includes load and store instructions called SWP and SWPB in a variant called version 2.
- It supports two more banked registers in fast interrupt mode.
- Version 2 is also obsolete.

(c) Version 3

- This version has 32 bit address space.
- In this version with a view to retain the contents of CPSR register in the case of an exception two new registers were added.
- The program status information which was stored in register R15 is now stored in the CPSR (Current Program Status Register) and the SPSR (Saved Program Status Register).
- In version 3 two new instructions (MRS and MSR) were added to access the CPSR and SPSR registers.
- It supports two new processor modes. The new processor modes help to use Data Abort, undefined instruction exceptions in the code.

(d) Version 4

- The version 4 is an extension of version 3.
- It includes additional features like :
  - (i) load as well as store instructions for half word data.
  - (ii) instruction transfer to Thumb state in T variants of ARM processors.
  - (iii) instruction to transfer to the new privileged processor mode using the user mode register.
- It also made distinction of instructions that could cause exception to be taken.
- It also has backward compatibility for the earlier versions of ARM processor.

(e) Version 5

- The version 5 is an upward extension of version 4.
- It includes additional instructions that improve the efficiency of ARM interworking in T variants.
- Version 5 allows same code generation techniques to be used for T and non T variants.
- It supports an additional software breakpoint instruction.
- Version 5 also supports a Count Leading Zeros (CLZ) instruction that allows efficient integer divide and interrupt prioritization.
- Version 5 tightens the definition of how flags are set by multiply instructions.
- It supports co-processor instructions.
- The variants are as follows shown in Fig. 1.6.1.

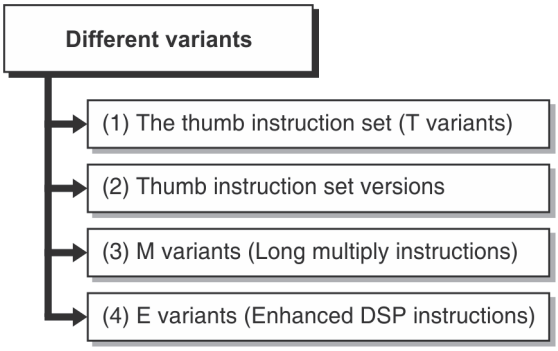


Fig. 1.6.1 : Different variants

(1) The Thumb Instruction Set (T variants)

- It is a subset of the ARM instruction set. The thumb instruction set has 16 bit instructions. i.e. the Thumb instructions are half the size of ARM instructions.
- The Thumb instruction set has greater code density than the ARM instruction set.
- The drawbacks of the Thumb instruction set are that it requires more instructions to complete the same task. Hence the ARM instruction set is best for maximizing the performance.
- The Thumb instruction set does not include instructions required for exception handling.
- Due to these drawbacks the Thumb instruction set is used in conjunction with a suitable version of the ARM instruction set.
- The Thumb instruction set is not valid prior to version 4 of the ARM architecture. It is signified by letter T.

(2) Thumb instruction set versions

- There are two versions of the Thumb instruction set i.e. version 1 and version 2.
- The version 1 of the Thumb instruction set is used in T variants of the ARM architecture version 4 and version 2 is used in T variants of the ARM architecture version 5.
- The Thumb instruction set version 2 modifies the instructions to improve the efficiency of ARM interworking.
- The Thumb instruction set tightens the definition of how flags are set by multiply instructions.
- The Thumb instruction set includes the breakpoint instruction.
- These modifications are related to the changes between ARM architecture versions 4 and 5.

(3) M variants (Long Multiply Instructions)

- The ARM instruction set includes four additional multiply instructions for producing 64 bit multiply and multiply and accumulate results.
- These instructions imply the existence of a multiplier that is larger than the minimum. Its presence is denoted by variant letter M.
- The long multiply instructions were first included in the variant of version 3 and exists in all after version 3.

(4) E variants (Enhanced DSP instructions)

- The E variants of the ARM instructions set consist of a number of ARM instructions that enhance the performance of an ARM processor for digital signal processing applications.
- The E variants support load, store, co-processor register transfer instructions that act on double word data.

- They include a cache preload instruction (PLD).
- They include several new multiply and multiply-accumulate instructions acting on 16 bit data.
- They were first included in the ARM architecture version 5T. They are not valid in non-T or non-M variants of the ARM architecture.

1.6.1 Terms Related to ARM Instruction Set

- Given below are some terms used in the ARM instruction set :

  {cond}: Condition field

- We know that an ARM instruction is 32 bit long. The 4 MSBs of the instructions are called as the condition field. The condition field is of 4 bits, hence there are 16 conditions as listed in Table 1.6.1.
- Fig. 1.6.2 shows the condition field.

Fig. 1.6.2 : ARM condition field

Table 1.6.1

Condition field	{cond}	
Suffix	**Description**	**Condition flag state**
EQ	Equal	Z = 1
NE	Not Equal	Z = 0
CS	Unsigned higher or same	C = 1
CC	Unsigned lower	C = 0
MI	Negative	N = 1
PL	Positive 0 or zero	N = 0
VS	Overflow	V = 1
VC	No overflow	V = 0
HI	Unsigned higher	C = 0 and Z = 0
LS	Unsigned lower or same	C =1 and Z = 1
GE	Signed greater than or equal	N = 1 and V = 1 or N = 0 and V = 0.
LT	Signed less than	N = 1 and V = 0 or N = 0 and V = 1.
GT	Signed greater than	Z = 0 and either N = 1 and V = 1 or N = 0 and V= 1.
LE	Signed less than or equal	Z = 1 or N = 1 and V = 0 or N = 0 and V = 1.
AL	Always (unconditional)	
NV	Reserved	

1.7 ARM Processor Family

- The ARM processor uses a nomenclature that describes the processor implementations.
- The word "ARM" is followed by letters and numbers indicating features of the processor as given below :

  ARM {x}{y}{z}{T}{D} {M}{I}{E}{J} {F}{S}.

  Such that

  x : ARM family.

  y : Memory management/protection unit.

  z : Cache.

  T : Thumb instruction set.

  D : Debug via JTAG interface.

  M : Long multiply instructions.

  I : Embedded ICE macrocell.

  E : Enhanced instructions (DSP applications).

  J : Jazelle.

  F : Vector Floating point unit.

  S : Synthesizable version.

Example : ARM7TDMI

- It implies that processor core is based on ARM7 family.
- It supports the Thumb instruction set. It supports debug via JTAG interface.
- It supports the long multiply instructions.
- It supports hardware breakpoints and watch points via the Embedded ICE macrocell.
- The nomenclature does not consist any information regarding architecture revision.
- A group of processors of processor family have the same hardware characteristics Example : ARM920T, ARM940T share ARM 9 family characteristics.
- The Thumb instruction set is supported for all versions after version 4. The variant T is automatic for ARM v6 and above.
- The core supports debug via JTAG interface. JTAG interface is described by IEEE 1149.1 standard Test Access Port and boundary scan architecture. It is used to send and receive debug information between processor core and the test equipment. The variant D is automatic for ARM v5 and above.
- The ARM core supports long multiply instructions for version 3 and above.

- Embedded ICE macrocell is the debug hardware that is built in processor for setting breakpoints and watchpoints.
- Enhanced DSP instructions are included in version 5 and above of ARM architecture. Variant E is included for all versions of ARM v 6 and above.
- The ARM core supports Jazelle Java acceleration architecture.
- It also supports vector floating point architecture.
- Synthesizable version implies that the processor core is supplied as source code that can be compiled into form easily used by EDA tools.

1.8 Development of ARM Architecture

- The ARM core is an engine within the system. It is responsible for fetching the instructions from the memory and executing them.
- The size of ARM core is very small; hence many additional components are added to the same chip.
- We will study the basic structure of the ARM core, as selection of an ARM core is the most critical decision while designing a system.
- Fig. 1.8.1 shows the ARM core data flow model. It supports Von-Neumann architecture.

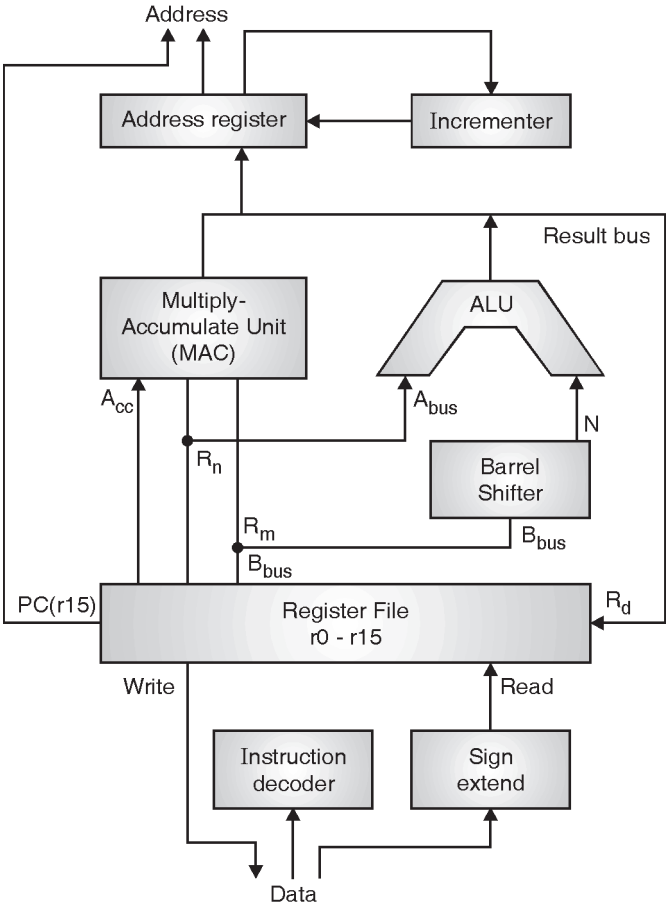


Fig. 1.8.1 : ARM core data flow model

Now, let us see the different functional units :

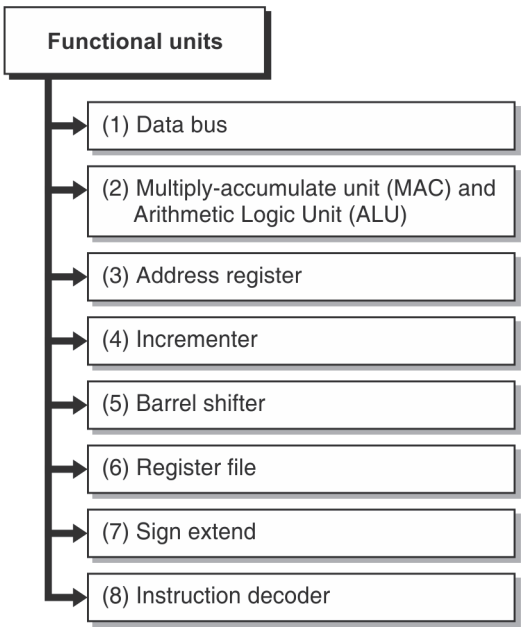


Fig. 1.8.2 : Functional units

(1) Data bus

The data enters the ARM core through the data bus. The same data bus is used by all the instructions as the ARM core uses Von-Neumann architecture. There are two buses A_{bus} and B_{bus}. The data can be an instruction opcode.

(2) Multiply-Accumulate Unit (MAC) and Arithmetic Logic Unit (ALU)

- Generally the ARM instructions are of three operands. In order to store the two input operands registers Rm and Rn are used.
- The ALU or MAC reads the two operand values from Rm and Rn registers using the Abus and Bbus.
- Then the desired operation is done and the result is stored using the result bus or Cbus in the destination register Rd.

(3) Address register

It is responsible for storing the address generated by the load and store instructions and placing it on the address bus.

(4) Incrementer

- The incrementer is responsible for upgrading the address register contents before the core reads the next register value from memory location or writes the next register value to memory location.
- In case of exception / interrupt the execution of instructions is changed.

(5) Barrel shifter

- The Rm register contents can be preprocessed in the barrel shifter before applying them to the ALU.
- A wide range of shift operations are possible using the barrel shifter and the ALU.

(6) Register file

The register file is a bank of sixteen 32-bit registers. The registers are used for storing the data.

(7) Sign extend

Some of the instructions require signed values. Whenever the processor reads an 8 or 16 bit signed number from the memory, the sign extend hardware converts the numbers to 32 bit values and places the number in the register file.

(8) Instruction decoder

- The instruction decoder decodes the instruction opcode read from the memory and then the instruction is executed.
- The ARM cores are of three types :
  (1) Application Core
  (2) Embedded Core
  (3) Secure Core
- Since the architecture of ARM based processor takes very less space and performs a lot of operations it is fit to be used in embedded systems.

1.9 Embedded System Hardware and Software

1.9.1 Embedded System Hardware

- Embedded System hardware includes various things like processor, memory, sensors, actuators etc. The following is a brief set of hardware requirements of an embedded system.

1. **Processor :** There are various processors like General Purpose Processor (GPP or MP), Microcontrollers (MC), Digital Signal Processors (DSP), Application Specific Processors ASP), etc. Even for a small application the minimum size of memory and I/O (s) must be interfaced to the CPU. This increases the hardware. In turn the PCB size and cost of the system also increases. In order to avoid these drawbacks **Microcontrollers** were developed. The Microcontroller is an on-chip true computer. It is optimized for specific applications. It consists of all the features of a microprocessor as well as the features required to build a complete computer.
2. **Memory :** There are various memories required for the embedded system. They include RAM (SRAM and DRAM), EPROM, Flash memory etc. RAM is used to store volatile data i.e. the run-time data. EPROM and flash memories are used to store permanent data including the program.
3. **Sensors :** An embedded system requires to take input from the environment or human, for this purpose it requires sensors that convert the physical quantity into electrical quantity.
4. **Actuators :** No embedded system is useful until it takes some input from the world and provides some output to the same. As sensors are used by embedded systems to take input from the user or the world, the actuators are used to provide the output to the world. Solenoid valves are automatic or electronically controlled valves to control the flow of various material especially liquids.

5. **Communication Interface :** An important element of an embedded system is a communication interface. There are various communication protocols covered in this book like USB, RS232, RS485 etc. An embedded system needs to communicate with different devices in the world for which it needs to follow some protocols like the TCP/IP, etc

1.9.2 Embedded System Software

- The firmware or the program of the embedded system are categorised as its boot-loader, RTOS, Drivers and application programs.
- The boot-loader is responsible for the execution of initial programs of the embedded system like memory checking etc.
- The next level of firmware is the operating system.
- The operating system required in the embedded system is the Real Time Operating System (RTOS). This will also be dealt in detail in this book.
- The next level is the drivers that are used to drive the various peripherals for the embedded system. There are various devices like LCD display, keyboard etc that require driver.
- Driver in this case means the program used to drive the devices discussed above. The application programs are the final stage code that are used to handle the application for the required system.
- Systems that are very small and have less complexity are normally called as super loops or foreground/background systems.
- It works in an infinite loop that has a huge number of functions to be executed. These functions in loop are called as background operations.
- There are also interrupts that are to be handled. These interrupt service routines are called as foreground operations.
- Hence the name given to such systems is called as foreground/background systems. Such small systems do not require RTOS. But if there are many tasks then we need a special operating system called as Real Time Operating System (RTOS).

❑❑❑

CHAPTER 2

UNIT II

ARM Architecture and Pipeline Structure

Syllabus

The Acorn RISC Machine, ARM Core data flow model, Architectural inheritance, The ARM7TDMI programmer's model: General purpose registers, CPSR, SPSR, ARM memory map, Data format, Load and store architecture, ARM 3 stage Pipeline, ARM 5 stage Pipeline, Pipeline Hazards, Data forwarding - a hardware solution, Stack implementation in ARM, Endianness, Condition codes. Processor core Vs CPU core, ARM7TDMI Interface signals

2.1 The Acorn RISC Machine

- The first ARM processor was developed at Acorn Computers Limited, England, between 1983 and 1985.
- At that time ARM stood for Acorn RISC Machine, until the formation of Advanced RISC Machines Limited when it was renamed Advanced RISC Machine in 1990.

2.2 ARM Core Data Flow Model

GTU - W-16, S-17, S-19

Q.	What is core ? Draw and Explain ARM core data flow mode.	**(W-16, 7 Marks)**
Q.	Draw the ARM core dataflow model.	**(S-17, 4 Marks)**
Q.	Explain concept of pipeline used in ARM processor. Explain three stage pipeline used in ARM7TDMI processor.	**(S-19, 7 Marks)**

2.2.1 5 Stage ARM9 Core Architecture

GTU - S-18

Q.	Explain 3 and 5 stage pipeline used in ARM processor.	**(S-18, 7 Marks)**

- Core of the microcontroller is the main part in it that performs the ALU, Control unit and register sections along with the buses.
- The ARM9 architecture is slightly different than the ARM architecture seen in the previous chapter. Fig. 2.2.1 shows the 5-stage pipeline organization of the ARM9 processor.

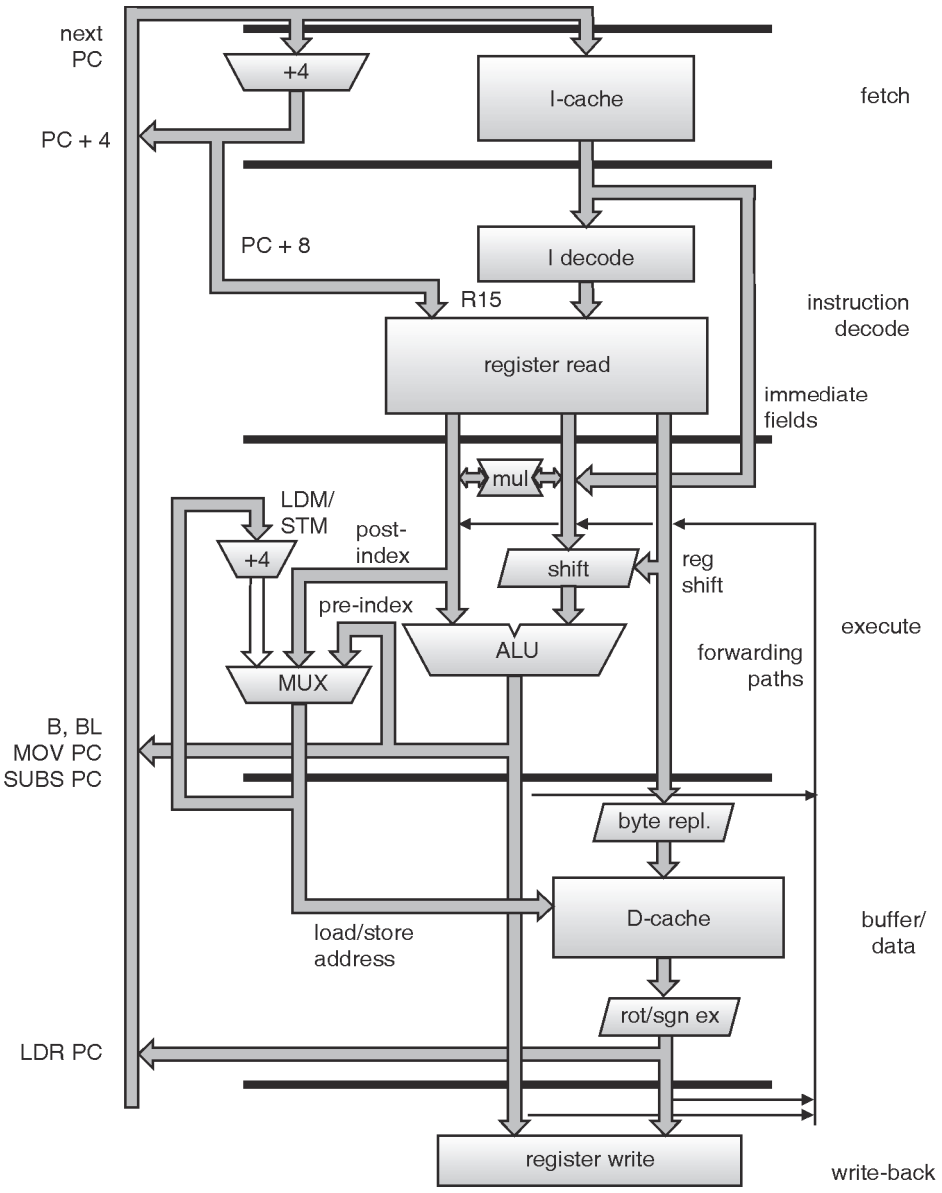


Fig. 2.2.1 : 5 stage pipeline of ARM9

2.2.2 Understanding the ARM Instruction Execution

The data and address flow for the execution of instructions in ARM9 processor is discussed in this module. The following common steps are taken for execution of all the instructions in ARM processor.

1. The Program counter issues the address that is given to the "increment" circuit.

   The increment circuit increments the value of program counter and gives the incremented value back to the program counter as well as to the address register from where it is issued on the address bus.

2. The instruction is then read from the memory and is stored into the "i.pipe".

   The flow of the address and the instruction is shown by dark arrows in Fig. 2.2.2.

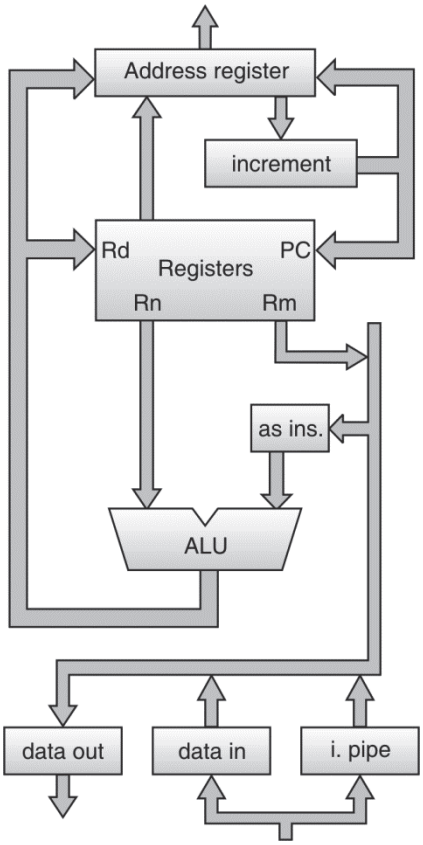


Fig. 2.2.2 : Instruction access in ARM9 processor

- The further execution of the instruction depends on the operation to be performed.
- For a data transfer instruction, simply the data will be transferred from one place to another.

2.2.3 Execution of Data Processing Instruction in ARM

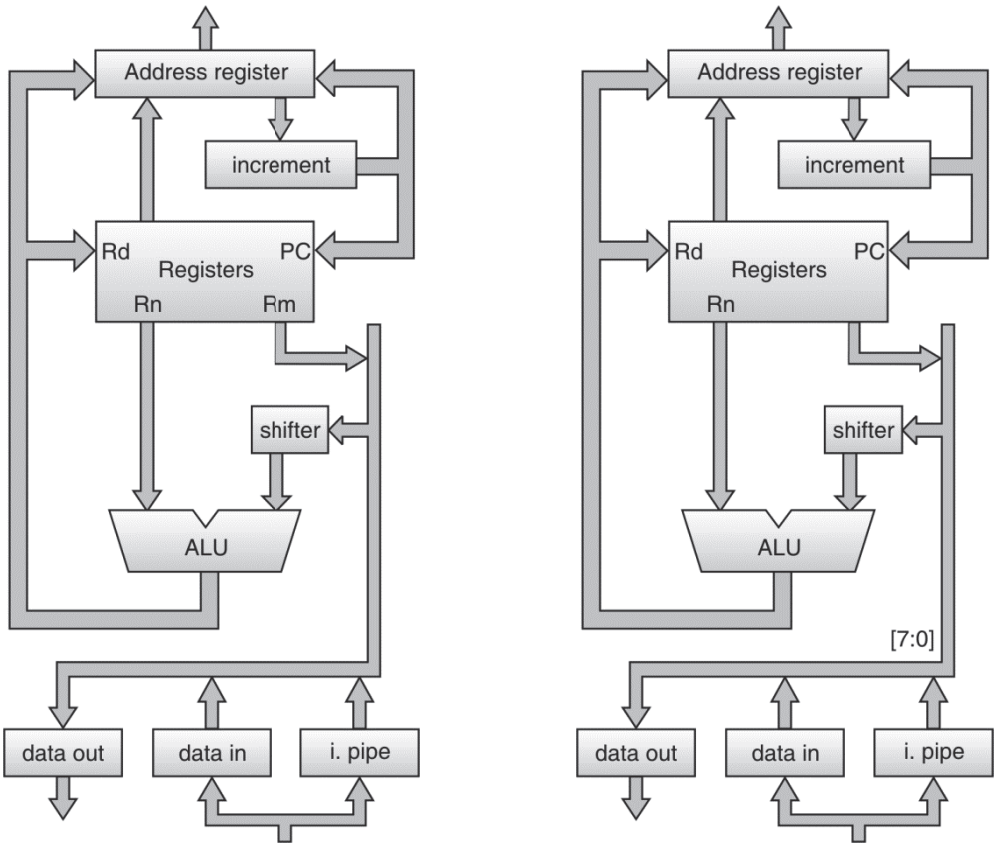
- For a data processing instruction the ALU will perform the execution of the instruction as required. The initial two steps will be same as discussed earlier i.e. the Program counter issuing the address and fetching the instruction in i.pipe.
- Then the registers are read and the ALU performs the operation as specified in the instruction. The source operands are the R_m and the R_n registers.
- The result is then given back to the destination register i.e. R_d. The steps are mentioned below :

1. The Program counter issues the address that is given to the "increment" circuit.

   The increment circuit increments the value of program counter and gives the incremented value back to the program counter as well as to the address register from where it is issued on the address bus.

2. The instruction is then read from the memory and is stored into the "i.pipe".
3. The source operands are given to the ALU, with one operand as register R_n. The second operand can be either a register i.e. R_m or its value shifted / rotated as specified in the instruction. Fig. 2.2.3 shows the flow with dark arrows, wherein the operation to be performed by the shifter and the ALU depends as specified in the instruction. There are two cases possible i.e. either the second operand is in the register or is an immediate data.

These are shown in the Fig. 2.2.3 (a) and Fig. 2.2.3(b).



(a) Register to register operation **(b) Register and immediate operation**

Fig. 2.2.3 : Data processing instruction execution

4. Finally the result is stored in the destination register i.e. R_d

- The flow of the address and the instruction is shown by dark arrows in Fig. 2.2.3. The two cases as discussed are based on the second operand being register or immediate data.

2.2.4 Execution of the Branch Instructions in ARM Processor

- In this case again the initial process of fetching the instruction into the i. pipe is same as discussed in the earlier subsection.
- Thereafter the 24-bit immediate value after being shifted left twice is added to the current value of program counter.
- This value is given to the address register and the new instruction is fetched from there. Also the return address value from the program counter is given to the link register, if required.
- This value given to the link address is also adjusted to point to the next instruction. Thus it is a three cycle operation.

The three cycles are required for the following operations :

1. The target address is computed in the first cycle. As discussed, this is done by adding the 24-bit immediate value left shifted twice with the value of PC.

2. In the second cycle the address result issued in the first cycle is fetched from and the instruction is filled in the i. pipe. Also in this cycle the return address is copied in the link register, if required.
3. In the third cycle a slight adjustment is done in the link address by decrementing its value with 4.

Fig. 2.2.4 shows the flow of the address and data in the first and second cycles using block arrows.

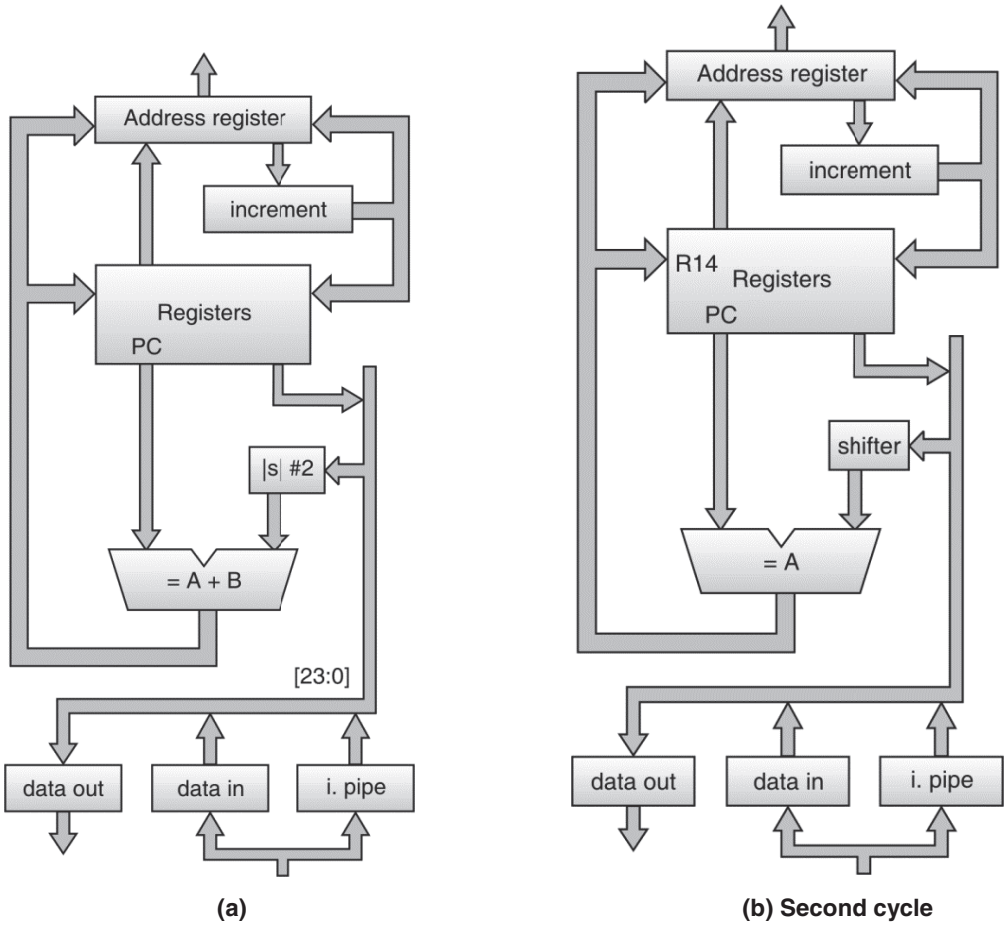


Fig. 2.2.4 : Branch instruction execution

2.3 Architectural Inheritance

The ARM architecture incorporated a number of features inherited from RISC design like :

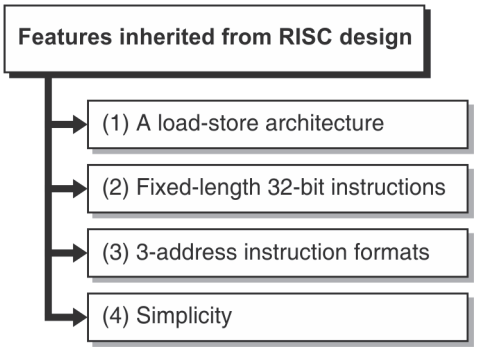


Fig. 2.3.1 : Features inherited from RISC design

(1) A load-store architecture

- A load-store architecture means that ALU instructions cannot access memory, they can operate only on registers.
- Data transfer between memory and processor is possible only by Load and Store instructions.
- LOAD instruction is used to load the data from memory into the processor's register.
- STORE instruction is used to store data in memory from the processor's register.
- The data in the registers can be processed by the ALU.

(2) Fixed-length 32-bit instructions

- All the instructions are 32-bit in length.
- This fixed length instructions make the design of the control unit simpler.
- Fixed length instructions also make it possible to use Hardwired control unit, which makes the operation of control unit faster.

(3) 3-address instruction formats

- This is also referred to as 3-opeand instructions
- The two source operands are different then the destination operand for e.g. in the add instruction given below

$$\text{ADD } d, s_1, s_2 ;$$

Operation of above instruction: d := s1 + s2

f bits	n bits	n bits	n bits
function	op 1 addr.	op 2 addr.	dest. addr.

Fig. 2.3.2 : A 3-address instruction format

(4) Simplicity

The greatest advantage of ARM processor is that it is not only having the great hardware but also very good instruction support making it highly efficient processor.

Although ARM is said to be a RISC processor, it has not implemented certain standard features of RISC processor. Here are some of them:

(1) Register Windowing

(2) Delayed Branching

(3) Single cycle execution of all instructions.

2.4 The ARM7TDMI Programmer's Model: General Purpose Registers, CPSR, SPSR

GTU - W-17, S-18, W-18, W-19

Q.	Describe the operating modes in ARM processor.	**(W-17, W-19, 7 Marks)**
Q.	Describe Exception and processor sequence followed when an Exception is recognized ARM processor.	**(W-17, 7 Marks)**
Q.	Explain ARM7TDMI programmer's model. Explain all registers of ARM7TDMI architecture.	**(S-18, 7 Marks)**
Q.	What are the exceptions in ARM processor? Explain mode of operation for different exceptions.	**(S-18, 7 Marks)**
Q.	Describe ARM register set? Also specify the flags in ARM-7 processor?	**(W-18, 7 Marks)**
Q.	Describe ARM7TDMI programmer's model. Explain all registers of ARM7TDMI architecture.	**(W-18, 7 Marks)**
Q.	Explain various exception handling schemes used in ARM processor.	**(W-18, W-19, 7 Marks)**
Q.	What are the exceptions in ARM processor? Explain process of exception entry and exception return.	**(S-19, 4 Marks)**

- The ARM processor can be operated in seven different modes. The current mode of the processor decides the availability of the registers to the programmer.

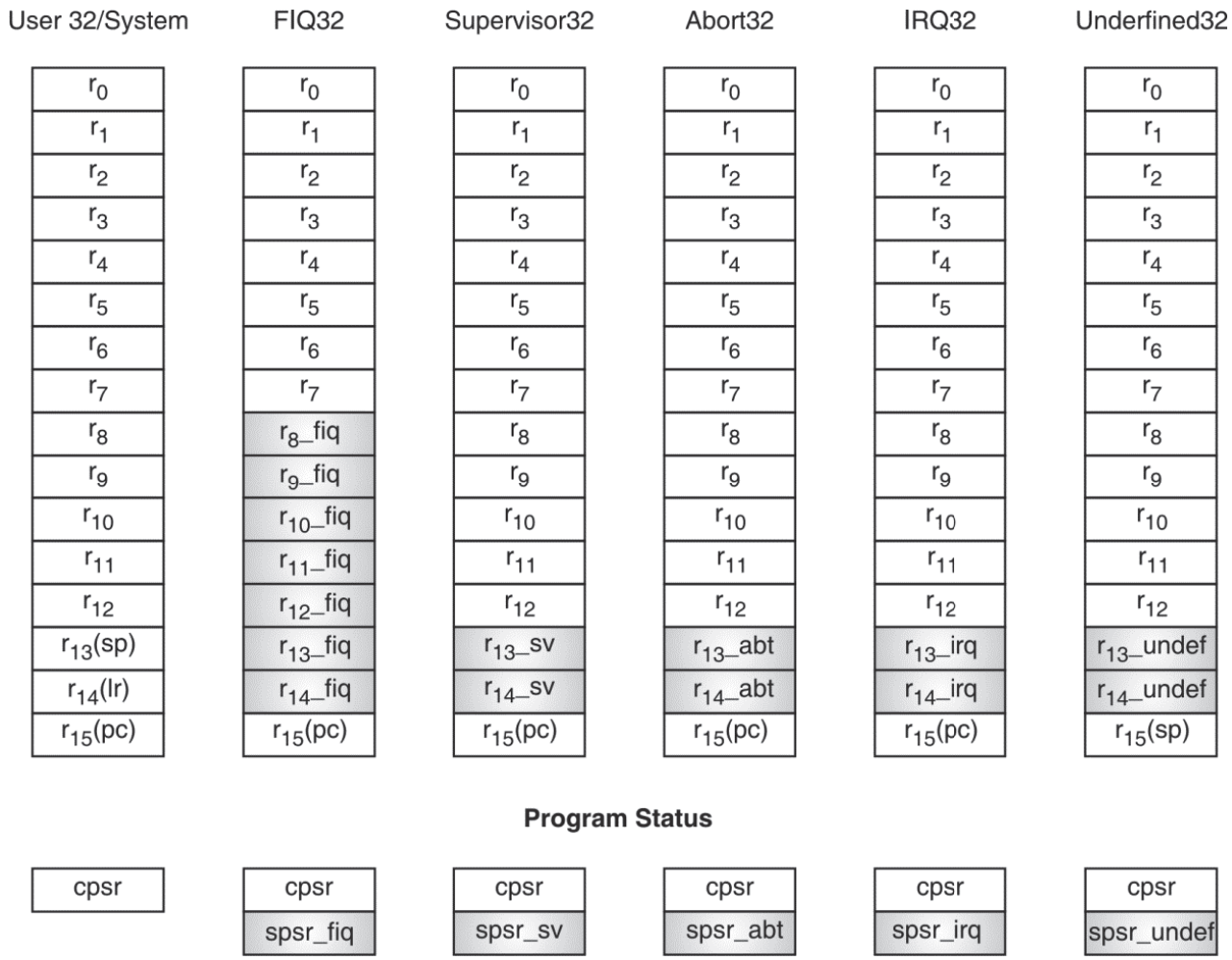


Fig. 2.4.1 : ARM programming model

- Fig. 2.4.1 shows the programming model of the ARM processor. The ARM processor has 37 thirty two bit registers.

- The registers include of
  - (i) a dedicated program counter.
  - (ii) a dedicated current program status register.
  - (iii) a dedicated saved program status registers.
  - (iv) 30 general purpose registers.
- However these registers are arranged into several banks, with the accessible bank being governed by the processor mode.
- Each processor mode can access a particular set of R0-R12 registers, R13 register (stack pointer) and link register (R14), register R15 (program counter) and CPSR (current program status register). Privileged modes can access SPSR (saved program status register).
- Fig. 2.4.1 shows the programming model of the ARM processor.

(1) General purpose registers

- The registers R_0 to R_{12} are general purpose registers i.e. they do not have any special function and can be used as required by the programmer to hold data as well as address.

(2) Stack Pointer (SP-R13)

- Register R13 is a stack pointer as shown in Fig. 2.4.1.
- It stores the top of the stack in the current processor mode.

(3) Link Register (LR-R14)

- Register R14 is a link register. It is used for storing the return address in case of subroutine calls.

(4) Program Counter (PC-R15)

- Register R15 is the program counter.
- It stores the address of the next instruction to be fetched from the memory by the processor.
- It can be used in place of registers R0 to R14. Hence it is considered as one of the general purpose registers.
- Its use is sometimes restricted depending on the type of instruction.
- Generally it is used as a pointer to the instruction which is two instructions after the instruction being executed.
- We know that all ARM instructions are 32 bit long. i.e. they are aligned on word boundary. Hence the PC value is stored in bits [31:2] with bits [1:0] equal to zero.

(5) Registers R0-R7

- These registers are called as **unbanked registers**. They are general purpose registers and can be used wherever general purpose registers can be used.
- Each register is a 32 bit physical register in all the processor modes. The registers R0-R7 do not have any specific use.

(6) Registers R8-R14

- These registers are called as **banked registers.** This is because the physical size of each of these registers depends on the current processor mode.
- Whenever a general purpose register is used, any of these registers can be accessed.
- Out of 37 registers, only 20 registers which are shown shaded in Fig. 2.4.1 are used as banked registers.
- Fig. 2.4.1 also indicates the mode of the bank registers. Banked registers of a particular mode are denoted by r number_mode Example abort mode has banked registers r13_abt, r14_abt and spsr_abt.

Note : Register R8 to R12 are subject to restrictions on the use of bank registers in systems where FIQ mode is never used. Hence, only one physical version of the register is always in use.

- Registers R8 to R12 are physically different registers when in the FIQ mode. This has an advantage of low context switching latency.
- This is because, there is no need of performing the PUSH and POP operations on these registers during an interrupt service routine execution.
- R13 and R14 are physically different for each of the six modes. Register R13 is a stack pointer (SP) that always points to the top of the stack.
- Register R14 is a link register that is used to store the return address from where the control has to resume in the caller mode. Register R14 It has two special functions. They are,

(a) In each mode, the modes own version of register R14 is used to hold the subroutine return address. The return task is achieved by loading R14 back to the program counter.

This task can be obtained by methods given below :

(i) Execute either of these instructions :

```
MOV   R15, R14
            Or
      MOV   PC, LR
            Or
      BX   LR
```

(ii) While entering the subroutine, store the contents of register R14 to the stacking using instruction

STMFD SP!, {<registers>, LR}

and to return use instruction

LDMFD SP! {<registers>, PC}.

(b) Incase of an exception, the exception mode's version of R14 is set to the exception return address. The exception return is done processed in the same way as the subroutine return, but using different instructions to ensure full restoration of the program state.

2.4.1 General Purpose Registers

GTU - S-16, S-17, S-18, W-18, S-19

Q.	Explain ARM processor mode.	**(S-16, 7 Marks)**
Q.	Which processor mode is non privileged in the case of ARM7TDMI core ?	**(S-17, 1 Mark)**
Q.	Operating system kernel operate in which mode of processor ?	**(S-17, 1 Mark)**
Q.	Draw the diagram showing complete ARM register set or programmable register model for ARM7TDMI core and describe it in brief.	**(S-17, 7 Marks)**
Q.	What is difference between FIQ and IRQ? Why FIQ response is fast than IRQ?	**(S-18, W-18, 4 Marks)**
Q.	Why FIQ response is fast in ARM processor compared to IRQ?	**(S-19, 3 Marks)**

- The ARM processor supports **seven** operating modes. The processor mode decides availability of the registers to the programmer and also access rights to the CPSR.
- The processor modes are classified as :
  - (i) privileged mode
  - (ii) non-privileged mode.
- Privileged mode supports full read and write access to the CPSR. The privileged modes supported by processor core are :
  - (i) Abort (ii) Fast interrupt request
  - (iii) Interrupt request (iv) Supervisor
  - (v) Undefined (vi) System.
- Non-privileged mode allows only read access to the control field in the CPSR. However it allows read-write access to the condition flags. Non-privileged mode supported by processor core is shown in Fig. 2.4.2 :

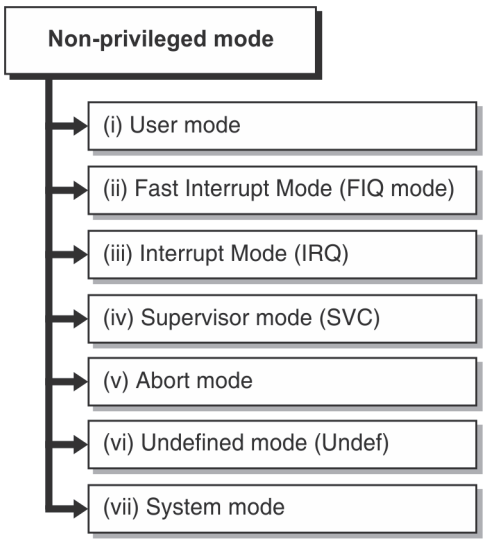


Fig. 2.4.2 : Non-privileged mode

(i) User mode

- It is a non privileged mode. Most of the tasks are executed in the user mode.
- Memory access is restricted in user mode. User cannot read directly from hardware device.

(ii) Fast Interrupt Mode (FIQ mode)

- This mode is entered whenever a high priority of interrupt is raised.
- FIQ mode is used to handle the peripherals that issue fast interrupts.
- This mode is privileged mode. The devices causing FIQs are floppy disc handling data, serial port etc.

(iii) Interrupt Mode (IRQ)

- This mode is a privileged mode.
- It is entered when a low priority interrupt is raised. Example keyboard, hard disk, floppy disc etc.

Difference between FIQ and IRQ

- An IRQ may be interrupted by an FIQ but an FIQ cannot be interrupted by IRQ.
- With FIQ we have to do processing fast. For achieving this, the processor has shadow registers.
- FIQs cannot call software interrupts (SWIs).
- If it becomes essential for an FIQ routine to re-enable interrupts, it will take longer time than it would take by an IRQ.
- FIQ should disable interrupts.

(iv) Supervisor mode (SVC)

- The ARM processor enters this mode on rest. This mode can also be entered if a software interrupt (SWI) instruction is executed.
- Supervisor mode has additional privileges that allow greater control of the system.
- An operating system kernel operates in this mode.
- In order to read from a I/O Module, programmer has to enter the supervisor mode.

(v) Abort mode

The abort mode is entered whenever an attempt access memory fails .

(vi) Undefined mode (Undef)

This mode is use to handle undefined instructions.

(vii) System mode

- This mode is a privileged mode. It is a special version of the user mode.
- It allows full read-write access to the CPSR.
- It is present in ARM architecture version 4 and above.

Summary of ARM processor modes is given in Table 2.4.1 :

Table 2.4.1 : ARM processor modes

Mode	Denotation	Privileged	Enter this mode in case of Exception	Mode select Bits in CPSR				
				B4	B3	B2	B1	B0
User mode	usr	No	No	1	0	0	0	0
Fast Interrupt request	fiq	Yes	Yes	1	0	0	0	1
Interrupt request	irq	Yes	Yes	1	0	0	1	0
Supervisor mode	svc	Yes	Yes	1	0	0	1	1
Abort	abt	Yes	Yes	1	0	1	1	1
Undefined	undef	Yes	Yes	1	1	0	1	1
System	sys	Yes	No	1	1	1	1	1

2.4.2 Program Status Registers (CPSR and SPSR)

GTU - S-16, W-16, S-17, W-17, S-18, W-18, W-19

Q.	Explain CPSR and SPSR register.	**(S-16, 7 Marks)**
Q.	What is CPSR? Draw an approximate format of SPSR register.	**(W-16, 3 Marks)**
Q.	State the status of lower five bits of CPSR for Abort mode in the case of ARM7TDMI core .	**(S-17, 1 Mark)**
Q.	Explain the importance of CPSR and SPSR register in ARM processor with an example.	**(W-17, 4 Marks)**
Q.	Explain program status registers CPSR and SPSR. Why two program status registers are used in ARM processors?	**(S-18, 3 Marks)**
Q.	Discuss the CPSR and SPSR with suitable example.	**(W-18, 4 Marks)**
Q.	Explain CPSR Register.	**(W-19, 4 Marks)**

- The ARM architecture supports two program status registers. They are the **Current Program Status Register (CPSR)** and the **Saved Program Status Register (SPSR)**.
- The CPSR is accessible in all the processor modes while the SPSR is accessible in privileged modes.
- The CPSR contains condition code flags, interrupt disable bits, current processor mode and other control and status information. Each exception mode contains a Saved Program Status Register (SPSR). It is responsible for holding the value of CPSR in case the exception occurs. Fig. 2.4.3 shows the format of the CPSR and SPSR registers.

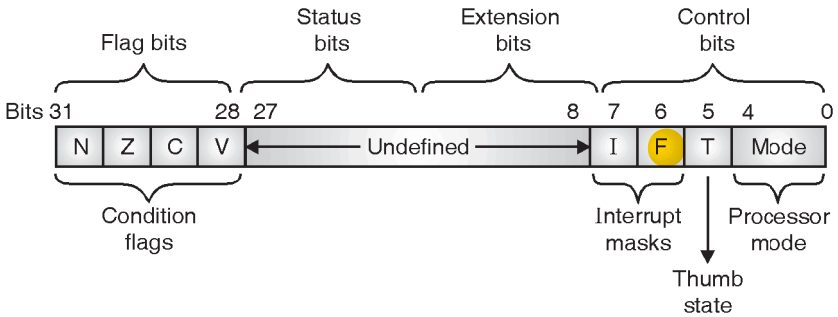


Fig. 2.4.3 : CPSR and SPSR register format

Now let us see description of each bit.

(1) Control Bits (Bits 0-7)

- Bits (0-7) are the control bits of CPSR and SPSR register. These bits change in case an exception occurs.
- The control bits can be altered by the software only when the processor is in privilege mode.

(2) Bits 0-4 (Processor Mode)

These bits determine the processor mode. The modes are listed in Table 2.4.2.

Table 2.4.2 : Processor modes

Processor mode	Bits				
	4	3	2	1	0
User	1	0	0	0	0
Fast Interrupt Request (FIQ)	1	0	0	0	1
Interrupt Request (IRQ)	1	0	0	1	0
Supervisor	1	0	0	1	1
Abort	1	0	1	1	1
Undefined	1	1	0	1	1
System	1	1	1	1	1

(3) Bit 5 (Thumb Bit) (Architecture version 4T only)

- This bit determines that the ARM processor is in Thumb mode or in ARM mode. Based on the mode is the instruction set to be used.
- ARM processor normally has 3 instruction sets : ARM, Thumb and Jazelle.
- When the T bit is 1, the processor is in the Thumb state and executes the Thumb instructions.
- In such processors the J and T bits decide the state of the processor.
- When the T and the J bits are '0', the processor is in the ARM state.
- To change the state of processor specialized branch instructions are used.

Comparison of ARM and Thumb instruction set

Sr. No.	Parameter	ARM instruction set	Thumb instruction set
1.	CPSR bit T	T = 0	T = 1
2.	Instruction size	32 bit	16 bit
3.	Core instructions	58	30
4.	Conditional execution	For most of the instructions	For branch instructions only.
5.	Program status register	Read-Write in privileged mode	No direct access.

Sr. No.	Parameter	ARM instruction set	Thumb instruction set
6.	Register access	All 15 general purpose registers + PC	8 general purpose registers + PC + 7 high registers.
7.	Data processing instructions	Access to Barrel shifter and ALU	Separate Barrel shifter and ALU instructions.
8.	Versions	Supported by all versions	Supported to all versions after version 4
9.	Exception handling instructions	Supported	Not supported
10.	Instructions required to do a task	Less	More than ARM instructions

(4) Jazelle instruction set

- The Jazelle instructions are 8 bit in size. These instructions are designed with the view to increase the speed of execution of Java bytecodes.
- The Jazelle instructions are a hybrid mixture of software and hardware. Over 60% of the Java bytecodes are implemented in hardware while remaining are implemented in software.

(5) Bits 6, 7 (Interrupt Masks)

- The ARM core has two interrupts. They are Fast Interrupt Request (FIQ) and Interrupt Request (IRQ).
- These interrupts are maskable. Bit 6 (F) controls the FIQ and Bit 7 (I) controls the IRQ. For their mask enable/disable.
- When bit is set i.e. F = 1 or I = 1 the respective interrupt is masked or disabled. When bit is clear, the interrupt is unmasked or available.

(6) Condition code flags

- The flags are updated by the operations performed by the ALU. The flags in the CPSR can be tested by the instructions to determine whether the instruction is to be executed or not.
- The flags can be updated by execution of a comparison instruction or by execution of arithmetic, logical or move instruction where the destination register is not R15.

(7) Bit 27 (Saturation Flag : Q)

- It is present on the DSP extension of ARM processor core.
- This bit is set if an overflow or saturation occurs. The hardware can only set this flag. In order to clear this bit we need to write to CPSR directly.
- In case of SPSR this bit is used to hold and restore the CPSR Q flag in case of exceptions.

(8) Bit 28 (Overflow Flag : V)

- This flag is set if an overflow occurs while addition or subtraction.
- For operations other than addition/subtraction overflow is unchanged.

(9) Bit 29 (Carry Flag : C)

This flag is set if one of the following conditions exist.

(i) For addition alongwith the comparison instruction CMP, if the addition produced a carry otherwise flag is clear.

(ii) For subtraction alongwith the comparison instruction CMN, if subtraction produced a borrow, then the carry flag is cleared otherwise it is set.

(iii) For operations not involved with addition/subtraction but include a shift operation, the carry flag is set to the last bit shifted out of the value by the shifter. For operations that do not include a shift, carry flag remains unaltered.

(10) Bit 30 (Zero flag : Z)

If the result of comparison is zero i.e. equal numbers are present then the zero flag is set to 1. Z flag is otherwise cleared.

(11) Bit 31 (Negative Flag : N)

- The negative flag is set if the result is negative and regarded as two's complement of a signed integer.
- Otherwise the flag is cleared if the result is positive or zero.
- Other methods to modify the N, Z, C and V flags are listed below :

(i) Execution of flag setting variants arithmetic and logical instructions such that the destination register is R15. The variants also copy the SPSR to CPSR. They are mainly used while returning from exceptions.

(ii) Execution of MRC instruction such that the destination register is R15.

(iii) Execution of MSR instruction, as part of its function of writing a new value to CPSR or SPSR.

(iv) Execution of some variants of LDM instruction.

(12) Reserved bits

These bits are reserved for further expansion in future.

2.4.3 Barrel Shifter

GTU - S-16

Q. Explain rotate and barrel shifter instruction. **(S-16, 7 Marks)**

- The ARM processor does not support actual shift instructions. But instead it supports a **Barrel shifter**. The Barrel shifter is responsible for carrying the shifts as a part of other instructions.
- The barrel shifter can be used to perform left shift operation by required amount. i.e. it multiplies by the powers of 1.

For Example : LSL #4 (multiply by 16).

Fig. 2.4.4 : Logical shift left operation

- The barrel shifter can also to used to perform logical and arithmetic right.

- In logical shift right the barrel shifter shifts the bits to the right by specified amount. i.e. it divides by power of 2

  Example : LSR #3 (divide by 8).

Fig. 2.4.5 : Logical shift right

- In arithmetic shift right the barrel shifter shifts the bits to the right by specified amount and preserves the sign bit, for 2's complement operations.

  Example : ASR #5 (divide by 32).

Fig. 2.4.6 : Arithmetic shift right

- The barrel shifter can also be used for performing rotations.
- In Rotate Right (ROR) the barrel shifter performs operation similar to arithmetic shift right but wraps the bits around as they leave the LSB and appear as MSB. The last bit rotated is also used as carry out alongwith LSB. Example ROR #2.

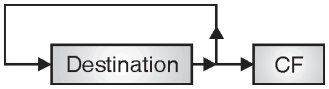


Fig. 2.4.7 : Rotate right

- For Rotate Right through carry or Rotate Right Extended (RRX) the barrel shifter uses the CPSR C flag as 33rd bit. In this operation the bits are rotated through the carry. The carry propagates to LSB and MSB to carry.
- Using the barrel shifter alongwith ALU a wide range of addresses can be calculated.

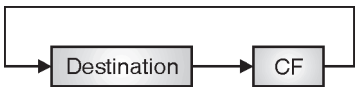


Fig. 2.4.8 : Rotate right through carry

- The contents of register R_m are preprocessed in the barrel shifter before applying them as input to the ALU. The register R_m can be a register, shifted register or immediate value.

2.5 ARM Memory Map#

- The LPC17xx incorporates several distinct memory regions, shown in the following Fig. 2.5.1 shows the overall map of the entire address space from the user program viewpoint following reset, the interrupt vector area supports address remapping.
- The AHB peripheral area is 2 MB in size, and is divided to allow for up to 128 peripherals. The APB peripheral area is 1 MB in size and is divided to allow for up to 64 peripherals. Each peripheral of either type is allocated 16 KB of space. This allows simplifying the address decoding for each peripheral.

Courtesy NxP

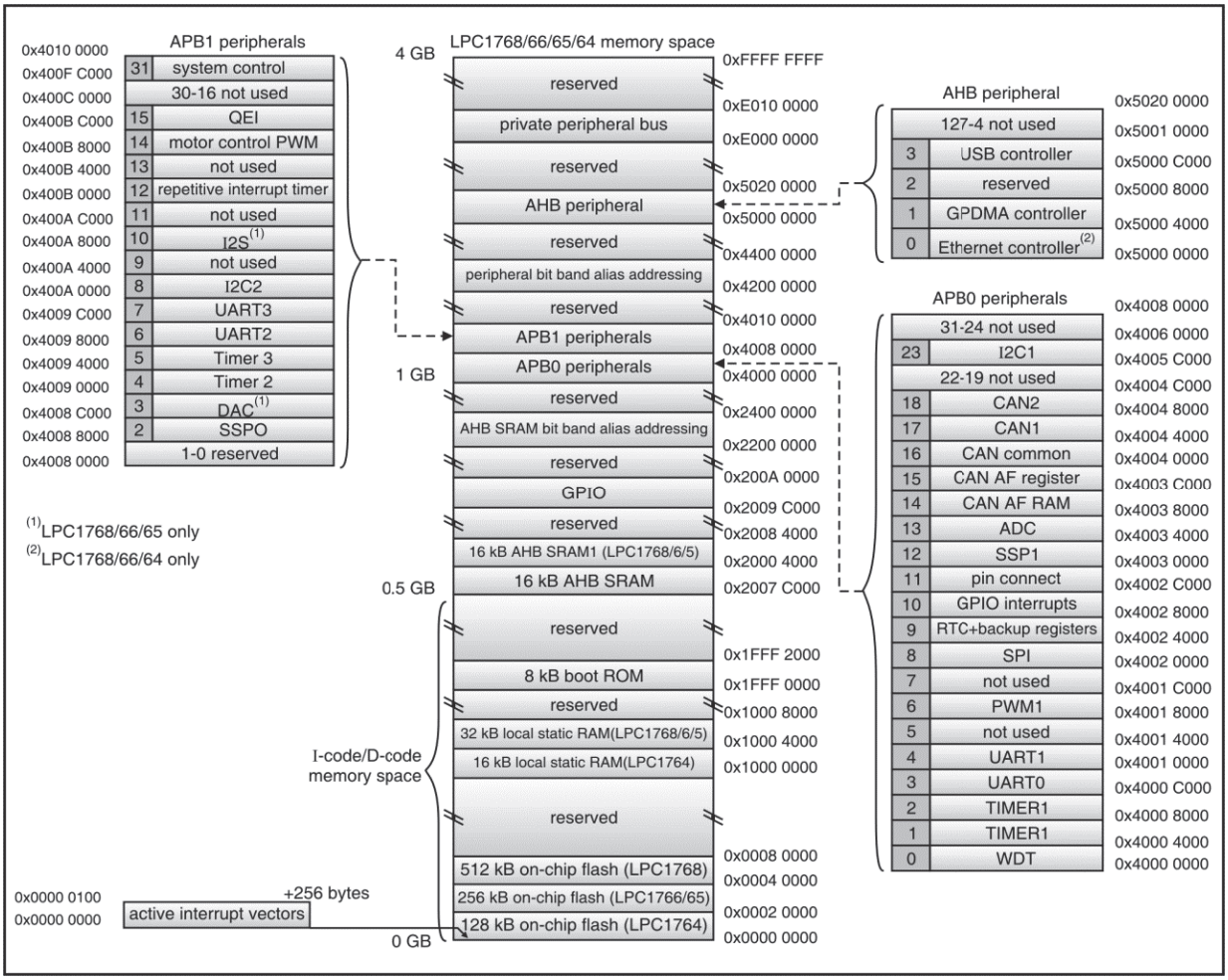


Fig. 2.5.1 : LPC1768/66/65/64 memory map#

2.6 Data Format

- The data types supported by the ARM processors are
  1. Byte i.e. 8 bits data.
  2. Half word i.e. 16 bit data.
  3. Word i.e. 32 bit data.
- The ARM instructions are one word instructions (32-bit) while the Thumb instructions are half word instructions i.e. 16-bit.
- All data operations are done on 32-bit data i.e. word.
- All the three data types are supported in the ARM architecture version 4 and above. Prior to ARM architecture version 4 only bytes and words were supported.

Courtesy NxP

2.7 Load and Store Architecture

GTU - W-18

Q. Explain addressing mode of ARM Processor with example. **(W-18, 7 Marks)**

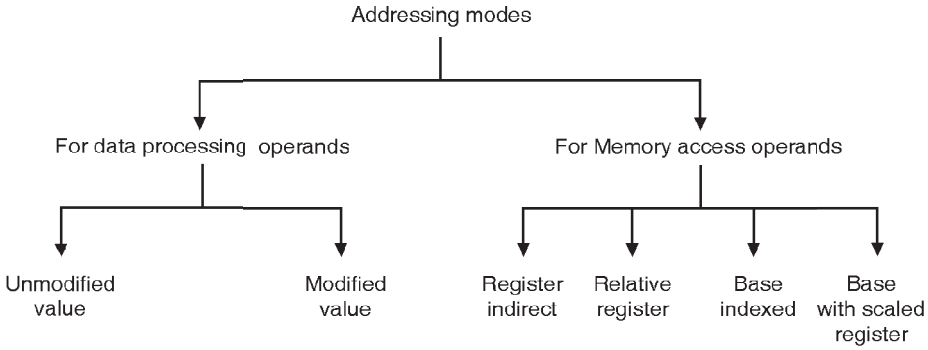


Fig. 2.7.1 : ARM addressing modes

- As already discussed Load and Store instructions are used to access memory. The different addressing modes to access memory and data processing instructions are shown in Fig. 2.7.1
- Addressing modes refer to the different methods of addressing (selecting) the operand. In ARM instruction set, the addressing modes are broadly divided into two categories viz for data processing operands and for memory access operands. Each of these categories have different methods as shown in Fig. 2.7.1.
- Each of the memory addressing modes accept for register indirect can also be used with pre-index and post index as explained in the subsequent subsections.

2.7.1 Addressing Modes for Data Processing Operands (i. e. op1)

These are two method for addressing these operands :

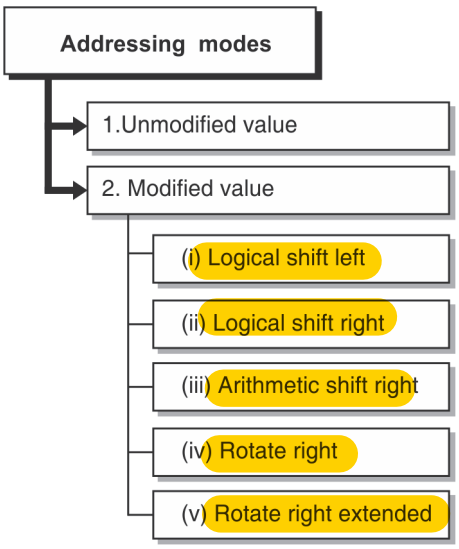


Fig. 2.7.2 Addressing modes

1. Unmodified value

In this addressing mode, the register or a value is given unmodified i.e. without any shift or rotation.

Example : (i) MOV R0, # 1234 H : This instruction will move the immediate constant value $(1234)_{10}$ into register R0.

(ii) MOV R0, R1 : This instruction will move the value in the register R1 into the register R0.

2. Modified value

In this addressing mode, the given value or register is shifted or rotated. These are different shift and rotate operations possible as listed below with examples.

(i) Logical shift left

This will take the value of a register and shift the value towards most significant bits, by n bits.

Example : MOV R0, R1, LSL # 2 :After the execution of this instruction R0 will become the value of R1 shifted by 2 bits.

(ii) Logical shift right

This will take the value of a register and shift the value towards right by n bits.

Example : MOV R0, R1, LSR R2 :After the execution of this instruction R0 will have the value of R1 shifted right by R2 times. R1 and R2 are not altered.

(iii) Arithmetic shift right

This is similar to logical shift right, except that the MSB is retained as well as shifted for arithmetic shift operation .

Example :MOV R0, R1, ASR #2 :After the execution of this instruction R0 will have the value of R1 Arithmetic shifted right by 2 bits.

(iv) Rotate right

This will take the value of a register and rotate it right by n bits.

Example : MOV R0, R1, ROR R2 :After the execution of this instruction R0 will have the value of R1 rotated right for R2 times.

(v) Rotate right extended

This is similar to Rotate right by one bit, with the carry flag moved into the MSB, i.e. it is similar to rotate right through carry.

Example :MOV R0, R1 RRX : After the execution of this instruction R0 will have the value of register R1 rotated right through carry by 1 bit.

2.7.2 Addressing Modes for Memory Access Operands

As already discussed load and store instructions are used to access memory. The different memory access addressing modes are

(i) Register indirect addressing mode
(ii) Relative register indirect addressing mode
(iii) Base indexed indirect addressing mode
(iv) Base with scale register addressing mode.

Each of these addressing modes have offset addressing, Pre-index addressing and post-index addressing as explained in the examples for each addressing mode.

(i) Register indirect addressing mode

- In this addressing mode, a register is used to give the address of the memory location to be accessed.

  Example LDR R0, [R1] :This instruction will load the register R0 with the 32-bit word at the memory address held in the register R1

(ii) Relative register indirect addressing mode

- In this addressing mode the memory address is generated by an immediate value added to a register. Pre index and post index are supported in this addressing mode.

  Example (a) LDR R0, [R1, #4]:This instruction will load the register R0 with the word at the memory address calculated by adding the constant value 4 to the memory address contained in the R1 register

  Example (b) LDR R0, [R1, #4]! : This is a pre-index addressing. This instruction is same as that in Example (a) this instruction also places the new address in R1 i.e. R1 ← R1 + 4.

  Example (c) LDR, [R1], #4

- This is post-index addressing. This instruction will load register R0 with the word at memory address given in register R1. It will then calculate the new address by adding 4 to R1 and place this new address in R1

(iii) Base indexed indirect addressing mode

- In this addressing mode the memory address is generated by adding the values of two registers. Pre-index and post-index are supported also in this addressing mode.

  Example (a) LDR R0, [R1, R2]

- This instruction will load the register R0 with the word at memory address calculated by adding register R1 to register R2.

  Example (b) LDR R0, [R1, R2]!

- This is pre-index addressing. This instruction is same as that in Example (a). this instruction also places the new address in R1 i. e. R1 ← R1 + R2.

  Example (c) LDR R0, [R1], R2

- This is a post-index addressing. This instruction will load register R0 with the word at memory address given in register R1. It will then calculate the new address by adding the value in register R2 to register R1 and Place this new address in R1.

(iv) Base with scaled register addressing mode

- In this addressing mode the memory address is generated by a register value added to another register shifted left. Pre-index and post-index are supported in this addressing mode.

  Example (a) LDR R0, [R1, R2, LSL #2]

- This instruction will load the register R0 with the word at the memory address calculated by adding register R1with register R2 shifted left by 2 bits.

  Example (b) LDR R0,[R1, R2, LSL #2]!

- This is a pre-indexed addressing. This instruction will load the register R0 with the word at the memory address calculated by adding register R1 with register R2 shifted left by 2 bits. The new address is placed in register R1.

  i. e. R1 ← R1 + R2 << 2.

  Example (c) LDR R0, [R1], R2, LSL #2.

- This is a post-indexed addressing. This instruction will load the register R0 with the word at memory address contained in register R1. It will then calculate the new address by adding register R1 with register R2 shifted left by two bits. The new address is placed in register R1.

2.8 ARM 3 and 5 Stage Pipeline

GTU - S-16, S-17

Q.	Draw and explain 3 stage pipeline Architecture of ARM Processor.	**(S-16, 7 Marks)**
Q.	Define pipeline concept for any processor.	**(S-17, 1 Mark)**
Q.	How many pipeline stages are there in ARM11 core ?	**(S-17, 1 Mark)**

- The process of fetching the next instruction while the current instruction is being executed is called as **"pipelining"**.
- Pipelining is supported by the processor to increase the speed of program execution. It also increases throughput.
- Several operations take place simultaneously, rather than serially in pipelining system.
- The pipeline has three stages fetch, decode and execute as shown in Fig. 2.8.1.

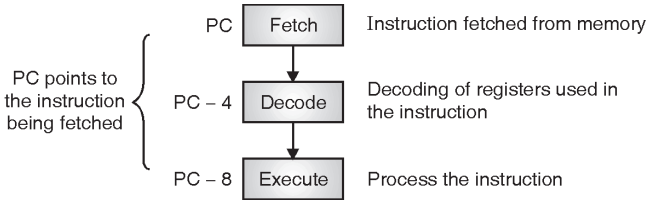


Fig. 2.8.1 : Three stage pipeline

- The three stages used in the pipeline are :

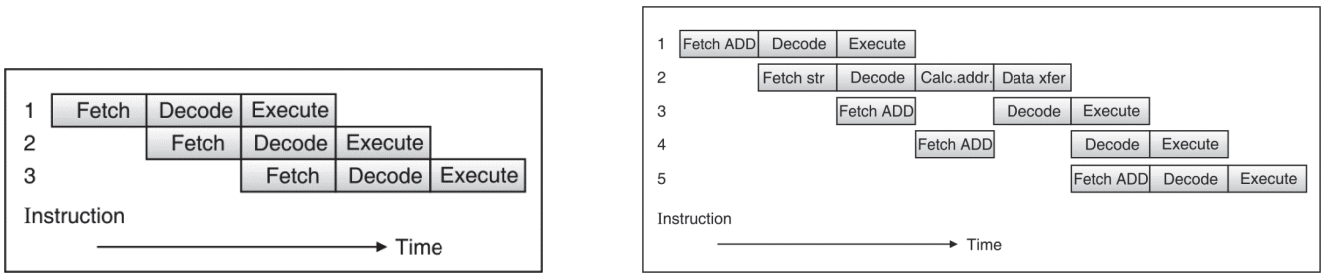
(i) **Fetch :** In this stage the ARM processor fetches the instruction from the memory.

(ii) **Decode :** In this stage recognizes the instruction that is to be executed.

(iii) **Execute :** In this stage the processor processes the instruction and writes the result back to desired register.

- If these three stages of execution are overlapped, we will achieve higher speed of execution. Such pipeline exists in version 7 of ARM processor.
- Once the pipeline is filled, each instruction requires one cycle to complete execution.

Fig. 2.8.2 shows a three stage pipelined instruction execution.



(a) Single cycle instruction execution for a 3-stage pipeline in ARM

(b) Multi cycle instruction execution for a 3-stage pipeline in ARM

Fig. 2.8.2 : Three stage pipelined instruction execution

- As shown in Fig. 2.8.2(a) in first cycle, the processor fetches instruction 1 from the memory. In the second cycle the processor fetches instruction 2 from the memory and decodes instruction 1. In the third cycle the processor fetches instruction 3 from memory, decodes instruction 2 and executes instruction 1.
- In the fourth cycle the processor fetches instruction 4, decodes instruction 3 and executes instruction 2. The pipeline thus executes an instruction in three cycles i.e. it delivers a throughput equal to one instruction per cycle.

- In case of a multi-cycle instruction as shown in Fig. 2.8.2(b), instruction 2 (i.e. STR of the store instruction) requires 4 clock cycles and hence the pipeline stalls for one clock pulse. The first instruction completes execution in the third clock pulse, while the second instruction instead of completing execution in fourth clock pulse completes the same in fifth clock pulse. Thereafter every instruction completes execution in one clock pulse as seen in the figure.
- The amount of work done at each stage can be reduced by increasing the number of stages in the pipeline. To improve the performance, the processor then can be operated at higher operating frequency.
- As more number of cycles are required to fill the pipeline, the system latency also increases. The data dependency between the stages can also be increased as the stages of pipeline increase. So the instructions need to be schedule while writing code to decrease data dependency.

The organization of an ARM processor with three stage pipeline consists of the following:

1. Register bank: This includes various registers as seen in the programmers model of ARM.
2. Barrel Shifter: We have also seen this barrel shifter that is used to do various operations as discussed earlier in this chapter.
3. The ALU: This unit performs the various arithmetic and logical operations.
4. Address register and incrementer: The register stores the address and the incrementer increments the same so as to point to the next instruction.
5. Data register: It is used as a buffer to store the data when written to the memory or read from the memory.
6. Instruction Decoder: As the name says, it decodes the instruction and issues the control signals accordingly. Hence, the Instruction decoder is associated with the control logic that issues the control signals.

- Fig. 2.8.3 shows the implementation of the three stage pipelining in ARM. In the figure, the flow of the instruction shows the three stages of pipelining. The instruction is first fetched by the PC (at the top) giving address to the address register and is incremented using the incrementer, so that the PC points to the next instruction. The instruction is fetched through the data bus (at the bottom) and is given to the Instruction Decode and Control unit. This unit decodes the instruction, which is the second stage of the pipeline and then the instruction is executed by the ALU, multiplier and barrel shifter using the registers from the register bank.

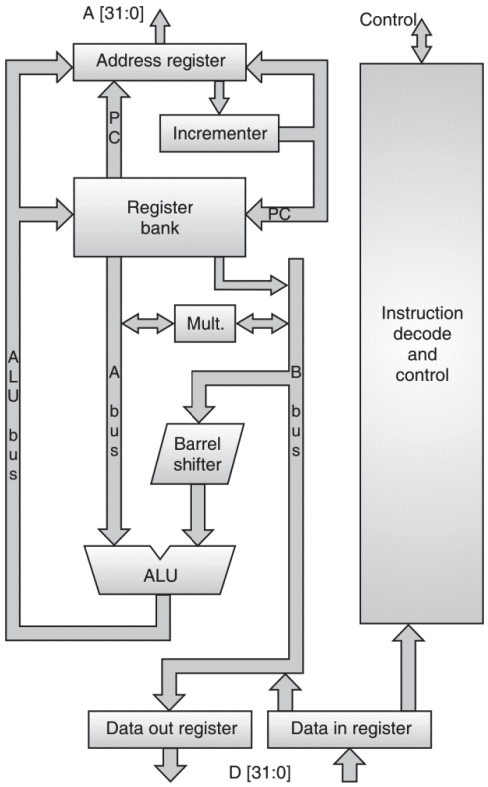


Fig. 2.8.3 : Organization of 3-stage pipelining in ARM

ARM9 pipeline

- The pipeline for ARM9 processor is a five stage pipeline as shown in Fig. 2.8.4.
- The five stages of pipeline are fetch, decode and register read, execute shift and ALU, memory access and multiply, write register.

- The instruction throughput increased by 13% as compared to ARM7 three stage pipeline.
- The core frequency of ARM9 is greater than the ARM7 processor.
- In case of 5-stage pipelining, ARM processor also uses data forwarding technology. This enables the data to be forwarded internally from one stage to next and hence reduce the chances of pipeline stalling.

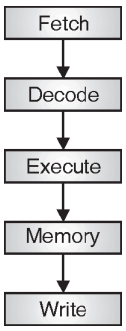


Fig. 2.8.4 : Five stage pipeline to be forwarded internally from one stage to next and of ARM9 TDMI processor

- Fig 2.8.5 shows the organization of the 5-stage pipeline organization in ARM9TDMI. In this case the fetching, decoding and execution are done in the same manner as in the 3-stage pipeline, by the respective blocks as seen in the Fig 2.8.5
- The "I-cache" as shown in the figure is instruction cache and provides the instruction to the decode unit and hence is involved in the fetch operation as shown in the figure. The "I decode" decodes the instructions and forwards to the execute stage after allocating the register read operation if any.

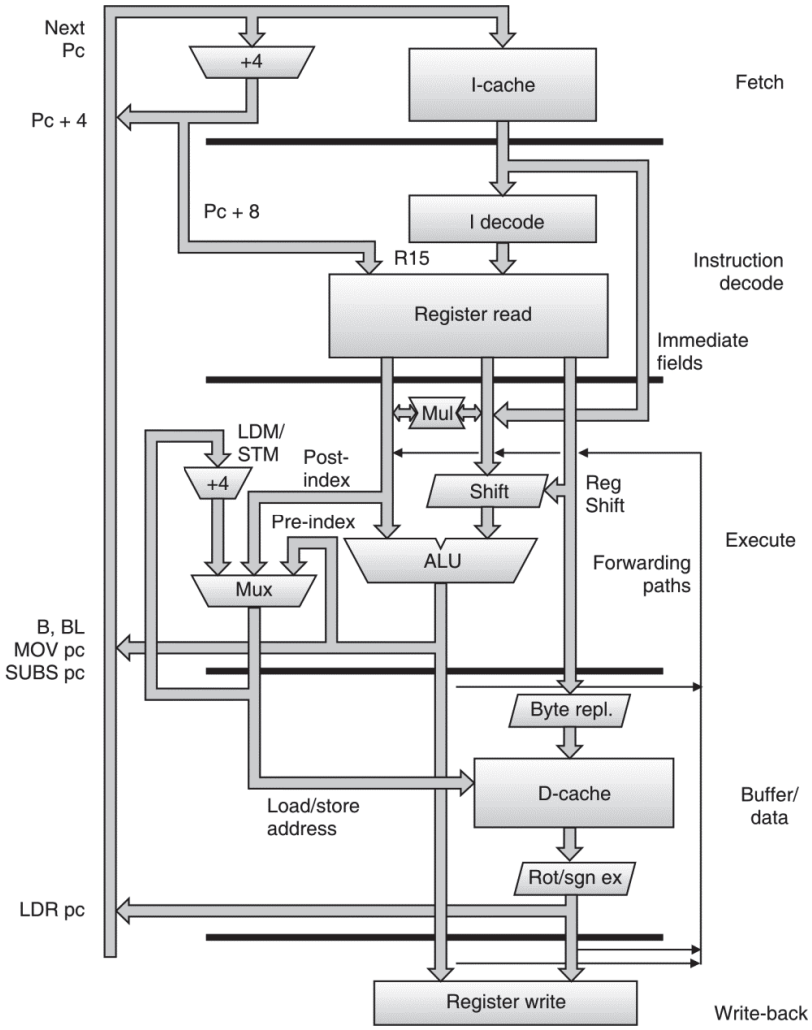


Fig. 2.8.5 : Organization of 5-stage pipeline in ARM

- In the stage three i.e. the execute stage, the ALU performs the operation to be carried out according to the instruction.
- The stage four i.e. the Memory access is used to perform memory read or write in case of the load or store instructions respectively. This is done with the help of the "D-Cache" i.e. Data cache.
- The stage five i.e. the Write stage, is used to perform the write operation i.e. to update the register for the result to be written.

ARM10 pipeline

- The pipeline of ARM10 processor has six stages fetch, issue, decode and read register, execute shift and ALU, memory access and multiply, write register as shown in Fig. 2.8.6.
- The instruction throughput increases by around 34% more than ARM7 processor. The system latency also increases.

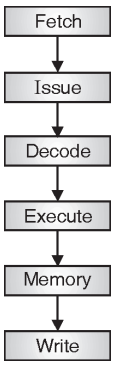


Fig. 2.8.6 : Six stage pipeline of ARM10 processor

2.9 Pipeline Hazards

- Pipelining increases processor performance by increasing instruction throughput, because several instructions are overlapped in the pipeline, cycle time can be reduced, increasing the rate at which instructions execute.
- Instruction Hazards (dependencies) occur when instructions read or write registers that are used by other instructions. The type of conflicts are divided into three categories :

(1) Structural hazards (resource conflicts)

(2) Data hazards (Data dependency conflicts)

(3) Branch difficulties (Control hazards)

(1) Structural Hazards (Resource conflicts)

- These hazards are caused by access to memory by two instructions at the same time. These conflicts can be slightly resolved by using separate instruction and data memories.
- Structural hazards occur when the processor's hardware is not capable of executing all the instructions in the pipeline simultaneously.
- Structural hazards within a single pipeline are rare on modern processors because the Instruction Set architecture is designed to support pipelining.

(2) Data Hazards (Data dependency)

- This hazard arises when an instruction depends on the result of a previous instruction, but this result is not yet available.
- These are divided into four categories :

(i) RAR – Hazard (Read after read Hazard)

(ii) RAW – Hazard (Read after write Hazard)

(iii) WAW – Hazard (Write after write Hazard)

(iv) WAR – Hazard (Write after read Hazard)

(i) RAR Hazard

RAR Hazard occurs when two instructions both read from the same register. This hazard does not cause a problem for the processor because reading a register does not change the register's value. Therefore, two instructions that have RAR Hazard can execute on successive cycles.

Example 1 : Instructions having RAR Hazard.

ADD r_1, r_2, r_3

SUB r_4, r_2, r_3 Both Instructions read r_3, creating RAR

(ii) RAW Hazard

This hazard occurs when an instruction reads a register that was written by a previous instruction. These are also called as **data dependencies** (or) **true dependencies**.

Example 2 : Instructions having RAW - Hazard.

ADD r_1, r_2, r_3

Subtract reads the output of the addition creating RAW hazard

SUB r_4, r_5, r_1

(iii) WAR and WAW are also called as name dependencies.

These hazards occur when the output register of an instruction has been either read or written by a previous instruction.

If the processor executes instructions in the order that they appear in the program and uses the same pipeline for all instructions, WAR and WAW hazards do not cause any problem in execution process.

Example 3 : Instruction having WAR Hazard.

ADD r_1, r_2, r_3

SUB r_2, r_5, r_6 WAR Hazard

Example 4 : Instructions having WAW Hazard

ADD r_1, r_2, r_3

SUB r_1, r_5, r_6

WAW Hazard

(3) Branch Hazards

- Branch instructions, particularly conditional branch instructions, create data dependencies between the branch instruction and the previous instruction, fetch stage of the pipeline.
- Since the branch instruction computes the address of the next instruction that the instruction fetch stage should fetch from, it consumes some time and also some time is required to flush the pipeline and fetch instructions from target location. This time wasted is called as branch penalty.

2.9.1 Methods to Resolve the Data Hazards and Advances in Pipelining

The methods used to resolve the data hazards are discussed in the following sub sections.

2.9.1(A) Pipeline Stalls

- The hardware inserts a special instruction called (NOP) i.e. no operation instruction known as a bubble into the flow of execution stage of pipeline to resolve the RAW hazard between two instructions.
- This method is also called as hardware interlocks.
- This approach detects the hazard and maintains the program sequence by introducing delays to resolve the data hazards (RAW).

2.9.1(B) Data Forwarding: An Hardware Solution

- This technique uses a special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline stages.
- Example of a RAW dependency exists between two instructions, instead of transferring an LAU result into a destination register, the hardware checks the destination operand, and if it is needed as a source in the next instructions, it passes the result directly into the LAU input; bypassing the register file.
- This method requires additional hardware paths through MUX (Multiplexers).
- In this case there is a multiplexer between the two stages, wherein the data required by the next instruction is forwarded.
- Let us see this with a program example.
- If we take the following sequence of instructions,

ADD r_1, r_2, r_3

SUB r_4, r_1, r_3

- We will notice that the destination of instruction 1 is the source for instruction 2. In this case the LAU stage or the execution stage of the pipeline will forward the data to the next instruction as shown in the Fig. 2.9.1. The Fig. 2.9.1 assumes that the system is 6 stage pipelined system.
- As shown in Fig. 2.9.1 the data i.e. the value of register r1 is passed from the first instruction to the second instruction. Actually the value of the register r1 is updated by the write operand stage of the first instruction. But, before that the same is required by the execute stage of second instruction. Hence the value of this register is passed to the second instruction.

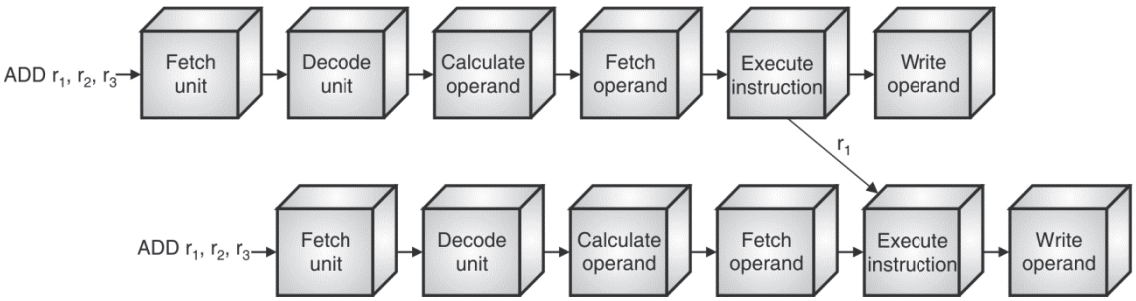


Fig. 2.9.1 : Data or Operand forwarding mechanism in pipelining for resolving a data hazard

2.9.1(C) Dynamic Instruction Scheduling (or) Out-Of-Order (OOO) Execution

- This is another interesting and very widely used technique because of the speed up given by it. It is used in Pentium IV processor.
- Here the execution of the instructions of a program is done out-of-order i.e. not in the sequence as the instructions were written by the programmer. As and when the resources of an instruction are available, the execution of that instruction is done. If, for an instruction the resources are not available, it is kept in waiting state and the further instructions whose resources are available will be executed.
- But, you would think that this approach will have a problem. The logic implemented by the programmer will not be followed properly i.e. wrong sequence of instructions will be executed. The answer to this is that, although the instructions are executed out-of-order, but the write-back is done in order, and hence the final result of the program is in sequence.
- The compiler is designed in such a way that, while translating from high-level language to machine language program, it detects the data dependencies and re-orders the instructions.
- If necessary to delay the loading of the conflicting data it inserts no-operation instruction (NOP).

2.9.2 Handling of Branch Instructions to Resolve Control Hazards

The methods used to resolve the control hazards are discussed in the following sub sections.

2.9.2(A) Pre-Fetch Target Instruction

- One way of handling a conditional branch is to prefetch the target instruction in addition to the instruction following the branch. Both are saved until the branch is executed.
- If the branch condition is successful, the pipeline continues from the branch target instructions else sequential instructions are executed.

2.9.2(B) Branch Target Buffer (BTB)

- The BTB is an associative memory included in the fetch segment of the pipeline.
- Each entry in BTB consists of the address of a previously executed branch instructions and the target instruction for that branch. It also stores the next few instructions after the branch target instructions. When the pipeline decodes a branch instruction, it searches the associative memory BTB for the address of the instruction.
- If it is in the BTB, the instruction is available directly and prefetch continues from the new path.
- If the instruction is not in BTB, pipeline shifts to a new instruction streams and stores the target instruction.

2.9.2(C) Loop Buffer

- This is like a BTB, but is a high speed register file maintained by the instruction fetch segment of the pipeline. When a program loop is detected in the program, it is stored in the loop buffer.
- The program loop can be executed directly without having to access memory until loop mode is removed by final branching out.

2.9.2(D) Branch Prediction

- A pipeline with branch prediction uses additional logic to speculate the outcome of a conditional branch instruction before it is executed.
- The pipeline then begins pre-fetching the instructions stream from the predicted path.
- A correct prediction eliminates the wasted time caused by branch penalties.
- A detailed operation of branch prediction logic will be discussed in the later sections of this chapter.

2.9.2(E) Pipeline Stall (Delayed Branch)

GTU - S-18,W-19

Q.	Explain concept of delayed branch. Is this concept adopted in ARM processor?	**(S-18, 3 Marks)**
Q.	Define delayed branch.	**(W-19, 3 Marks)**

Complier detects branch instruction and rearranges the machine language code sequence by inserting useful instructions and rearranges the code sequence to reduce the delays incurred by Branch Instruction.

2.9.2(F) Loop Unrolling Technique

- This is a very superb solution to handle the stalls due to branching in loops.
- In this case a code which has a loop that has to be executed multiple times, will be actually stored multiple times (or unrolled) so as to remove the need of branching.
- The **advantage** clearly seen of this method is that there is no scope for pipeline stall and hence the performance will increase.
- The major **disadvantage** of this method is that the memory required will be more as the loop has to be unrolled and stored in memory. In our example the loop was to be repeated for only 5 times, but if the loop was larger and had to be repeated for say 100 or 1000 times, the memory consumed would be very huge.

2.9.2(G) Software Scheduling or Software Pipelining

- In case of software pipelining, the iterations of a loop of the source program are continuously initiated at regular intervals, before the earlier iterations complete. Thus taking advantage of the parallelism in data path.
- It can be said that software scheduling, schedules the operations within a loop, such that an iteration of the loop can be pipelined to yield optimal performance.

2.10 Stack Implementation in ARM

- Stack implementation in ARM processor is done using the stack pointer R13.
- It stores the top of the stack in the current processor mode. For each of the 6 modes of operation of ARM processor, the R13 i.e. the stack pointer is a physically different register i.e. it is one of the banked register.

2.11 Endianess

GTU - S-18, W-18, S-19

Q. Explain difference between : Little-Endian & Big-Endian. **(S-18, W-18, S-19, 2 Marks)**

- When dealing with multibyte numbers, there is an issue regarding how the number is stored in memory. If the least significant byte of the number is stored at the lowest address, then the byte order is called little-endian. Otherwise, the byte order is called big-endian
- There are two kinds of memory organizations based on how the data is stored in memory. They are called as:
  (i) Little Endian memory organization and
  (ii) Big Endian memory organization.
- These memory organizations indicate the way the data will be stored in the memory, especially for multi-word data i.e. the data that cannot fit in a single location. For example to store a 16-bit data in a memory with each location of a byte (i.e. 8-bit), we will need to store the data in two consecutive memory locations. If the lower byte is stored in the first location and the higher byte in the consecutive next location it is called as little Endian memory organization.
- If the higher byte is stored in the first location and the lower byte in the consecutive next location then it is called as big Endian memory organization.

2.12 Condition Codes

- We know that an ARM instruction is 32 bit long. The 4 MSBs of the instructions are called as the **condition field**. The condition field is of 4 bits, hence there are 16 conditions as listed in Table 2.12.1.
- Fig. 2.12.1 shows the condition field.

Fig. 2.12.1 : ARM condition field

Table 2.12.1

Condition field	{cond}	
Suffix	**Description**	**Condition flag state**
EQ	Equal	Z = 1
NE	Not Equal	Z = 0
CS	Unsigned higher or same	C = 1
CC	Unsigned lower	C = 0
MI	Negative	N = 1
PL	Positive 0 or zero	N = 0
VS	Overflow	V = 1
VC	No overflow	V = 0
HI	Unsigned higher	C = 0 and Z = 0
LS	Unsigned lower or same	C =1 and Z = 1
GE	Signed greater than or equal	N = 1 and V = 1 or N = 0 and V = 0.
LT	Signed less than	N = 1 and V = 0 or N = 0 and V = 1.
GT	Signed greater than	Z = 0 and either N = 1 and V = 1 or N = 0 and V= 1.
LE	Signed less than or equal	Z = 1 or N = 1 and V = 0 or N = 0 and V = 1.
AL	Always (unconditional)	
NV	Reserved	

2.13 Processor Core Vs. CPU Core

- Processor is the entire chipset of all the hardware included in the processor, including all the CPU cores.
- A CPU is the main core of the processor that can execute an instruction.
- Multiple CPU cores will execute multiple instructions simultaneously. Hence, each core executing an instruction simultaneously, results in parallel instruction execution.
- This increases the processor's overall speed of executing a task.
- Most of the modern processors have multiple CPU core i.e. 2, 3, 4, 5, 6, 8 or more cores.

2.14 ARM7TDMI

- The LPC1768/66/65/64 are ARM Cortex-M3 based microcontrollers for embedded applications featuring a high level of integration and low power consumption. The ARM Cortex-M3 is a next generation core that offers system enhancements such as enhanced debug features and a higher level of support block integration.
- The LPC1768/66/65/64 operate at CPU frequencies of up to 100 MHz. The ARM Cortex-M3 CPU incorporates a 3-stage pipeline and uses a Harvard architecture with separate local instruction and data buses as well as a third bus for peripherals.

- The ARM Cortex-M3 CPU also includes an internal prefetch unit that supports speculative branching.
- The peripheral complement of the LPC1768/66/65/64 includes up to 512 KB of flash memory, up to 64 KB of data memory Ethernet MAC, USB Device/Host/OTG interface, 8-channel general purpose DMA controller, 4 UARTs, 2 CAN channels, 2 SSP controllers, SPI interface, 3 I2C-bus interfaces, 2-input plus 2-output I2S-bus interface, 8-channel 12-bit ADC, 10-bit DAC, motor control PWM, Quadrature Encoder interface, 4 general purpose timers, 6-output general purpose PWM, ultra-low power Real-Time Clock (RTC) with separate battery supply, and up to 70 general purpose I/O pins.
- The LPC1768/66/65/64 are pin-compatible to the 100-pin LPC236x ARM7-based microcontroller series.

2.14.1 Features of ARM7TDMI

- ARM Cortex-M3 processor, running at frequencies of up to 100 MHz. A Memory Protection Unit (MPU) supporting eight regions is included.
- ARM Cortex-M3 built-in Nested Vectored Interrupt Controller (NVIC).
- Up to 512 KB on-chip flash programming memory. Enhanced flash memory accelerator enables high-speed 100 MHz operation with zero wait states.
- In-System Programming (ISP) and In-Application Programming (IAP) via on-chip boot loader software.
- On-chip SRAM includes :32/16 kB of SRAM on the CPU with local code/data bus for high-performance CPU access.
- Two/one 16 kB SRAM blocks with separate access paths for higher throughput. These SRAM blocks may be used for Ethernet (LPC1768/66/64 only), USB, and DMA memory, as well as for general purpose CPU instruction and data storage.
- Eight channel General Purpose DMA controller (GPDMA) on the AHB multilayer matrix that can be used with the SSP, I2S-bus, UART, the Analog-to-Digital and Digital-to-Analog converter peripherals, timer match signals, and for memory-to-memory transfers.
- Multilayer AHB matrix interconnect provides a separate bus for each AHB master. AHB masters include the CPU, General Purpose DMA controller, Ethernet MAC (LPC1768/66/64 only), and the USB interface. This interconnect provides communication with no arbitration delays.
- Split APB bus allows high throughput with few stalls between the CPU and DMA. Serial interfaces :
- Ethernet MAC with RMII interface and dedicated DMA controller (LPC1768/66/64 only).
- USB 2.0 full-speed device/Host/OTG controller with dedicated DMA controller and on-chip PHY for device, Host, and OTG functions. The LPC1764 includes a device controller only.
- Four UARTs with fractional baud rate generation, internal FIFO, DMA support, and RS-485 support. One UART has modem control I/O, and one UART has IrDA support.

  CAN 2.0B controller with two channels.
- SPI controller with synchronous, serial, full duplex communication and programmable data length.
- Two SSP controllers with FIFO and multi-protocol capabilities. The SSP interfaces can be used with the GPDMA controller.
- Two I2C-bus interfaces supporting fast mode with a data rate of 400 kbits/s with multiple address recognition and monitor mode.
- One I2C-bus interface supporting full I2C-bus specification and fast mode plus with a data rate of 1 Mbit/s with multiple address recognition and monitor mode.

- On the LPC1768/66/65 only, I2S (Inter-IC Sound) interface for digital audio input or output, with fractional rate control. The I2S-bus interface can be used with the GPDMA. The I2S-bus interface supports 3-wire and 4-wire data transmit and receive as well as master clock input/output.

Other peripherals

- 70 General Purpose I/O (GPIO) pins with configurable pull-up/down resistors and a new, configurable open-drain operating mode. 12-bit Analog-to-Digital Converter (ADC) with input multiplexing among eight pins, conversion rates up to 1 MHz, and multiple result registers. The 12-bit ADC can be used with the GPDMA controller.
- 10-bit Digital-to-Analog Converter (DAC) with dedicated conversion timer and DMA support (LPC1768/66/65 only). Four general purpose timers/counters, with a total of eight capture inputs and ten compare outputs. Each timer block has an external count input and DMA support.
- One motor control PWM with support for three-phase motor control. Quadrature encoder interface that can monitor one external quadrature encoder. One standard PWM/timer block with external count input
- RTC with a separate power domain and dedicated RTC oscillator. The RTC block includes 64 bytes of battery-powered backup registers.
- Watchdog Timer (WDT) resets the microcontroller within a reasonable amount of time if it enters an erroneous state.
    - System tick timer, including an external clock input option.
    - Repetitive interrupt timer provides programmable and repeating timed interrupts.
    - Each peripheral has its own clock divider for further power savings.
- Standard JTAG test/debug interface for compatibility with existing tools. Serial Wire Debug and Serial Wire Trace Port options.
- Emulation trace module enables non-intrusive, high-speed real-time tracing of instruction execution.
- Integrated PMU (Power Management Unit) automatically adjusts internal regulators to minimize power consumption during Sleep, Deep sleep, Power-down, and Deep power-down modes.
- Four reduced power modes : Sleep, Deep-sleep, Power-down, and Deep power-down.
- Single 3.3 V power supply (2.4 V to 3.6 V).
- Four external interrupt inputs configurable as edge/level sensitive. All pins on PORTO and PORT2 can be used as edge sensitive interrupt sources.
- Non-maskable Interrupt (NMI) input.
- Clock output function that can reflect the main oscillator clock, IRC clock, RTC clock, CPU clock, and the USB clock.
- The Wakeup Interrupt Controller (WIC) allows the CPU to automatically wake up from any priority interrupt that can occur while the clocks are stopped in deep sleep, Power-clown, and Deep power-down modes.
- Processor wake-up from Power-down mode via interrupts from various peripherals.
- Brownout detect with separate threshold for interrupt and forced reset.
- Power-On Reset (POR).
- Crystal oscillator with an operating range of 1 MHz to 25 MHz.
- 4 MHz internal RC oscillator trimmed to 1 % accuracy that can optionally be used as a system clock.
- **PLL** allows CPU operation up to the maximum CPU rate without the need for a high-frequency crystal. May be run from the main oscillator, the internal RC oscillator, or the RTC oscillator.

- **USB** PLL for added flexibility.
- Code Read Protection (CRP) with different security levels.
- Available as 100-pin LQFP package (14 × 14 × 1.4 mm).

2.14.2 Block Diagram and ARM7TDMI Interface Signals

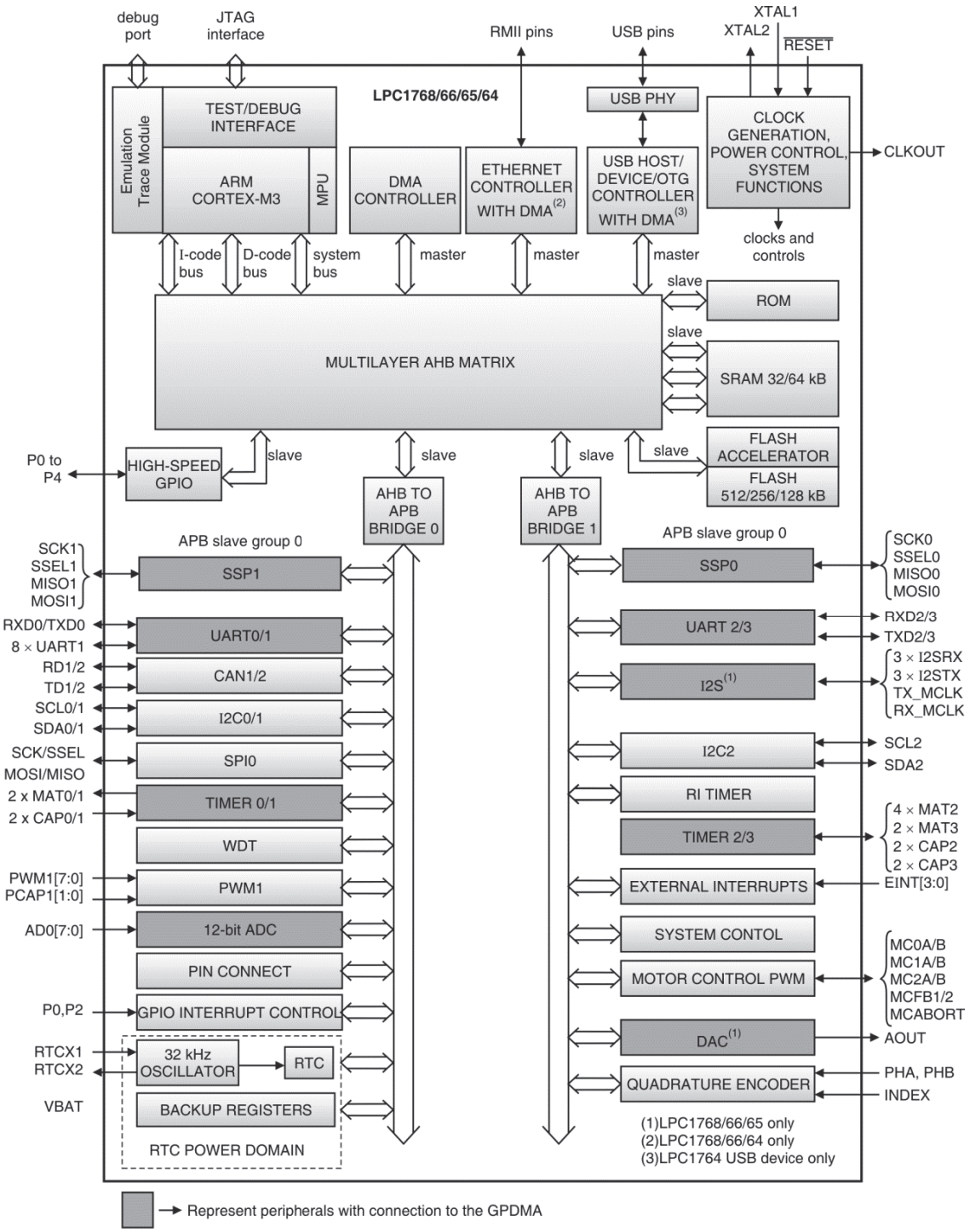


Fig. 2.14.1: Block diagram and ARM7TDMI Interface Signals

Table 2.14.1 : Signal Interface description#

Symbol	Pin	Type	Description
P0[0] to P0[31]		I/O	Port 0 – Port 0 is a 32-bit I/O port with individual direction controls for each bit. The operation of port 0 pins depends upon the pin function selected via the pin connect block. Pins 12, 13, 14, and 31 of this port are not available.
P0[0]/RD1/TXD3/SDA1	46	I/O	P0[0] – General purpose digital input/output pin.
		I	RD1 – CAN1 receiver input.
		O	TXD3 – Transmitter output for UART3.
		I/O	SDA1 – I^2C1 data input/output (this is not an I^2C – bus compliant open – drain pin)
P0[1]/TD1/RXD3/SCL1	47	I/O	PO[1] – General purpose digital input/output pin.
		O	TD1 – CAN1 transmitter output.
		I	RXD3 – Receiver input for UART3.
		I/O	SCL1 – I2C1 clock input/output (this is not an I2C-bus compliant open-drain pin).
P0[2]/TXD0/AD0[7]	98	I/O	P0[2] – General purpose digital input/output pin.
		O	TXD0 – Transmitter output for UART0.
		I	AD0[7] – A/D converter 0, input 7.
P0[3]/RXD0/AD0[6]	99	I/O	P0[3] – General purpose digital input/output pin.
		I	RXD0 – Receiver input for UARTO.
		I	AD0[6] – A/D converter 0, input 6.
P0[4]/I2SRX_CLK/RD2/CAP2[0]	81	I/O	P0[4] – General purpose digital input/output pin.
		I/O	I2SRX_CLK – Receive Clock. It is driven by the master and received by the slave. Corresponds to the signal SCK in the I2S-bus specification. (LPC1768/66/65 only).
		I	RD2 – CAN2 receiver input.
		I	CAP2[0] – Capture input for Timer 2, channel 0.
P0[5]/I2SRX_WS/TD2/CAP2[1]	80	I/O	P0[5] – General purpose digital input/output pin.
		I/O	I2SRX_WS – Receive Word Select. It is driven by the master and received by the slave. Corresponds to the signal WS in the I^2S-bus specification. (LPC1768/66/65 only).
		O	TD2 – CAN2 transmitter output.
		I	CAP2[1] – Capture input for Timer 2, channel 1.

Courtesy NxP

Symbol	Pin	Type	Description
P0[6]/I2SRX_SDA/SSEL 1/MAT2[0]	79	I/O	P0[6] – General purpose digital input/output pin.
		I/O	I2SRX_SDA – Receive data. It is driven by the transmitter and read by the receiver. Corresponds to the signal SD in the I2S-bus specification. (LPC1768/66/65 only).
		I/O	SSEL1 – Slave Select for SSP1.
		O	MAT2[0] – Match output for Timer 2, channel 0.
P0[7]/I2STX_CLK/SCK1/MAT2[1]	78	I/O	P0[7] – General purpose digital input/output pin.
		I/O	I2STX_CLK – Transmit Clock. It is driven by the master and received by the slave. Corresponds to the signal SCK in the I2S-bus specification. (LPC1768/66/65 only)
		I/O	SCK1 – Serial Clock for SSP1.
		O	MAT2[1] – Match output for Timer 2, channel 1.
P0[8]/ I2STX_WS/ MISO1/MAT2[2]	77	I/O	P0[8] – General purpose digital input/output pin.
		I/O	I2STX_WS – Transmit Word Select. It is driven by the master and received by the slave. Corresponds to the signal WS in the I2S-*bus specification.* (LPC1768/66/65 only).
		I/O	MISO1 – Master In Slave Out for SSP1.
		O	MAT2[2] – Match output for Timer 2, channel 2.
P0[9]/I2STX_SDA/MOSI1/MAT2[3]	76	I/O	P0[9] – General purpose digital input/output pin.
		I/O	I2STX_SDA – Transmit data. It is driven by the transmitter and read by the receiver. Corresponds to the signal SD in the I2S-bus specification. (LPC1768/66/65 only).
		I/O	MOSI1 – Master Out Slave In for SSP1
		O	MAT2[3] – Match output for Timer 2, channel 3.
P0[10]/TXD2/SDA2/MAT3[0]	48	I/O	P0[10] – General purpose digital input/output pin.
		O	TXD2 – Transmitter output for UART2.
		I/O	SDA2 – I^2C2 data input/output (this is not an open-drain pin).
		O	MAT3[0] – Match output for Timer 3, channel 0.
P0[11]/RXD2/SCL2/MAT3[1]	49	I/O	P0[11] – General purpose digital input/output pin.
		I	RXD2 – Receiver input for UART2.
		I/O	SCL2 – I2C2 clock input/output (this is not an open-drain pin).

Symbol	Pin	Type	Description
		O	MAT3[1] – Match output for Timer 3, channel 1.
P0[15]/TXD1/SCK0/SCK	62	I/O	P0[15] – General purpose digital input/output pin.
		O	TXD1 – Transmitter output for UART1.
		I/O	SCKO – Serial clock for SSPO.
		I/O	SCK – Serial clock for SPI.
P0[16]/RXD1/SSEL0/SSEL	63	I/O	P0[16] – General purpose digital input/output pin.
		I	RXD – Receiver input for UART1.
		I/O	SSEL0 – Slave Select for SSPO.
		I/O	SSEL – Slave Select for SPI.
P0[17]/CTS1/MISO0/MISO	61	I/O	P0[17] – General purpose digital input/output pin.
		I	CTS1 – Clear to Send input for UART1.
		I/O	MISO0 – Master In Save Out for SSP0.
		I/O	MISO Master In Slave Out for SPI.
P0[18]/DCD1/MOSI0/MOSI	60	I/O	P0[18] – General purpose digital input/output pin.
		I	DCD1 Data Carrier Detect input for UART 1.
		I/O	M0SI0 – Master Out Slave In for SSP0.
		I/O	MOSI – Master Out Slave In for SPI.
P0[19]/DSR1/SDA1	59	I/O	P0[19] – General purpose digital input/output pin.
		I	DSR1 – Data Set Ready input for UART1
		I/O	SDA1 – I^2C1 data input/output (this is not an I^2C bus compliant open drain pin).
P0[20]/DTR1/SCL1	58	I/O	P0[20] – General purpose digital input/output pin.
		O	DTR1 – Data Terminal Ready output for UART1.
		I/O	SCL1 – I^2C1 clock input/output (this is not an I^2C-bus compliant open-drain pin).
P0[21]/RI1/RD1	57	I/O	P0[21] – General purpose digital input/output pin.
		I	RI1 – Ring Indicator input for UART1
		I	RD1 – CAN1 receiver input
P0[22]/RTS1/TD1	56	I/O	P0[22] – General purpose digital input/output pin.
		O	RTS1 – Request to Send output for UART1.
		O	TD1 – CAN1 transmitter output.

Symbol	Pin	Type	Description
P0[23]/AD0[0]/I2SRX_CLK/CAP3[0]	9	I/O	P0[23] – General purpose digital input/output pin.
		I	AD0[0] – A/D converter 0 input 0
		I/O	I2SRX_CLK – Receive Clock. It is driven by the master and received by the slave. Corresponds to the signal SCK in the I^2S-bus specification. (LPC1768/66/65 only).
		I	CAP3[0] – Capture input for Timer 3, channel 0.
P0[24]/AD0[1]/I2SRX_WS/CAP3[1]	8	I/O	P0[24] – General purpose digital input/output pin.
		I	AD0[1] – A/D converter 0, input 1
		I/O	I2RX_WS – Receive Word Select. It is driven by the master and received by the slave. Corresponds to the signal WS in the I^2S-bus specification. (LPC 1768/66/65 only).
		I	CAP3[1] – Capture input for Timer 3, channel 1.
P0[25]/AD0[2]/I2SRX_SDA/TXD3	7	I/O	P0[25] – General purpose digital input/output pin.
		I	AD0[2] – A/D converter 0, input 2
		I/O	I2SRX_SDA – Receive data. It is driven by the transmitter and read by the receiver. Corresponds to the signal SD in the I^2S-bus specification. (LPC1768/66/65 only).
		O	TXD3 – Transmitter output for UART3
P0[26]/AD0[3]/AOUT/RXD3	6	I/O	P0[26] – General purpose digital input/output pin.
		I	AD0[3] – A/D converter 0, input 3.
		O	AOUT – DAC output (LPC1768/66/65 only)
		I	RXD3 – Receiver input for UART3.
P0[27]/SDA0/USB_SDA	25	I/O	P0[27] – General purpose digital it input/output pin. Output is open-drain.
		I/O	SDA0 – I^2C0 data input/output. Open-drain output (for I2C-bus compliance).
		I/O	USB_SDA – USB port I^2C serial data (OTC transceiver, (LPC1768/66/65 only).
P0[28]/SCL0/USB SCL	24	I/O	P0[28] – General purpose digital input/output pin. Output is open-drain.
		I/O	SCL0 I^2C0 clock input/output. Open drain output (for I^2C bus compliance).
		I/O	USB_SCL – USB port I^2C serial clock (OTG transceiver, LPC1768/68/66/65 only).

Symbol	Pin	Type	Description
P0[29]/USB_D+	29	I/O	P0[29] – General purpose digital input/output pin.
		I/O	USB_D+ – USB bidirectional D– line.
P0[30]/USB_D–	30	I/O	P0[30] – General purpose digital input/output pin.
		I/O	USD_D - USD bidirectional D-line.
P1[0] to P1[31]		I/O	Port 1 : Port 1 is a 32-bit I/O port with individual direction controls for each bit. The operation of port 1 pins depends upon the pin function selected via the pin connect block Pins 2, 3, 5, 6, 7, 11, 12 and 13 of this port are not available.
P1[0]/ENET_TXD0	95	I/O	P1[0] – General purpose digital input/output pin.
		O	ENET_TXD0 – Ethernet transmit data 0. (LPC1768/66/64 only).
P1[1]/ENET_TXD1	94	I/O	P1[1] – General purpose digital input/output pin.
		O	ENET_TXD1 – Ethernet transmit data 1 (LPC1768/66/64 only).
P1 [4]/ENET_TX_EN	93	I/O	P1[4] – General purpose digital input/output pin.
		O	ENET_TX_EN – Ethernet transmit data enable. (LPC1768/66/64 only).
P1[8]/ENET_CRS	92	I/O	P1[8] – General purpose digital input/output pin.
		I	ENET_CRS – Ethernet carrier sense. (LPC1768/66/64 only).
P1[9]/ENET_RXD0	91	I/O	P1[9] – General purpose digital input/output pin.
		I	ENET_RXD0 – Ethernet receive data. (LPC1768/66/64 only).
P1[10]/ENET_RXD1	90	I/O	P1[10] – General purpose digital input/output pin.
		I	ENET_RXD1 – Ethernet receive data. (LPC1768/66/64 only).
P1[14]/ENE1_RX_ER	89	I/O	P1[14] – General purpose digital input/output pin.
		I	ENET_RX_ER – Ethernet receive error. (LPC1768/66/64 only.)
P1[15]/ENET_REF_CLK	88	I/O	P1[15] – General purpose digital input/output pin.
		I	ENET_REF_CLK – Ethernet reference clock. (LPC1768/66/64 only).
P1[16]/ENET_MDC	87	I/O	P1[16] – General purpose digital input/output pin.
		O	ENET_MDC – Ethernet MIIM clock (LPC1768/66/64. only).

Symbol	Pin	Type	Description
P1[17]/ENET_MDIO	86	I/O	P1[17] — General purpose digital input/output pin.
		I/O	ENET_MDIO — Ethernet MIIM data input and output. (LPC1/68/66/64 only).
P1[18]/USB UP LED/PWM1[1]/CAP1[0]	32	I/O	P1[18] – General purpose digital input/output pin.
		O	USB_UP_LED USB GoodLink LED indicator. It is LOW when devices is configured (non-control endpoints enabled). It is HIGH when the device is not configured or during global suspend.
		O	PWM1[1] – Pulse width modulator 1, channel 1 output.
		I	CAP1[0] – Capture input for Timer 1, channel 0
P1[19]/MC0A/ $\overline{\text{USB}}$ –PPWR/CAP1[1]	33	I/O	P1[19] – general purpose digital input/output pin.
		O	MC0A – Motor control PWM channel 0, output A.
		O	$\overline{\text{USB–PPWR}}$ – port power enable signal for USB port (LPC 1768/66/65 only).
		I	CAP1[1] – Capture input for timer 1, channel 1.
P1[20]/MCFB0/PWM1[2]/SCK0	34	I/O	P1[20] – general purpose digital input/output pin.
		I	MCFB0 – Motor control PWM channel 0, feedback input. Also Quadrature Encoder interface PHA input.
		O	PWM1[2] – Pulse width Modulator 1, channel 2 output.
		I/O	SCK0 – Serial clock for SSP0.
P1[21]/MCABORT/PWM1[3]/SSEL0	35	I/O	P1[21] – General purpose digital input/output pin.
		O	MCABORT – Motor control PWM, emergency abort.
		O	PWM1[3] – Pulse width Modulator 1, channel 3 output.
		I/O	SSEL0 – Slave Select for SSP0.
P1[22]/MC0B/USB_PWRD/MAT1[0]	36	I/O	P1[22] – General purpose digital input/output pin.
		O	MC0B – Motor control PWM channel 0, output 0.
		I	USB_PWRD – Power Status for USB port (host power switch LPC1768/66/65 only).
		O	MAT1[0] – Match output for Timer 1, channel 0.
P1[23]/MCFB1/PWM1[4]/MISO0	37	I/O	P1[23] – General purpose digital input/output pin.
		I	MCFB1 – Motor control PWM channel 1, feedback input. Also Quadrature Encoder Interface PHB input.
		O	PWM1[4] – Pulse Width Modulator 1, channel 4 output.
		I/O	MISO0 – Master in Slave Out for SSP0.

Courtesy NxP

Symbol	Pin	Type	Description
P1[24]/MCFB2/PWM1[5]/MOSI0	38	I/O	P1[24] – General purpose digital input/output pin.
		I	MCFB2 – Motor Control PWM channel 2, feedback input. Also Quadrature Encoder Interface INDEX input.
		O	PWM1[5] – Pulse Width Modulator 1, channel 5 output.
		I/O	MOSI0 – Master Out Slave in for SSP0.
P1[25]/MC1A/MAT1[1]	39	I/O	P1[25] – General purpose digital input/output pin.
		O	MC1A – Motor control PWM channel 1, output A.
		O	MAT1[1] – Match output for Timer 1, channel 1.
P1[26]/MC1B/PWM1[6]/CAP0[0]	40	I/O	P1[26] – General purpose digital input/output pin.
		O	MC1B – Motor Control PWM channel 1, output B.
		O	PWM1[6] – Pulse Width Modulator 1, Channel 6 output.
		I	CAP0[0] – Capture input for Timer 0, channel 0.
P1[27]/CLKOUT/ $\overline{\text{USB_OVRCR}}$ / CAP0[1]	43	I/O	P1[27] – General purpose digital input/output pin.
		O	CLKOUT – Clock output pin.
		I	$\overline{\text{USB_OVRCR}}$ – USB port Over-Current status (LPC 1768/66/65 only).
		I	CAP0[1] – Capture input for Timer 0, channel 1.
P1[28]/MC2A/1[0]/MAT0[0]	44	I/O	P1[28] – General purpose digital input/output pin.
		O	MC2A – Motor control PWM channel 2, output A.
		I	PCAP1[0] – Capture input for PWM1, channel 0.
		0	MAT0[0] – Match output for Timer 0, channel 0.
P1[29]/MC2B/PCAP1[1]/MAT0[1]	45	I/O	P1[29] – General purpose digital input/output pin.
		O	MC2B – Motor control PWM channel 2, output B.
		I	PCAP1[1] – Capture input for PWM1, channel 1.
		O	MAT0[1] – Match output for Timer 0, channel 0.
P1[30]/V_{BUS}/AD0[4]	21	I/O	P1[30] – General purpose digital input/output pin.
		I	V_{BUS} – Monitors the presence of USB bus power. Note : This signal must be HIGH for USB reset to occur.
		I	AD0[4] – A/D converter 0, input 4.
P1[31]/SCK1/AD0[5]	20	I/O	P1[31] – General purpose digital input/output pin.
		I/O	SCK1 – Serial Clock for SSP1.
		I	AD0[5] – A/D converter 0, input 5.

Courtesy NxP

Symbol	Pin	Type	Description
P2[0] to P2[31]		I/O	Port 2 : Port 2 is a 32-bit I/O port with individual direction controls for each bit. The operation of port 2 pins depends upon the pin function selected via the pin connect block. Pins 14 through 31 of this port are not available.
P2[0]/PWM1[1]/TXD1	75	I/O	P2[0] – General purpose digital input/output pin.
		O	PWM1[1] – Pulse Width Modulator 1, channel 1 output.
		O	TXD1 – Transmitter output for UART1.
P2[1]/PWM1[2]/RXD1	74	I/O	P2[1] – General purpose digital input/output pin.
		O	PWM1[2] – Pulse Width Modulator 1, channel 2 output.
		I	RXD1 – Receiver input for UART1.
P2[2]/PWM1[3]/CTS1/ TRACEDATA[3]	73	I/O	P2[1] – General purpose digital input/output pin.
		O	PWM1[3] – Pulse Width Modulator 1, channel 3 output.
		I	CTS1 – Clear to Send input for UART1.
		O	TRACEDATA[3] – Trace data, bit 3.
P2[3]/PWM1[4]/DCD1/ TRACEDATA[2]	70	I/O	P2[3] – General purpose digital input/output pin.
		O	PWM1[4] – Pulse Width Modulator 1, channel 4 output.
		I	DCD1 – Data Carrier Detect input for UART1.
		O	TRACEDATA[2] – Trace data, bit 2.
P2[4]/PWM1[5]/DSR1/ TRACEDATA[1]	69	I/O	P2[4] – General purpose digital input/output pin.
		O	PWM1[5] – Pulse Width Modulator 1, channel 5 output.
		I	DSR1 – Data Set Ready input for UART1.
		O	TRACEDATA[1] – Trace data, bit 1.
P2[5]/PWM1[6]/DTR1/ TRACEDATA[0]	68	I/O	P2[5] – General purpose digital input/output pin.
		O	PWM1[6] – Pulse Width Modulator 1, channel 6 output.
		O	DTR1 – Data Terminal Ready output for UART1.
		O	TRACEDATA[0] – Trace data, bit 0.
P2[6]/PCAP1[0]/RI1/TRACECLK	67	I/O	P2[6] – General purpose digital input/output pin.
		I	PCAP1[0] – Capture input for PWM1, channel 0.
		I	RI1 – Ring indicator input for UART1.
		O	TRACECLK – Trace Clock.

Courtesy NxP

Symbol	Pin	Type	Description
P2[7]/RD2/RTS1	66	I/O	P2[7] – General purpose digital input/output pin.
		I	RD2 – CAN2 receiver input.
		O	RTS1 – Request to Send output for UART1.
P2[8]/TD2/TXD2	65	I/O	P2[8] – General purpose digital input/output pin.
		O	TD2 – CAN2 transmitter output
		O	TXD2 – Transmitter output for UART2.
P2[9]/USB_CONNECT/RXD2	64	I/O	P2[9] – General purpose digital input/output pin.
		O	USB_CONNECT – Signal used to switch an external 1.5 Ω resistor under software control. Used with the Softconnect USB feature.
		I	RXD2 – Receiver input for UART2
P2[10]/$\overline{\text{EINT0}}$ /NMI	53	I/O	P2[10] – General purpose digital input/output pin. Note : LOW on this pin while $\overline{\text{RESET}}$ is LOW forces on chip boot loader to take over control of the part after a reset.
		I	$\overline{\text{EINT0}}$ – External interrupt 0 input.
		I	NMI – Non maskable interrupt 0 input.
P2[11]/$\overline{\text{EINT1}}$ /I2STX_CLK	52	I/O	P2[11] – General purpose digital input/output pin.
		I	$\overline{\text{EINT1}}$ – External interrupt 1 input.
		I/O	I2STX_CLK – Transmit Clock it is driven by the master and received by the slave. Corresponds to the signal SCK in the I2S-bus specification. (LPC1768/66/65 only)
P2[12]/$\overline{\text{EINT2}}$ /I2STX_WS	51	I/O	P2[12] – General purpose digital input/output pin.
		I	$\overline{\text{EINT2}}$ – External interrupt 2 input.
		I/O	I2STX_WS – Transmit word Select. It is driven by the master and received by the slave. Corresponds to the signal WS in the I^2S-bus specification. (LPC1768/66/65 only)
P2[13]/$\overline{\text{EINT3}}$ /I2STX_SDA	50	I/O	P2[13] – General purpose digital input/output pin.
		I	$\overline{\text{EINT3}}$ – External interrupt 3 input.
		I/O	I2STX_SDA – Transmit data. It is driven by the transmitter and ready by the receiver. Corresponds to the single SD in the I^2S-bus specification. (LPC1768/66/65 only)

Symbol	Pin	Type	Description
P3[0]/to P3[31]		I/O	Port 3 : Port 3 is a 32 bit I/O port with individual direction controls for each bit. The operation of port 3 pins depends upon the pin function selected via the pin connect block pins 0 through 24 and 27 through 31 of this port are not available.
P3[25]/MAT0[0]/PWM1[2]	27	I/O	P3[25] – General purpose digital input/output pin.
		O	MAT0[0] – Match output for Timer 0, channel 0.
		O	PWM1[2] – Pulse Width Modulator 1, output 2.
P3[26]/STCLK/MAT0[1]/PWM1[3]	26	I/O	P3[26] – General purpose digital input/output pin.
		I	STCLK – system tick timer clock input.
		O	MAT0[1] – Match output for Timer 0, channel 1.
		O	PWM1[3] – Pulse Width Modulator 1, output 3.
P4[0]to P4[31]		I/O	Port 4 : Port 4 is a 32-bit I/O port with individual direction controls for each bit. The operation of port 4 pins depends upon the pin function selected via the pin connect block. Pins 0 through 27, 30 and 31 through 31 of this port are not available.
P4[28]/RX_MCLK/MAT2[0]/TXD3	82	I/O	P4[28] – General purpose digital input/output pin.
		I	RX_MCLK I^2S receive master clock. (LPC1768/66/65 only)
		O	MAT2[0] – Match output for Timer 2, channel 0.
		O	TXD3 – Transmitter output for UART3.
P4[29]/TX_MCLK/MAT2[1]/RXD3	85	I/O	P4[29] – General purpose digital input/output pin.
		I	TX_MCLK – I^2S transmit master clock. (LPC1768/66/65 only)
		O	MAT2[1] – Match output for Timer 2, channel 1.
		I	RXD3 – Receiver input for UART3.
TDO/SWO	1	O	TDO – Test Data out for JTAG interface.
		O	SWO – Serial wire trace output.
TDI	2	I	TDI – Test data in for JTAG interface.
TMS/SWDIO	3	I	TMS – Test mode select for JTAG interface.
		I/O	SWDIO – Serial wire debug data input/output.
$\overline{\text{TRST}}$	4	I	$\overline{\text{TRST}}$ – Test Rest for JTAG interface.

Symbol	Pin	Type	Description
TCK/SWDCLK	5	I	TCK – Test Clock for JTAG interface.
		I	SWDCLK – Serial wire clock
RTCK	100	I/O	RTCK – JTAG interface control signal.
$\overline{\text{RSTOUT}}$	14	O	$\overline{\text{RSTOUT}}$ – This is a 3.3 V pin. LOW on this pin indicates LPC1768/66/65/64 being in Reset state.
$\overline{\text{RESET}}$	17	I	External reset input : A LOW on this pin resets in the device, causing I/O ports and peripherals to take on their default states, and processor execution to begin at address0). TTL with hysteresis, 5 V tolerate.
XTAL1	22	I	Input to the oscillator circuit and internal clock generator circuits.
XTAL2	23	O	Output from the oscillator amplifier.
RTCX1	16	I	Input to the RTC oscillator circuit.
RTCX2	18	O	Output from the RTC oscillator circuit.
V_{SS}	32, 41, 55,72, 97, 83	I	Ground : 0 V reference.
V_{SSA}	11	I	Analog ground : 0 V reference. This should nominally be the same voltage as V_{SS} but should be isolated to minimize noise and error.
$V_{DD(3V3)}$	28, 54, 71, 96	I	3.3 V supply voltage : This is the power supply voltage for the I/O ports.
$V_{DD(REG)(3V3)}$	42, 84	I	3.3 V voltage regulator supply voltage : This is the supply voltage for the on-chip voltage regulator only.
V_{DDA}	10	I	Analog 3.3 V pad supply voltage : This should be nominally the same voltage as $V_{DD(3V3)}$ but should be isolated to minimize noise and error. This voltage is used to power the ADC and DAC.
VREFP	12	I	ADC positive reference voltage: This should be nominally the same voltage as V_{DDA} but should be isolated to minimize noise and error. Level on this pin is used as a reference for ADC and DAC.

Courtesy NxP

Symbol	Pin	Type	Description
VREFN	15	I	ADC negative reference voltage : This should be nominally the same voltage as V_{SS} but should be isolated to minimize noise and error. Level on this pin is used as a reference for ADC and DAC.
VBAT	19	I	RTC pin power supply : 3.3 V on this pin supplies the power to the RTC peripheral.
n.c.	13	–	Not connected

❑❑❑

CHAPTER 3

UNIT III

ARM7TDMI Assembly Language Instructions and Programming

Syllabus

Different Types of Instructions, ARM Instruction set, Data processing instructions. Shift Operations, shift Operations using RS lower byte, Immediate value encoding. Data processing Instructions. Addressing modes with examples. Instruction Timing, Swap Instructions, Swap Register related Instructions, Loading Constants. Program Control Flow, Control Flow Instructions, B & BL instructions, BX instruction. Interrupts and Exceptions, Exception Handlers, Reset Handling. Aborts, Software Interrupt Instruction, Undefined instruction exception. Interrupt Latency, Multiply Instructions,. Thumb state, Thumb Programmers model, Thumb Implementation, Thumb Applications. Thumb Instructions, Interrupt processing. Interrupt Handling schemes, Examples of Interrupt Handlers, Co-processor instructions

3.1 Different Types of Instructions

- A processor has many instructions. These instructions are broadly classified into various groups based on their functions.
- Some of the groups are listed below :

1. Data Transfer instructions are used to perform various data transfer operations i.e. transfer of data from one location to another
2. Arithmetic instructions to perform arithmetic operations. As the name says, these instructions perform operations like addition, subtraction etc.
3. Logical instructions to perform logical operations like AND, OR, EXOR etc.
4. Control Flow Instructions are used to control the sequence of execution of the program. They are mainly branching instruction which includes conditional branching.

3.2 ARM Instruction Set

The instruction set of ARM processor is divided into the following categories :

(1) Control Flow instructions i.e. Branch instructions and Branch with link instructions.

(2) Data Processing Instructions.

(3) Multiply Instructions.

(4) Load and Store Instructions or the Data transfer Instructions

(5) Program Status Register Access Instructions.

(6) Swap/Semaphore Instructions.

(7) Counting Leading Zeros Instruction.

(8) Exception/Software Interrupt generating Instructions.

(9) Multiple Register Transfer Instructions.

(10) Co-processor Instructions.

3.3 Data Processing Instructions

GTU - S-16, W-16

Q. Write a operation of following instruction : RSC r0,r1,r2. **(S-16, 7 Marks)**

Q. Explain Addressing Modes used for ARM data transfer Operation. Explain the types of Base plus offset addressing used for ARM data transfer operation with examples. **(W-16, 7 Marks)**

The ARM instruction set has 16 data processing instructions. The data processing instructions include following instructions.

(1) Arithmetic instructions.
(2) Comparison and test instructions.
(3) Logical instructions.
(4) Data movement between registers.

Table 3.3.1 lists the data processing instructions.

Table 3.3.1

Mnemonic	Operation	Action
ADD	Add	R_d : = R_n + <operand2>
ADC	Add with carry	R_d : = R_n + <operand2> + carry
SUB	Subtract	R_d : = R_n – <operand2>
SBC	Subtract with carry	R_d : = R_n – <operand2> – NOT (carry)
RSB	Reverse Subtract	R_d : = <operand2 > – R_n
RSC	Reverse Subtract with carry	R_d : = <operand2> – R_n – NOT (carry)
CMP	Compare	CPSR flags : = R_n – <operand2>
CMN	Compare Negated	CPSR flags : = R_n + <operand2>
TST	Test	CPSR flags : = R_n AND <operand2>
TEQ	Test Equivalence.	CPSR flags : = R_n EOR <operand2>
AND	Logical AND	R_d : = R_n AND <Operand2>
EOR	Logical Exclusive OR	R_d : = R_n EOR <operand2>
ORR	Logical OR	R_d : = R_n OR <operand2>
BIC	Bit Clear (Logical AND)	R_d : = R_n AND NOT <operand2>
MOV	Move	R_d : = <operand2>
MVN	Move Not	R_d : = NOT <operand2>

- The data processing instructions operate on registers only. They cannot be operated on memory.

- They perform a specific operation on one or more operands. The first operand is always register Rn and second operand is sent to the ALU through the barrel shifter.
- The compare and test instructions update the condition flags only. While the other data processing instructions store the register into a register and update the condition flags.
- The operand2 is an immediate number or a register.

3.4 Arithmetic and Logical Instructions

The arithmetic instructions are listed in the Table 3.4.1.

Table 3.4.1 : Arithmetic instructions

Mnemonic	Operation	Action
ADD	Add	R_d : = R_n + <oprnd 2>
ADC	Add with carry	R_d : = R_n + <oprnd 2> + carry
SUB	Subtract	R_d : = R_n – <oprnd 2>
SBC	Subtract with carry	R_d : = R_n – <oprnd 2> – NOT (carry)
RSB	Reverse subtraction	R_d : = <oprnd 2> – R_n
RSC	Reverse subtraction with carry	R_d : = <oprnd 2> – R_n – NOT (carry)

(1) ADD : Addition

GTU - W-16, W-18

Q. Explain the following ARM instruction with examples : ADD **(W-16, 1 Mark)**

Q. Explain following instructions: ADD **(W-18, 1 Mark)**

Mnemonic	ADD {<cond>} {S} R_d, R_n, <oprnd 2>	**Algorithm**	R_d = R_n + <oprnd 2>, if condition is true, condition is optional
Description	R_d ← R_n + <oprnd 2> This instruction will add the two operands. The result is placed in the destination register R_d. Operand2 can be a register, shifted register or an immediate number.		
Example : (i) ADD R0,R1,R2 R0 ← R1 + R2 (ii) ADD R0, R1, #256 R0 ← R1 + 256 (iii) ADD R0, R2, R3, LSL #1 R0 ← R2 + (R3 << 1)	This instruction will add the contents of registers R1 and R2 and place the result in the destination register R0. This instruction will add the contents of register R1 with the immediate value 256 and store the result in register R0. This instruction will shift the contents of register R3 by 1 bit position to the left. Then it will add the contents of registers R2 and R3 and store the result in register R0.		

(2) ADC : Addition with Carry

Mnemonic	ADC {<cond>} {S} R_d, R_n, <oprnd 2>	**Algorithm**	$R_d = R_n$ + <oprnd 2> + carry if the condition is true
Description	$R_d \leftarrow R_n$ + <oprnd 2> + carry. This instruction adds the two operands, placing the result in the destination register. It uses a carry bit, inorder to add numbers greater than 32 bits.		

Example

Addition of 64 bit numbers.

64 bit result	;	registers R2 and R3.
64 bit first	;	registers R0 and R1.
64 bit second	;	registers R2 and R3.
ADDS R2, R2, R0	;	Add lower 32 bits and save status of carry.
ADC R3, R3, R1	;	Add upper 32 bits and carry of LSB addition.

Note : While adding numbers greater than 32 bits, suffix S is added, so that the status of carry flag is updated.

(3) SUB : Subtraction

Mnemonic	SUB {<cond>} {S} R_d, R_n, <oprnd 2>	**Algorithm**	$R_d = R_n$ – <oprnd 2>, if condition is true
Description	$R_d \leftarrow R_n$ – <oprnd 2> This instruction will subtract the operand2 from register R_n, placing the result in the destination register. The operand2 can be a register, shifted register or an immediate number		
Example : SUB R0, R1, R2 R0 ← R1 – R2.	This instruction will subtract the contents of register R2 from the contents of register R1, placing the result in register R0.		

(4) SBC : Subtract with Carry

Mnemonic	SBC {<cond>} {S} R_d, R_n, <oprnd 2>	**Algorithm**	$R_d = R_n$ – <oprnd> – ! carry. if the condition is true.
Description	$R_d \leftarrow R_n$ – <oprnd> – ! carry. This will subtract the two operands, placing the result in the destination register. The carry bit is used to represent "borrow" inorder to subtract numbers greater than 32 bits.		

Note : SUB and SBC generate the carry flag in a inverted way i.e. if a borrow is required then the carry flag is not set. Thus, this instruction requires a NOT carry flag. The carry flag is automatically inverted during the instruction.

(5) RSB : Reverse Subtract

GTU - S-17, S-18, W-19

Q.	Describe the function performed by the following ARM mnemonics : RSB	**(S-17, 7 Marks)**
Q.	Explain following instructions with at least one example for each : RSB	**(S-18, W-19, 1 Mark)**

Mnemonic	RSB {<cond>} {S} R_d, R_n, <oprnd 2>	**Algorithm**	R_d = <oprnd 2> – R_n, if condition is true.
Description	R_d ← <oprnd 2> – R_n. This instruction will perform reverse subtraction i.e. the contents of register R_n are subtracted from operand2 and the result is stored in the destination register. Operand2 can be a register, shifted register or an immediate number. The subtraction may be performed on signed or unsigned numbers.		

(6) RSC : Reverse Subtraction with Carry

Mnemonic	RSC {<cond>} {S} R_d, R_n, <oprnd 2>	**Algorithm**	R_d = <oprnd 2> – R_n – ! carry if condition is true
Description	This instruction will subtract the two operands, placing the result in destination register. It is same like SBC instruction, except that subtraction is carried in reverse order.		

3.4.1 Comparison and Test Instructions

GTU - S-16

Q.	Write a operation of following instruction : TST r1,r2	**(S-16, 7 Marks)**

The comparison and test instructions are listed in Table 3.4.2.

Table 3.4.2 : Comparison and test instructions

Mnemonic	Operation	Action
CMN	Compare Negative	CPSR flags = R_n + <oprnd 2>
CMP	Compare	CPSR flags = R_n – <oprnd 2>
TEQ	Test Equivalence	CPSR flags = R_n EOR <oprnd 2>
TST	Test bits	CPSR flags = R_n AND <oprnd 2>

Let us see these instructions.

(1) CMN : Compare Negative

GTU - W-18, W-19

Q.	Explain following instructions: CMN	**(W-18, W-19, 1 Mark)**

Mnemonic	CMN {<cond>} R_n, <oprnd 2>	**Algorithm**	CPSR flags are updated on the operation R_n + <oprnd 2> i.e. R_n – <– oprnd 2>, if the condition is true
Description	This instruction compares the contents of the register R_n with the logical NOT of operand i.e. against small negative values. It performs a subtraction, but the result of comparison is not stored anywhere. The flags are updated. The instruction does not affect the other registers.		
Example CMN R0, R2	This instruction will compare the contents of register R0 with R2 and update flags according to the result.		

(2) CMP : Compare

Mnemonic	CMP {<cond>} R_n, <oprnd 2>	Algorithm	CPSR flags are updated on the operation R_n – <oprnd 2>, if the condition is true.
Description	This instruction compares the contents of register R_n with the operand 2, updating the status flags. It performs a subtraction, but the result is not stored anywhere. The flags are updated.		
Example CMP R0,R1.	This instruction will perform subtraction, R0 – R1, and update flags according to the result.		

(3) TEQ : Test Equivalence

GTU - S-17, S-18, W-18

Q. Describe the function performed by the following ARM mnemonics : TEQ **(S-17, 7 Marks)**

Q. Explain following instructions with at least one example for each : TEQ **(S-18, W-18, 1 Mark)**

Mnemonic	TEQ {<cond>} R_n, <oprnd 2>	Algorithm	CPSR flags = R_n EOR <oprnd 2>, if the condition is true
Description	This instruction performs logical EOR (exclusive OR) operation on the two operands. The result of the operation is not stored anywhere. The result of operation is reflected in the status flags. This instruction is used with the P suffix to alter the flags in R15 (in 26 bit mode). It provides a way to see if the bits in both the operands are same or not without affecting the carry flag. Operand2 can be a register, shifted register or immediate value.		
Example TEQ R2, #3	This instruction will logically perform EOR operation on register R2 and immediate number 3. The result will be updated in the status flags.		

(4) TST : Test bits

GTU - W-18

Q. Explain following instructions: TST **(W-18, 1 Mark)**

Mnemonic	TST {<cond>} Rn, <oprnd 2>	Algorithm	CPSR flags = R_n AND <oprnd 2>, if condition is true.
Description	This instruction performs a logical AND operation on the two operands. The result updates the status flags. This instruction is used to see if a particular bit is set. After testing the zero flag will be set upon a match, otherwise it is cleared.		
Example TST R0, R1, SL#3.	This instruction will logically AND R0 and left shifted contents of R1 by 3 bit positions and update the flags in CPSR according to the result.		

3.4.2 Logical Instructions

Table 3.4.3 lists the logical instructions.

Table 3.4.3

Mnemonic	Operation	Action
AND	Logical AND	$R_d = R_n$ AND <oprnd 2>
BIC	Bit clear (Logical NAND)	$R_d = R_n$ AND NOT <oprnd 2>
EOR	Logical Exclusive OR	$R_d = R_n$ EOR <oprnd 2>
ORR	Logical (inclusive) OR	$R_d = R_n$ OR <oprnd 2>

Let us study these instructions one by one.

(1) AND : Logical AND

Mnemonic	AND {<cond>} {S} R_d, R_n, <oprnd 2>	**Algorithm**	R_d = R_n AND <oprnd 2>, if condition is true.
Description	$R_d \leftarrow R_n$ AND <oprnd 2> This instruction will logically AND the two operands and place the result in the destination register specified in the instruction. This instruction is useful for masking the bits the user wants. The operand2 can be a register, shifted register or an immediate number. **AND Table** R_n | Operand2 | Results 0 | 0 | 0 0 | 1 | 0 1 | 0 | 0 1 | 1 | 1		
Example : AND R0, R0, #3	This instruction will logically AND the contents of register R0 with immediate data, 3 and store the result in register R0.		

(2) BIC : Bit Clear

GTU - S-17, S-18

Q. Describe the function performed by the following ARM mnemonics : BIC **(S-17, 7 Marks)**

Q. Explain following instructions with at least one example for each : BIC **(S-18, 1 Mark)**

Mnemonic	BIC {<cond>} {S} R_d,R_n, <oprnd 2>	**Algorithm**	R_d = R_n AND NOT <oprnd 2>, if condition is true
Description	$R_d \leftarrow R_n$ AND ! <oprnd 2>. This instruction will logically AND the contents of register R_n and complemented contents of operand2. The result of operation will be stored in the destination register. Operand2 is a 32 bit mask. If a bit is set in the mask, it will be cleared. Unset mask bits indicate bits to be left as it is. This instruction provides a way to clear bits within a word.		
Example : BIC R0, R1, #%1011	This instruction will clear the bits zero, one and three in register R1 and leave the remaining bits as it is. The result will be stored in register R0.		

(3) EOR : Logical Exclusive OR

GTU - S-18, W-19

Q. Explain following instructions with at least one example for each : EOR **(S-18, W-19, 1 Mark)**

Mnemonic	EOR {<cond>} {S} R_d, R_n, <oprnd 2>	**Algorithm**	R_d = R_n EOR <oprnd 2>, if condition is true
Description	R_d ← R_n EOR <oprnd 2>. This instruction will perform a logical EOR operation between register R_n and operand2, the result will be stored in the destination register. Operand2 can be a register, shifted register or an immediate value. This instruction is useful for inverting some bits.		
Example : EOR R0, R0, #3.	This instruction will logically EOR the register R0 with 3 i.e. it will inverts bits zero and one in register R0 and store the result in register R0.		

EOR Table

R_n	Operand2	Result
0	0	0
0	1	1
1	0	1
1	1	1

(4) ORR : Logical (inclusive) OR

Mnemonic	ORR {<cond>} {S} R_d, R_n, <oprnd 2>	**Algorithm**	R_d = R_n OR <oprnd 2>, if condition is true
Description	Rd ← Rn OR <oprnd 2>. This instruction will perform logical OR operation between register R_n and operand2 and the result will be stored in destination register R_d. Operand2 can be a register, shifted register or an immediate value. This instruction is useful for setting certain bit to be set.		
Example : ORR R0, R1, R2, LSL #3.	This instruction will logically OR the register R1 with the left shifted contents of register R2 by three locations. The result of operation is stored in the register R0.		

OR Table

R_n	Operand2	Result
0	0	0
0	1	1
1	0	1
1	1	1

3.4.3 Shift Operation using RS Lower Byte

- In the immediate value encoding, the immediate value or the operand 2 can be replaced with a register Rm and a shift to be applied to the register Rm.
- The structure of this encoding is as shown in the Fig. 3.4.1 :

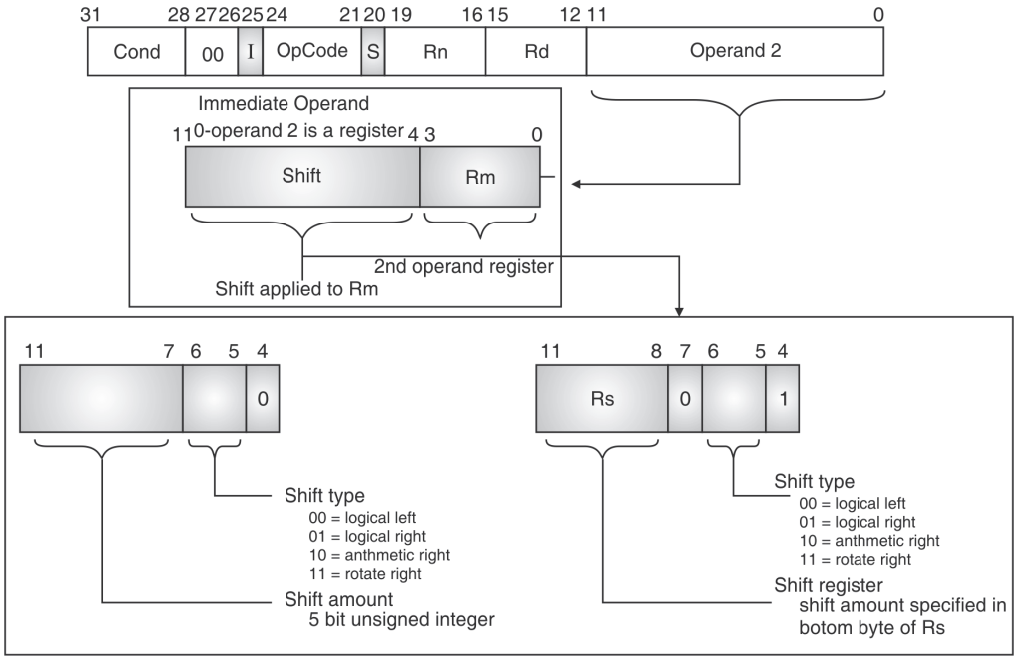


Fig. 3.4.1 : Structure of this encoding

- As shown in Fig. 3.4.1, Rs is the register (R0 to R15) that is used to indicate the number of shifts to be performed on Rm.
- The bits 5 and 6 are used to indicate the type of shift as shown in the Fig. 3.4.1

  Example: MOV R2,R1,LSL R0
- This instruction logically shifts the register R1 for R0 times in the left direction and thereafter copies the value in to register R2
- Similarly other shift operations can be performed as shown in the next section.

3.5 Rotate and Barrel Shifter

GTU - W-17, S-18, S-19,W-19

Q.	Describe the importance of barrel shifter in ARM processor architecture.	**(W-17, 3 Marks)**
Q.	Explain barrel shifter of ARM processor and its related assembly language instructions with suitable example.	**(S-18, 7 Marks)**
Q.	Explain barrel shifter instructions LSL, LSR and ASR with example.	**(S-19, 6 Marks)**
Q.	Explain shift instruction with example.	**(W-19, 7 Marks)**

- The ARM processor does not support actual shift instructions. But instead it supports a **Barrel shifter**. The Barrel shifter is responsible for carrying the shifts as a part of other instructions.
- The barrel shifter can be used to perform left shift operation by required amount. i.e. it multiplies by the powers of 2.

Example : LSL #4 (multiply by 16).

Fig. 3.5.1 : Logical shift left operation

- The barrel shifter can also to used to perform logical and arithmetic right.
- In logical shift right the barrel shifter shifts the bits to the right by specified amount. i.e. it divides by power of 2.

Example : LSR #3 (divide by 8).

Fig. 3.5.2 : Logical shift right

- In arithmetic shift right the barrel shifter shifts the bits to the right by specified amount and preserves the sign bit, for 2's complement operations.

Example : ASR #5 (divide by 32).

Fig. 3.5.3 : Arithmetic shift righ

- The barrel shifter can also be used for performing rotations.

In Rotate Right (ROR) the barrel shifter performs operation similar to arithmetic shift right but wraps the bits around as they leave the LSB and appear as MSB. The last bit rotated is also used as carry o t along with LSB.

Example : ROR #2.

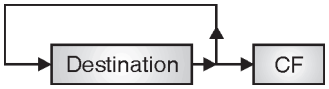


Fig. 3.5.4 : Rotate right

- For Rotate Right through carry or Rotate Right Extended (RRX) the barrel shifter uses the CPSR C flag as 33rd bit. In this operation the bits are rotated through the carry. The carry propagates to LSB and MSB to carry.

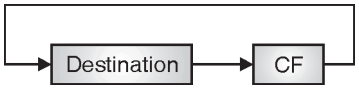


Fig. 3.5.5 : Rotate right through carry

- Using the barrel shifter alongwith ALU a wide range of addresses can be calculated.
- The contents of register R_m are preprocessed in the barrel shifter before applying them as input to the ALU. The register R_m can be a register, shifted register or immediate value.

3.6 Program Control Flow/ Branch Instructions

GTU - S-17, S-18, S-19, W-19

Q.	State the instruction used to come back the program sequence to the next instruction after the BL instruction is stored.	**(S-17, 1 Mark)**
Q.	Explain looping and branch instructions of ARM processor with suitable examples.	**(S-18, S-19, 4 Marks)**
Q.	Explain following ARM instruction with example: BGT	**(W-19, 1 Mark)**

(1) B. Branch

Mnemonic	B {< cond > } < target address > or B { < cond >} < label >	**Algorithm**	R15 = target address
Description	The ARM instruction allows a conditional branch forward or backward upto 32 MB. Upon entering a branch instruction the ARM processor will jump to the target address given in the instruction, and it will resume execution from there.		
Example BCC label BNE label	This instruction will branch to label if carry flag is clear. This instruction will branch to label if zero flag is clear		

Note : The actual value stored in the branch instructions is an offset from the current value of R15, rather than an absolute address.

The various conditions possible in the branch instruction are listed in Table 3.6.1.

Table 3.6.1 : Conditions of branch instructions in ARM

Sr. No.	Branch Instruction		Branch on condition
	Mnemonic	Meaning	
1.	B	Unconditional	Branch is always taken
2.	BAL	Always	
3.	BEQ	Equal	Branch if equal i.e. result is zero
4.	BNE	Not Equal	Branch if not equal i.e. result is non-zero
5.	BPL	Plus	Branch if result is non-negative.
6.	BMI	Minus	Branch if result is negative.
7.	BCC	Carry clear	Branch if result has no carry generated.
8.	BLO	Lower	
9.	BCS	Carry Set	Branch if result has carry generated.
10.	BHS	Higher or Same	
11.	BVC	Overflow clear	Branch if result has no overflow is generated.
12.	BVS	Overflow set	Branch if result has overflow is generated.
13.	BGT	Greater Than	Branch if signed comparison is greater than.

Sr. No.	Branch Instruction		Branch on condition
	Mnemonic	Meaning	
14.	BGE	Greater or Equal	Branch if signed comparison is greater than or equal
15.	BLT	Less Than	Branch if signed comparison is less than
16.	BLE	Less or Equal	Branch if signed comparison is less than or equal
17.	BHI	Higher	Branch if unsigned comparison is higher
18.	BLS	Lower or same	Branch if unsigned comparison is lower or same

(2) BL : Branch with Link

GTU - S-17

Q. Describe the function performed by the following ARM mnemonics : BL **(S-17, 7 Marks)**

Mnemonic	BL { < cond > } label.	**Algorithm**	R14 = R15, R15 = address of label.
Description	This instruction loads the contents of register R15 to the LR (R14) register just before the branch i.e. it loads the return address for the branch in register R14. This allows us to initiate subroutine calls. The register R14 can be reloaded to register R15, to return to the main program execution after the execution of the branch.		
Example BL func	subroutine call to function.		

(3) BLX : Branch with Link and Exchange

GTU - W-16

Q. Explain the following ARM instruction with examples : BLX **(W-16, 1 Mark)**

Mnemonic	BLX < label >	**Algorithm**	R14 = R15, T = 1
Description	This instruction is used to call a Thumb subroutine from the ARM instruction set at an address that is specified in the instruction. This instruction is unconditional. It stores the address of the instruction following the branch instruction in the link register R14. The instruction sets the T flag. The execution of the Thumb instruction begins at the address specified.		
Example BLX T_Func	Thumb subroutine call to function.		

(4) BX : Branch and Exchange Instruction

Mnemonic	BX {< cond > } < R_m >.	**Algorithm**	R15 = R_m
Description	This instruction branches the program control to the address specified in register R_m, with an optional switch to Thumb execution. The register R_m holds the value of branch address. Bit ϕ of the register R_m is 0 in order to select a target ARM instruction or 1 to select a target Thumb instruction.		
Example BX func	This instruction will transfer the program control to target ARM instruction or target Thumb instruction depending on the status of Bit [0] of R_m register.		

3.7 Load and Store Instructions

GTU - S-16

Q. Write a operation of following instruction : STR r^0,[r^1] **(S-16, 1 Mark)**

- The ARM architecture supports two types of instructions which load the value of register from or to memory.
- They are :
  - (i) To load or store a 32 bit word or an unsigned byte.
  - (ii) To load or store a 16 bit unsigned halfword or load and sign extend a 16 bit halfword or a byte. These instruction are available in ARM architecture version 4 and higher versions.

3.7.1 Load and Store Word or Unsigned Byte Instruction

GTU - S-17

Q. Describe big endian configuration for various LDR and STR mnemonics. **(S-17, 4 Marks)**

Mnemonic	LDR / STR {<cond>} {B} {T} R_d, <address> Where, B : Distinguishes between an unsigned byte (B = = 1) and a word (B = = 0) access. T : Load with user mode privilege.
Description	The load instructions load a data from memory and place it in the processor register while the store instructions store the data from a register to a memory location.
Example (i) LDR R1, [R0] (ii) LDR R12, [R14, # – 3] (iii) STR R2, [R1, # 0x100]	This instruction will load value from address in register [R0] to ARM register R1. This instruction will load value from address [R14] – 3 and store it on general purpose register R12. This instruction will store the value in ARM register R2 to address in [R1] + 0x100.

3.7.2 Load and Store Half Word and Load Signed Byte

GTU - S-17, W-19

Q.	Describe the function performed by the following ARM mnemonics : LDRSB	**(S-17, 7 Marks)**
Q.	Explain following ARM instruction with example : LDR	**(W-19, 1 Mark)**

Mnemonic	LDR / STR {<cond>} H / SH / SB. R_d, <address> where H : Distinguishes between a halfword (H = = 1) and a signed byte (H = = 0) access. S : Distinguishes between a signed (S = = 1) and an unsigned (S = = 0) halfword access. Note : (i) If the L bit is zero and S bit is one, the instruction is unpredictable. (ii) If the S and H bit are both zero, then the instruction encodes Multiply or SWAP instruction.
Description	The load instructions load data from memory and store it to an ARM register. The store instructions read data from an ARM register and store it to memory. Table 3.7.1 lists the instructions in this group.

Table 3.7.1

Mnemonic	Operation
LDRB	Load Byte
LDRBT	Load byte with user mode privilege.
LDRH	Load unsigned halfword.
LDRSB	Load signed byte
LDRSH	Load signed halfword
LDRT	Load word with user mode privilege
STRB	Store byte
STRBT	Store byte with user mode privilege.
STRH	Store halfword
STRT	Store word with user mode privilege.

Example (i) LDRB R4, [R8]	This instruction will load byte from address in R8 to ARM register R4. The upper 3 bytes will be zero.
(ii) LDRH R1, [R0]	This instruction will load the halfword from address in R0 to ARM register R1. The upper two bytes in register R1will be zero.
(iii) LDRSB R3, [R4, #3]	This instruction will load signed byte from address R8 + 3 to register R3.
(iv) LDRSH R4, [R8]	This instruction will load the signed halfword from address in R8 to register R4.
(v) STRB R5, [R4, # – 2] !	This instruction will store the byte form ARM register R5 to address R4 – 2 and then R4 = R4 – 2.
(vi) STRH R2, [R7, – R3]	This instruction will store halfword from ARM register R2 to address R7 – R3

Table 3.7.2 lists the instructions use to transfer data between registers.

Table 3.7.2

Mnemonic	Operation	Action
MOV	Move	R_d = <operand2>
MVN	Move Not	R_d = ! <operand2>

(1) MOV : Move

GTU - W-16

Q. Explain the following ARM instruction with examples : MOVEQ **(W-16, 1 Mark)**

Mnemonic	MOV {<cond>} R_d, <oprnd 2>	**Algorithm**	R_d = <oprnd 2>, if condition is true
Description	$R_d \leftarrow$ <oprnd 2>. This instruction loads value from operand2 to the destination register. Operand2 may be a register, shifted register or an immediate value. For the effect of NOP instruction the same register can be specified or shifted.		
Example (i) MOV R0, R0 R0 ←R0 (ii) MOV R0, R1 R0 ← R1	This instruction will load the contents of R0 from R0. This instruction will load the contents of register R1 to register R0.		

(2) MVN : Move Negative

GTU - S-17

Q. Describe the function performed by the following ARM mnemonics : MVN **(S-17, 7 Marks)**

Mnemonic	MVN {<cond>} R_d, <oprnd 2>	**Algorithm**	R_d = ! <oprnd 2>, if condition is true
Description	R_d = ! <oprnd 2>. This instruction will load the inverted value of operand2 (i.e. negative number or 1's complement of number to the destination register.). Operand2 can be a register, shifted register or an immediate number.		
Example MVN R0, #4	This instruction will load the register R0 with one's complement of 4.		

3.7.3 Counting Leading Zeros Instruction

GTU - W-16, S-17

Q. Explain the following ARM instruction with examples : CLZ **(W-16, 1 Mark)**

Q. Describe the function performed by the following ARM mnemonics : SMULL **(S-17, 7 Marks)**

Mnemonic	CLZ {<cond>} R_d, R_n
Description	The versions 5 and above of ARM processor support this instruction. It returns the number of 0 bits at the most significant end of its operand before the first '1' is encountered. It is used to locate the highest priority bit in a bit mask. It is used to determine how many bits the operand should be shifted left so that it can be normalized such that MSB is 1.

The multiplication instructions are different from the normal arithmetic instructions. This is because there are restrictions on the operands. The restrictions are :

(i) All the operands and the destination should be registers.

(ii) For the operand2 immediate values or shifted registers cannot be used. Operand2 must be a simple register.

(iii) The destination register and register R_m must both be different.

(iv) Register R15 cannot be used as the destination register.

There are two types of multiplication instructions shown in Fig. 3.7.1. They are :

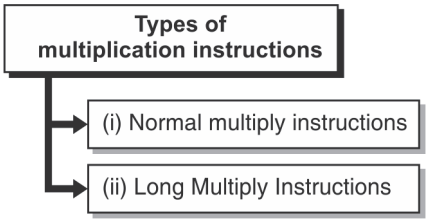


Fig. 3.7.1 : Types of multiplication instructions

(1) Normal multiply instructions

- These instructions produce 32 bit results. The normal multiply instructions are listed in Table 3.7.3.

Table 3.7.3

Mnemonic	Operation	Action
MLA	Multiply and Accumulated	$R_d = (R_m * R_s) + R_n$
MUL	Multiply	$R_d = R_m * R_s$

- Both these instructions can optionally set the Negative (N) and Zero (Z) condition flags.
- There is no distinction between signed and unsigned variants.
- Register $\mathbf{R_m}$ specifies the register that contains the first operand to be multiplied. Register $\mathbf{R_s}$ specifies the second operand. Register R_n contains the value that is added to the product of R_s and R_m.

(a) MUL : Multiply

Mnemonic	MUL {<cond>} {S}, R_d, R_m, R_s.	**Algorithm**	$R_d = R_m * R_s$., if condition is true
Description	$R_d \leftarrow R_m * R_s$. This instruction multiplies the 32 bit values in the two registers and stores the result in destination register. If the operands are signed, then the result is also signed.		

Example (i) MUL R3, R2, R1. (ii) MULS R3, R2, R1. R3 ← R2 * R1.	This instruction will multiply the contents of registers R1 and R2 and store the result in destination register R3. It will perform the same operation as example 1 and in addition to that it will set the N and Z flags.

(b) MLA : Multiply and Accumulate

Mnemonic	MLA {<cond>} {S} R_d, R_m, R_s, R_n.	**Algorithm**	$R_d = R_m * R_s + R_n$., if condition is true
Description	$R_d \leftarrow R_m * R_s + R_n$. This instruction multiplies the values of two registers, adds the value of third register, truncates the result to 32 bit and stores the result in the destination register. This instruction is useful for running totals.		
Example MLA R4, R3, R7, R8. R4 = R3*R7 + R8.	This instruction will multiply the two registers R3 and R7, add value of R8 to result of multiplication. The total result is truncated to 32 bits and stored in the destination register R4.		

(2) Long Multiply Instructions

GTU - W-16, W-19

Q. Explain the following ARM instruction with examples : UMULL **(W-16, W-19, 1 Mark)**

These instructions produce 64 bit results. They can optionally set the N(negative) and Z (zero) condition code flags.

- There are 4 Long Multiply Instructions.
- Two of the variants multiply the values of two registers and store the 64 bit result in third and fourth registers. One of the variant is signed (SMULL) and the other variant is unsigned (UMULL). The signed variants produce a different result in MSB 32 bits if one or both of the operands is/are negative.
- The other two variants multiply the two registers, add the 64 bit value from third and fourth registers and store the 64 bit result back in those registers. One of variant is signed (SMLAL) and other is unsigned (UMLAL). These variants perform long multiply and accumulate.
- Table 3.7.4 lists the Long Multiply Instructions.

Table 3.7.4

Mnemonic	Operation	Action
SMLAL	Signed multiply and accumulate long	[RdHi, RdLo] = [RdHi, RdLo] + (R_m *R_s)
SMULL	Singed multiply long	[RdHi, RdLo] = R_m *R_s
UMULL	Unsigned multiply long	[RdHi, RdLo] = R_m *R_s
UMLAL	Unsigned multiply and accumulate long	[RdHi, RdLo] = [RdHi, RdLo] + (R_m*R_s)

Mnemonic	{<cond>} {S}, RdLo, RdHi, R_m, R_s.
Example SMULL R3, R4, R2, R1. R3 ← Bits 0 to 31 of R2 × R3. R4 ← Bits 32 to 63 of R2 × R3.	This instruction will multiply the lower 32 bits of registers R2 and R3 and store the LSB result in R3. Then it will multiply the upper 32 bits of registers R2 and R3 and store the MSB result in register R4.

3.7.4 Semaphore / Swap Register Instructions

GTU - S-16, W-19

Q.	Write a operation of following instruction : SWPB r2,r1,[r4]	**(S-16, 1 Mark)**
Q.	Explain following ARM instruction with example : SWAPB	**(W-19, 1 Mark)**

The Table 3.7.5 list the semaphore/swap instructions.

Table 3.7.5

Mnemonic	Operation
SWP	Swap Word
SWPB	Swap Byte

- This instruction will load the byte from location pointed by R4 into the register R2 and the lower byte of R1 in the location pointed by R4.
- These instructions are used for process synchronization.
- They generate an automatic load and store operation, allowing a memory semaphore to be loaded and altered without interruption.
- Both the instructions support a single addressing mode.
- The swap instructions do not provide a compare and conditional write facility.
- In order to specify the value to be stored and the destination where data is to loaded, separate registers must be used. If same registers are used for loading and storing then value is exchanged with value in memory.

Mnemonic	SWP {<cond>} R_d, R_m, [R_n] : for swap word SWPB {<cond>} R_d, R_m, [R_n] : for swap byte.
Example (i) SWP R1, R2, [R3] (ii) SWPB R2, R1, [R4]	This instruction will load register R1 with the word addressed by register R3, and store register R2 at R3. This instruction will load register R2 with the byte addressed by R4 and store bits 0 to 7 of R1 at R4.

3.8 Software Interrupt Instructions

GTU - W-17, W-19

Q.	Describe Software Interrupt instruction SWI with its importance.	**(W-17, 4 Marks)**
Q.	Describe Software Interrupt instruction with example.	**(W-19, 3 Marks)**

The ARM instruction set supports two instructions whose main purpose is to cause a processor exception to take place. They are :

(1) SWI (Software Interrupt)
(2) BKPT (Breakpoint)

(1) SWI Software Interrupt

Mnemonic	SWI {<cond>} <imm_24>.
Description	This instruction causes a software interrupt process exception. Using this mechanism user can make calls to privileged operating system code.

Note : <imm_24> is a 24 bit immediate value that is put into bits [23 : 0] of the instruction. The 24 bit value is ignored by the processors. It can be used by the operating system to determine which operating system service is being requested.

(2) BKPT : Breakpoint

Mnemonic	BKPT <imm>
Description	This instruction is used for inserting software breakpoints. It causes a prefetch abort exception to occur. It is used in the version 5 and above of ARM processor. A debug monitor program that had been installed on prefetch abort vector can handle the exception caused.

Note : <imm> is a 16 bit immediate value. The 12 MSB bits are placed in bits 19-8 of the instruction and 4 LSB bits are placed in bits 3-0 of the instruction. This value is used by debugger to store information about the breakpoint.

3.8.1 Co-processor Instructions

GTU - S-18

Q.	Explain co-processor instructions to communicate ARM processor with co-processors.	**(S-18, 7 Marks)**

- For communication with the coprocessors the ARM processors, instruction set supports different instructions. They are :

(1) CDP : Coprocessor Data Operation
(2) LDC : Load Coprocessor register
(3) MCR : Move to Coprocessor from ARM Register.
(4) MRC : Move to ARM Register from Coprocessor.
(5) STC : Store Coprocessor Register.

- When the ARM processor executes a coprocessor instruction, it will offer it to any coprocessor that is attached to the system.

- All the coprocessor instructions can be conditionally executed.
- The coprocessor instructions allow the ARM processors to generate addresses for the coprocessor Load and Store instructions.
- They allow the ARM processor to initiate a coprocessor data processing operation.
- The coprocessor instructions support data transfer to/from ARM processors register to coprocessor registers.

(1) CDP : Coprocessor Data Operation

Mnemonic	CDP {<cond>} <coproc>, <opcode_1> <CR_d>, <CR_n>, <CR_m>, <opcode_2> CDP <coproc>, <opcode_1>, <CR_d>, <CR_n>, <CR_m>, <opcode_2>. Where, <coproc> : Specifies the name of the coprocessor and causes the corresponding coprocessor number to be placed. The coprocessor names are p0, p1, p2, p3, …, p15. <opcode_1> and <opcode_2> : Specifies the coprocessor operation that is to be done. <CR_d> : Specifies coprocessor destination register for instruction. <CR_n> : Specifies the coprocessor register that contains the first operand for the instruction. <CR_m> : Specifies the coprocessor register containing the second operand.
Description	It instructs the coprocessor to do some processing.
Example CDP p4, 3, C12, C13, C3, 4	This instruction indicates that it is a coprocessor 4 data operation such that opcode_1 = 3, opcode_2 = 4, destination register is C12 and source register are C13 and C3.

(2) MCR : Move to Coprocessor from ARM register

Mnemonic	MCR <coproc> <opcode_1>, <R_d>, <CR_n>, <CR_m> <opcode_2> Where, R_d : ARM destination register. If R15 is used as destination register then the result is unpredictable. CR_n : Destination coprocessor register. CRm : Addition destination or source coprocessor register.
Description	This instruction transfers data from an ARM register to a coprocessor register.
Example MCR p3, 1, R8, C7, C5, 4.	This instruction will transfer data from ARM register R8 to coprocessor 3 such that opcode_1 = 1, opcode_2 = 4, coprocessor destination registers are C7 and C5.

Note : Other fields have same meaning as CDP instruction.

(3) MRC : Move to ARM register from coprocessor register

Mnemonic	MRC <coproc>, <opcode_1>, <R_d>, <CR_n>, <CR_m>, <opcode_2> where fields have same meaning like MCR instruction.
Description	This instruction will transfer data from a coprocessor register to an ARM register
Example MRC p2, 5, R2, C0, C5, 3.	This instruction will transfer data from coprocessor 2 to ARM register R2 such that opcode_1 = 5, opcode_2 = 3, Source registers are C0 and C5.

(4) LDC : Load coprocessor register

Mnemonic	LDC {<cond>} {L} <coproc>, <CR_d>, <address> LDCL <coproc>, <CR_d>, <address>. Where, L sets N bit (bit 21) in instruction to 1 indicating a long load instruction. If L bit is omitted, then the N bit is zero and instruction is a short load instruction.
Description	This instruction loads data from memory to coprocessor register. If no coprocessor is present for execution of instruction, an undefined instruction exception is generated. The data is loaded form the memory from the sequence of consecutive memory addresses calculated by the addressing mode. The addressing mode can be one of the following : (i) Immediate offset : {<R_n>, # + / – <offset_8> * 4] (ii) Immediate preindexed : [<R_n>, # + / – <offset_8> * 4] ! (iii) Immediate postindexed : <R_n>, # + / – <offset_8> * 4 (iv) Unindexed : [<R_n>], <option>.
Example LDC p2, CR0, [R3]	This instruction will load the data from memory to coprocessor 2. ARM register R3 consists the address to load in register CR0

(5) STC : Store coprocessor register

HN	STC {<cond>} {L} <coproc>, <CR_d>, <address> STCL <coproc>, <CR_d>, <address> Fields have same meaning as in LDC instruction.
Description	This instruction stores data from a coprocessor register to memory. If no coprocessor is present for the execution of instruction, an undefined instruction exception is generated. The data is loaded to the memory from the sequence of consecutive memory addresses calculated by the addressing mode. The addressing mode can be one of the following : (i) immediate offset : [<Rn>, # + / – <offset_8> * 4] (ii) immediate pre-indexed : [<Rn>, # + / – <offset_8> * 4] ! (iii) immediate post indexed : [<Rn>] # + / – <offset_8> * 4. (iv) unindexed : [<Rn>, <option>]

Example STC P2, CR8, [R3], # – 8	This instruction will store data from coprocessor 8 to memory. The ARM register R3 holds the address after the transfer R3 = R3 – 16. The data is stored from coprocessor register 8.

3.9 Program Status Register Instructions

For transferring the contents of program status register to a general purpose register or vice versa, two instructions are used. The CPSR and SPSR can be accessed.

The instructions in this category are :

(1) MRS {<cond>}R_d, CPSR

GTU - W-16

Q. Explain the following ARM instruction with examples : MRS **(W-16, 1 Mark)**

Move data from CPSR to register R_d.

Mnemonic	MRS {<cond>} R_d, CPSR.	**Algorithm**	R_d = CPSR, if condition is true
Description	$R_d \leftarrow$ CPSR This instruction will transfer the contents of CPSR to destination register R_d i.e. it will read the CPSR.		
Example MRS R0, CPSR.	This instruction will copy the contents of CPSR in register R0.		

(2) MRS {<cond>} Rd, SPSR

Move data from SPSR to register

Mnemonic	MRS {<cond>} R_d, SPSR.	**Algorithm**	R_d = SPSR, if condition is true
Description	$R_d \leftarrow$ SPSR. This instruction will transfer the contents of SPSR to destination register R_d. i.e. it will read the SPSR.		
Example : MRS R1, SPSR.	This instruction will transfer the content of SPSR to register R1		

(3) MSR {<cond>} CPSR_<fields>, #<immediate>

Move immediate data to CPSR flags

Mnemonic	MSR {<cond>} CPSR_ <fields>, # <immediate>	**Algorithm**	CPSR = # 32 bit immediate data, if condition is true
Description	CPSR ← # 32 bit immediate data. This instruction will copy the 32 bit immediate data to the CPSR flags.		

Example : MSR CPSR_f, # FE002134H.	This instruction will set the field mask bit in CPSR and copy 32 bit FE002134 H in the CPSR.

Note : Fields is a sequence of one or more of the following

c : It sets the control field mask bit (bit 16)

x : It sets the extension field mask bit (bit 17).

s : It sets the status field mask bit (bit 18).

f : It sets the flags field mask bit (bit 19).

(4) MSR {<cond>} CPSR_<fields>, R_m

Move data from register to CPSR

Mnemonic	MSR {<cond>} CPSR_<fields>, R_m	**Algorithm**	CPSR = R_m, if condition is true
Description	CPSR ← R_m. This instruction will transfer the contents of register R_m to CPSR flags. R_m is any general purpose register.		

Example

```
MRS R0, CPSR        ;  Read CPSR
BIC R0, R0, # 0x1F  ;  Clear mode bits
ORR R0, R0, # 0x11  ;  Set mode bits to FIQ mode
MSR CPSR_C, R0      ;  Update the control field mask bit in the CPSR. Processors will now be in FIQ mode
```

(5) MSR {<cond>} SPSR_<fields>, # <immediate>

Move immediate data to SPSR flags

Mnemonic	MSR {<cond>} SPSR_<fields>, # <immediate>	**Algorithm**	SPSR = 32 bit immediate data, if condition is true
Description	SPSR ← 32 bit immediate data. This instruction will copy the 32 bit immediate data to register SPSR.		

(6) MSR {<cond>} SPSR_<fields>, R_m

Move data from register to SPSR flags

Mnemonic	MSR {<cond>} SPSR_<fields>, R_m	**Algorithm**	SPSR = R_m, if condition is true
Description	SPSR ⟵ R_m. This instruction will transfer data from register R_m to SPSR flags. Register R_m is any general purpose register.		

3.10 Conditional Execution

GTU - S-16, S-17, W-17, S-18, S-19

Q. Write a operation of following instruction : ADDEQ r0,r1,r2. **(S-16, 1 Mark)**

Q. State the significance of conditional execution. **(S-17, 1 Mark)**

Q. Explain the benefits of conditional execution of instructions in ARM processor. **(W-17, 4 Marks)**

Q. What is conditional execution? What are the mnemonics extension codes used for conditional execution? Explain with example. **(S-18, S-19, 7 Marks)**

- A very special feature of ARM instruction set is that almost all the instructions have a condition associated with it. The instruction will be executed only if the condition associated with is true. This concept is called as condition execution.
- This is not in many processors. The major advantage of this condition execution is that it dramatically reduces the branches and hence the pipeline stalls. Every instruction has the first four bits dedicated for condition check.
- Since there are 4-bits, we can have a maximum of 16 conditions to be checked.
- The Table 3.10.1 lists these conditions that can be checked by the ARM instructions.

Table 3.10.1 : ARM condition codes

Op code [31:28]	Mnemonic extension	Interpretation	Status flag stage for execution
0000	EQ	Equal/Equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/ Unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/Negative	N set
0101	PL	Plus/Positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HL	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear and Z set
1010	GE	Signed greater than or equal	N equals V

Op code [31:28]	Mnemonic extension	Interpretation	Status flag stage for execution
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always (need not to use !)	Any
1111	NV	Never (do not use !)	None

- If nothing is mentioned in the instruction, then the "Always" case is used. The last case i.e. "Never" should never be used, else the instruction will never be executed.
- All other case, check a particular flag or status bit to decide, whether the instruction should be executed or not, based on the condition given. The conditions or the status flags can be checked to be set or clear. The different status bits that can be checked are Zero flag, Carry flag, Negative flag, overflow flag and their combination as shown in Table 3.10.1.
- For example if the first case i.e. 'EQ' is used with ADD instruction, then the instruction will be written as ADDEQ in this case the ADD operation will be performed only if the condition 'EQ' is true i.e. zero flag is set to '1.'

3.11 Multiple Register Load and Store Instructions

GTU - W-16, S-17

Q. Explain the following ARM instruction with examples : STMFD **(W-16, 1 Mark)**

Q. Describe various addressing methods for stack operations for ARM architecture with the help of suitable LDM and STM mnemonics. **(S-17, 7 Marks)**

- The multiple register transfer instructions load or store a subset or all of the general purpose registers to or from the memory.
- The main use of these instructions is to dump registers that need to be preserved onto the stack.

Mnemonic	LDM {<cond>} <address> R_n {!}, <R list> {∧} STM {<cond>} <address> Rn {!}, <Rlist> {^} R_n : express for evaluating valid register number. <Rlist> : is a list of registers and register changes enclosed in s{ } e.g. {R1, R2-R7, R10} {!} : if present requests write back (W = 1), otherwise W = 0. {^} : if present set S bit to load the CPSR along with the PC or force transfer of user bank when in privileged mode.

3.12 Stack Instructions

GTU - W-16, S-17, S-18, W-19

Q.	Explain Stack operation in ARM with it's types in detail. Also explain instructions used for Stack.	**(W-16, 7 Marks)**
Q.	Describe various addressing methods for stack operations for ARM architecture with the help of suitable LDM and STM mnemonics.	**(S-17, 7 Marks)**
Q.	Explain STACK operations in ARM processor. Explain STACK related instructions.	**(S-18, W-19, 7 Marks)**

- The location where the data was last stored in the stack is called as the **top of the stack**.
- Stack works in the Last In First Out (LIFO) manner. The PUSH operation writes onto the top of the stack while the POP operation is used to read the most recent data pushed on the stack top.
- The microprocessor stores the information/data like stacking books. Fig. 3.12.1 shows stacked books. If we want to remove the first stack book, then we have to remove all the books above the first i.e. we have to remove the fourth book, third book, second book and then finally the first book. This indicates that the first book pushed onto the stack is the last one to be popped from the stack. This operation is called as **First In Last Out (FILO)**.

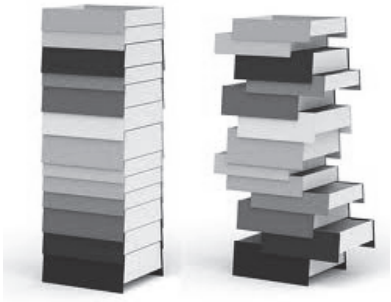


Fig. 3.12.1 : Stacked books

- The instructions used for stack operations in case of ARM processor are the same i.e. LDM and STM. We can store the contents of multiple registers in to the memory using a single STM instruction; this can be considered as PUSH operation for contents of multiple registers on to the stack memory. Similarly using a single LDM instruction, we can load the contents into multiple registers; this can be referred to as POP operation from stack memory. Let us take some examples for the same.

Ex. 3.12.1 : POP the contents from memory location 0x12345678 into registers r5 to r8.

Soln. :

```
LDMIA r3, {r5-r8}
```

In this case r3 must be pointing to the location 0x12345678. Thus all the registers will be loaded from the memory location 0x12345678 onwards. The above instruction will do the same operation as the following set of instructions:

```
LDR  r5, [r3]
LDR  r6, [r3, #4]
LDR  r7, [r3, #8]
LDR  r8, [r3, #12]
```

Ex. 3.12.2 : Write a operation of following instruction : LDMIA r0,{r2-r6} **GTU - S-16**

Soln. :

LDMIA r0,{r2-r6} : This instruction will load the contents of memory location pointed by R0 onwards into the **registers R2 to R6**

3.13 Thumb Instruction Set

GTU - W-16, S-18, W-18, W-19

Q.	List the properties of Thumb Instruction Set. Write a note on Thumb Implementation.	**(W-16, W-19, 7 Marks)**
Q.	What is the difference between ARM state and Thumb state?	**(S-18, W-18, 3 Marks)**

3.13.1 Introduction to Thumb Instruction Set

- The ARM7TDMI processor has two sets of instruction sets:
  - (i) standard 32-bit ARM instructions, studied until this section
  - (ii) 16-bit Thumb instruction set, to be studied in this section.
- Thus the processor is said to operate in two states viz.
  - (i) The ARM state to execute the 32-bit, word aligned ARM instructions
  - (ii) The Thumb state to execute the 16-bit half-word aligned Thumb instructions.
- The Thumb instruction set is available from the ARM versions above v4T. The functionality of these Thumb instructions is a subset of the functionality of the ARM instructions. In other words, the Thumb instructions do some of the operations that ARM instruction set can perform. But there are certain differences that will be seen in this section.
- An important thing to note is that the processor can execute the Thumb instructions when in Thumb state, while ARM instructions when in ARM state. The processor in ARM state cannot execute the Thumb instructions of 16-bit and also when the processor is in Thumb state it cannot execute the ARM instructions of 32-bit. There are instructions available in each set to switch the processor state. The default state of ARM processor when switched on is the ARM state. In Thumb instruction application, we have
  1. Two operand (or address) instructions i.e. one of source is destination.
  2. Conditional execution is not executed
  3. Inline barrel shifter cannot be used
  4. The size of the constants is limited
  5. The lower part of registers are only used.
- It should be noted that Thumb in itself is not a complete architecture of the processor, it needs the main state of processor operation. Also all the ARM development tools support the Thumb instruction set.

3.13.2 Switching between ARM State and Thumb State

GTU - W-17, S-18.,W-19

Q.	Describe the rules defined in ARM-Thumb Procedure Call Standard for calling a function.	**(W-17, 3 Marks)**
Q.	How switching between ARM and Thumb state is done?	**(S-18, 3 Marks)**
Q.	Write a difference between ARM instruction set and thumb instruction set.	**(W-19, 3 Marks)**

- Thumb applications are normally the 16-bit applications of embedded systems. The embedded system can be such designed that during the 32-bit ARM state the processor uses the on-chip 32-bit memory, while the processor uses the 16-bit off-chip memory when in the Thumb state. The Thumb state should be for large non-critical sub-routines.

- The instruction *BX* is used for switching between the two states. The *BX* instruction stands for Branch and Exchange, which is discussed earlier in this chapter. This instruction sets the T bit in the program status register, if the bottom bit of the specified register was set and hence enters into the Thumb state.
- In this state the program counter switches address given by the remainder part of the register specified in the instruction. Thus this causes change in the value of the program counter i.e. a branching occurs and hence the flushing of the pipeline has to be done.
- Switching back to the ARM state from the Thumb state can be done by the processor using the same instruction i.e. the *BX.* Also the processor returns automatically to the ARM state if the exception occurs in the Thumb state. When the processor returns from the Thumb state to the ARM state the previous value of the SPSR is copied into the CPSR register.

3.13.3 Thumb Programmer's Model

GTU - W-16

Q. Discuss the thumb programmer's model in brief. **(W-16, 7 Marks)**

The Thumb programmers model includes the following:

1. The registers R0 to R7 (only lower part)
2. Few instructions require R8 to R15, some of which also have a special function.
   (i) R13 works as a stack pointer
   (ii) R14 works as a link register
   (iii) R15 works as a program counter.

Fig. 3.13.1 shows the programmers model of the thumb state of the ARM processor.

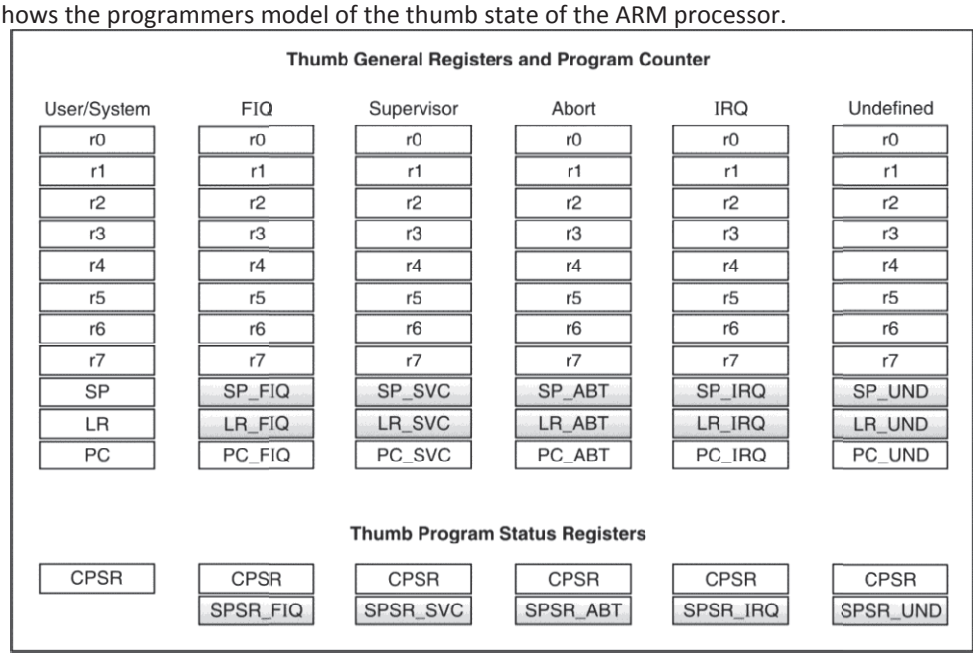


Fig. 3.13.1 : Thumb programmer's model

3.13.3(A) Branching Instructions of Thumb

The various branching instructions in the Thumb instruction set are as shown in Fig. 3.13.2

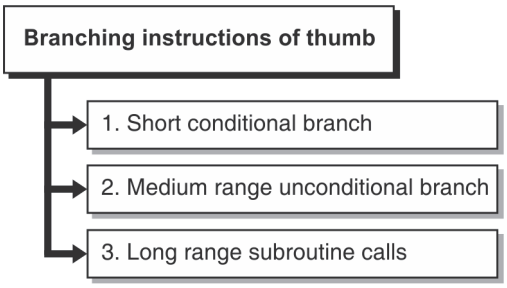


Fig.3.13.2 : Branching Instructions of Thumb

1. Short conditional branch

The format of the short conditional branch is shown in the Fig 3.13.3. It has 8-bit offset address that is added to the current value of Program counter if the condition is correct. It is used for short jumps like in loops of if-else condition implementation.

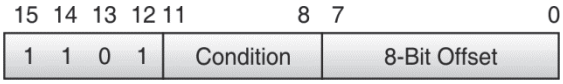


Fig. 3.13.3 : Short conditional branch instruction format

2. Medium range unconditional branch

This format has a 11-bit offset address as shown in the Fig. 3.13.4. This value is added to the current value of program counter if the condition is correct. It is normally used for slightly longer jumps to implement instructions like *GOTO* of high level languages.

Fig. 3.13.4 : Medium range unconditional branch instruction format

3. Long range subroutine calls

In this case the instruction is used for branch and link operation. The 'H' bit controls operation for this instruction. If the 'H' bit is '0', then the Link Register gets a new value which is equal to the current value of PC added with the offset given in the instruction shifted 12 bits left. If the 'H' bit is '1', then the program counter register gets a value of Link Register added with the offset given in the instruction shifted one bit left; and the Link register gets the older value of the Program counter added with 3.

If H ='0', LR = PC + (sign extended offset shifted left 12 times)

If H ='1', PC = LR+ (offset shifted left one time)

LR = (Old value of PC) + 3

Fig 3.13.5 shows the format of the BL instruction.

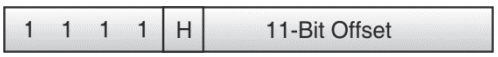


Fig. 3.13.5 : Long range subroutine call instruction format

4. BLX instruction is same as that of the ARM instruction set. This is studied earlier in this chapter.

3.13.3(B) Thumb Software Interrupt Instruction

GTU - W-16

Q. Explain Thumb Software interrupts instruction. **(W-16, 7 Marks)**

The Fig. 3.13.6 shows the instruction format for the Thumb software interrupt.

Fig. 3.13.6 : Thumb software interrupt instruction format

Whenever a software interrupt instruction is given in the Thumb mode, the following events occur:

1. The address of the next instruction to be executed is stored in the Link register i.e. r14_svc
2. The value of CPSR is copied into the SPSR.
3. The interrupt is disabled by clearing the IRQ bit and the processor enters into the supervisor mode.
4. The program counter is forced to the address 08H

Thus the software interrupt instruction executes the interrupt.

3.13.3(C) Thumb Data Processing Instructions

The Thumb instruction set for the data processing has both 3-operand and 2-operand instructions. The mnemonics of the instructions for data processing in the ARM state and the Thumb state are same. The operand options are either two address (operand) instructions of 3-address (operand) instructions.

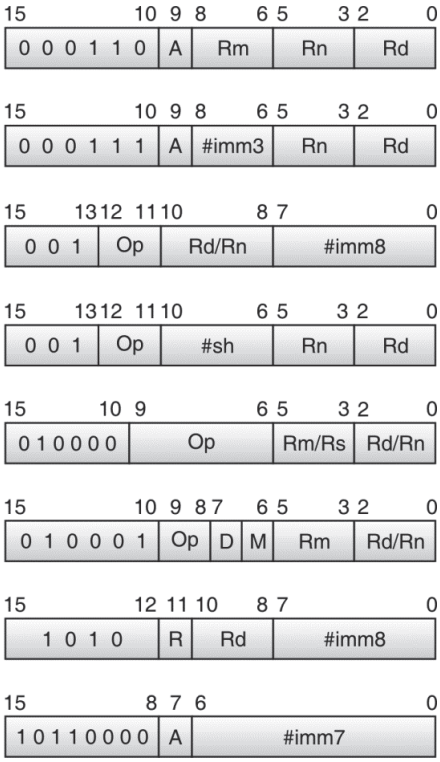


Fig. 3.13.7 : Data processing instruction formats of thumb

The data processing instructions of the Thumb instruction set has complex formats that cover most of the operations required by the compiler. The various formats of these data processing instructions are shown in the Fig. 3.13.7.

The examples for some of these instruction formats can be as given below in the same sequence.

1. The first format is a three address format i.e. has three operands. It is a register addressing mode i.e. all the operands are in the registers. Example of such an instruction is: ADD r3,r4,r5.
2. The second format is an immediate addressing mode example and has one immediate operand. This is also The example of such an instruction is: ADD r2,r3,#34.

   The remaining instructions are the two address format instructions. The example of instruction of the format 6 is: ADD r3,r4. Similarly the example for the 7th instruction format is: ADD r3,#34.

3.13.4 Advantage of Thumb Instructions

GTU - S-17

Q. Describe the advantages of Thumb instructions by developing the division program using ARM instructions as well as Thumb instructions. **(S-17, 3 Marks)**

The Thumb instructions as discussed earlier are similar to that of the ARM instruction set with 16-bit registers and the Thumb instruction set is a subset of ARM instruction set. The Table 3.13.1 shows some of the differences of ARM state and the Thumb state of the processor with respect to the instruction set

Table 3.13.1 : ARM and thumb instruction set differences

Sr. No.	ARM state	Thumb state
1.	All the instructions are 3-operand or 3-address format	Most of the instructions are 2-address or 2-operand format
2.	Many of the ARM instructions are executed conditionally	Many of the instructions in the Thumb mode are executed unconditionally
3.	The instruction formats of the ARM state do not have too many variations	The instruction formats of the Thumb state have a lot of variations

The following are some of the similarities of ARM state and the Thumb state of the processor.

(i) Both use the Load-Store architecture to access memory

(ii) They support 8-bit, 16-bit and 32-bit data

The features of Thumb instructions set with respect to ARM instructions set in terms of performance is as given below:

1. The Thumb instructions require only 70% of the memory space compared to that of the ARM instructions. Hence the power requirement for accessing the memory is also lesser by 30% in case of the Thumb instructions compare to the ARM instructions.
2. The Thumb code requires 40% more instructions to perform the same task as that of the ARM instructions.
3. Since ARM instruction set has 32-bit memory, it is 40% faster compared to the Thumb instructions to access memory.
4. Since Thumb instruction set has 16-bit memory, it is 45% faster compared to the ARM instructions for code execution.

3.13.5 Instruction Timing

1. T- State :

- One subdivision of an operation. A T-state lasts for one clock period.
- An instruction's execution length is usually measured in a number of T-states. (Clock cycles). Fig. 3.13.8 shows a T-state.

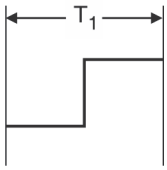


Fig. 3.13.8 : T-state

- If operating frequency is 3 MHz then ,

1. **Time period (t-state)** = $\frac{1}{\text{operating frequency}}$

$= \frac{1}{3}$ MHz

= 333.33 ns

2. **Machine Cycle** :

- The time required to complete one operation of accessing memory, I/O, or acknowledging an external request is machine cycle.
- The Fig. 3.13.9 shows a machine cycle of four T-states.

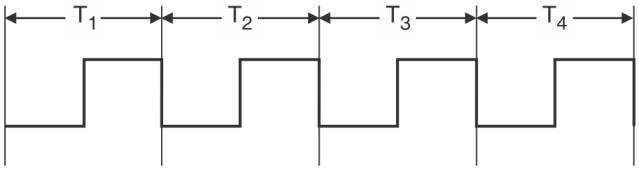


Fig. 3.13.9: Four T-state Machine Cycle

3. **Instruction Cycle :**

- The time required to complete the execution of an instruction is called instruction cycle. Fig. 3.13.10 shows an instruction cycle of two machine cycles (one machine cycle of 4 T-states and another of 3 T-states).

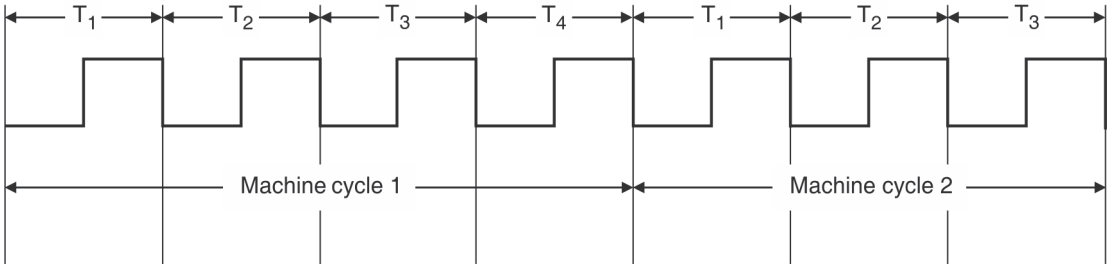


Fig. 3.13.10: Instruction cycle with two machine cycles

- One operation is surely to be done for all instructions i.e. fetching of opcode and hence the opcode fetch machine cycle will be there for all instructions. Besides, there may be some additional machine cycles as required for a given instruction.

3.14 Assembler Rules and Directives

GTU - S-18

Q. What are assembler directives? Explain any three assembler directives used for ARM assembly programming.

(S-18, 4 Marks)

- The assembly language instructions have a definite structure.

- The assembly language statement is divided into various fields. These fields are separated by spaces and tabs.
- The general format of an assembler instruction is as follows :

Label	Mnemonic	Operands	Comment
(Optional) used to identify the instruction	Instruction name and type	The data which is being operated upon	(Optional)

Label <space> opcode <space> operands <space> ; comment

- The comments allow user to annotate each line of source code in the program i.e. make it simple for user to understand the program.
- The following instruction illustrates the various fields clearly.

e.g. up : MOV PC, LR ; Load data from R14 to R15

Label Mnemonic operands comment

AREA, ENTRY and END Assembler Directives

- Assembly language consists of a number of statements that help us to control the manner in which a source program assembles and lists. Such statements are called as **directives.**

Let us see the directives.

(i) **AREA** : This directive specifies the chunks of code or data that are manipulated by the linker. One of more areas can exist in an application. A single CODE area is minimum required in order to produce an application.

(ii) **ENTRY** : This directive marks the first instruction to be executed within an application. An application can consist a single entry point.

(iii) **END** : This directive marks the end of the module.

3.15 Assembly Language Programs

Program 3.15.1 : To transfer block of data from one address to other address.

Soln. :

We will transfer data from BLOCK 1 to BLOCK 2.

Label	Instructions	Comments
	AREA BLOCKCOPY, CODE, READONLY	Declare code area.
	SWI_Write CEQU & 0	Output character in R0.
	SWI_Exit EQU & 11	End program.
	ENTRY	Code entry point.
START :	ADR R1, BLOCK 1	Load address of BLOCK 1 in register R1 R1.
	ADR R2, BLOCK 2	Load address of BLOCK 2 in register R2 R2.
	ADR R3, T1END	Load the address of END of BLOCK 1 in register R3
LOOP1 :	LDR R0, [R1], #4	Obtain 1st word of BLOCK 1.

Label	Instructions	Comments
	STR R0, [R2], #4	Store the 1st word of block 1 to BLOCK 2.
	CMP R1, R3	Is storing task finished ?
	BLT LOOP1	If not, continue else copy data.
	ADR R1, BLOCK 2	Load address of block 2 in register R1.
LOOP2 :	CMP R0, #0	
	SWI NE	Check for end of text string if not print and loop back
	SWI_Write C	
	BNE LOOP2	
	SWI	
	SWI_Exit	End.
BLOCK 1	"This is the correct string !", & 0a, & 0d, 0.	
B1END		
	ALIGN	Check word alignment.
BLOCK2	This is the wrong string !", 0	
	END.	

Note : SWI 0x00 : Write the character in bottom 8 bits of R0 to the display.

SWI 0x11 : exits the program.

Program 3.15.2 : Hex to ASCII conversion. Write ALP for HEX to ASCII conversion. Write ARM assembly language program for HEX number to ASCII conversion. Assume HEX number is stored in register R1. **GTU - S-18, S-19, 4 Marks**

Soln. :

Label	Instructions	Comments
	AREA Hex_ASCII, Code, Read only	Declare code area.
	SWI_Write CEQU & 0	Output character in R0.
	SWI_Exit EQU & 11	End program.
	ENTRY	
LDR :	R1, DATA	Load data in R1.
	BL Hex_ASCII	Call subroutine ex_ASCII.
	SW SWI_Exit	End.
	DATA DCD & 87654321	
	Hex_ASCII	
	MOV R2, #8	Load R2 with iteration count.
LOOP :	MOV R0, R1, LSR #28	Move the upper 4 bits of R1 to R0.
	CMP R0, #9	Compare R0 with 9
	ADDGT R0, R0, #'A' – 10	If R0 > 9, add 37 H.
	ADDLE R0, R0, #0	If R0 < 9, add 30 H.

Label	Instructions	Comments
	SWI SWI_Write C	Print value in R0.
	MOV R1, R1, LSL #4	Shift R4 by four bits to left to display next nibble.
	SUBS R2, R2, #1	decrement 2
	BNE LOOP	If not zero, continue.
	MOV PC, LR	
	END	End.

Program 3.15.3 : To unpack bits 8-15 of register R0 and place the result in R1 register.

Soln. :

Instructions	Comments
Area Unpack, Code, Readonly	Declare code area.
SWI_Write C EQU & 0	Output character in R0.
SWI_Exit EQU & 11	End program
ENTRY	.
AND R1, R0, #00F0 H	Mask bits 8-15 of register R0 and store result in register R1.
SWI SWI_Exit	
END	End.

Program 3.15.4 : To divide a 32 bit number by an 8 bit number.

Soln. :

Instructions	Comments
Area Unpack, Code, Readonly	Declare code area.
SWI_WriteC EQU & 0	Output character in R0.
SWI_Exit EQU & 11	End program
ENTRY	.
MOV R1, #23567890 H	Load 32 bit number in register R1. register R1.
MOV R2, #8	Load 8 bit number in register R2.
MOV R1, R1, LSR #3	divide 32 bit number by 8 bit number and store the result in register R1.
SWI SWI_Exit	
END	End

Program 3.15.5 : Write a program to load and shift a data in R0 towards left by one bit and store the result in R1.

Soln. :

Instructions	Comments
Area DataShift, Code, Readonly	Declare code area.
SWI_WriteC EQU & 0	Output character in R0.
SWI_Exit EQU & 11	End program

Instructions	Comments
ENTRY	.
MOV R0, #0x11	Load a number into register R0
MOV R1, R0, LSL #1	Shift left by one bit the contents of register R0 and store the result in R1
SWI SWI_Exit	
END	End

Program 3.15.6 : Factorial calculation.

Soln. :

Instructions	Comments
Area Factorial, Code, Read only	Declare code area.
SWI_WriteC EQU & 0	Output character in R0.
SWI_Exit EQU & 11	End program
ENTRY	.
MOV R6,#5	Load a number 5 in register R6
MOV R4, R6	Copy the number in register R4. R4 will work as a multiplier register
SUBS R4,R4,#1	Decrement the multiplier register
MUL R3,R6,R4	Multiply the multiplier register with the earlier result and store result in R3.
MOV R6,R3	Copy the result in the register R6 for next iteration
SUBS R4,R4,#1	Decrement the multiplier register
BNE back	Repeat the loop if R4 is not equal to zero. There is a condition followed after the branch instruction i.e. "NE", thus the branch will happen only if R4 is not equal to zero
SWI SWI_Exit	
END	End

Program 3.15.7 : Write a program to load some values in registers R0 and R1 and swap them.

Soln. :

Instructions	Comments
Area Swapping, Code, Readonly	Declare code area.
SWI_WriteC EQU & 0	Output character in R0.
SWI_Exit EQU & 11	End program
ENTRY	.
MOV R1, #23567890 H	Load 32 bit number in register R1. register R1.
MOV R2, #1ABCDEF4 H	Load 32 bit number in register R2.

Instructions	Comments
EOR R0,R0,R1	The registers R0 and R1 are EX-ORed and the result is stored in register R0
EOR R1,R0,R1	The registers R0 and R1 are EX-ORed and the result is stored in register R1
EOR R0,R0,R1	The registers R0 and R1 are EX-ORed and the result is stored in register R0
SWI SWI_Exit	
END	End

Program 3.15.8 : Write a program to move a floating point value from register R3 to an integer register R4.

Soln. :

Instructions	Comments
Area Unpack, Code, Readonly	Declare code area.
SWI_WriteC EQU & 0	Output character in R0.
SWI_Exit EQU & 11	End program
ENTRY	.
MOV R4, R3	Load 32 bit number from register R3 to register R4.
SWI SWI_Exit	
END	End

Program 3.15.9 : Write an ALP to find the factorial of a given number. **GTU - W-16, 7 Marks**

Soln. :

Instructions	Comments
MOV R3, R0	Number whose factorial is required
MOV R1, #1	Counter
MOV R4, R0	Copy of number
LOOP :	.
SUB R0, R0, #1	Decrement multiplier
MUL R3,R0, R3	Multiply multiplier to previous result
ADD R1, R1, #1	Increment counter
SUBS R1, R4	Check if counter = the given number
BEG LOOP	
MOV R0, R3	Store result in R0

Program 3.15.10 : Write an ALP to find the larger of the two 32 bit numbers. **GTU - W-16, 7 Marks**

Soln. :

Instructions	Comments
LRD R1, value1	Load first number in R1
LDR R2, value2	Load second number in R2
CMP R1, R2	Compare R1 with R2
BHI OVER	Branch to label over if R1 is higher than R2
MOV R1, R2	If not higher move R2 to R1
OVER:	
STR R1, RESULT	Store higher value that is R1 in result variable.

3.16 Interrupt Latency

Interrupt is a asynchronous event that stops the normal routine of microprocessor/ microcontroller and forces the processor to execute service routine for that event. The microprocessor / microcontroller can be interrupted by three sources.

1. External I/O device who wants immediate service from processor called as hardware interrupt.
2. Software run time exception or trap that cause error condition.
3. Execution of certain instruction which are called software interrupt. Most of the processors have a instruction(s) that cause interrupt.

Whenever the processor is interrupted by any one of above three interrupt sources, first it completes the execution of current instruction and save the address of next instruction on the stack and branch to service routine. After executing the instruction of service routine, processor returns back to the program where it was interrupted by retrieving return address from stack.

3.16.1 Interrupt Service Routine

- The subroutine written for servicing interrupt is known as Interrupt Service Routine (ISR). The address where the interrupt service routine is located is called as interrupt vector address or simply interrupt vector.
- One question that may be raised in your mind that how does the microprocessor/microcontroller knows where to find the interrupt routine when interrupt occur ? This depends on the processor.
- In some microprocessor/microcontroller there is predefined fixed interrupt vector address for each interrupt source. In 8051, for $\overline{INT0}$ interrupt, interrupt vector address is fixed at memory location 0003H. It is the responsibility of the user to write $\overline{INT0}$ ISR at location 0003H.

- Some other processor have more sophisticated methods. The most typical is that a table somewhere in the memory contains interrupt vectors, the address of the interrupt routines.
- When an interrupt occurs, the processor will look up the address of the interrupt routine in this interrupt vector table. Again it is responsibility of user to set up interrupt vector table properly.

3.16.2 Interrupt Priorities

- Generally, there are multiple interrupt sources that send the interrupt requests to the microprocessor/ microcontroller, and some of the interrupt requests can come at the same time.
- Another situation is when one request is being served, another request can come during that time. These condition addressed the need of interrupt priorities.
- Generally non maskable interrupts and the software interrupts have higher priorities than the general purpose interrupts. Similarly vectored interrupts assign higher priorities than non-vectored interrupt.

For example

In 8085 microprocessor TRAP interrupt has highest priority followed by RST 7.5, RST 6.5, RST 5.5 and INTR. Here TRAP is non maskable interrupt. RST 7.5 to RST 5.5 are the maskable vectored interrupt whereas INTR is maskable nonvectored interrupt. In some microprocessors/microcontrollers, we can change the default interrupt priorities by loading appropriate control word into the priority register.

3.16.3 Interrupt Nesting

- Generally microprocessors/microcontrollers have prioritized multiple interrupt sources. In most processor higher priority interrupt can interrupt a lower priority interrupt service routine. This results in interrupt nesting.
- If the processor is executing a higher priority interrupt service routine when the hardware (I/O device) asserts the lower –priority interrupt signal, the processor will finish the higher interrupt service routine and then execute the lower priority interrupt routine.
- Interrupt nesting is useful and effective in real time embedded system. Here some devices may require urgent or immediate service from processor. This can be done by assigning higher priority interrupt to such devices and low priorities to other.
- In real time embedded system the tasks are to be completed in stringent time limit. If the processor is executing low priority interrupt service routine and I/O device needs immediate service from the processor. At this time, the device will assert signal to higher priority interrupt to get service. If the interrupt nesting is allowed then device will be serviced immediately.

3.16.4 Interrupt Latency

- The time gap between the occurrence of the interrupt and the beginning of execution of the corresponding ISR is called as the Interrupt latency.
- Following methods can be used to reduce interrupt latency.
  1. Keep the interrupt routines short. If the interrupt routines are short, waiting time for same or lower priority interrupt will be reduced.
  2. Disabling the interrupts for short period of time.
  3. Usage of nested interrupt handler, which allows further interrupts to occur even servicing the existing interrupt. This can be accomplished by reenabling the interrupt system after completion of work done for currently servicing interrupt. After the completion of nested interrupts, then control goes back to the original interrupt service routine.
  4. By prioritizing interrupts, interrupt controller can be configured in such way that microprocessor/ microcontroller ignore interrupts of the same or lower priority. To get the chance to same or lower priority interrupt, reenable interrupts at the end of ISR. The microprocessor/microcontroller gives time for lower-priority interrupts until a higher-priority interrupts occur. Therefore higher-priority interrupts have a lower average interrupt latency than the lower-priority interrupts which is essential for real time systems.

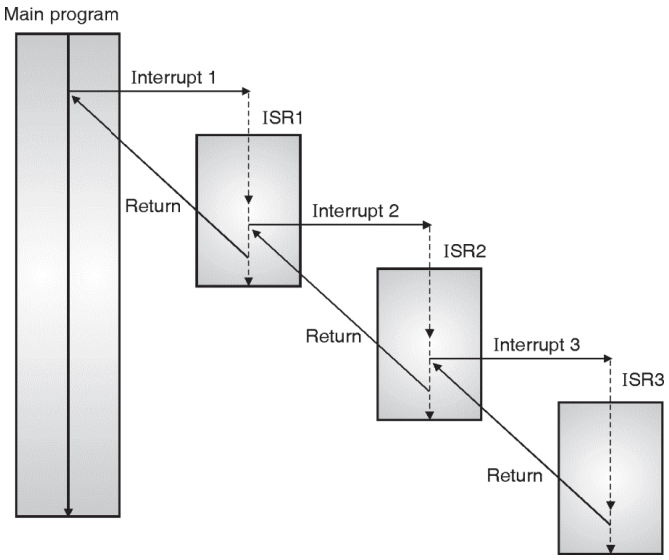


Fig. 3.16.1 : Nested interrupt handler execution sequence

3.17 Interrupts/Exception Handling in ARM

- In ARM processor there are five types of exception and the privileged option for each of these exceptions.
- The types of exception are :
  (i) Normal interrupt.
  (ii) Fast interrupt.
  (iii) Memory aborts used to implement memory protection or virtual memory.
  (iv) Undefined instruction execution.
  (v) software interrupts

- Software interrupt instructions that are used to make a call to the operating system.
- Whenever an exception occurs, the standard registers are replaced with registers that are specific to the exception mode.
- The exception modes have replacement banked registers for R13 and R15.
- For fast interrupt processing there are more registers as the processing has to be done fast.
- R14 contains the return address for exception processing. Once the exception is processed R14 returns to the address that caused the exception.
- The system mode uses the user mode registers in order to run the tasks that require privilege access to memory and/or coprocessors without limitations.
- R13 provides each exception handler an individual separate stack pointer.

3.17.1 Exception Process

- When an exception occurs, the ARM processor stops the execution of current instruction and continues execution from one of fixed addresses referred as exception vector in memory. For each exception there is a separate vector location.
- Processor copies the CPSR register to the SPSR. Then appropriate CPSR bits are set. If the core implements ARM architecture 4T and is in the Thumb state, then the ARM state is entered.
- It stores the return address in R14.
- Maps the appropriate banked registers.
- To return from the exception handler the CPSR is restored from the SPSR and PC from R14.
- At a time more than one exception can take place. Table 3.17.1 consists the types of exception and the processor mode used to process that exception.

Table 3.17.1 : Exception types

Exception type	Mode	Priority	Normal address	High vector address
Reset Handling	Supervisor	**1**	0x00000000	0xFFFF0000
Undefined instructions	Undefined	**6**	0x00000004	0xFFFF0004
Software Interrupt (SWI)	Supervisor	**6**	0x00000008	0xFFFF0008
Prefetch Abort (instruction fetch memory abort)	Abort	**5**	0x0000000C	0xFFFF000C
Data Abort (data access memory abort)	Abort	**2**	0x00000010	0xFFFF0010
IRQ (interrupt)	IRQ	**4**	0x00000018	0xFFFF0018
FIQ (fast interrupt)	FIQ	**3**	0x0000001C	0xFFFF001C

3.17.1(A) Reset Handling

- When Reset pin is activated, the current execution is stopped by the processor. After reset the processor begins execution at an address 0x00000000 or 0xFFFF0000 is the supervisor mode with interrupts disabled.
- The RESET vector is called only once during the initialization of the processor.

- In certain cases, the RESET vector may be called multiple times like in case of a RESET applied by the Watchdog timer. The watchdog timer is a timer that keeps a track of the processor, and in case the processor hangs in some code execution, it resets the processor to bring it out of the hanged mode.
- Watchdog timer also resets the processor in case of prolonged inactivity.

3.17.1(B) Undefined Instructions

- Whenever an attempt is made to execute an undefined instruction, an undefined instruction exception occurs.
- If a coprocessor instruction is executed, the processor waits for any external coprocessor to acknowledge that it can execute the instruction. Incase no external processor acknowledges, an undefined instruction exception occurs.

3.17.1(C) Software Interrupt

Whenever a software interrupt instruction is executed a software interrupt exception is caused. The processor enters the supervisor mode inorder to request a particular supervisor function.

3.17.1(D) Prefetch Abort Vector

- This interrupt automatically occurs whenever the processor tries to execute an unimplemented instruction of ARM processor.
- The binary codes corresponding to the assembly instruction in case is wrong i.e. no such instruction exists, then the interrupt will be generated.
- This interrupt may also be caused due to breakpoint in case of ARM 5 and above versions of ARM.

3.17.1(E) Data Abort

- A memory abort is signalled by the memory system. Activating an abort in response to data access (i.e. load or store) marks the data as invalid.

3.17.1(F) Interrupt Request (IRQ)

- This exception is generated by asserting the IRQ input on the processor. When the I bit in CPSR is set, this interrupt is disabled.
- It has a lower priority than the FIQ.
- If the I bit in CPSR is clear then the processor checks for an IRQ at instruction boundaries.

3.17.1(G) Fast Interrupt Request (FIQ)

- If the FIQ input is asserted, an FIQ exception is generated.
- FIQ mode has many banked register and hence not requiring major context switching i.e. the PUSH and POP operations. This is why it is called as fast interrupt.
- It is mainly used for data transfers like DMA operation.
- FIQ or fast interrupts can be disabled by clearing the F bit in the CPSR register.

3.17.1(H) High Vectors

- Some of the ARM implementations allow the exception vector locations to be shifted from their normal address 0x00000000-0x0000001C to address 0xFFFF0000-0xFFFF001C. These locations are called as **high vectors**.

3.18 Exception Priorities

Table 3.18.1 shows the exception priorities

Table 3.18.1

Priority	Exception
Highest 1	Reset
2	Data Abort
3	FIQ
4	IRQ
5	Prefetch Abort
Lowest 6	Undefined instruction, SWI

- The priority of a data abort exception is higher than FIQ. The priority of SWI and undefined instruction is same. But, both of them cannot take place at the same time.

3.18.1 Interrupt Handlers and Interrupt Handling Schemes

- Interrupt handler is the hardware that handles multiple interrupt sources.
- It is especially important to decide which interrupt is to be serviced in case of multiple interrupts occurring simultaneously.
- To do this job of interrupt handling, the interrupt handler uses various interrupt handling schemes.
- Interrupt handling scheme as the name states, is a method to handle multiple interrupts. There are various interrupt handling schemes or Interrupt Handlers.
- Some of the different interrupt handling schemes are listed below:

1. **Nested interrupt Handling Scheme :** In this case, an interrupt of higher priority can interrupt the servicing of an interrupt of lower priority
2. **Non-nested Interrupt handling scheme :** In this case an interrupt service routine cannot be interrupted by any other interrupt
3. **Re-entrant Interrupt handling scheme :** In this case an interrupt service routine can be re-entered by either the same or another interrupt routine
4. **Non Re-entrant Interrupt Handling Scheme :** In this case an interrupt that is already in service cannot be re-entered, even if the routine is suspended currently.
5. **Prioritized Interrupt Handling Scheme :** In this case the interrupts are allocated priorities and the same are used to decide which interrupt will be executed first and which will be executed later
6. **Vectored Interrupt controller :** In this case the vector entry of the Interrupt is modified and thereby deciding the priority of the interrupt.

- We will study an example of Nested Vectored Interrupt Controller (VIC) of ARM processor in later section

3.18.2 Example of Interrupt Handler : NVIC (Nested Vector Interrupt Controller) in LPC17xx

The NVIC is an integral part of the Cortex-M3. The tight coupling to the CPU allows for low interrupt latency and efficient processing of late arriving interrupts.

Features

- Controls system exceptions and peripheral interrupts
- In the LPC1768/66/65/64, the NVIC supports 33 vectored interrupts.
- 32 programmable interrupt priority levels, with hardware priority level masking
- Relocatable vector table
- Non-Maskable interrupt (NMI)
- Software interrupt generation

Interrupt sources

- Each peripheral device has one interrupt line connected to the NVIC but may have several interrupt flags. Individual interrupt flags may also represent more than one interrupt source.

  Any pin on PORT0 and PORT2 (total of 42 pins) regardless of the selected function, can be programmed to generate an interrupt on a rising edge, a falling edge, or both.

❑❑❑

CHAPTER 4

UNIT IV

Embedded C Programming for ARM

Syllabus

ARM Development Environment Embedded Software, Overview of C compiler and optimization, Basic C data types, C Looping structures, Register allocations, Function calls, Pointer aliasing, structure arrangement, bit-fields, Unaligned data and Endianness, Division, Floating point, Lnline functions and inline assembly, Portability issues. C programs for General purpose I/O, General purpose timer, PWM Modulator, UART, I2C Interface, SPI Interface, ADC, DAC

4.1 ARM Development Environment Embedded Software

- ARM offers Keil MDK which is best suited for commercial, off-the-shelf embedded microcontrollers. There is also ARM Development Studio for software development on ARM application processors and custom SoC designs.
- Alternatively, the ARM Mbed Device OS and ARM Pelion IoT Platform provides the operating system, cloud services, tools and developer ecosystem for commercial, standards-based IoT solutions. Some of ARM's ecosystem partners also provide development tools, and a range of partners also offer RTOS supporting the Arm architecture.
- In most of the cases, C language is used for programming in Embedded Systems. Let us see the basics of C programming before embedded programming.

4.2 Overview of C Compiler and Optimization

GTU - W-17, W-18, S-19, W-19

Q.	Discuss techniques for optimization of memory needs while writing C program for ARM processor.	**(W-17, 4 Marks)**
Q.	List and explain the various ARM optimization techniques.	**(W-18, 7 Marks)**
Q.	What are the advantages and disadvantages of assembly language programming compared to C programming?	**(S-19, 3 Marks)**
Q.	Discuss any three ARM optimization techniques.	**(S-19, 3 Marks)**
Q.	Explain techniques used for memory optimization in embedded systems.	**(S-19, 7 Marks)**
Q.	Describe ARM optimization techniques.	**(W-19, 3 Marks)**

- The embedded software is responsible for controlling the various peripherals of the embedded hardware to get desired result as per the specified functional requirement. The embedded software is generally embedded into ROM, hence it is also called embedded firmware.

- Incorporating intelligence to an embedded system is a onetime process and it can be done at any stage, it can be immediately after the fabrication of embedded hardware or at later stage. Once intelligence is incorporated into embedded product, by embedding the software in hardware, the product starts functioning properly and will continue serving the assigned task till the hardware failure occurs or corruption in the embedded software happens.
- In case of hardware failure the damaged component may need to be replaced by new component and for software corruptions, the software should be reloaded. The adaptability in embedded systems is implemented by storing the configurable parameters in the alterable permanent memory like NVRAM / FLASH.
- Embedded software can be developed by using processor / controller specific language such as assembly language or machine language or a processor / controller independent high level language like C, C++, JAVA etc. or combination of assembly language and high level language.
- In this chapter we will discuss embedded software programming in assembly language and in high level language 'C'.

4.2.1 Disadvantages of Assembly Programming

- Each technology has its own advantages and limitations. For the certain functional requirement, the assembly language is the most efficient technique. But it has following technical limitations.

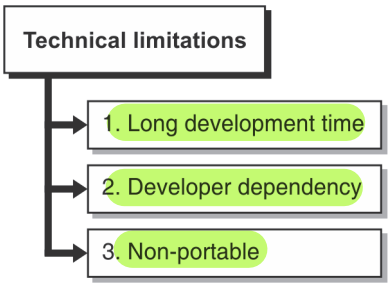


Fig. 4.2.1 : Technical limitations

1. Long development time

- Assembly language programming is a very rigorous and time consuming process.
- It is not a structured programming and hence consumes a lot of time i.e. it requires long development time.

2. Developer dependency

- Another problem with assembly programming is that to change or edit the program the same developer is required and hence the dependency on developer.
- It is very difficult for a new person to make changes in the program as it is difficult to understand the same.

3. Non-portable

- Assembly language is specific to a given processer. It is machine dependent and not portable from one system to another.
- It the target processor/controller changes, it is required to rewrite the application using assembly instructions of new processor/controller.

4.2.2 C Programming

- The main concern in microcontroller programming is program memory space. The machine code generated by the assembly language program is much smaller than the machine code generated by the 'C' program.
- But programming in C, has following advantages.
  1. Programming in C is less tedious and less time consuming as compared to assembly.
  2. It is easier to modify and update.
  3. The program requires slight modifications to be written for some other microcontroller i.e. the entire program need not be changed.
  4. C also provides some library functions.
- When we write a program in ‘C’ we need a compiler to convert it into the machine language. Here we will be using Pentium processor based compiler to do this programming.
- The compiler has to make a code of 8051 processor from the C program given to it such a compiler is called as cross compiler i.e. it produces the machine code for another processes, and not as that used in the system.
- The widely used cross compiler for 8051 and many other microcontrollers is “Keil μ Vision”.
- Keil μ vision has a header file for every microcontroller. This header file contains variable declaration, macro definitions and definitions for special function registers (SFRs).
- The header file for 8051 is reg 51.h. This file has given names for all the SFRs and hence it becomes convenient for the programmer to write the program directly by using these names.
- The names given to the SFRs are as used by us in normal programming; such as P0 for Port 0, P1 for Port1, TCON for timer control register etc.
- Embedded software is usually developed using system called Host system such as desktop PC or laptop. The reason is that the embedded systems generally don’t have the hardware such as keyboard, monitor, high capacity memory for embedded software development.
- Using tool chain such as text editor, cross compiler, linker / locator, object code to Hex file convertor, embedded ‘C’ code can be converted into target specific processor / controller Hex file which then dumps into target processor / controller memory via serial link.

4.2.3 Using ARM GCC Tool Chain

- **Tool chain** is series of development tools for creation of any software product. It may involve compilation, attaching libraries and other objects, debugging etc.
- Most of the Linux systems have their common native tool chain. A native tool chain on your x86 platform will generate code for x86.
- For embedded system development, native tool chain will not benefit us. We will probably be using our Linux x86 platform to generate code for ARM development board (cross-compilation).
- A specific tool chain needed for such cross compilation is called **cross-compiling tool chain.** This will run on out native platform but will generate code for target platform.

- Fig. 4.2.2 shows a Linux x86 platforms as our development or compiling machine. This machine will be used to compile our source code program and generate code for both :
  1. Native code which will run on same platform (x86)
  2. Target code for ARM development board
- The Linux native tool chain in our x86 platform will suffice the need of generating native code.
- A cross compiling tool chain installed in our native machine (compiling machine), is used to generate the code for ARM.
- One can build their own tool chain as per their requirement but it is tedious task and require significant amount of time. We generally used pre-compiled tool chain.

For Example : Linaro tool chain for ARM.

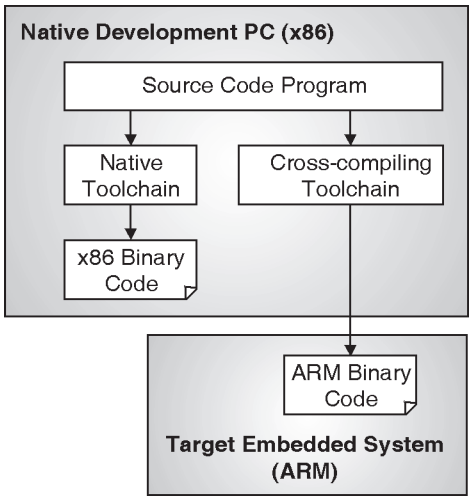


Fig. 4.2.2 : Tool chains generating binary for native x86 and target ARM

4.3 Speed of Code or Optimization of Codes

Embedded application software using C++ can be optimized using following techniques.

1. Declare classes as private if possible. It helps in optimizing the generated codes.
2. Use char, int and boolean scalar data types in place of objects (user defined data types) as arguments and use local variables as much as possible.
3. Releasing memory used by unused object with NULL pointer assignment to them.

Host and target systems

- In the embedded world there are many reasons to do actual programming work on a system other than one on which the software will eventually run.
- The embedded system may or may not have a keyboard, a video display unit (screen), a secondary storage drive, and other peripherals necessary for programming.
- It may not have enough memory to run a programming editor, or its programming editor may not be available for processor / controller uses in embedded system.

- Therefore, the programming work for embedded system is done on computer system called as host system on which all the programming tools run.
- Only after the program has been written, compiled, linked an Hex file is generated. Hex file is then dumped into the program memory of embedded system called as target system via ethernet or serial link. Fig. 4.3.1 shows embedded software development system.

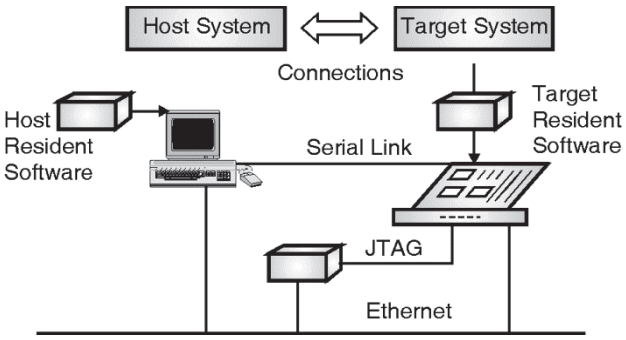


Fig. 4.3.1 : Embedded software development system

4.4 Source Code Engineering Tools for Embedded C / C++

- For embedded software development, the source code engineering tool is required for program writing, compiling, testing and debugging of embedded system software. SNiFF+ is source code engineering tool for C and C++ from Windriver® systems.
- SNiFF+ PRO version has full SNiFF+ code as well as debug module. Main features of this tool are as follows.
  1. It searches and provides the information about the dependencies of symbols and defined symbols, variables, functions.
  2. It monitor, configure (enable or disable) the implementation of virtual functions.
  3. Virtual functions are opted for dynamic run-time binding.
  4. It facilitate automatic search and replacement for given information.

4.5 Integrated Development Environment (IDE)

- IDE is a toolkit to write and execute a program. The most widely used IDE with the C programs is the Turbo C IDE. This IDE contains of five tools namely : Editor, Preprocessor, Compiler, Linker and Loader.
- **Editor :** It is used to type and edit the program.
- **Preprocessor :** It is responsible for performing certain operations before the compiling, like including header files, declaring constants, etc. It always begins with the '#' sign.

  For Example #include<stdio.h> #define pi 3.14;
- **Compiler :** It is used to convert the C program to machine language. As the processor cannot understand the language of C, we need to convert it into a machine understandable language i.e. machine language.
- **Linker :** It is responsible for linking (connecting) the different functions in a program. We may write multiple functions in the program or conditional/loop operations in our program. These kinds of operations are to be handled by the linker.

- **Loader** : The binary file generated by the compiler is loaded by the loader and the loader also gets this file executed by the processor

An IDE must have the following features

1. It has a facility for selecting a processor family as well as it's version (family member). It has source code engineering tools which integrates the editor, compiler for C, embedded C++, assembler, linker, locator, logic analyser etc. into a single package.
2. It provides the flexibility to select a user-definable assembler to support a new version or type of processor.
3. The design process divides the project (task) into number of subtasks. Each programmer is assigned independent but linked subtask.
4. It has ability to simulate hardware unit like emulator, peripherals and input / output devices on a host system.
5. It provides the single stepping debug facility. It has facility for synchronizing the internal peripherals with system clock.
6. It provides the multi windows display for source code with labels and symbolic arguments, registers window, the detailed information of the status of peripherals devices, status of RAM and ports, stack, program memory etc.
7. It evaluates the performance of a target system. It has inbuilt emulator with development system that remains independent of a particular targeted system and a logic analyzer for capturing up to 256 or more transitions on the address and data buses after triggering.

4.6 Optimization of Memory Needs

Fig. 4.6.1 shows techniques used for memory optimization in embedded system.

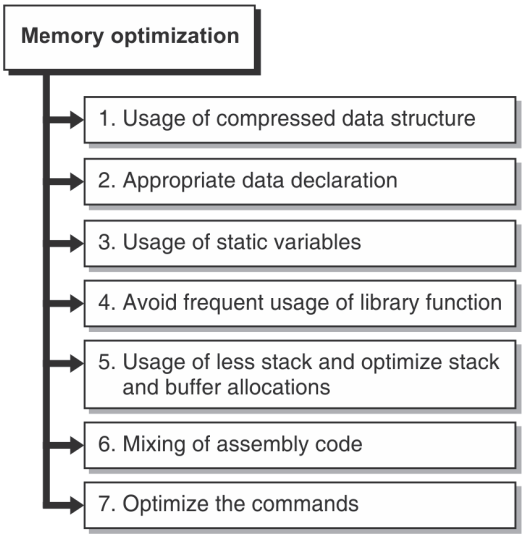


Fig. 4.6.1 : Memory optimization

1. Usage of compressed data structure

Complex data structure can be used when decompression algorithm and compressed data structure together takes less amount of the memory in embedded system compare to uncompressed data structure.

2. Appropriate data declaration

The declaration of the variables should be with suitable data types so as to save memory. For example, if a variable will not have a value less than zero and more than 255, then it should be declared as unsigned char instead of int.

3. Usage of static variables

For local variable the compiler use the local stack. Use static (global) variables if shared data problems are resolved and use static variable that use reserve memory and stops the frequent use of local stack.

4. Avoid frequent usage of library function

Try to avoid the use of library functions if a simpler coding is possible.

5. Usage of less stack and optimize stack and buffer allocations

If possible try to reduce use of frequent function calls and nested calls. Allocated stack and buffer are first filled with the specific bytes. Then check the how many filled stack locations, do not check for worst cases of running of embedded system. Reduce the allocated size of stack and buffer for finalized code.

6. Mixing of assembly code

As discussed some operations happen better in assembly compared to high level languages. In such cases the two programming languages can be mixed.

7. Optimize the commands

Try to combine the two functions if possible when both functions have more or less similar codes. If feasible, use array of pointers to the functions instead of switch statements. C++ programming for embedded system avoid the use of multiple inheritance, template, runtime identification and throwable exceptions with classes.

4.7 Basic C Data Types

- Before going for programming, let us see a very important thing required for programming i.e. the tokens of a programming language.
- The C programs are made up of different things termed as tokens.
- The basic C data types are shown in Fig. 4.7.1.

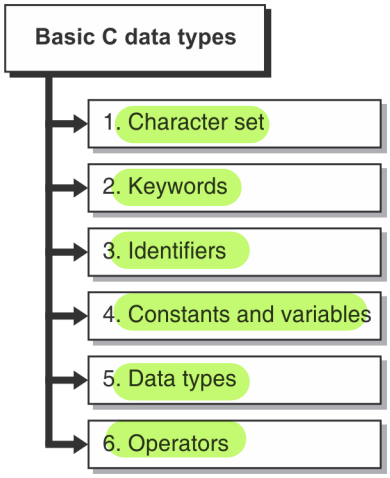


Fig. 4.7.1 : Basic C data types

We will see these tokens in the subsequent sub-sections.

4.7.1 Character Set of C

- When we will be writing C programs, all these will be found in our program. Table 4.7.1 gives a list of C character set.

Table 4.7.1 : Character Set of C

Sr. No.	Characters	List included
1.	Alphabets (Upper case and lower case)	A, B, C .....Z a, b, c, ...z
2.	Digits (numbers)	0,1,2,....9
3.	Special symbols (all those seen on a keyboard, nothing besides that)	< > { } ( ) [ ] , . ; : ! ? ' " / + * = % & # @ | \ ~ ` $ ^ _ - (The names used for these symbols are given in the Table 4.6.2)
4.	Other special characters	Blank Space, Tab, Carriage Return (Enter Key)

Table 4.7.2 : Special Symbols and their names

Symbol	Name of Symbol	Symbol	Name of Symbol
,	Comma	&	Ampersand
.	Period or dot	*	Asterisk
;	Semicolon	-	Minus Sign
:	Colon	+	Plus Sign
?	Question Mark	<	Opening Angle (Less than sign)
!	Exclamation Mark	>	Closing Angle (Greater than sign)
|	Pipe	(	Left Parenthesis
/	Forward Slash	)	Right Parenthesis
\	Back Slash	[	Left Square Bracket
'	Single Quotes	]	Right Square Bracket
"	Double Quotes	{	Left Brace
~	Tilde	}	Right Brace
_	Underscore	#	Hash
^	Caret	$	Dollar
%	Percentage		

Each of these special symbols may have some significance which will be discussed in this chapter.

4.7.2 Keywords

- These are some special words that have a predefined meaning for the C compiler. Hence, these words cannot be used as identifiers (identifiers are discussed in Section 4.7.3).
- These are a set of words which are reserved for the certain operations and hence are also sometimes referred as reserved words.
- All keywords are in lower case.
- The keywords used in C are as shown in Table 4.7.3.

Table 4.7.3 : Keywords used in C

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_packed
double			

- These keywords will be used in different places in programming. Their significance and use will be studied wherever required in this chapters.
- These keywords include all those keywords also as declared by ANSI (American National Standards Institute) C. This institute has published the standards to be used in C programming language.

4.7.3 Identifiers

Identifiers are names given to different user defined things like variables, constants, functions, classes, objects, structures, unions, etc. While making these identifiers we need to follow some rules. These rules are stated below :

1. The identifier can consist of alphabets, digits and a special symbol i.e. '_' (underscore).
2. An identifier cannot start with a digit. It can start either with an alphabet or underscore.
3. It cannot contain any special symbol except underscore. Blank spaces are also not allowed.
4. It cannot be a keyword.
5. It is case sensitive i.e. an alphabet capital in one identifier with same name in another identifier with that alphabet small case will be considered different (for more details see examples in this section).
6. Earlier; there was a limit of the length of the identifier to be 32 characters, but now this limit is removed. Hence, an identifier can be as long as required and minimum of one character.

Ex. 4.7.1 : A list of valid and invalid identifiers is given below with reasons wherever required.

1. simple_interest: Valid
2. char : Invalid, because it is a keyword
3. 3friends: Invalid, because starts with a digit
4. _3friends: Valid
5. Simple interest: Invalid, because blank spaces are not allowed
6. #3friends: Invalid, because no special symbol except underscore is allowed.
7. void: Invalid, because keyword not allowed.
8. void: Valid, case sensitive.

4.7.4 Scalar Data Types in C

- The data type decides the type of data and the memory locations required for storing that type of data in the memory.
- The data types of C can be divided into three types: Primitive, Derived and User defined data types.
- The different primitive types of data that can be used in C are integer, character, fraction type numbers, etc.
- Table 4.7.4 shows the different primitive data types and the memory space required for storing them.

Table 4.7.4 : Data types in C

Sr. No.	Data type	Type of data to be stored	Range	Space required in memory (bytes)
1.	char	1 character in ASCII form	– 127 to 128	1
2.	signed char	1 character in ASCII form	– 127 to 128	1
3.	unsigned char	1 character in ASCII form	0 to 255	1
4.	int	Integer nos.	– 32768 to 32767	2
5.	signed int	Integer nos.	– 32768 to 32767	2
6.	unsigned int	Integer nos.	0 to 65535	2
7.	short int	Integer nos.	– 32768 to 32767	2
8.	long int	Integer nos.	– 2147483648 to 2147483647	4
9.	signed short int	Integer nos.	– 32768 to 32767	2
10.	signed long int	Integer nos.	– 2147483648 to 2147483647	4
11.	unsigned short int	Integer nos.	0 to 65535	2
12.	unsigned long int	Integer nos.	0 to 4294967296	4
13.	float	Fraction nos.	$3.4e^{-38}$ to $3.4e^{38}$	4
14.	double	Fraction nos.	$1.7e^{-308}$ to $1.7e^{308}$	8
15.	long double	Fraction nos.	$3.4e^{-4932}$ to $1.1e^{4932}$	10

Ex. 4.7.2 : Some examples of data to be stored are listed below with the data types that will be best suited for them

1. Age in years: int type data must be used, as years is an integer value.
2. Rate of interest: float data must be used, as rate of interest is a fraction type value.
3. Alphabet : char type of data is to be used, as a character is to be stored.
4. Principal amount: float type data must be used, as principal is in rupee and paisa.
5. Runs made by a batsman: int type data must be used, as no. of runs will always be integer.
6. Factorial of a number: int type data must be used, as the factorial is always an integer.
7. Radius of a circle: float type data must be used, as radius will mostly be a fraction number.
8. Area of a circle: float type data must be used, as area is radius2 X 3.14, which has to be fraction number.
9. User input as 'Y' or 'N': char type data must be used, as there is only a character to be taken from the user as yes or no. Boolean type of data may also be used for this.

Note : The arithmetic operations can be done even on char type of data. This is because the processor stores ASCII (American Standard Code for Information Interchange) to store the characters.

- The ASCII value for capital 'A' is 65, 'B' is 66, 'C' is 67 and so on. While the ASCII values for small alphabets are 97 for 'a', 98 for 'b', 99 for 'c' and so on.
- The user defined data types are structure, union, class and enumeration. The derived data types are array, function, pointer and reference.
- The derived and user defined data types will be studied in the subsequent chapters wherever it is needed.

4.7.5 Constants and Variables

- Constants are values given to the identifiers that do not change their values throughout the execution of the program.
- Constants can be defined either by writing the keyword const before the data type or by using #define.
- Constants are used to declare the values that remain constant, for example value of pi.
- The use of constants in program will be discussed later wherever required.
- Variables are values given to identifiers that can change their values during the execution of the program.

Ex. 4.7.3 : A variable can be defined with the data type as shown in the examples below :

1. For declaring age as an integer type data the syntax (grammar or method of writing) is: int age;
2. For declaring rate of interest as float data type the syntax is: float rate;
3. For declaring a character data type element the syntax is: char x;

4.7.6 Escape Sequences

- Escape sequence is a character followed by a backslash (\).
- They are used especially to perform some special operations like going to new line, providing a horizontal tab, vertical tab etc.
- The following is a list of escape sequences.

1. \n Newline
2. \t Horizontal Tab
3. \v Vertical Tab
4. \b Backspace

5. \r Carriage Return
6. \f Form feed
7. \a Audible Alert (bell)
8. \\ Backslash
9. \? Question mark
10. \'Single quote
11. \" Double quote

4.7.7 Unary Operators

These are operators that require only one operand. Let us discuss these operators one by one.

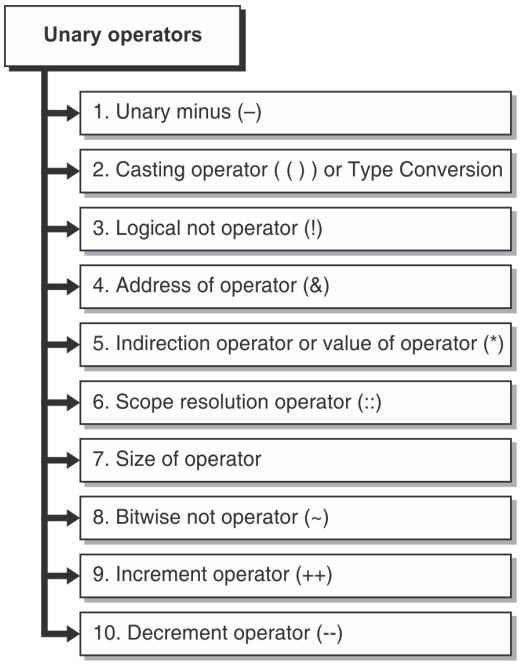


Fig. 4.7.2 : Unary operators

1. Unary minus (–)

- The symbol shown in the bracket is used as unary minus operator.
- It returns the negative value of the variable to which it is preceded.
- For Example if x = 3 and y = 6 then

  y = – x;

  will make the value of y as – 3.

2. Casting Operator (( )) or Type Conversion

- It is many a times required to convert one data type to another.
- The casting operator is used for type conversion i.e. it can be used to convert a data of one type to another data type.
- For Example if we have int x=3;

  float y=5.6;

  then the statement, x=(int) y;

  will result in the value of x as 5.
- We will see the use of this operator in the programming

3. Logical Not operator (!)

- The symbol shown in the brackets i.e. the exclamation mark is used as a logical not operator. It is used to check certain conditions in a condition statement. It performs the logical not of the result obtained from the expression on its right.
- For Example if x=1, then y=!x; will result with y having the value 0.
- We will see some more logical not operator based expressions and program followed by this section.

4. Address of operator (&)

- This operator returns the address of the variable associated with it. It will be studied more deeply in the chapter on Pointers.
- For Example if we write y=&x;

  then the memory address allocated to the variable x will be copied into the variable y. For this the variable y must be a pointer variable.

5. Indirection operator or value of operator (*)

- This operator returns the value of the data at the given address. This operator will also be studied in more details in the chapter on pointers.
- For Example z=*y;

  will give the value of the data stored at the address given by y.

6. Scope Resolution operator (::)

- It resolves the scope of a variable to be inside the function or outside the function.
- For Example if an external variable x is to be accessed then the same can be accessed using the scope resolution operator by the statement, y=::x;

7. Size of operator

- This operator is used to know the size of a variable as required to store its value in memory. It can also be used to find the size of a data type.
- As seen in the Section 4.7.4, the space required to store different data type is different ranging from 1 byte to 10 bytes.
- For Example the statement sizeof(int);

  will return the value 2, as int requires 2 bytes.
- Another Example : int x,y;

  char a;

  x=sizeof(a);

  y=sizeof(float);

  then x will have 1, and y will have 4

8. Bitwise not operator (~)

- This operator is used to perform bitwise NOT operation. We have also seen some examples of NOT operations.
- The bitwise NOT operator can be used to perform binary NOT operation. This operation can be performed by using the operator given in brackets i.e. ~.

- For Example If x = 3,

  then y=~x;

  will result in y having a value of -4

9. Increment Operator (++)

- It returns the value of the variable added with one and stores the result in the variable itself.
- For Example if x=5,

  then x++;

  will make the value of x equal to 6.
- This "x++;" (also called as post increment operator) can also be written as "++x;" (also called as pre increment operator).
- It can also be used to store the result in another variable. But in this case the post increment and pre increment statements will have different behavior as explained below with examples.
- In post increment case, For Example if x=5,

  then y=x++;

  will make the value of y equal to 5 and x equal to 6.
- As the name says post increment, it first gives the previous value and then increments.
- In pre increment case, For Example if x=5,

  then y=++x;

  will make the value of y equal to 6 and x equal to 6.
- As the name says pre increment, it first increments and then gives the incremented value.
- More such examples will be seen with programs.

10. Decrement Operator (--)

- It returns the value of the variable subtracted with one and stores the result in the variable itself.
- For Example if x=5,

  then x--;

  will make the value of x equal to 4.
- This "x--;" (also called as post decrement operator) can also be written as "--x;" (also called as pre decrement operator).
- It can also be used to store the result in another variable. But in this case the post decrement and pre decrement statements will have different behavior as explained below with examples.
- In post decrement case, For Example if x=5,

  then y=x--;

  will make the value of y equal to 5 and x equal to 4.
- As the name says post decrement, it first gives the previous value and then decrements.
- In pre decrement case, For Example if x=5,

  then y=--x;

  will make the value of y equal to 4 and x equal to 4.
- As the name says pre decrement, it first decrements and then gives the decremented value.

4.7.8 Binary Operators

GTU - S-16, W-19

Q.	Write a short note on Binary operators in C language.	**(S-16, 7 Marks)**
Q.	Write a note on Arithmetic operators in C language.	**(W-19, 3 Marks)**

- Operators that require two operands are called as binary operators.
- These operators are further classified into various types namely the Arithmetic operators, Logical operators, Bitwise operators and Relational operators.
- We will see these operators one by one in this section.

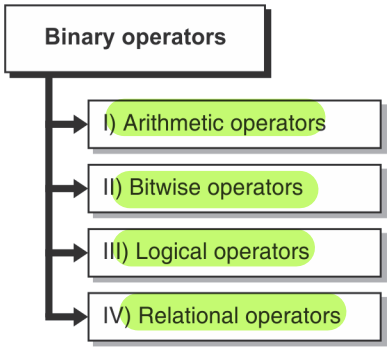


Fig. 4.7.3 : Binary operators

(I) Arithmetic operators

- This set includes the basic arithmetic operators to perform basic arithmetic operations like addition, subtraction, multiplication and division. There are five operators in this set. They are:
  1. * to find the product
  2. / to find the quotient after division
  3. % to find the remainder after division
  4. + to find the sum
  5. – to find the difference
- One important thing to be noted here is that the '/ ' operator returns only the quotient, while the '%' operator (also called as MOD operator) returns the remainder after division.
- MOD operator is possible only for int or char type of data. It doesn't work on float and double type of data.
- For example of each of these operators
  1. 2 * 2 = 4;
  2. For int type of data, 5 / 3 =1;

     For float type of data, 5 / 3 = 1.67
  3. 5 % 3 =2;
  4. 2 + 2 = 4;
  5. 3 – 2 = 1;

(II) Bitwise operators

- These operators work bitwise on a data.
- They perform different operations on bits of a data like AND, OR, EXOR and NOT.
- The operators are listed below :
  1. ~ to perform bitwise NOT operation.
  2. & to perform bitwise AND operation.
  3. | to perform bitwise OR operation.
  4. ^ to perform bitwise EXOR operation.
  5. << to perform bitwise left shift operation.
  6. >> to perform bitwise right shift operation.
- These operators are use to perform bitwise binary operations on the data.
- As we have addition, subtraction operations in decimal data for performing arithmetic operations; similarly AND, OR, NOT are basic bitwise operations on the binary data.
- The result of these operations can be understood from the following examples :

1. 5 & 3 = 1

$(5)_{10}$ = (0 1 0 1)$_2$
$(3)_{10}$ = (0 0 1 1)$_2$
(0 0 0 1)$_2$ = $(1)_{10}$

2. 12 | 9 = 13

$(12)_{10}$ = (1 1 0 0)$_2$
$(9)_{10}$ = (1 0 0 1)$_2$
(1 1 0 1)$_2$ = $(13)_{10}$

3. 8 ^ 10 = 2

$(8)_{10}$ = (1 0 0 0)$_2$
$(10)_{10}$ = (1 0 1 0)$_2$
(0 0 1 0)$_2$ = $(2)_{10}$

4. ~7 = – 8

$(7)_{10}$ = (0 1 1 1)$_2$
(1 0 0 0)$_2$ = $(-8)_{10}$

(According to C , wherein it will take more no. of bits and hence -8 will be the result. Hence; in general the ~x is always equal to – (x+1).

5. 10 << 2 = 40

Assuming the data to be char i.e. 8 bit data

$(10)_{10}$ = (0 0 0 0 1 0 1 0)$_2$

After shifting left once (0 0 0 1 0 1 0 0)$_2$

After shifting left for the second time

(0 0 1 0 1 0 0 0)$_2$ = $(40)_{10}$

Note : When shifting to left, each of the bit is shifted left. The first bit is lost and the last bit is inserted as 0.

6. 13 >> 3 = 1

Assuming the data to be char i.e. 8 bit data

$(13)_{10} = (0\ 0\ 0\ 0\ 1\ 0\ 1\ 1)_2$

After shifting left once	$(0\ 0\ 0\ 0\ 0\ 1\ 0\ 1)_2$
After shifting left for the second time	$(0\ 0\ 0\ 0\ 0\ 0\ 1\ 0)_2$
After shifting for the third time	$(0\ 0\ 0\ 0\ 0\ 0\ 0\ 1)_2 = (1)_{10}$

Note : When shifting to right, each of the bit is shifted right. The first bit is inserted as 0 and the last bit is lost.

III) Logical operators

- Logical operators follow the same truth table as for the bitwise operators; but they are used to check conditions instead of performing operations on a data.
- The logical operators are AND and OR. The symbols used for these operators in C are && and || respectively.
- For example a statement

  y > 5 && y < 10;

  will result in "true" i.e. 1 if the value of y is greater than 5 AND less than 10, else it will result in false i.e. 0.
- Another example a statement

  y > 5 || y==2;

  will result in "true" i.e. 1 if the value of y is greater than 5 OR equal to 2, else it will result in false i.e. 0.
- Logical operators will be understood in more details with the expressions and program examples followed by this section.

IV) Relational operators

- The operators like '<' (less than) and '>' (greater than) used in the above examples are relational operators. We have seen the example also of the "==" (equality operator) in the above logical operators.
- A list of all the relational operators is given below :
  1. == used to check if the two things are equal.
  2. != used to check if the two things are not equal.
  3. < used to check if the first data is less than the second one.
  4. > used to check if the first data is greater than the second one.
  5. <= used to check if the first data is less than or equal to the second one.
  6. >= used to check if the first data is greater than or equal to the second one.
- The examples of these relational operators are as shown with the logical operators above.

4.7.9 Ternary Operator

- An operator that requires three operands is called as a ternary operator.
- There is only one ternary operator in C. This operator is used to check a condition and accordingly do one of the two things based on the condition being true or false.

- The syntax (way of writing) of this operator is as given below :

  (condition) ? <value if condition is true> : <value if condition is false>;
- Hence; as shown in the syntax, first the condition is to be written in brackets followed by a question mark (?). Then the operation that is to be performed if condition is true and then a colon (:) followed by the operation to be performed if the condition is false.
- For example :

  z = (x>y)?x:y;
- This statement will put the value of x into z if the given condition i.e. x > y is true. Else; the value of y will be put into z. Hence, the value of the greater variable will be put into z.
- A slightly complicated use of this operator is to find the greatest of three numbers as shown in the example

  g=(x>y)?((x>z)?x:z)):((y>z)?y:z);
- This statement will give the largest of x, y and z into the variable g.

4.7.10 Assignment Operators and Statements

- These operators are used to assign the value of the expression or variable on the right of the assignment operator to the variable on its left.
- The simple assignment operator is '='. But there are some more assignment operators called as composite assignment operators.
- The different assignment operators are as listed below :

  1. **=** : This operator assigns the value of the expression or variable on its right to the variable on its left. For Example y=x+2;
  2. **+=** : This operator adds the variable on its left and right and the result is put into the variable on its left. For Example y+=x; is same as y=y+x;
  3. **– =** : This operator subtracts the variable on its right from the variable on its left and the result is put into the variable on its left. For Example y - =x; is same as y=y-x;
  4. ***=** : This operator multiplies the variable on its left and right and the result is put into the variable on its left. For Example y*=x; is same as y=y*x;
  5. **/=** : This operator divides the variable on its right from the variable on its left and the result is put into the variable on its left. For Example y/=x; is same as y=y/x;
  6. **%=** : This operator finds the remainder by dividing the variable on its right from the variable on its left and the result is put into the variable on its left. For Example y%=x; is same as y=y%x;
  7. **&=** : This operator ANDs the variables on its left and right and the result is put into the variable on its left. For Example y&=x; is same as y=y&x;
  8. **|=** : This operator ORs the variables on its left and right and the result is put into the variable on its left. For Example y|=x; is same as y=y|x;
  9. **^=** : This operator EXORs the variables on its left and right and the result is put into the variable on its left. For Example y^=x; is same as y=y^x;
  10. **<<=** : This operator shifts in left direction the variable on its left for the number of times indicated by the variable or value on right and the result is put into the variable on its left. For Example y<<=x; is same as y=y<<x;

11. **>>=** : This operator shifts in right direction the variable on its left for the number of times indicated by the variable or value on right and the result is put into the variable on its left. For Example y>>=x; is same as y=y>>x;

4.7.11 Selection Operators

- These operators are used to select certain element of a set of elements. Here, we will make a list of these operators and see the detailed use of these operators when we study the corresponding topic where these operators are required.
- The different operators in this set are listed below:
  1. **[]** : This operator is used to select an element of Array.
  2. **.** : This is called as period operator and is used to select an element of a structure or union.
  3. **->** : This is used to select an element of a structure or union pointed by a pointer.
  4. **( )** : This is called as a function call operator and is used to call or select a function.
  5. **,** : The comma (,) operator is used to separate the different values etc.

4.8 C Looping Structures

4.8.1 if-else Selective Statement

- This is a very important statement used to check the condition and accordingly execute a set of statements, based on whether the condition is true or false.
- Hence it is called as selective statement. It selectively executes some statements and doesn't executes some.
- The syntax of the if-else condition is as given below

```
if(condition)
{
      -
      -
      statements1;
      -
      -
}
else
{
      -
      -
      statements2;
      -
      -
}
```

- The set of statements named as statements1 in the above syntax are executed if the condition given with the if statement is true.

- The set of statements statements2 are not executed in this case.
- If the condition specified in the if statement is false then the statements named as statements2 in the above syntax are executed.
- The set of statements statements1 are not executed in this case.

Note : The else part is optional i.e. we can have a if statement without the else statement. As seen in the above given syntax, only the first half i.e. we can have if statement as shown below.

```
if (condition)
{
      -
      -
      Statements
      -
      -
}
 will be there
```

- In the previous section we have seen the if-else ladder. A better solution of this is switch-case. Using switch-case the if-else ladder can be implemented in a much better way.
- The syntax of switch-case is given below :

```
switch(expression / variable)
{
case label1: statements;
          break;
case label2:statements;
          break;
case label3:statements;
          break;
      |
      |
case labeln:statements;
          break;
default : statements;
}
```

Note : Break statement transfers the control outside the current loop.

- The expression or variable can be given in the brackets associated with the switch statement. The values of this expression / variable are the labels associated with the cases inside the switch statement.
- The value of the expression / variable is first compared with the label1. If they are equal, then the statements followed by the corresponding case are executed. The break statement followed by these statements, transfers the control after the switch statement.
- If the expression / variable is not equal to label1, then it is directly compared to label2 without executing the statements followed by the label1.

- Hence the statements of only that case are executed with the correct label value of the expression / variable.

Note :

1. Break statement is not compulsory. But if the break statement is nut written everything will be executed after the case statements
2. Default is not necessary. But in case if default is not written and some case occurs which is not considered then there will be no operation performed and the user will not be able to realize the problem in the program.
3. The label can only be value, it cannot be a condition or an expression.
4. The brackets associated with the switch statement can have either the variable or expression and no condition.

4.8.2 Looping Statements

- In programming we need to sometimes control the flow of operation other than just the sequential statements. In this case we need the control statements
- Control statements are classified into two types viz. the **iterative statements** and **conditional statements**.
- **Iterative statements** are use to perform certain operation repetitively for certain number of times. In C we have three iterative statements viz. for loop, while loop and do-while loop. Iterative statements are also called as repetitive statements as they repeat a set of statements for a given number of times.
- **Conditional statements** are use to perform the operations based on a particular condition i.e. if a condition is true perform one task else perform another task. In C we have two conditional statements namely if-else statements and switch-case statements. Conditional statements are also called as selective statements i.e. these statement select a particular statement to be executed based on the condition.
- In the subsequent sections we will see all the iterative statements one by one in detail. We will be writing only the Pseudo code type algorithms from here onwards. For some program the stepwise algorithm will be given for ease of students to understand the program.

4.9 C- Control Structures for Iteration

4.9.1 for Loop

- for is a iterative statement. It is used to repeat a set of statements number of times. The syntax (method of writing) **for** statement is given below :

```
for(initializations; condition; increment / decrement / updating)
        {
            -
            -
            -
```

```
        statements;
        -
        -
        -
    }
```

- The sequence of execution of the for loop is such that the **initialization statements are executed first**. These initialization statements are executed only once. They are used to initialize the values of the variables.
- The **second step** is **checking the condition** specified. There can be only one condition. If more than one conditions are required they can be combined using the logical AND, OR operators. If the condition is false the execution of the loop will be terminated i.e. the execution will go outside the braces of for loop, if the condition is not true.
- The **third step** is to **execute all the statements** inside the curly braces. These statements will be executed sequentially. The number of statements can be of any count. There can be another control statement if required inside one control statement.
- The **fourth step** is the **increment / decrement** or updating operations. These operations are not necessarily increment or decrement operations, but mostly these are increment decrement and hence called so. We can update the variables over here before starting the next iteration of the iterative statements.
- **Finally the control goes back to the second step**. As said the first step is executed only once, the steps that are repeated continuously are the second, third and fourth steps. After the fourth step the condition is checked again. If the condition is true the execution continues, else the control goes outside the for loop i.e. the curly braces.
- while and do-while loops are also for repetitive operations.
- The operations are slightly different than the for loop, but the same operations can be implemented by for, while or do-while loops. Although there is one major difference in a do-while loop wherein one particular operation cannot be implemented. This will be discussed later in this section. The syntax of while loop is given below :

```
while(condition)
{
    -
    -
    statements;
    -
    -
    -
}
```

- The operation of the while loop is such that, first the condition is checked. If the condition is true, then the statements are executed. Once the statements are executed, the condition is again checked and this keeps on repeating, until the condition is false. If the condition is false, the statements inside the loop are not executed, instead the control directly comes out of the loop.

- The syntax of do-while loop is given below :

```
do
{
    -
    -
    statements;
    -
    -
    -
} while(condition);
```

- One major point to be noted is that in case of do-while loop, the statements are executed atleast once even if the condition is not true for the first statement. On the other hand, in case of for loop and while loop, even for the first time the statements are executed only if the condition is true.
- The implementation of a for loop converted to while and similarly do-while loop is shown in Fig. 4.9.1(a) and 4.9.1(b) respectively.

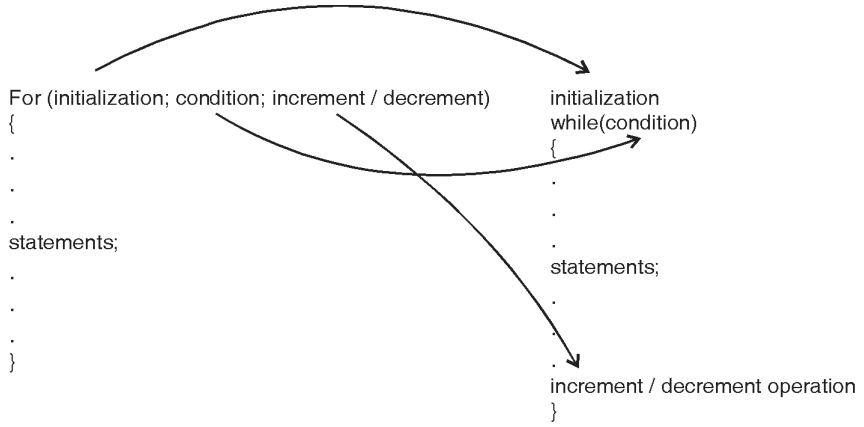


Fig. 4.9.1(a) : Implementation of a while loop related to for loop

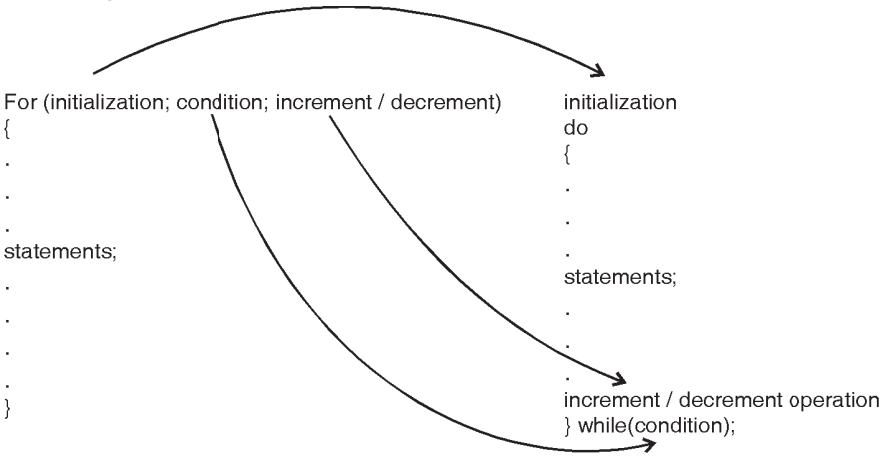


Fig. 4.9.1(b) : Implementation of a do-while loop related to for loop

- You will note (in Fig. 4.9.1) that the initialization statements are removed outside the loop, the condition is in the brackets associated with the while part and increment decrement are to be inside the loop at the end so as to implement a similar operation as that in the for loop.
- Differences between the while and do-while loop is given in Table 4.9.1.

Table 4.9.1 : Difference between while and do-while loop

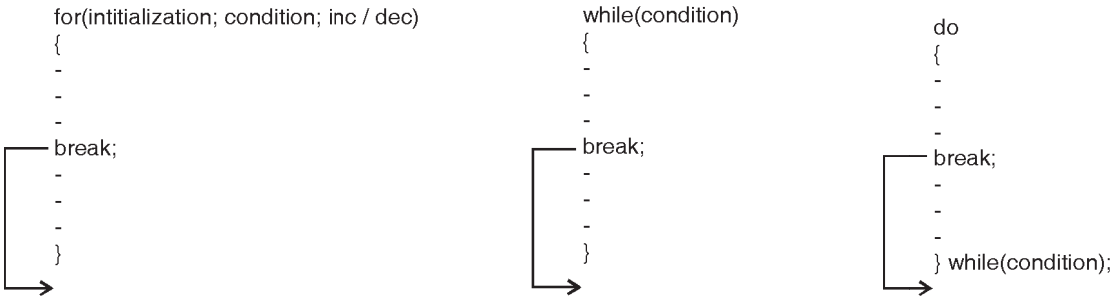
Sr. No.	while loop	do-while loop	for loop

1.	Syntax of while loop:
	while(condition)
	{
	-
	-
	statements;
	-
	-
	-
	}
		Syntax of do-while loop:
		do
		{
		-
		-
		statements;
		-
		-
		-
		} while(condition);
			Syntax of for loop:
			for (initializations; condition; updations)
			{
			-
			-
			statements;
			-
			-
			-
			}

2.	This is called as a entry controlled loop, as the entry inside the loop is possible only if the condition is true.	This is called as exit controlled loop, as the entry inside this loop is sure i.e. no condition is checked to enter inside the loop. Nut the exit is possible only if the condition is false.	This is called as a entry controlled loop, as the entry inside the loop is possible only if the condition is true.
3.	If the condition is not true for the first time, the control will never enter into the loop. Hence there is a possibility that the control never enters into the loop and the statements inside the loop are never executed.	Even if the condition is not true for the first time the control will enter into the loop. Hence the statements inside the loop will be executed atleast once even if the condition is not true for the first time.	If the condition is not true for the first time, the control will never enter into the loop. Hence there is a possibility that the control never enters into the loop and the statements inside the loop are never executed.
4.	There is no semicolon (;) after the condition in the syntax of the while loop	There is a semicolon (;) after the condition in the syntax of the do-while loop.	There is no semicolon (;) after the condition in the syntax of the for loop
5.	The initializations and the updating is not a part of the syntax.	The initializations and the updating is not a part of the syntax.	The initializations and the updating is a part of the syntax.

- These statements transfer the control to a different place in the program. We will see the operation of each of these. We have already seen the use of break statement in switch-case. Let us see the use of continue and break in other loop statements

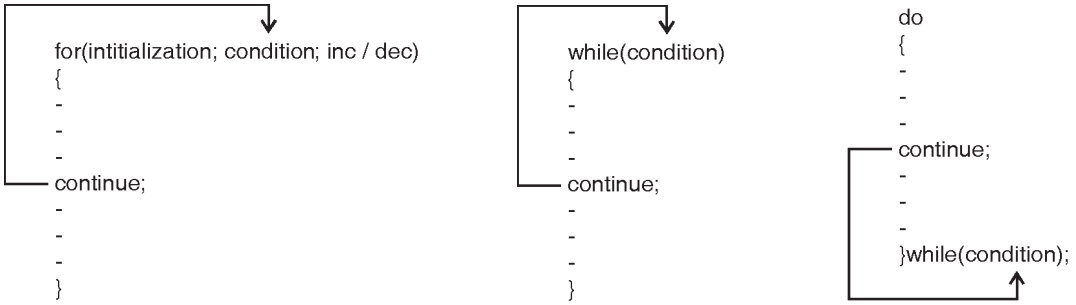
- The break statement neglects the statements after it in the loop and transfers the control outside the loop as shown in Fig. 4.9.2. The Fig. 4.9.2 shows the operation of break statement in each of the loop statements i.e. for, while and do-while statements.



(a) Operation of break statement in a for loop **(b) Operation of break statement in a while loop** **(c) Operation of break statement in a do-while loop**

Fig. 4.9.2

- The continue statement also neglects the statements after it in the loop and transfers the control back to the starting of the loop for next iteration. The Fig. 4.9.3 shows the operation of continue statement in each of the loop statements i.e. for, while and do-while statements.



(a) Operation of continue statement in a for loop **(b) Operation of continue statement in a while loop** **(c) Operation of continue statement in a do-while loop**

Fig 4.9.3

- The goto statement transfers the control to the label specified with the goto statement. A label is any name given to a particular part in the program. The label is to be followed with a colon (:).
- The syntax of a goto statement is as shown below :

```
-                          -
goto label1                label2:
-                          -
-          or              -
label1:                    -
-                          goto label2
-                          -
```

- Thus it shows that the label can be before or after the goto statement and hence the control can be transferred to earlier or next statements using a goto statement.

- Let's consider a typical example for a looping statement in embedded C application. Assume that we have a device. We want to read the various registers of it. We can read the content of various registers only if device is ready and that indicate with the help of it's STATUS register.

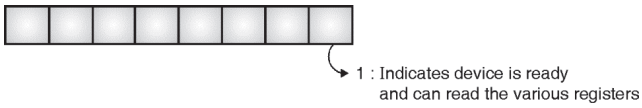


Fig. 4.9.4 : Format of STATUS register

We can read the various registers of device using various loop statement.

```
// using while loop
while (STATUS ! = 0 × 01) ; // Wait till STATUS register = 0 × 01

// using do while loop
do
{
} while (STATUS ! = 0 × 01) ; // Loop till STATUS register = 0 × 01

// using for loop
for (; STATUS ! = 0 × 01 ;) ; // Repeat until STATUS register = 0 × 01
```

4.10 Register Allocations

- The compiler allocates not only memory locations but also registers to the variables declared in the program.
- The memory locations have only one disadvantage that the access time i.e. the read time and write time are more than those for registers.
- The registers have this as the advantage i.e. faster access. But the disadvantage for the registers is that there is a limited number of registers and hence there will be a limited number of variables that can be declared as register type.
- This limitation is not much in case of memory. Thus there should be care taken that:
  1. variables required frequently are stored in registers to increase the speed and
  2. less number of variables must be used i.e. the same variables must be used again once their previous work is over.
- ARM registers are normally used by the compiler as shown in the Table 4.10.1.

Table 4.10.1

Register Name	Alternate function	Remarks
R0	A0	Argument registers (A0 to A3): These registers are normally used to pass the parameters to the function called, hence are called as argument registers
R1	A1	
R2	A2	
R3	A3	

Register Name	Alternate function	Remarks
R4	V0	General Purpose Variable Registers (V0 to V7): These registers as the name says are used to store general purpose or local variables for a function.
R5	V1	
R6	V2	
R7	V3	
R8	V4	
R9	V5	
R10	V6	
R11	V7	
R12	IP	IntraProcedure (IP) : It is used to store some value for an intraprocedure call, that can be preserved even in case of call to the same function without affecting the value.
R13	SP	Stack Pointer(SP) : It is used to point to the top of stack
R14	LR	Link Register(LR) : It is used to store the link in the function on returning from the called function. Its value is to be copied in the program counter
R15	PC	Program Counter (PC): It points to the current instruction to be executed.

- Thus accordingly there are various storage methods or classes to store the variables. These are discussed in the subsequent sub-section.

4.10.1 Storage Class

- The different locations in the computer where we can store data and their accessibility, initial values etc. vary based on the way they are declared. These different ways are termed as different storage classes.
- In C, we have four storage classes, namely

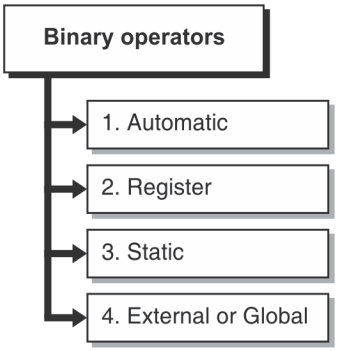


Fig.4.10.1 : Storage class

Let us see these storage classes one by one

1. Automatic storage class

- In this case data is stored in memory
- The initial value of such a variable is garbage.
- The scope of the variable is local i.e. limited to the function in which it is defined.
- The life of such variables is till the control remains in the particular function where it is defined.
- For Example:

  int i; or auto int i;
- In all our programs till now we have been using the automatic storage class for our variables.

2. Register storage class

- In this case data is stored in CPU register
- The initial value of such a variable is garbage.
- The scope of the variable is local i.e. limited to the function in which it is defined.
- The life of such variables is till the control remains in the particular function where it is defined.
- For Example:

  register int i;
- In this case the data is stored in a small memory inside the processor called as its registers.
- The advantage of such storage class is that since the data is in the processor itself, its access and operations on such data is faster.
- There is a limitation on the size of the data that can be declared to be register storage class. The data should be such that it doesn't require more than 4 bytes. Hence double and long double data types cannot be declared as register.
- Also there is a limitation on the maximum number of variables in a function that can be of register class. The limitation is that a maximum of 3 register class variable can be declared in a function.

3. Static storage class

- In this case data is stored in memory
- The initial value of such a variable is zero.
- The scope of the variable is local i.e. limited to the function in which it is defined.
- The life of such variables is till the program is alive.
- For Example:

  static int i;
- If a variable is declared static, its value remains unchanged even if the function execution is completed.
- When the execution to that function returns, the previous value is retained.
- Thus it says the initialization is only once. If you have an initialization statement of a static member, it will be executed only once i.e. for the first time when this function is called.

4. External or Global storage class

- In this case data is stored in memory
- The initial value of such a variable is zero.
- The scope of the variable is global i.e. it is accessible from anywhere in the program.
- The life of such variables is till the program is alive.
- Let us see some program examples to understand the global variables.
- The operators are one of the tokens of C. There are different types of operators classified based on their functions.
- Many of these operators although listed here, but will be understood in the subsequent chapters as and when we use them. So you may just go through the operators and understand them later while programming.
- The data on which the operation is performed are called as operands. If an operator requires one operand, it is called as Unary operator; if it requires 2 operands then binary operator and if three operands then ternary operator.

4.11 Function Calls

- Function is a self-contained block of statements that performs a given task. The programs we were writing until now had only one function i.e. the main function. In this chapter most of our programs will have two or more functions.
- The syntax of a function is given below :

```
return_type function_name (argument_list)
{
      -
      -
      statements;  -

      -
}
```

Let us see the meaning of each of these terms used in the syntax

1. "return_type" is the data type of the data to be returned by the function. The main function we have always been writing has return type of void i.e. no data is returned. The return type of the data can be any of the data types like int, char, float etc.
2. The "function_name" can be anything as far as the rules of identifiers seen in chapter one are considered. For example "main" is the name of function we have been writing until now.
3. In the brackets we write the "argument_list". Argument list is a set of data types along with the identifiers to accept a set of data passed to the function. Until now we were not passing any data to the main function, and hence the brackets have always been empty. In the programs we will be writing in this chapter we will be having functions that accept the data from another functions.
4. Statements inside the function are the statements to be executed when that function is called. We can call a function using the function call operator. We will see about all these in the subsequent programs.

   Declaration and usage of a function pointer with typedef is given below

```
typedef int (* fncptr) () ;
fncptr fptr1 ;
```

The following program illustrates the declaration, definition and the usage of function pointer.

```
# include <stdio.h>
void cube (int x) ;
void main ()
{
void (*fptr1) (int) ;          // Declare a function pointer
fptr1 = cube ;                     // Assign the address of function cube to
                                   // function pointer variable
fptr1 (2) ;                        // Invoke the function through function pointer
}
                    // Function for printing the cube of a number.
void cube (int x)
{
printf ("cube of % d = % d \ n", x, x * x * x) ;
}
```

- Function pointer can be useful in late binding. Based on the need arises in the application, we can invoke the required function by the binding the function with the right function pointer. This less use of 'if ' and 'switch-case' statements with function names. Function pointers are useful for handling the situations which require passing of a function as argument to another function.
- Function pointers are frequently used within the functions where the function should be able to work with a number of functions whose name are known only at the run time.
- The following sample code snippet shows the usage of function pointer as parameter to a function.

```
# include <stdio.h>
// Function prototype declaration
void square (int a) ;
void cube (int a) ;
void compute (void (*fptr1) (int), int a) ;
void main ()
{
// Declare a function pointer
void (*fptr2) (int) ;
//Assign the function address to the function pointer
fptr2 = square ;
// Invoke the function 'square' through function pointer fptr2 compute (fptr2, 3) ;
// Assign the function 'cube' address to the function pointer
fptr2 = cube ;
// Invoke the function 'cube' through function pointer fptr2 compute (fptr2, 3) ;
// Function implementations
void compute (void (* fptr2) (int), int a)
{
      fptr2 (a) ;
}
```

```
      void square (int a)
{
      printf ("square of %d = %d \n", a, a * a) ;
}
      void cube (int a)
{
      printf ("cube of % d = % d \ n", a, a * a * a) ;
}
```

Array of function pointers declaration using 'typedef' qualifier.

For example

```
typedef int (* fncptr1) () ;
fncptr1 fptral  [5] () ;
```

- The above statement declare an array of pointer to functions, which returns int and takes no parameters, using typedef function pointer declaration.
- We can do declaration and initialisation of array of pointers to functions at the same time.

For example

```
typedef int (* fncptr1) () ;
fncptr1 fnptrarr [] () = {/ * intialisation */} ;
```

- Following statement declare and initialise an array of pointers to functions to NULL, which return int and takes no parameters using typedef function pointer declaration.

```
typedef int (*fncptr1) () ;
fncptr1 fnptrarr [4] () = {NULL} ;
```

- The following code snippet shows the usage of function pointer arrays :

```
# include <stdio.h>
                                        // Function declaration
void square (int a) ;
void cube (int a) ;
void main ()
{
                                        // Declare a function pointer array of size 2 and initialises to "NULL"
void (* fptr1 [2]) (int) = {NULL} ;
                                        // Assign the address of function "square" to fptr1 [0]
fptr1 [0] = square ;
                                        // Invoke the function square using first element of array of
                                        // function pointer
      fptr1 [0] (3);
                                        // Assign the address of function "cube" to fptr1 [1]
      fptr1 [1] = cube ;
                                        // Invoke the function cube through function pointer fptr1 [1](3)
      fptr1 [1] (3) ;
```

```
    }
    void square (int a)
{
    printf ("square of % d = % d \n", a, a * a) ;
}
    void cube (int a)
{
    printf ("cube of % d = %d \n", a, a * a * a) ;
}
```

4.12 Pointers

- An array is a group of related data items. Arrays are usually declared with data type of array, name of the array and the number of related data items (elements) to be placed in the array. For example, the following array declaration.
- char array1 [5] ; declares a character array with name 'array1' and reserves memory for 5 character elements as shown in Fig. 4.12.1.

a[0]	1
a[1]	2
a[2]	3
a[3]	4
a[4]	5
a[5]	6
a[6]	7
a[7]	8
a[8]	9
a[9]	10

Fig. 4.12.1 : Memory allocation for an array 'a' of 10 integer elements

- A pointer is a variable that contains the address of another variable in the memory. Pointer is a memory pointing based technique for variable access and modification. Pointers are very useful in

1. Accessing and modifying variables that are defined outside the function.
2. Improves the speed of execution.
3. Accessing contents within a block of memory.
4. Reducing the length and complexity of a program.
5. Dynamic memory allocation.

- To understand the pointer concept, let us look at the data memory organisation of microcontroller AT 89C51. AT 89C51 has 128 bytes of internal data memory, the memory is organised as

  Consider the following statement

  char input = 0 $\times$ 20 ;

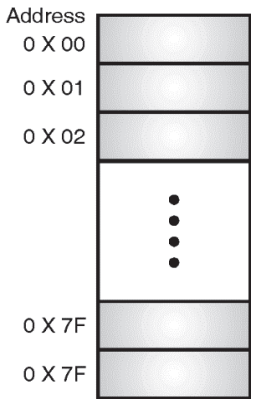


Fig. 4.12.2 : Data memory organisation for 8051

- Compiler will assign a memory location to the variable anywhere within the internal memory 0×00 to $0 \times 7F$. Let us assume that compiler has choosen the address location 0×55 for input Fig. 4.12.3 shows the representation of variable.

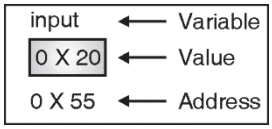


Fig. 4.12.3 : Representation of a variable

- We can access to the value 0×20 using either name of the variable or the address 0×55.
- Suppose we assign the address of input variable to a variable p. The Link between the variable p and input shown in Fig. 4.12.4. The address of p is 0×00.

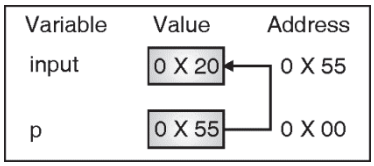


Fig. 4.12.4 : Pointer as variable

- Address of variable can be accessed with the help of operator '&'.
- The statement p = & input ; will assign address of variable input to variable p. Since the pointers are the variables that must declared before usage. The declaration of a pointer variable takes the following form.

  data type * ptr_ name.

- This tells the compiler three things about variable ptr_name.

  1. The asterisk (*) tells that the variable ptr_name is a pointer name.
  2. ptr_name needs a memory location.
  3. ptr_name points to a variable of type data type.

  In our example, statement

  char * p ;

  declares the variables p as a pointer variable that points to a character data type.

- Using the above pointer declaration and initialisation we can access the data memory of AT 89C51 microcontroller as follows.

```
char input  = 0 × 20 ;    // Declaring input as character variable with initial value 0 × 20
char * p ;                // Declaring a character pointer p
p = & input ;             // Assign the address of input to p
char a ;                  // Define variable a
a = * p ;                 // indirection operator returns the
                          // value of variable where address specify
                          // in variable p. This statement will
                          // assign 0 × 20 to variable a
```

4.12.1 Pointer Aliasing

GTU - W-17

Q. Describe Pointer aliasing in C programming for ARM processor. **(W-17, 7 Marks)**

- Two or more pointers pointing to the same location is called as pointer aliasing. If an operation is done on one of the pointers the others will get affected. The compiler is not capable of deciding the changes to be done for the pointers that are alias.
- In this case special care should be taken by the programmer that there is a proper method or function used whenever a pointer value is to be changed so that all the other alias pointers also are updated accordingly.

4.13 Structure Arrangement

- Structure is a collection of multiple data elements that can be of different data types.
- Array is a collection of data items of same type, while structure can have data items of different type.
- The memory space required to store one variable of structure is equal to the memory space required by all the elements independently to be stored in memory.
- Let us see the syntax of a structure declaration and declaration of a variable of the structure type.

Syntax of structure declaration

```
struct structure_name
{
      data_type variable_name;
      data_type variable name;
      -
      -
};
```

For example a structure to store the some information like name, roll number and fees of a student in college can be declared as shown below.

```
struct student
{
    char name[20];
    int roll_no;
    float fees;
};
```

Syntax of declaration of a variable of the structure

```
struct structure_name structure_variable_name;
```

For example, to declare a variable of the above defined structure student, we can write it as:

```
struct student s1;
```

- The variables declared inside the structure are called as the member elements of the structure. For example name, roll number and fees are member elements of the structure student.
- Structure is normally used when a set of data items corresponding to a common thing are to be stored together. For example, when information of a student is to be stored, we need to store his name, roll number, fees etc. In this case we can make a structure element and store the corresponding data of every student in a separate structure element.
- To access a member element of a structure variable we use the dot operator (“.”), also called as the period operator. The syntax of accessing a structure element is given below:

```
structure_variable_name.varaible_name;
```

- For example, in the above case to access the fees of the student “s1”, the period operator will be used as shown below:

```
s1.fees;
```

4.14 Bitfields

- These are special data types or types of variables that are never seen in the standard C programming; these are specifically used only in Embedded C programming.
- The significance of the bit fields is more important in embedded systems due to the memory constraint. For example to store a flag in case of C programming language we normally use an "int" type of data.
- This flag variable will require 16 bits or two bytes of memory space; wherein the value to be stored is just a '0' or a '1'. This value can be stored in simply one bit of data field, which is implemented in Embedded C using the bitfield type of data, using the memory resource optimally.

4.15 Unaligned Data and Endianness

4.15.1 Unaligned Data

- This is an important point to consider i.e. the unaligned data.
- The ARM processor pointer assumes that the data is stored aligned in multiple of 4 bytes data boundaries. But when the compiler stores the data, it may not store in proper boundaries as required by the processor.
- This is a very important issue that is to be addressed in the programming. Proper care has to be taken in the program for the data to be aligned.

4.15.2 Endianness

GTU - W-16, S-17

Q.	Define the following : Big Endian Memory.	**(W-16, 1 Mark)**
Q.	Describe big endian configuration for various LDR and STR mnemonics.	**(S-17, 4 Marks)**

- There are two types of endianness namely: Little endianness and big endiannness.
- In case of little endian the lower byte is stored in smallest address, while the higher byte in higher address. In case of big endian, this is exactly reverse i.e. higher byte in lower address and the higher byte in lower address.
- The ARM core architecture can be configured to work in little endian or big endian mode. This configuration of the little endian or big endian has to be done on reset and then it cannot be changed until next reset i.e. it remains fixed throughout the operation once reset.

4.15.3 Division

- The speciality of this aspect is that the ARM processor doesn't have a hardware to perform the division operation. Instead, the compiler provides with a special subroutine to perform the division operation by repetitive subtract operation.
- Thus special care is to be taken so as to implement the division operation by different methods and thus reduce the overheads for timing and increase the speed of operation. An example for the same is given below.
- Let's say we have to do an operation like finding the offset pointer in a buffer for a circular queue.

  The standard method to do this will be:

```
offset_ptr = (offset_ptr  + increment_value) % size_of_buffer.
```

- In this case offset_ptr is the pointer pointing to a current location. This pointer is incremented by an increment value equivalent to the data size. But if it crosses the buffer, then it should point back to the beginning of the buffer. For doing this the incremented value is first divided by the buffer size and the remainder is taken.
- This method may take 50+ cycles as it will call the subroutine to perform the divide operation and return the remainder. Instead the following method will be very efficient and will be able to perform the operation in around 3 cycles.

```
offset_ptr = offset+ptr +increment_value
if(offset_ptr > size_of_buffer) offset_ptr = offset_ptr - buffer_size;
```

- Thus as far as possible, the divide operation must be avoided and instead an alternative method must be used to perform similar operation. The division operation can also be converted into multiply operations by taking the inverse of the divisor.

4.16 Floating Point

- Similar to the issue of division, there is also no floating point unit in most of the ARM processors. Thus the compiler attaches or calls a subroutine for the operations to be performed on floating point data.
- There has to be special care taken to reduce the number of floating point operations and hence reduce the time required for the same. Thus, the variables must be declared as floating point only if there is an utmost requirement for the same else it must be declared as integer type data.

4.17 Inline Functions and Inline Assembly

GTU - S-17

Q. Describe inline function and inline assembly and also state advantages of using it. **(S-17, 3 Marks)**

- Inline functions have an advantage that the function call can be replaced by the entire function and hence reduce the time for calling the function. The function to be called will be replaced with the function call statement.
- The advantage of such inline function is that the execution speed increases. A function can be declared as inline by the keyword inline preceded to the function.
- Following is an example of inline function

```
_inline void get_data()
{
    ...
    ...
}
```

4.17.1 Inline Assembly

- Also some compilers have inline assemblers. This makes it possible for the programmer to write some assembly language statements if required in the C program.
- This kind of programming helps in performing those operations that are not possible in high level language like C. This can be understood better in the following explanation.
- 'C' generates an object code that is extremely fast and compact, but it is not as fast as the object code generated by a good programmer using assembly language.
- It is true that the time needed to write a program in assembly language is much more than the time taken in Higher Level Languages like C.
- Some operations like floating point operations can be much faster if done in assembly language then to include the floating point header file and use the high level language functions to do the same.
- Turbo C (TC) has inline assembler.

Case 1 : You can prefix the keyword asm for and assembly instruction in a 'C' or 'C++' program.

Example

```
#include<iostream.h>
void main()
{
  int a, b, c;
  cout<<"Enter two numbers";
  cin>>a>>b;
  asm mov ax,a;
  asm mov bx,b;
  asm add ax,bx;
  asm mov c,ax;
  cout<<"The  sum is "<<c;
}
```

Case 2 : You can prefix the keyword asm for a function and write assembly instruction in the curly braces in a 'C' or 'C++' program.

Example

```
#include<iostream.h>
void main()
{
  int a, b, c;
  cout<<"Enter two numbers";
  cin>>a>>b;
  asm{
  mov ax,a;
  mov bx,b;
  add ax,bx;
  mov c,ax;
  }
  cout<<"The  sum is "<<c;
}
```

- The other option is to write the assembly code and the C code separately.
- The assembly code is to be assembled and the C code is to be compiled.
- The two object files that are created by assembling and compiling will then be linked together.

4.18 Portability Issues

There are various issues related to portability in ARM based compiling of the programs. Some of these issues are listed below:

1. The "int" type data is mostly taken as a 16-bit data format, but for the 32-bit ARM processors it is a 32-bit format. This issue has to be specially taken care in case of signed data.
2. The "char" type data in most of the microcontrollers including ARM is by default unsigned, while for C programmers it is by default signed. This can cause severe issues like infinite loop in case of the condition being checked is i<0. The ARM compiler generates a warning in this case so that the programmer can take care of the same.
3. The two types of endian supported by ARM has to be taken care of. In case if one function is written considering little endian, while the other considering big endian, the system will not work. As discussed earlier, the endian mode can be initialised only once during the reset of the system.
4. The endian problem will also result into issues for the bit fields, as they access a bit in a higher size of data like byte, double byte etc. Thus if the wrong bit is accessed based on endian, the entire system will work in an abrupt manner.
5. The inline assembly feature of the ARM compilers must be used very carefully. It may access the registers that are assigned as another variable by the C program. Hence separate functions must be written using assembly so as to avoid the re-usage of same registers for multiple variables.
6. The alignment of the data pointers also has to be taken care of. The pointers must be aligned according to the data size of the processor being used.

4.19 LPC2148#

- The LPC2141/42/44/46/48 microcontrollers are based on a 16-bit/32-bit ARM7TDMI-S CPU with real-time emulation and embedded trace support, that combine the microcontroller with embedded high-speed flash memory ranging from 32 KB to 512 KB. A 120-bit wide memory interface and a unique accelerator architecture enable 32-bit code execution at the maximum clock rate.
- For critical code size applications, the alternative 16-bit thumb mode reduces code by more than 30 % with minimal performance penalty. Due to their tiny size and low power consumption, LPC2141/42/44/46/48 are ideal for applications where miniaturization is a key requirement, such as access control and point-of-sale.
- Serial communications interfaces ranging from a USB 2.0 Full-speed device, multiple UARTs, SPI, SSP to I^2C-bus and on-chip SRAM of 8 KB up to 40 KB, make these devices very well suited for communication gateways and protocol converters, soft modems, voice recognition and low end imaging, providing both large buffer size and high processing power.
- Various 32-bit timers, single or dual 10-bit ADC(s), 10-bit DAC, PWM channels and 45 fast GPIO lines with up to nine edge or level sensitive external interrupt pins make these microcontrollers suitable for industrial control and medical systems.

Courtesy NxP

4.20 Features of LPC 2148#

- 8 KB to 40 KB of on-chip static RAM and 32 KB to 512 KB of on-chip flash memory. 128-bit wide interface/accelerator enables high-speed 60 MHz operation.
- In-System Programming/In-Application Programming (ISP/IAP) via on-chip boot loader software. Single flash sector or full chip erase in 400 ms and programming of 256 B in 1 ms.
- Embedded ICE RT and Embedded Trace interfaces offer real-time debugging with the on-chip Real Monitor software and high-speed tracing of instruction execution.
- USB 2.0 Full-speed compliant device controller with 2 kB of endpoint RAM. In addition, the LPC2146/48 provides 8 KB. of on-chip RAM accessible to USB by DMA.
- One or two (LPC2141/42 vs. LPC2144/46/48) 10-bit ADCs provide a total of 6/14 analog inputs, with conversion times as low as 2.44 μs per channel.
- Single 10-bit DAC provides variable analog output (LPC2142/44/46/48 only).
- Two 32-bit timers/external event counters (with four capture and four compare channels each), PWM unit (six outputs) and watchdog.
- Low power Real-Time Clock (RTC) with independent power and 32 kHz clock input.
- Multiple serial interfaces including two UARTs (16C550), two Fast I^2C-bus (400 kbit/s), SPI and SSP with buffering and variable data length capabilities.
- Vectored Interrupt Controller (VIC) with configurable priorities and vector addresses.
- Up to 45 of 5 V tolerant fast general purpose I/O pins in a tiny LQFP64 package.
- Up to 21 external interrupt pins available.
- 60 MHz maximum CPU clock available from programmable on-chip PLL with settling time of 100 μs.
- On-chip integrated oscillator operates with an external crystal from 1 MHz to 25 MHz.
- Power saving modes include Idle and Power-down.
- Individual enable/disable of peripheral functions as well as peripheral clock scaling for additional power optimization.
- Processor wake-up from Power-down mode via external interrupt or BOD.
- Single power supply chip with POR and BOD circuits:
- CPU operating voltage range of 3.0 V to 3.6 V (3.3 V ± 10 %) with 5 V tolerant. I/O pads.

Courtesy NxP

4.21 Architecture of LPC2148#(Block Diagram and its Description)

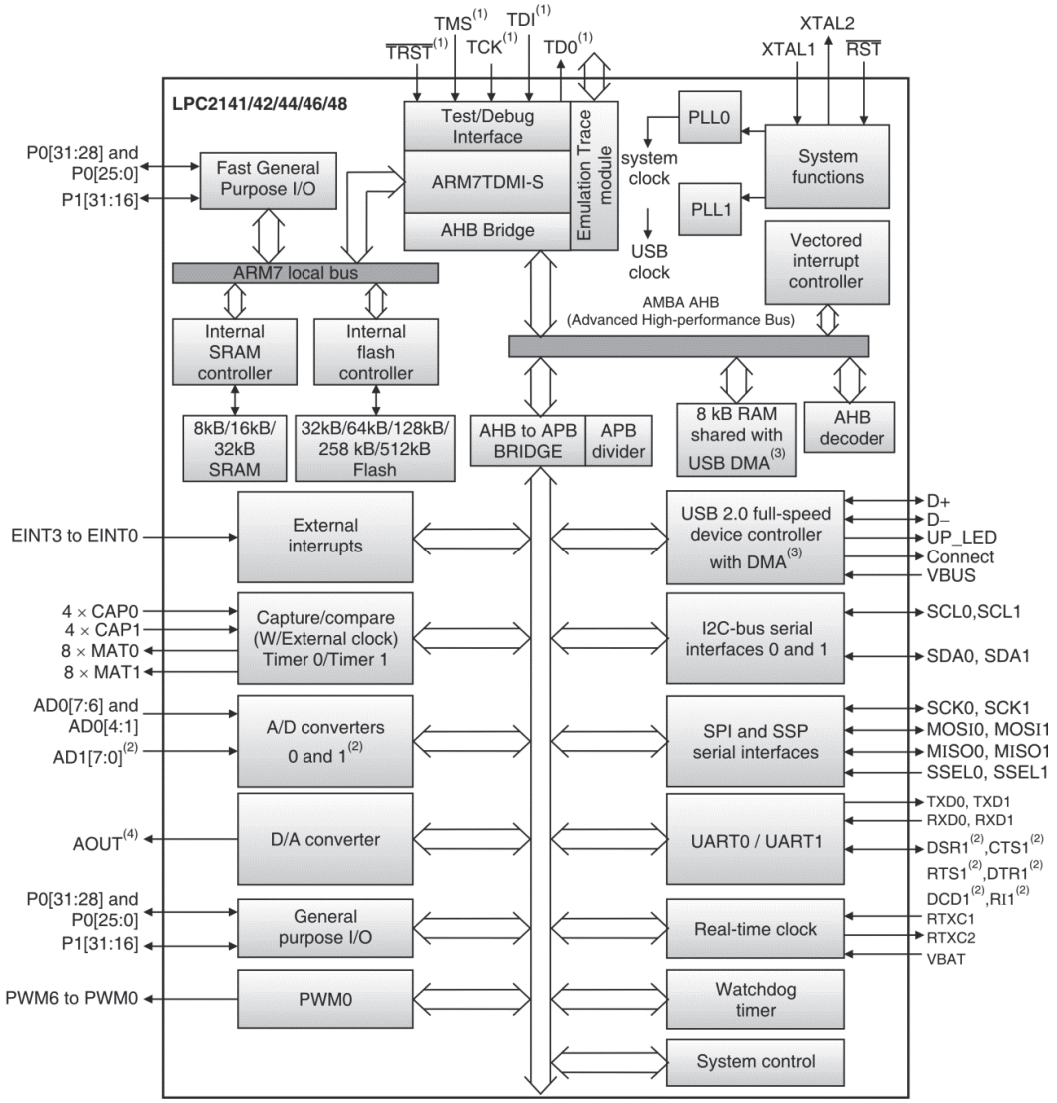


Fig. 4.21.1 : Block diagram of LPC2148

Table 4.21.1 : Pin description (Courtesy NxP)

Symbol	Pin	Type	Description
P0.0 to P0.31		I/O	Port 0 : Port 0 is a 32-bit I/O port with individual direction controls for each bit. Total of 31 pins of the Port 0 can be used as a general purpose bidirectional digital I/Os while P0. 31 is output only pin. The operation of port 0 pins depends upon the pin function selected via the pin connect block. Pins P0.24, P0.26 and P0.27 are not available.
P0.0/TXD0/PWM1	19	I/O	P0.0 – General purpose input/output digital pin (GPIO).
		O	IXD0 – Transmitter output for UART 0.
		O	PWM1 – Pulse With Modulator output 1.

Courtesy NxP

Symbol	Pin	Type	Description
P0.1/RXD0/PWM3/EINT0	21	I/O	P0.1 – General purpose input/output digital pin (GPIO).
		I	RXD0 - Receiver input for UART 0
		O	PWM3 – Pulse Width Modulator output 3.
		I	EINT0 – External interrupt 0 input.
P0.2/SCL0/CAP0.0	22	I/O	P0.2 – General purpose input/output digital pin (GPIO).
		I/O	SCL0 – I^2C0 clock input/output. Open-drain output (for I^2C-bus compliance)
		I	CAP0.0 – Capture input for Timer 0, channel 0
P0.3/SDA0/ MAT0.0/EINT1	26	I/O	P0.3 – General purpose input/output digital pin (GPIO).
		I/O	SDA0 – I^2C0 data input/output open-drain output (for I^2C-bus compliance).
		O	MAT0.0 – Match output for Timer 0, channel 0
		I	EINT1 – External interrupt 1 input
P0.4/SCK0/CAP0.1/AD0.6	27		
		I/O	P0.4 – General purpose input/output digital pin (GPIO)
		I/O	SCK0 – Serial clock for SPI0. SPI clock output from master or input to slave.
		I	CAP0.1 – Capture input for Timer 0. Channel 1
		I	AD0.6 – ADC0.input 6
P0.5/MISO0/MAT0.1/AD0.7	29	I/O	P0.5 – General purpose input/output digital pin (GPIO)
		I/O	MISO0 – Master in Slave Out for SPI0. Data input to SPI master or data output from SPI slave
		O	MAT0.1 – Match output for Timer 0, channel 1
		I	AD0.7 ADC0, input 7
P0.6/MOSI0/CAP0.2/AD1.0	30	I/O	P0.6 – General purpose input/output digital pin (GPIO)
		I/O	MOSI0 – Master out slave in for SPI0. Data output from SPI master or data input to SPI slave.
		I	CAP0.2 – Capture input for Timer 0, channel 2
		I	AD1.0 – ADC1, input 0. Available in LPC2144/46/48 only.
P0.7/SSEL0/PWM2/EINT2	31	I/O	P0.7 – General purpose input/output digital pin (GPIO)
		I	SSEL0 – Slave select for SPI0. Selects the SPI interface as a slave.
		O	PWM2 – Pulse width modulator output 2
		I	EINT2 – External interrupt 2 input

Courtesy NxP

Symbol	Pin	Type	Description
P0.0/TXD1/PWM4/AD1.1	33	I/O	P0.0 – General purpose input/output digital pin (GPIO)
		O	TXD1 Transmitter output for UART1.
		O	PWM4 – Pulse width modulator output 4
		I	AD1.1 – ADC1, input 1. Available in LPC2144/46/48 only.
P0.9/RXD1/PWM6/EINT3	34	I/O	P0.9 – General purpose input/output digital pin (GPIO)
		I	RXD1 Receiver input for UART1
		O	PWM6 – Pulse width modulator output 6
		I	EINT3 – External interrupt 3 input
P0.10/RTS1/CAP1.0/AD1.2	35	I/O	P0.10 – General purpose input/output digital pin (GPIO)
		O	RTS1 – Request to send output for UART1.LPC2144/46/48 only.
		I	CAP1.0 – capture input for Timer 1, channel 0
		I	AD1.2 – ADC 1, input 2. available in LPC2144/46/48 only.
P0.11/CTS1/CAP1.1/SCL1	37	I/O	P0.11 – General purpose input/output digital pin (GPIO)
		I	CTS1 – Clear to send input for UART1. Available in LPC2144/46/48 only.
		I	CAP1.1 – Capture input for Timer 1, Channel 1
		I/O	SCL1 – I^2C1 clock input/output. Open drain output (for I^2C bus compliance)
P0.12/DSR1/MAT1.0/AD1.3	38	I/O	P0.12 – General purpose input/output digital pin (GPIO).
		I	DSR1 – Data set Ready input for UART1. Available in LPC2144/46/48 only.
		O	MAT1.0 – Match output for Timer 1, channel 0.
		I	AD1.3 – ADC 1 input 3. Available in LPC2144/46/48 only.
P0.13/DTR1/MAT1.1/AD1.4	39	I/O	P0.13 – General purpose input/output digital pin (GPIO)
		O	DTR1 – Data Terminal ready output for UART1. LPC2144/46/48 only.
		O	MAT1.1 – Match output for Timer 1, channel 1.
		I	AD1.4 – ADC 1 input 4. Available in LPC2144/46/48 only.
P0.14/DCD1/EINT1/SDA1	41	I/O	P0.14 – General purpose input/output digital pin (GPIO)
		I	DCD1 – Data carrier detect input for UART1. LPC2144/46/48 only.
		I	EINT1 – External interrupt 1 input
		I/O	SDA1 – I^2C1 data input/output. Open-drain output (for I^2C-bus compliance) Note : LOW on this pin while $\overline{\text{RESET}}$ is LOW forces on-chip boot loader to take over control of the part after reset.

Symbol	Pin	Type	Description
P0.15/RI1/EIN12/AD1.5	45	I/O	P0.15 – General purpose input/output digital pin (GPIO)
		I	RI1 – Ring indicator input for UART1. Available in LPC2144/46/48 only.
		I	EINT2 – External interrupt 2 input
		I	AD1.5 – ADC1 input 5 Available in LPC2144/46/48 only.
P0.16/EINT0/MAT0.2/CAP0.2	46	I/O	P0.16 – General purpose input/output digital pin (GPIO).
		I	EINT0 – External interrupt 0 input
		O	MATA0.2 – Match output for Timer 0, Channel 2
		I	CAP0.2 – Capture input or Timer 0, Channel 2.
P0.17/CAP1.2/SCK1/MAT1.2	47	I/O	P0.17 – General purpose input/output digital pin (GPIO).
		I	CAP1.2 – Capture input for Timer 1, channel 2
		I/O	SCK 1 – Serial clock for SSP. Clock output from master or input to slave.
		O	MAT1.2 – Match output for Timer 1. Channel 2.
P0.18/CAP1.3/MISO1/MAT1.3	53	I/O	P0.18 – General purpose input/output digital pin (GPIO)
		I	CAP1.3 – Capture input for Timer 1, Channel 3
		I/O	MISO1 – Master In slave out for SSP. Data input to SPI master or data output from SSP slave.
		O	MAT1.3 – Match output for Timer 1, channel 3.
P0.19/MAT1.2/MOSI1/CAP1.2	54	I/O	P0.19 – General purpose input/output digital pin (GPIO)
		O	MAT1.2 – Match output for Timer 1, channel 2
		I/O	MOSI1 – Master out slave in for SSP. Data output from SSP master or data input to SSP slave.
		I	CAP1.2 – Capture input for Timer 1, channel 2
P0.20/MAT1.3/SSEL1/EINT3	55	I/O	P0.20 – General purpose input/output digital pin (GPIO)
		O	MAT1.3 – Match output for Timer 1, channel 3.
		I	SSEL1 – Slave select for SSP. Selects the SSP interface as a slave.
		I	EINT3 – External interrupt 3 input.

Courtesy NxP

Symbol	Pin	Type	Description
P0.21/PWM5/AD1.6/CAP1.3	1	I/O	P0.21 – General purpose input/output digital pin (GPIO)
		O	PWM5 – Pulse width Modulator output 5
		I	AD1.6 – ADC 1, input 6 Available in LPC2144/46/48 only
		I	CAP1.3 – Capture input for Timer 1, channel 3
P0.22/AD1.7/CAP0.0/MAT0.0	2	I/O	P0.22 – General purpose input/output digital pin (GPIO)
		I	AD1.7 – ADC 1, input 7. Available in LPC2144/46/48 only.
		I	CAP0.0 – Capture input for Timer 0, Channel 0
		O	MAT0.0 – Match output for Timer 0, Channel 0.
P0.23/V_{BUS}	58	I/O	P0.23 – General purpose input/output digital pin (GPIO)
		I	V_{BUS} – Indicates the presence of USB bus power Note : This signal must be HIGH for USB reset to occur.
P0.25/AD0.4/AOUT	9	I/O	P0.25 – General purpose input/output digital pin (GPIO)
		I	AD0.4 – ADC 0, input 4
		O	AOUT – DAC output. Available in LPC2142/44/46/48 only
P0.28/AD0.1/CAP0.2/MAT0.2	13	I/O	P0.28 – General purpose input/output digital pin (GPIO)
		I	AD0.1 – ADC 0, input 1
		I	CAP0.2 – Capture input for Timer 0, Channel 2
		O	MAT0.2 – match output for Timer 0, channel 2
P0.29/AD0.2/CAP0.3/MAT0.3	14	I/O	P0.29 – General purpose input/output digital pin (GPIO)
		I	AD0.2 – ADC.0 input 2
		I	CAP0.3 – Capture input for Timer 0, Channel 3
		O	MAT0.3 – Match output for Timer 0, Channel 3.
P0.30/AD0.3/EINT3/CAP0.0	15	I/O	P0.30 – General purpose input/output digital pin (GPIO)
		I	AD0.3 – ADC 0, input 3
		I	EINT3 – External interrupt 3 input
		I	CAP0.0 – Capture input for Timer 0, channel 0

Courtesy NxP

Symbol	Pin	Type	Description
P0.31/UP_LED/CONNECT	17	O	P0.31 – General purpose output only digital pin (GPO)
		O	UP_LED – USB Goodlink LED indicator. it is LOW when device is configured (non-control endpoints enabled). It is HIGH when the device is not configured or during global suspend.
		O	CONNECT – Signal used to switch an external 1.5 kΩ resistor under the software control. Used with the softconnect USB feature. Important : This is an digital output only pin. This. pin MUST NOT be externally pulled LOW when $\overline{\text{RESET}}$ pin is LOW or the JTAG port be disabled.
P1.0 to P1.31		I/O	Port 1 : Port 1 is a 32-bit bidirectional I/O port with individual direction controls for each bit. The operation of port 1 pins depends upon the function selected via the pin connected block. Pins 0 through 15 of port 1 are not available.
P1.16/TRACEPKT0	16	I/O	P1.16 – General purpose input/output digital pin (GPIO). Standard I/O port with internal pull-up.
		O	TRACEPKT0 – Trace Packet, bit 0
P1.17/TRACEPKT1	12	I/O	P1.17 – General purpose input/output digital pin (GPIO) standard I/O port with internal pull up.
		O	TRACEPKT1 – Trace packet. bit 1
P1.18/TRACEPKT2	8	I/O	P1.18 – General purpose input/output pin (GPIO) standard I/O port with internal pull-up
		O	TRACEPKT2 – Trace packet, bit 2
P1.19/TRACEPKT3	4	I/O	P1.19 – General purpose input/output digital pin (GPIO) standard I/O port with internal pull up.
		O	TRACEPKT3 – Trace packet. bit 3
P1.20/TRACESYNC	48	I/O	P1.20 – General purpose input/output digital pin (GPIO). Standard I/O port with internal pull-up
		O	TRACESYNC – Trace synchronization **Note :** LOW on this pin while $\overline{\text{RESET}}$ is LOW enables pin P1.25 : 16 to operate as Trace port after reset.

Courtesy NxP

Symbol	Pin	Type	Description
P1.21/PIPESTAT0	44	I/O	P1.21 – General purpose input/output digital pin (GPIO). Standard I/O port with internal pull-up.
		O	PIPESTAT0 – Pipeline status, bit 0
P1.22/PIPESTAT1	40	I/O	P1.22 – General purpose input/output digital pin (GPIO) standard I/O port with internal pull-up
		O	PIPESIAI1 – Pipeline status, bit 1
P1.23/PIPESTAT2	36	I/O	P1.23 – General purpose input/output digital pin (GPIO), standard I/O port with internal pull-up
		O	PIPESTAT2 – Pipeline status, bit 2
P1.24/TRACECLK	32	I/O	P1.24 – General purpose input/output digital pin (GPIO) standard I/O port with internal pull-up
		O	TRACECLK – Trace clock.
P1.25/EXTIN0	28	I/O	P1.25 – General purpose input/output digital pin (GPIO). Standard I/O port with internal pull-up
		I	EXTIN0 – External Trigger input
P1.26/RTCK	24	I/O	P1.26 – General purpose input/output digital pin (GPIO)
		I/O	RTCK – Returned test clock output. Extra signal added to the JTAG port. Assists debugger synchronization when processor frequency varies Bidirectional pin with internal pull-up. Note : LOW on RTCK while $\overline{\text{RESET}}$ is LOW enables pin P1[31:26] to operate as Debug port after reset.
P1.27/TDO	64	I/O	P1.27 – General purpose input/output digital pin (GPIO)
		O	TDO – Test Data out for JTAG interface
P1.28/TDI	60	I/O	P1.28 – General purpose input/output digital pin (GPIO)
		I	TDI – Test data in for JTAG interface
P1.29/TCK	56	I/O	P1.29 general purpose input/output digital pin (GPIO).
		I	TCK — Test Clock for JTAG interface. This clock must be slower than $^{1}?_{8}$ of the CPU clock (CCLK) for the JTAG interface to operate.

Courtesy NxP

Symbol	Pin	Type	Description
P1.30/TMS	52	I/O	P1.30 – General purpose input/output digital pin (GPIO).
		I	TMS — Test Mode Select for JTAG interface.
P1.31/$\overline{TRST}$	20	I/O	P1.31 – General purpose input/output digital pin (GPIO).
			$\overline{TRST}$ – Test Reset for JTAG interface.
D+	10	I/O	USB bidirectional D+ line.
D–	11	I/O	USB bidirectional D+ line.
$\overline{RESET}$	57	I	External reset input : A LOW on this pin resets the device, causing I/O ports and peripherals to take on their default states, and processor execution to begin at address 0. TTL with hysteresis, 5 V tolerant.
XTAL1	62	I	Input to the oscillator circuit and internal clock generator circuits.
XTAL2	61	O	Output from the oscillator amplifier.
RTCX1	3	O	Input to the RTC oscillator circuit.
RTCX2	5	O	Output from the RTC oscillator circuit.
V_{SS}	6, 18, 25, 42, 50	I	Ground : 0 V reference
V_{SSA}	59	I	Analog ground : 0 V reference. This should nominally be the same voltage as V_{SS}, but should be isolated to minimize noise and error.
V_{DD}	23, 43, 51	I	3.3 V power supply : This is the power supply voltage for the core and I/O ports.
V_{DDA}	7	I	Analog 3.3 V power supply : This should be nominally the same voltage as V_{DD} but should be isolated to minimize noise and error. This voltage is only used to power the on-chip ADC(s) and DAC.
V_{REF}	63	I	ADC reference voltage : This should be nominally less than or equal to the V_{DD} voltage but should be isolated to minimize noise and error. Level on this pin is used as a reference for ADC(s) and ADC.
V_{BAT}	49	I	RTC power supply voltage : 3.3 V on this pin supplies the power to the RTC.

Courtesy NxP

4.22 System Control Block (PLL and VPB Divider)

4.22.1 PLL

- The PLL accepts an input clock frequency in the range of 10 MHz to 25 MHz.
- The input frequency is multiplied up into the range of 10 MHz to 60 MHz with a current controlled Oscillator (CCO). The multiplier can be an integer value from 1 to 32 (in practice, the multiplier value cannot be higher than 6 on this family of microcontrollers due to the upper frequency limit of the CPU).
- The CCO operates in the range of 156 MHz to 320 MHz, so there is an additional divider in the loop to keep the CCO within its frequency range while the PLL is providing the desired output frequency.
- The output divider may be set to divide by 2, 4, 8 or 16 to produce the output clock. Since the minimum output divider value is 2, it is insured that the PLL output has a 50% duty cycle. The PLL is turned off and bypassed following a chip reset and may be enabled by software.
- The program must configure and activate the PLL, wait for the PLL to Lock, then connect to the PLL as a clock source. The PLL settling time is 100 μs.

4.22.2 VPB Divider

- The LPC2141/42/44/46/48 include 2-stage monitoring of the voltage on the V_{DD} pins. If this voltage falls below 2.9 V, the BOD assets an interrupt signal to the VIC. This signal can be enabled for interrupt; if not, software can monitor the signal by reading dedicated register.
- The second stage of low voltage detection asserts reset to inactivate the LPC2141/42/44/46/48 when the voltage on the V_{DD} pins falls below 2.6 V. This reset prevents alteration of the flash as operation of the various elements of the chip would otherwise become unreliable due to low voltage.
- The BOD circuit maintains this reset down below 1V, at which point the POR circuitry maintains the overall reset.
- Both the 2.9V and 2.6V thresholds include some hysteresis. In normal operation, this hysteresis allows the 2.9V detection to reliably interrupt, or a regularly-executed event loop to sense the condition.

4.23 Memory Map

- Fig. 4.23.1 shows the different regions in the memory of LPC 2148.
- As seen in the Fig. 4.23.1 locations 0x00000000 to 0x0007FFF is an on-chip non-volatile memory. There after some space is reserved and then there is on-chip static RAM from 0x40000000 to 0x40007FFF.
- Then there is again some reserved space followed by the USB DMA RAM. This is again followed by reserved space and then the space from 0x7FFFD000 to 0x7FFFFFFF is used by on-chip flash memory.

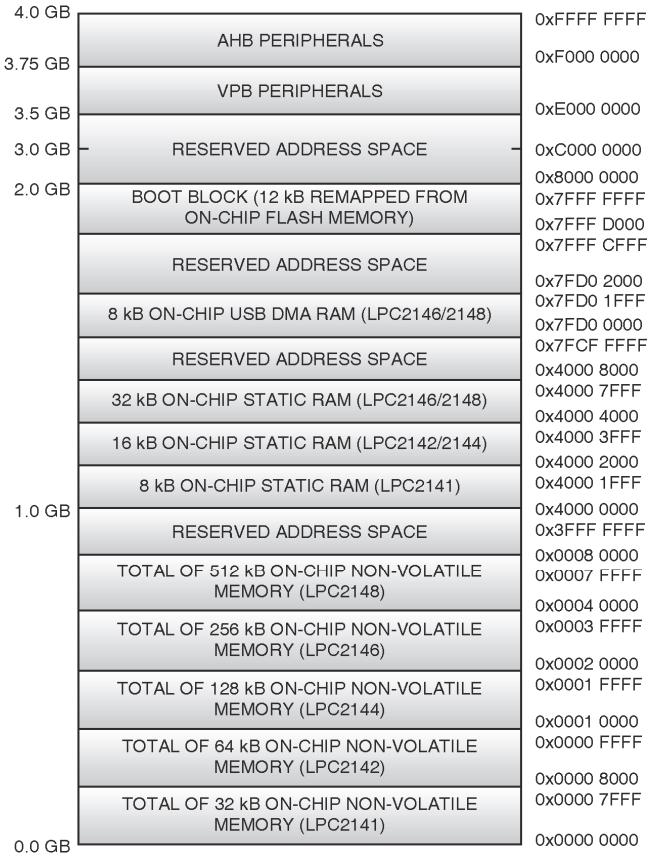


Fig. 4.23.1 : LPC 2141/42/44/46/48 Memory Map (Courtesy NxP)

4.24 Fast General Purpose Parallel I/O (GPIO)

- Device pins that are not connected to a specific peripheral function are controlled by the GPIO registers. Pins may be dynamically configured as inputs or outputs. Separate registers allow setting or clearing any number of outputs simultaneously.
- The value of the output register may be read back, as well as the current state of the port pins. LPC2141/42/44/46/48 introduce accelerated GPIO functions over prior LPC2000 devices.
- GPIO registers are relocated to the ARM local bus for the fastest possible I/O timing.
- Mask registers allow treating sets of port bits as a group, leaving other bits unchanged.
- All GPIO registers are byte addressable.
- Entire port value can be written in one instruction.

4.24.1 Features of GPIO

- Bit-level set and clear registers allow a single instruction set or clear of any number of bits in one port.
- Direction control of individual bits.
- Separate control of output set and clear.
- All I/O default to inputs after reset.

4.25 Pin Connect Block

- The pin connect block allows selected pins of the microcontroller to have more than one function. Configuration registers control the multiplexers to allow connection between the pin and the on chip peripherals.
- Peripherals should be connected to the appropriate pins prior to being activated, and prior to any related interrupt (s) being enabled. Activity of any enabled peripheral function that is not mapped to a related pin should be considered undefined.
- The pin control module with its pin select registers defines the functionality of the microcontroller in a given hardware environment.
- After reset all pins of Port 0 and Port 1 are configured as input with the following exceptions : If debug is enabled, the JTAG pins will assume their JTAG functinality; if trace is enabled, the Trace pins will assume their trace functionality. The pins associated with the I^2C0 and I^2C1 interface are open drain.

4.26 Timers

- The Timer/Counter is designed to count cycles of the peripheral clock (PCLK) or an externally supplied clock and optionally generate interrupts or perform other actions at specified timer values, based on four match registers.
- It also includes four capture inputs to trap the timer value when an input signal transitions. Optionally generating an interrupt.
- Multiple pins can be selected to perform a single capture or match function, providing an application with 'or' and 'and', as well as 'broadcast' function among them.
- The LPC2141/42/44/46/48 can count external events on one of the capture inputs if the minimum external pulse is equal or longer than a period of the PCLK.
- In this configuration, unused capture lines can be selected as regular timer capture inputs, or used as external interrupts.

4.26.1 Features of Timers

- A 32-bit timer/counter with a programmable 32-bit prescaler.
- External event counter or timer operation.
- Four 32-bit capture channels per timer/counter that can take a snapshot of the timer value when an input signal transitions. A capture event may also optionally generate an interrupt.
- Four 32-bit match registers that allow :
- Continuous operation with optional interrupt generation on match.
- Stop timer on match with optional interrupt generation.
- Reset timer on match with optional interrupt generation.
- Four external output per timer/counter corresponding to match registers, with the following capabilities :

  1. Set LOW on match. 2. Set HIGH on match
  3. Toggle on match 4. Do nothing on match

4.27 Interfacing GPIO using Timer

Fast general purpose parallel I/O (GPIO)

- Device pins that are not connected to a specific peripheral function are controlled by the GPIO registers. Pins may be dynamically configured as inputs or outputs. Separate registers allow setting or clearing any number of outputs simultaneously. The value of the output register may be read back, as well as the current state of the port pins. LPC2141/42/44/46/48 introduce accelerated GPIO functions over prior LPC2000 devices.
- GPIO registers are relocated to the ARM local bus for the fastest possible I/O timing.
- Mask registers allow treating sets of port bits as a group, leaving other bits unchanged.
- All GPIO registers are byte addressable.
- Entire port value can be written in one instruction.

Features of GPIO

- Bit-level set and clear registers allow a single instruction set or clear of any number of bits in one port.
- Direction control of individual bits.
- Separate control of output set and clear.
- All I/O default to inputs after reset.

4.27.1 Interfacing Light Emitting Diode (LED) and Seven Segment Display (SSD)

GTU - S-17, S-19, W-19

Q. Develop C program to read the status of the pull up switch connected with pin number of 16 of port 0 and display the LED connected with pin number 23 of port 0. The cathode of LED is connected with port 0 pin and the anode is connected with +5V through resistor. **(S-17, 4 Marks)**

Q. Develop C program to turn all eight LED connected with Port 0 Pin number 16 to 23 one by one from left to right and then right to left continuously. Assume that the anode of all the eight LED connected with port 0 pins and the cathode of all eight LED connected with ground through current limiting register. **(S-17, 7 Marks)**

Q. Write C program to turn ON all eight Electrical devices which are controlled by Port pins P0.0 to P0.7 one by one at the interval of 1 second. Assume that there is relay driver used to control electrical devices and logic 1 on the port pin turns on the electrical device. **(S-19, 4 Marks)**

Q. Write a C program to toggle led which is connected to pin 30 of LPC 2148. **(W-19, 4 Marks)**

Q. Draw interfacing diagram of one CA seven segment on port 0.0 to 0.7. Write a C program to display A to F with some time delay. **(W-19, 7 Marks)**

As shown in the Fig. 4.27.1 the seven segment display uses seven LEDs to make any digit. If all the LEDs are on, it shows the digit 8. There are two types SSDs available.

(i) Common cathode i.e. the cathode of all the LEDs are given as a common pin. In this case the anode is connected to the port pins. As already discussed in the section for LEDs, this method requires port pins to source large current. But 8051 cannot source current beyond 2 mA.

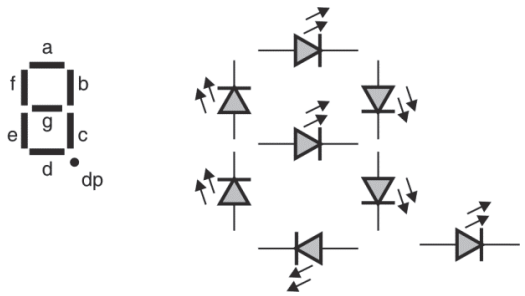


Fig. 4.27.1 : Structure of Seven Segment Display (SSD)

(ii) Common anode i.e. the anode of all the LEDs are given as a common pin. In this case the cathode is connected to the port pins. Hence port pins have to sink current of 20 mA.

Hence we use common anode SSD, and connect it to 8051 as shown in Fig. 4.27.2.

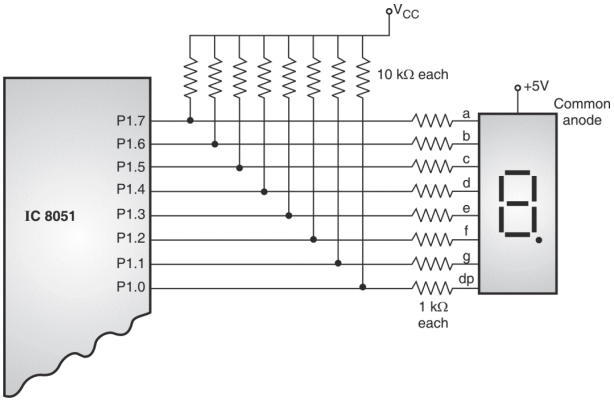


Fig. 4.27.2 : Interfacing common anode SSD to 8051

The 'dp' stands for decimal point. The Table 4.27.1 illustrates the binary and hexadecimal data given to the pins of SSD, to display different digits. A pin should be given logic '0', to switch on the corresponding LED.

Table 4.27.1 : Code for 0 to F given to Common Anode SSD

Digit	g	f	e	d	c	b	a	Hexadecimal form
0	1	0	0	0	0	0	0	0x40
1	1	1	1	1	0	0	1	0x79
2	0	1	0	0	1	0	0	0x24
3	0	1	1	0	0	0	0	0x30
4	0	0	1	1	0	0	1	0x19
5	0	0	1	0	0	1	0	0x12
6	0	0	0	0	0	1	0	0x02
7	1	1	1	1	0	0	0	0x78
8	0	0	0	0	0	0	0	0x00
9	0	0	1	0	0	0	0	0x10
A	0	0	0	1	0	0	0	0x08
B	0	0	0	0	0	1	1	0x03
C	1	0	0	0	1	1	0	0x46
D	0	1	0	0	0	0	1	0x21
E	0	0	0	0	1	1	0	0x06
F	0	0	0	1	1	1	0	0x0E

Program 4.27.1 : Write a program to implement two digit counter from 00H to FFH and show the interfacing of the these two SSDs with LPC2148. **GTU : S-16, W-16, S-17, 7 Marks**

Soln. :

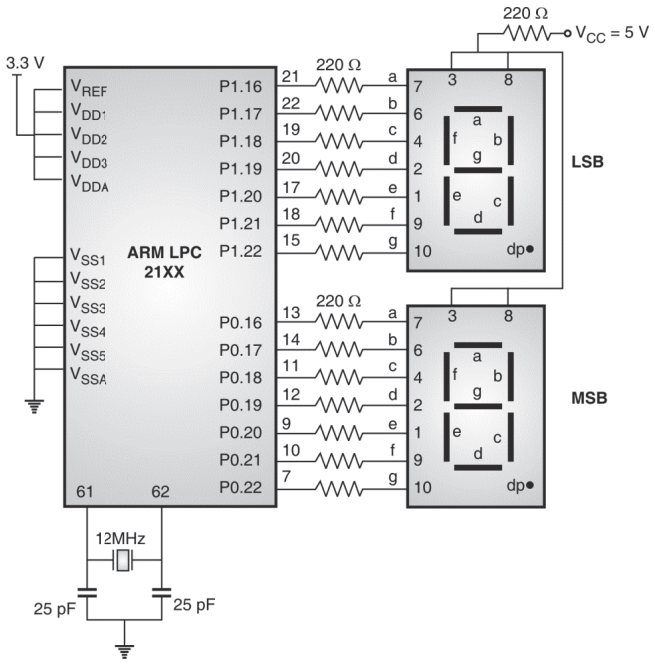


Fig. P. 4.27.1

```
#include<lpc21xx.h>
unsigned char seg[16] ={0x40,0x79,0x24,0x30,0x19,0x12,0x02,0x78,0x00,0x10,0x08,
0x03,0x46,0x21,0x06,0x0E};          //From Table 4.26.1
void main()
{
      unsigned long int k, var=0, var1=0;
      unsigned char seg_val,seg_val1,i,j;
      PINSEL0=0X00000000;           // Select Port 0  pins as General Purpose I/O lines
      PINSEL1=0X00000000;           // Select Port 1  pins also as General Purpose I/O lines
      IODIR0 = 0X00FF0000;          // Make pins of Port 0 as output pins
      IODIR1 = 0X00FF0000;          // Make pins of Port 1 as output pins
      for (i=0;i<=15;i++)           // Counter for upper digit

      {
          IOCLR1 = var;
          seg_val = seg[i];
          var = seg_val;
          var = var<<16;
          IOSET1 = var;

          for(j=0;j<=15;j++)        // Counter for lower digit
          {
              IOCLR0=var1;
```

```
            seg_val1=seg[j];
            var1=seg_val1;
            var1=var1<<16;
            IOSET0=var1;
            for(k=0;k<50000;k++);
        }
    }
}
```

Program 4.27.2 : Draw interfacing diagram of one CC seven segment on port 0.0 to 0.7. Write a C program to display 0 to 9 & A to F with every one second time delay. **GTU : S-16, 7 Marks**

Draw interfacing diagram to interface push-button switch at port pin P1.0 and common anode seven segment display with port pins P0.0 to P0.7. Write C program to read switch and display number of seven segment display. Initially 0 should be displayed. When switch is pressed, number should increase (UP counting). **GTU : S-18, 7 Marks**

Soln. :

```
#include<lpc21xx.h>
unsigned char seg[16] ={0x40,0x79,0x24,0x30,0x19,0x12,0x02,0x78,0x00,0x10,0x08,
0x03,0x46,0x21,0x06,0x0E};
void main()
{
    unsigned long int k, var=0, var1=0;
    unsigned char seg_val,seg_val1,i,j;
    PINSEL1=0X00000000;                 // Select Port 1  pins also as General Purpose I/O lines
    IODIR1 = 0X00FF0000;                // Make pins of Port 1 as output pins
    for(j=0;j<=15;j++)                  // Counter
    {
            IOCLR0=var1;
            seg_val1=seg[j];
            var1=seg_val1;
            var1=var1<<16;
            IOSET0=var1;
            delay(1000);                //delay of 1000 milli seconds
    }
}
```

Interfacing Diagram

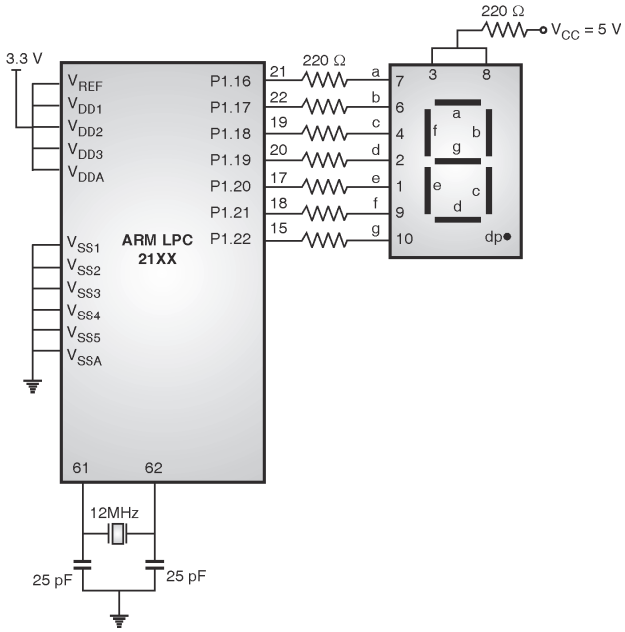


Fig. P. 4.27.2

Program 4.27.3 : Write C language instructions to define port pins P1.0 to P1.7 output pins and set all these pins.

GTU - S-18, 3 Marks

Soln. :

```
#include<lpc21xx.h>
void main()
{
      long  int var=0xFFFFFFFF;
      PINSEL1=0X00000000;              // Select Port 1  pins also as General Purpose I/O lines
      IODIR1 = 0X00FF0000;             // Make pins of Port 1 as output pins
      IOSET1=var;
}
```

Program 4.27.4 : Give the C language instructions to define port pins P1.0 to P1.3 as output pins and P1.4 to P1.7 as input pins.

GTU - W-18, 3 Marks

Soln. :

```
#include<lpc21xx.h>
void main()
{
      long  int var=0x0F;
      long int var1=0xF0
      PINSEL1=0X00000000;              // Select Port 1  pins also as General Purpose I/O lines
      IODIR1 = 0X000F0000;
      IOSET1=var;
      IOCLR1=var1;
}
```

4.28 Interfacing DC Motor Using PWM

- A DC motor works by converting electric power into mechanical work. This is accomplished by forcing current through a coil and producing a magnetic field that spins the motor.

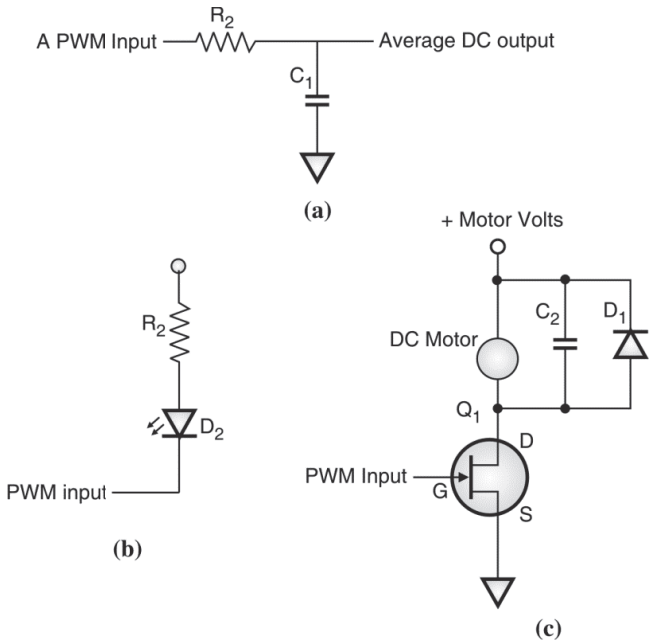


Fig. 4.28.1 : Use of PWM output

- The speed of the motor is dependent on the voltage given to it.
- Hence by varying the voltage to DC motor we can control the speed of DC motor.
- This is mainly implemented by PWM (Pulse Width Modulation)
- In PWM, a variation in the duty cycle generates variation in the average DC voltage.
- Pulse Width Modulation (PWM) is one of the methods to provide Digital to Analog Conversion, by controlling the duty cycle of the square wave provided at the output the average voltage varies and hence a DC voltage at the output of the filter available is dependent on the pulse width (counter digital value).
- A RC circuit provides filtering and gives the average DC voltage proportional to the count as shown in Fig. 4.28.1(a). The Fig. 4.28.1 (b) shows how it can be used to turn on a LED and then to control its intensity by varying the voltage (inversely proportional) across it. The Fig. 4.28.1 (c) shows how it can be used to turn on a DC motor and then to control its speed by varying the voltage across it.

Program 4.28.1 : Write a program for switching ON and OFF a DC motor using a relay for a common delay and show the interfacing of the same.

OR

Write a program for switching ON and OFF a relay for a common delay and show the interfacing of the same. **GTU : S-16, 7 Marks**

Soln. :

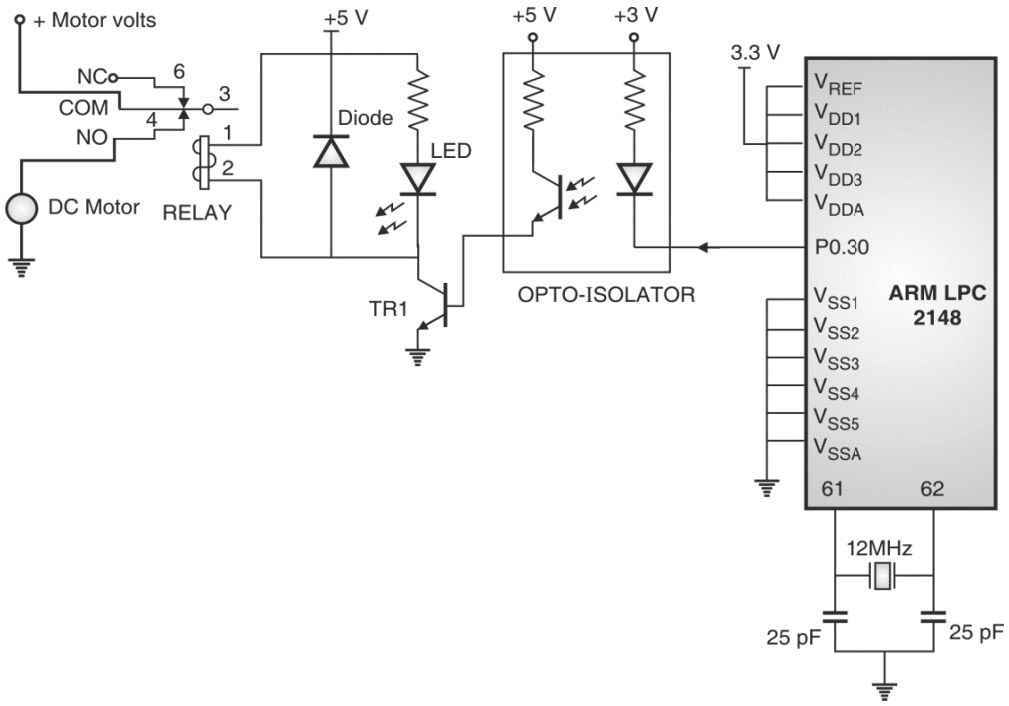


Fig. P. 4.28.1

```
# include  <lpc214X.H>
void delay()                          // Delay function
 {
      unsigned int i=20000;           //by changing the delay you can control the speed of the motor
      while(i--)  ;
}
void main()
 {
      IODIR0=1<<30                    // Configure the port 0 pin 30 as output
      while(1)
      {
           IOSET0=1<<30;               //Switch off motor
          delay();                            //Call delay function
          IOCLR0=1<<30;                       // Switch on motor
          delay();                            //Call delay function
      }
 }
```

4.29 Interfacing GSM Module Using UART

- The LPC2141/42/44/46/40 each contain two UARTs. In addition to standard transmit and receive data lines, the LPC2144/46/48 UART1 also provides a full modem control handshake interface.
- Compared to previous LPC2000 microcontrollers, UARTs in LPC2141/42/44/46/48 introduce a fractional baud rate generator for both UARTs, enabling these microcontrollers to achieve standard baud rates such as 115200 with any crystal frequency above 2 MHz. In addition, auto-CTS/RTS flow-control functions are fully implemented in hardware (UART1 in LPC2144/46/48 only).

Features

- 16 B Receive and Transmit FIFOs
- Register locations conform to 16C550 industry standard.
- Receiver FIFO trigger points at 1 B, 4 B, 8 B, and 14 B
- Built-in fractional baud rate generator covering wide range of baud rates without a need for external crystals of particular values.
- Transmission FIFO control enables implementation of software (XON/XOFF) flow control on both UARTs.
- LPC2144/46/48 UART1 equipped with standard modem interface signals. This module also provides full support for hardware flow control (auto-CTS/RTS).

4.29.1 GSM Modem and its Interfacing with Microcontroller

- GSM (Global System for Mobile Communication) is a digital mobile telephony system. GSM modem digitizes and compresses the data, then sends it through a channel.
- Fig. 4.29.1 shows interfacing of a GSM modem with microcontroller through UART. To communicate over UART, we just need three signals viz., RXD (receive), TXD (transmit), GND (common ground). GSM modem interfacing with microcontroller can be used for various applications like SMS control of industrial equipments.

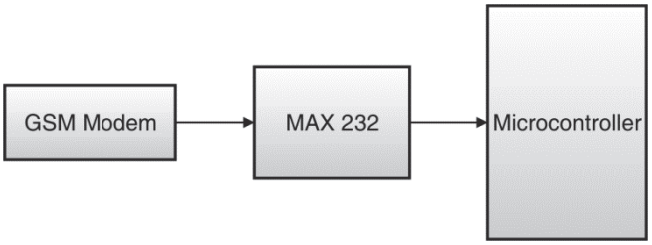


Fig. 4.29.1 : Interfacing GSM modem to a microcontroller

4.29.2 Interfacing GSM with LPC2148

In LPC2148 microcontroller we have two serial interfaces that i.e. the UART0 and the UART1. We can use either of them, but in the following example we will be using UART0.

Program 4.29.1 : Write a program to send a SMS using a GSM modem and show the interface for the same.

Soln. :

Fig. P. 4.29.1 shows the interfacing of the LPC2148 with the GSM modem through MAX232 a UART driver.

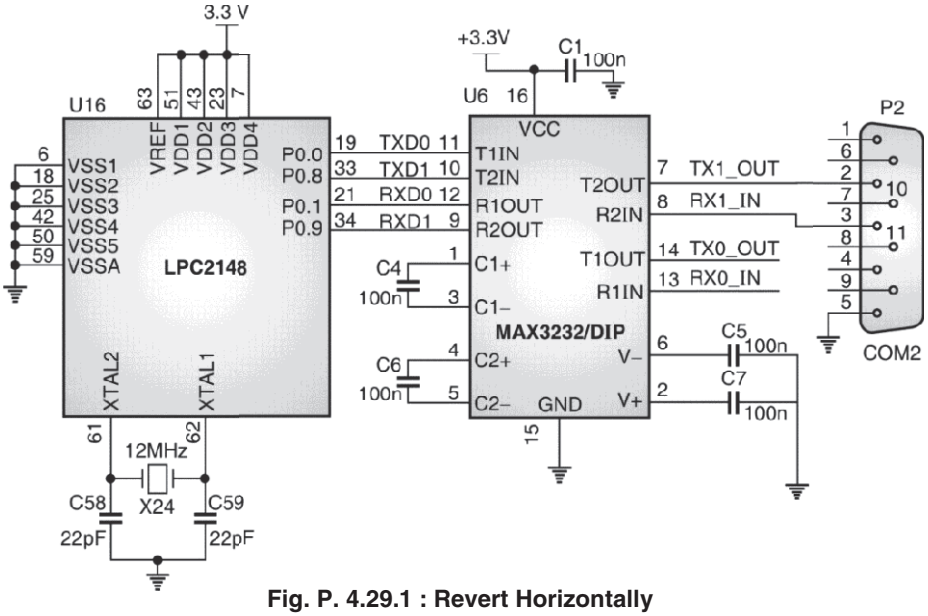


Fig. P. 4.29.1 : Revert Horizontally

```
#include <lpc21xx.H>
#include <stdio.h>
void input(unsigned char *array)
{
    unsigned char temp=0, i=0;
    do
    {
        temp = getchar();
        *array = temp;
        array++;
    }
    while((temp != '\r') && (temp != '\n'));
    *array = '\0';
}
char getchar ()  /* Read a character from the serial Port
{
     while (!(U0LSR & 0x01))
     {
    }
     return (U0RBR);
}
```

```
void status()
{
      input(y);
      while(!(strstr(y,"OK")))
      {
           input(y);
      }
}
void InitSerial()
{
       PINSEL0 = 0X00000005;                    //Enabling TxD0 & RxD0
      U0LCR    = 0x00000083;                    //Configure in 8-bit data, no parity and 1-stop bit mode
       U0DLL   = 0x00000061;                    //Initialize baud rate=9600
           U0LCR = 3;
}
void delay(unsigned int n)
{
       unsigned int i,j;
       for(i=0;i<n;i++)
       for(j=0;j<10000;j++);
}

void main()
{
       unsigned int cnt=0x80,m;
       char xx;
       InitSerial();
       delay(50);
       while(1)
       {
            printf("AT\r");                     // Send GSM AT command for initializing GSM modem
           status();                            //Check is sent properly.
```

```
        printf("AT+IPR=9600\r");            // Send GSM AT command to initialize the baud rate
        status();
        printf("AT+CMGR=2\r");              // Send command to read the message detail
                                            // at Index 1 with phone number, data and time
        status();
        delay(250);                         //Call delay function

        printf("ATD9790550124;\r");         //Send command for     call dialing
        delay(250);
        status();
        delay(3500);
        printf("ATH\r");                    // Send command for call disconnecting
        delay(250);
        status();
         delay(1000);
        printf("ATDL\r");                     // Send AT command for redialing
        delay(250);
        status();
        delay(1000);
        printf("ATH\r");                    // Send AT command for answering the call
        delay(250);
        status();
        delay(1000);
    }
}
```

4.30 Interfacing EEPROM using I2C Communication Protocol

- The I2C (Inter-IC) bus is a bi-directional serial communication protocol used for communication between the integrated circuits (ICs).
- It is developed by Phillips and is a synchronous protocol for serial communication. LPC2148 has a I2C port, hence it can be easily interfaced with the I2C EEPROM. Fig. 4.30.1 shows the interfacing of LPC2148 with I2C EEPROM.

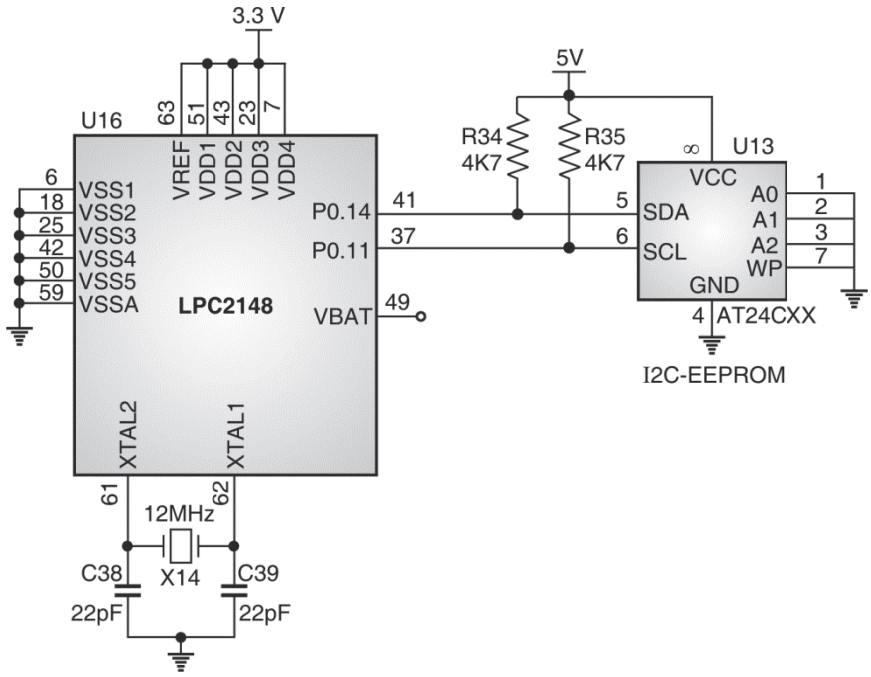


Fig. 4.30.1 : I2C EEPROM interfacing with LPC2148

4.31 Interfacing Memory Using SPI Protocol

- Let us first understand the interfacing technique of the SPI.

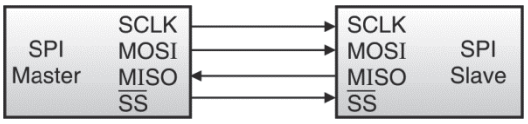


Fig. 4.31.1

- The Serial Peripheral Interface (SPI) bus is a synchronous serial communication protocol that supports full duplex (simultaneous bi-directional) communication. The SPI communication works in a special manner wherein, one device works as a master while the other works as slave. There can also be multiple slave devices.

The SPI bus specifies four logic signals

- SCLK: Serial Clock (output from master);
- MOSI; SIMO: Master Output, Slave Input (output from master);
- MISO; SOMI: Master Input, Slave Output (output from slave);
- SS: Slave Select (active low, output from master).
- Data transmission between the master and the slave is done as shown in the Fig. 4.31.2.
- In this case the first task is the configuration of clock which is done by the master device. The slave device is then selected by the master. The frequency commonly used is in the range of 1 to 70 MHz.

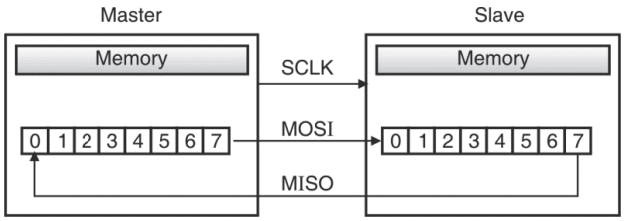


Fig. 4.31.2

The communication is then done in the following steps:

1. The master gives out a data bit on the MOSI line on which the slaves reads this data, as shown in Fig. 4.31.2.
2. The slave sends the data bit as shown in Fig. 4.31.2, which is received by the master.

   This process is implemented using the two shift registers, one in each the master and the slave.

4.31.1 Interfacing SD-MMC with LPC2148 using SPI

- Thus the interfacing of the SD-MMC memory card as discussed earlier can be done using the SPI protocol with LPC2148. Fig. 4.31.3 shows the interfacing of the SD-MMC card with LPC2148.
- The SD-MMC card can be connected at the jumper shown at the right end in the Fig. 4.31.3. LPC2148 has a SPI port whose pins are available on port 0 pins. Hence the SD card is connected on this port pins of LPC2148.
- These pins are port 0 pins 4, 5, 6 and 7 working as SCLK, MISO, MOSI and SD_CS respectively.

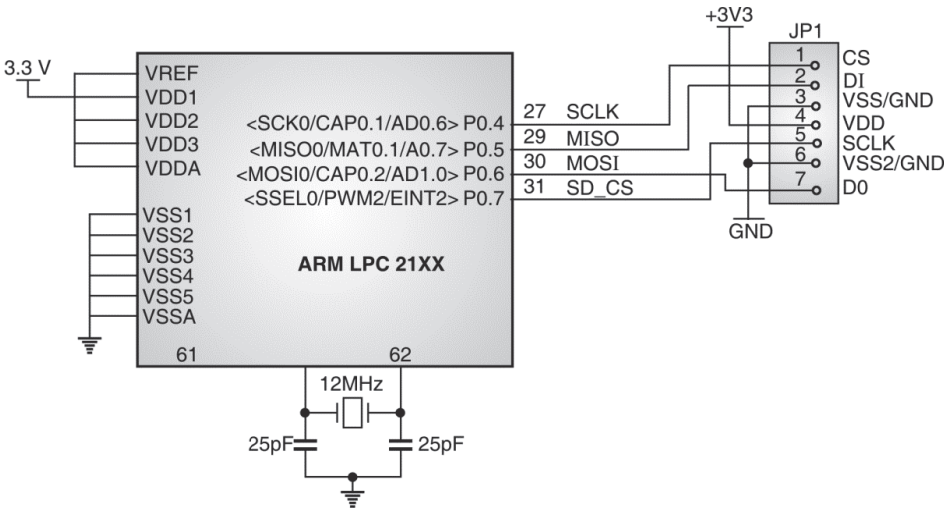


Fig. 4.31.3 : SD-MMC card interfacing with LPC2148

4.32 Interfacing Internal ADC

- LPC2148 has two on-chip ADCs. We will make use of the ADC0 and convert the analog signal given on its channel 3, to its equivalent digital value.
- In the Program 4.32.1, we will connect a potentiometer to the analog input of the ADC and convert it to its digital equivalent value.

Program 4.32.1 : Write a program to interface a potentiometer to the LPC2148 and measure the analog voltage and convert to its digital value.

Soln. :

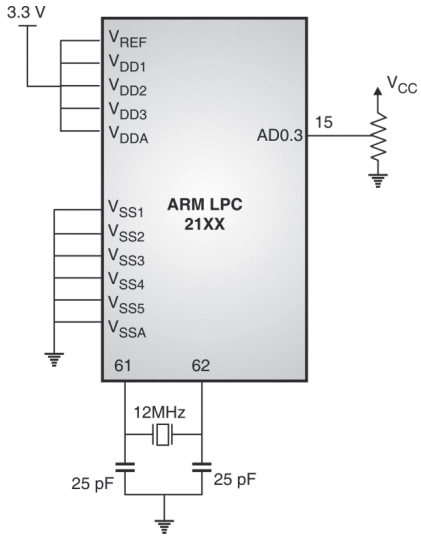


Fig. P. 4.32.1

```
#include "lpc214x.h"                                    //This is the header file that contains all the
                                                        //information about the microcontroller LPC2148
void InitADC()                                          //Function to initialize ADC
{
      PCB_PINSEL1 = (PCB_PINSEL1 & ~PCB_PINSEL1_P030_MASK) |
      PCB_PINSEL1_P030_AD03;                            //Enable power supply for ADC0
      SCB_PCONP |= SCB_PCONP_PCAD0;                     //Initialise ADC
      AD0_CR = AD_CR_CLKS10 | AD_CR_PDN | ((3 - 1) << AD_CR_CLKDIVSHIFT)| AD_CR_SEL3;
                                                        /*Enable 10-bit precision ADC, Exit power-down mode
                                                                4.0MHz & Use channel 3*/
}
 int ReadADC()                                          // Function to read the digital equivalent value from
ADC
{
      AD0_CR &=  ~(AD_CR_START_MASK | AD_CR_SELMASK);
                                                        //Stop all conversions and deselect all channels
      {
          AD0_CR |=  (AD_CR_START_NONE | AD_CR_SEL3);       // Select channel 3 of ADC0
          AD0_CR |=   AD_CR_START_NOW;
```

```
                                                    //Start conversion
        while (!(AD0_DR3 & AD_DR_DONE))  ;
                                                    // Wait for the completion of the conversion
        return ((AD0_DR3 & AD_DR_RESULTMASK)>>AD_DR_RESULTSHIFT);
                                                    // Return the digital result read from ADC
    }
}
void  main()
{
    InitADC() ;                                     // Call the function to initialise ADC 0, Channel 3
    int results = 0;                                // Initialize the result to zero
    while (1)                                       //Always true infinite loop to read the ADC digital data
    {
        results = ReadADC();
    }
}
```

4.32.1 Interfacing DAC

GTU - S-19

Q. Explain interfacing of DAC with ARM processor. **(S-19, 2 Marks)**

- The DAC 0808 is an 8-bit current output monolithic DAC manufactured by the National semiconductor corporation.
- It is a 16 - pin IC available in dual in line DIP plastic package. The analog output is available in the form of current I_o. That means I_o is proportional to the 8-bit digital input. The important features of DAC 0808 are as follows :

Features of DAC 0808

- Fast settling time — 150 nsec. typically
- Power supply voltage range — ± 4.5 mW at ± 5V
- Low power consumption. — 33 mW at ± 5V.
- High speed multiplying input slew rate — 8 mA / μ sec.

Interfaces directly with TTL, DTL and CMOS logic levels.

Pin configuration and functional block diagram

- The pin configuration and functional block diagram of DAC 0808 are as shown in Figs. 4.32.1(a) and (b) respectively.

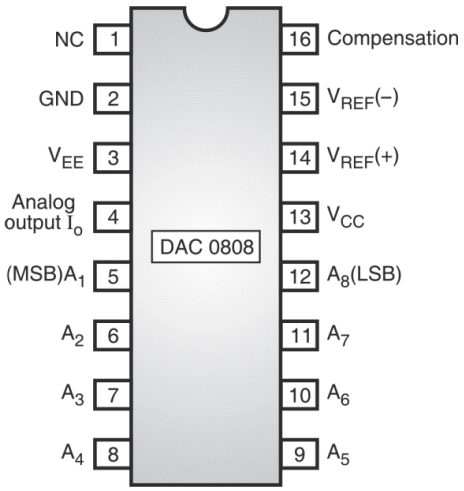


Fig. 4.32.1(a) : Pin configuration of DAC 0808

- The internal block diagram shows that DAC 0808 consists of R-2R ladder along with current switches and reference current amplifier.
- A_1 to A_8 are the 8-digital input lines with A_1 as the most significant bit and A_8 as the least significant bit.
- The analog output is available in the form of current I_o, therefore we need to use an external current to voltage converter if the analog output in the form of voltage is required.
- DAC 0808 requires a dual polarity (±) supply voltage, typically ± 15V, for its operation. The reference voltage can be either positive or negative.
- An external reference voltage should be applied to either V_{REF} (+) or V_{REF} (–) depending on the polarity of reference voltage.

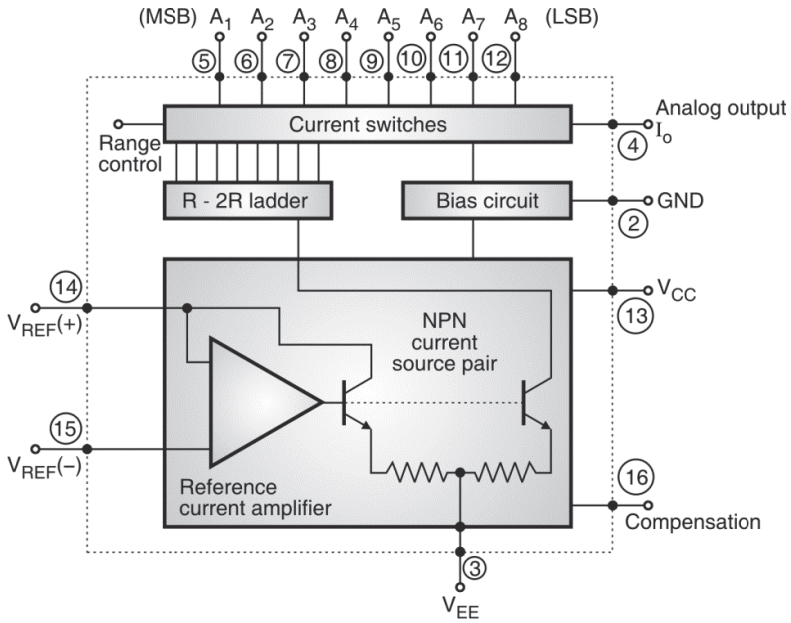


Fig. 4.32.1(b) : Functional block diagram of DAC 0808

Program 4.32.2 : Write a program to generate a square wave and show the interfacing of a DAC 0808 with LPC2148.

Soln. :

Fig. P. 4.32.2 shows the interfacing of LPC2148 with DAC0808.

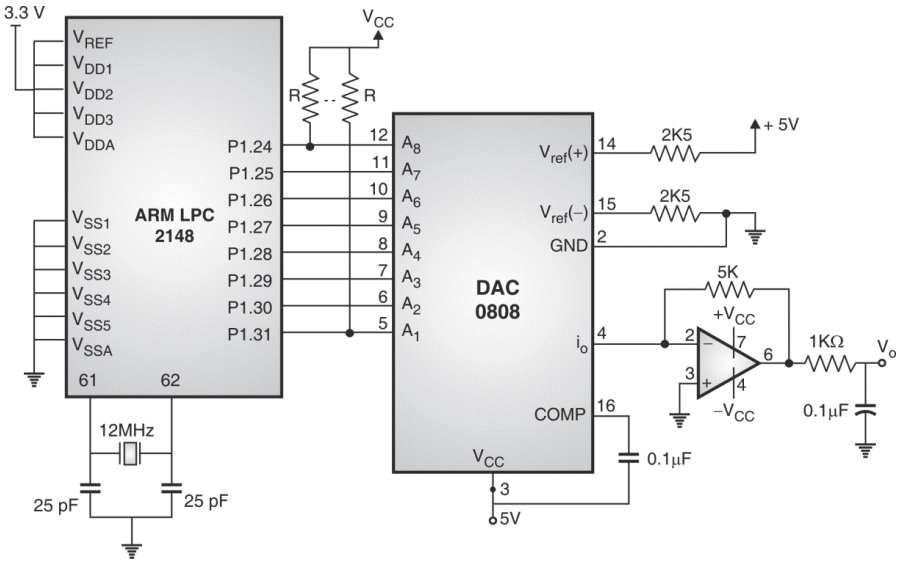


Fig. P. 4.32.2

The waveform generated will be seen on the V_0 pin of the OPAMP across the capacitor. The program will be same as the LED blinking. The delay can be varied to vary the frequency of the square wave. The program is given below.

```
#include<lpc2148.H>                 //This is the header file that contains all the
                                    //information about the microcontroller LPC2148
 void delay()                       // Delay function
{
      int i, j;
      for(i=0;i<=250;i++)
      for(j=0;j<=250;j++)  ;
}
void main()
{
       PINSEL2 = 0X00000000;        // Set P1.24 to P1.31 as General purpose I/O
       IO1DIR = 0XFF000000;         // Configure Port pins P1.24 to P1.31 as
                                    // Output pins
      while(1)                      // Always true infinite loop
      {
          IO1SET=0XFF000000;
                                    // Make the output pins to logic 1
          delay();                  // call delay function
          IO1CLR=0XFF000000;        // Make the output pins to logic 0
          delay()   ;               // call delay function
      }
}
```

Program 4.32.3 Write a program to generate a ramp wave and show the interfacing of a DAC 0808 with LPC2148. Write program to generate ramp waveform. **GTU- W-16, S-19**

Soln. :

Fig. P. 4.32.3 shows the interfacing of LPC2148 with DAC0808.

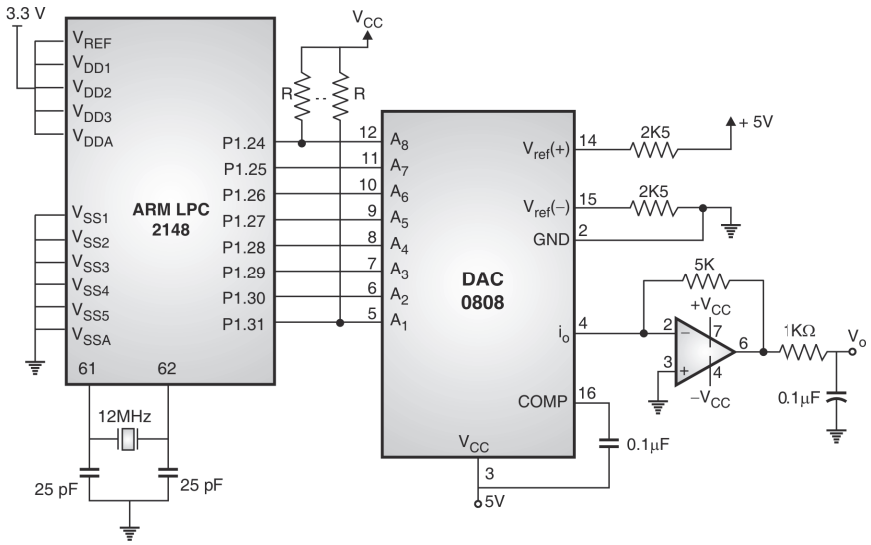


Fig. P. 4.32.3

```
#include<lpc2148.H>                    //This is the header file that contains all the
                                       //information about the microcontroller LPC2148
 void delay()                          //Delay function
      {
           int i;
           for(i=0;i<=25;i++);     }
      void main()
      {
           unsigned int x=0;
           PINSEL2 = 0X00000000;       //Set   P1.24 TO P1.31 as General purpose I/O
           IO1DIR = 0XFF000000;        //Configure Port pins  P1.24 to P 1.31 as Output pins
           while(1)                    //Always true infinite loop
      {
           IO1SET=x<<24;               // Give value of x to Port 1
           delay();                    // Call delay function
           x++;                        // Increment the count and hence the equivalent analog
                                       // value
      if(x==255) x=0;                  // If count reached max value (i.e. 0xff or 255), reset it.
      }
}
```

4.32.2 Interfacing Sensors to LPC 2148

- In most of the industrial applications the parameter to be measured is temperature. Thus we need a temperature sensor that will convert the physical quantity i.e. temperature into its equivalent electrical value. They are used to measure the accurate temperature and hence control it in various industrial applications.
- The semiconductor IC LM34 is a precision temperature sensor, whose output voltage is directly proportional to the temperature in Fahrenheit. The semiconductor IC LM35 is a precision temperature sensor, whose output voltage is directly proportional to the Celsius (Centigrade) temperature.
- Another special advantage of LM35 is that it does not require any external calibration and the temperature range for which it operates is −55 to +150°C.
- The LM35 is rated to operate over a – 55°C to +150°C temperature range.
- As shown in the Fig. 4.32.2 the connection for LM35 is very simple. Also the output scale is linear with a change of 10mV/°C.

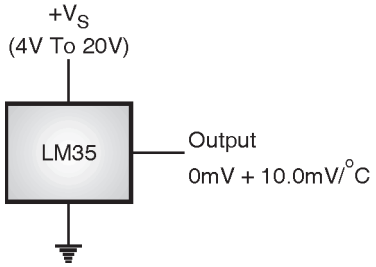


Fig. 4.32.2 : Circuit diagram for the LM35 basic temperature sensor (+2° C to +150° C)

Program 4.32.4 : Write a program to read the temperature and show the interfacing of a sensor with LPC2148.

Soln. :

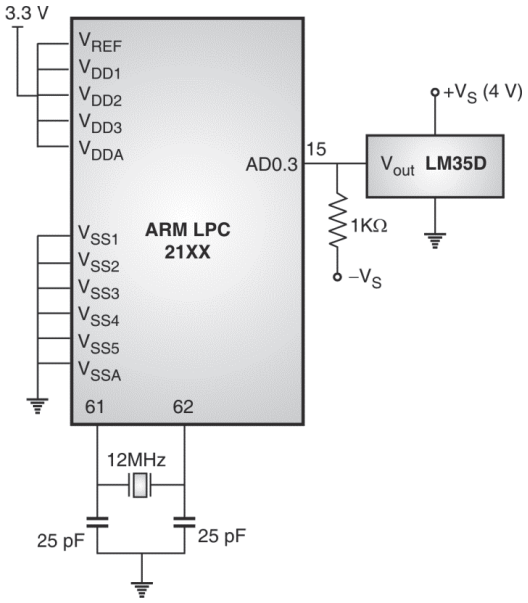


Fig. P. 4.32.4

As discussed earlier, the output of the sensor LM35 is a voltage proportional to the temperature. This analog voltage can be measured using an ADC. Thus the program for the interfacing of a temperature sensor will be same as that of the interfacing of ADC as seen in Section 4.32. The program is given below. The variable result holds the digital value equivalent to the temperature measured by the ADC.

```
#include "lpc214x.h"                    //This is the header file that contains all the
                                        //information about the microcontroller LPC2148
void InitADC()                          //Function to initialize ADC
{
    PCB_PINSEL1 = (PCB_PINSEL1 & ~PCB_PINSEL1_P030_MASK) |
    PCB_PINSEL1_P030_AD03;              // Enable power supply for ADC0
    SCB_PCONP |= SCB_PCONP_PCAD0
                                        // Initialise ADC
    AD0_CR = AD_CR_CLKS10 | AD_CR_PDN |
    ((3 - 1) << AD_CR_CLKDIVSHIFT)
    | AD_CR_SEL3;
                        // Enable 10-bit precision ADC, Exit power-down mode 4.0MHz & Use channel 3
}
int ReadADC()
                        // Function to read the digital equivalent value from ADC
{
    AD0_CR &=  ~(AD_CR_START_MASK | AD_CR_SELMASK);
                        //Stop all conversions and deselect all channels
    {
        AD0_CR |=  (AD_CR_START_NONE |
                    AD_CR_SEL3);
                        // Select channel 3 of ADC0
        AD0_CR |=   AD_CR_START_NOW;
                        //Start conversion
        while (!(AD0_DR3 & AD_DR_DONE))  ;
                        // Wait for the completion of the conversion
        return ((AD0_DR3 & AD_DR_RESULTMASK)>>AD_DR_RESULTSHIFT);
                        // Return the digital result read from ADC
    }
```

```
}
void  main()
{
     InitADC() ;
                                // call the function to initialise ADC 0, Channel 3
     int results = 0;           // Initialize the result to zero
     while (1)                  //Always true infinite loop to read the ADC digital data
     {
          results = ReadADC();
     }
}
```

❑❑❑

CHAPTER 5

UNIT V

Cache and Memory Management and Protection

Syllabus

Memory Technologies, Need for memory Hierarchy, Hierarchical Memory Organization, Virtual Memory. Cache Memory, Mapping Functions. Cache Design, Unified or split cache, Multiple level of caches, ARM cache features, Coprocessor 15 for system control. Processes, Memory Map, Protected Systems, ARM systems with MPU, memory Protection Unit (MPU). Physical Vs Virtual Memory, Paging, Segmentation. MMU Advantage, Virtual memory translation, Multitasking with MMU, MMU organization, Tightly Coupled Memory (TCM).

5.1 Memory Technologies

When a memory is taken then there are various characteristics of this memory that are considered. The characteristics of memory are based on the following :

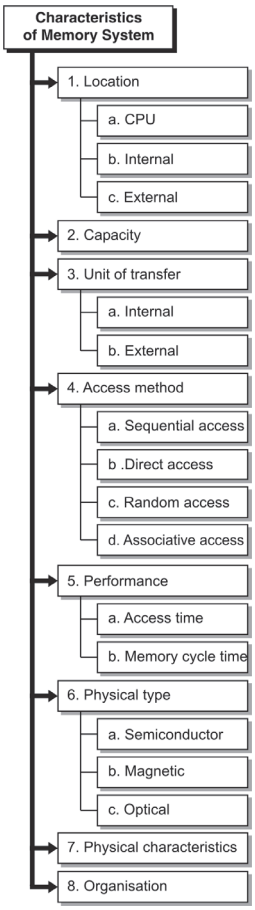


Fig. 5.1.1 : Characteristics of memory system

1. **Location :** The memory can be located in one of the following :
   - (a) **CPU :** This includes CPU registers and on-chip cache memory.
   - (b) **Internal :** This includes the memory that the processor can directly access.
   - (c) **External :** This is normally removable or virtual memory and hence access is slower.
2. **Capacity :** It is measured in terms of the word size and the number of words. Word size is the size of each location. Number of words is the number of locations.
3. **Unit of transfer :** This refers to the size of the data that is transferred in one clock cycle. It mainly depends on the data bus size. The data as discussed earlier may be internal or external and accordingly will be the data to be transferred in one clock pulse :
   - (a) **Internal :** It is related to the communication of data with the memory directly accessible. It is usually governed by data bus width.
   - (b) **External :** This is the data communication with the external removable memory or virtual memory. It is usually a block which is much larger than a word.
4. **Access method :** There are various methods of accessing the memory based on the memory organization. These methods are listed below with examples :
   - (a) **Sequential access :** The sequential access means start from the beginning and read through in order until the byte to be read is reached. Hence the access time depends on location of data and previous location accessed. For Example, magnetic tape. If one wants to listen to the third stanza of the fourth song stored in an audio cassette, he has to go through the entire first song second and the third song, and then the first stanza, second stanza of the third song and then reaches to the third stanza of that song.
   - (b) **Direct access :** Here individual blocks have unique address and the access is done by jumping to vicinity plus sequential search. Hence access time depends on location and previous location. For Example magnetic or optical disk. Let take the same example that a person wants to listen to the third stanza of the fourth song on a CD, then he can directly reach to the fourth song, but thereafter he has to access the stanzas of the song sequentially to reach to the third stanza.
   - (c) **Random access :** In case of random access individual addresses identify locations exactly. Hence the access time is independent of location or previous access. For example ROM. In case of a ROM, any location to be accessed can be directly reached to without going through the locations sequentially.
   - (d) **Associative access :** Here the data is located by a comparison with contents of a portion of the stored data(address). Hence the access time is independent of location or previous access. For example cache. In case of cache memory, each location has a tag associated with it, and to reach to the required location the tags are to be compared with the location to be accessed. There are techniques used to reach to the required tagged location at a faster speed.
5. **Performance :** The performance of the memory depends on its speed of operation or the data transfer rate. The data transfer rate is the rate at which the data is transferred. The speed of operation depends on two things :
   - (a) **Access time :** The time between providing the address and getting the valid data from memory is called as its access time i.e. the address to data time.
   - (b) **Memory cycle time :** The time that is required for the memory to "recover" before next access i.e. the time between two addresses is called as memory cycle time.

6. **Physical type :** The physical material using which the memory is made can be different like :
   - (a) **Semiconductor :** Memory can be made using semiconductor material i.e. ICs, for example ROM.
   - (b) **Magnetic :** Memory can also be made using magnetic read and write mechanism, for example Magnetic disk and Magnetic tape.
   - (c) **Optical :** Optical memories i.e. memories that use optical methods to read and write have become famous these days, for example CD and DVD.
   - (d) There are some other methods using which data was stored in early days like Bubble and Hologram.
7. **Physical characteristics :** The physical characteristic of memory is also an important aspect to be considered. This includes the volatility, power consumption, erasable / not erasable, etc.
8. **Organization :** It is not that always the memory will be organized sequentially. There are some other types of memory organization like interleaved memory, etc. Interleaved memory is used in microprocessor 8086.

5.2 Memory Hierarchy : Need and Organization

GTU - W-14

Q. Explain in details Memory Hierarchy with examples. **(W-14, 6 Marks)**

- Memory Hierarchy explains that the nearer the memory to the processor, faster is its access. But costlier the memory becomes as it goes closer to the processor. The following sequence is in faster to slower or costlier to cheaper memory:
  1. Registers i.e. inside the CPU.
  2. Internal memory that includes one or more levels of cache and the main memory. Internal memory is always ROM, SROM for cache and DROM for main memory. The differences between the SROM and DROM will be seen in a later section in this chapter. This is also called as the primary memory.
  3. External memory or removable memory includes the hard disk, CDs, DVDs etc. This is the secondary memory.

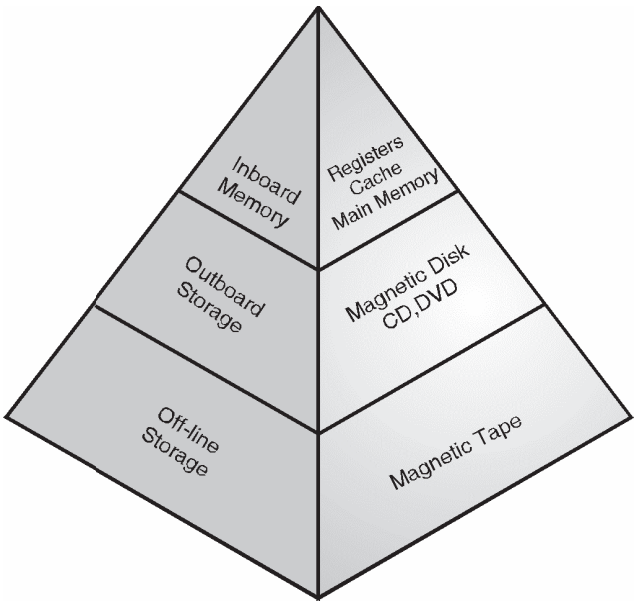


Fig. 5.2.1 : Memory Hierarchy

- Fig. 5.2.1 shows the memory hierarchy based on the closeness to the processor. The registers as discussed are the closest to the processor and hence are the fastest while off-line storage like magnetic tape are the farthest and also the slowest. The list of memories from closest to the processor to the farthest is given as below :

  1. Registers
  2. L1 Cache
  3. L2 Cache
  4. Main memory
  5. Magnetic Disk
  6. Optical
  7. Tape

- To have a large faster memory is very costly and hence the different memory at different levels gives the memory hierarchy. How does this memory hierarchy give faster operation and some other terms like cache etc. will be understood in the subsequent sections.

5.3 Cache Memory : Concept and Cache Consistency

GTU - W-17, S-19, W-19

Q.	Define cache memory. Explain the importance of cache memory.	**(W-17, W-19, 3 Marks)**
Q.	What is cache performance? On which factor it depends ?	**(S-19, 3 Marks)**

We will see some basics of the cache like its operation, advantage, principles of locality of reference, cache architectures, write policies etc.

5.3.1 Cache Operation

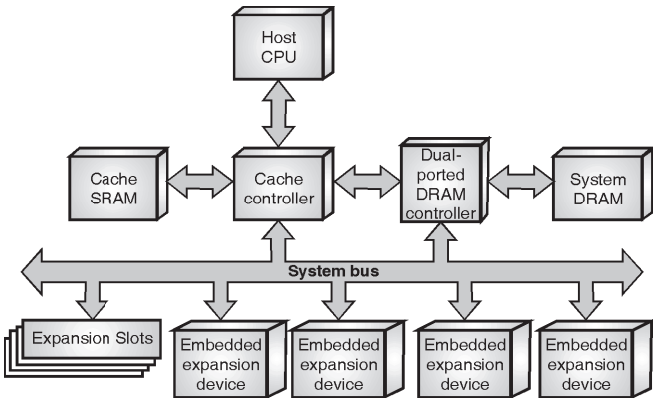


Fig. 5.3.1 : Cache Memory connection in the System

1. Implementation of cache memory subsystem is an attempt to achieve almost all accesses with zero wait state while accessing memory, but with an acceptable system cost.

2. The cache controller maintains a directory to keep a track of the information and it has copied into the cache memory.
3. When the processor initiates a memory read bus cycle, the cache controller checks the directory to determine if it has a copy of the requested information in cache memory.
4. If the copy is present, the cache controller reads the information from the cache, sends it to the processors data bus, and asserts the processor's ready signal. This is called as **READ HIT**.
5. If the cache controller determines that it does not have a copy of the requested information in its cache, the information is now read from main memory (DROM). This is known as **READ MISS** and causes wait states due to slow access time of DROM.
6. The requested information is from the DROM given to the processor. The information is also copied into the cache memory by cache controller and it updates its directory to track the information stored in cache memory.

   Assume the cache memory is empty, in the beginning (after reset).

The following sequence takes place :

1. The processor performs a memory read cycle to fetch the first instruction from memory.
2. The cache controller uses the address issued by the processor to determine if a copy of the requested information is already in the cache memory. But a cache miss occurs as the cache memory is empty.
3. The cache controller initiates a memory read cycle to fetch the requested information from DROM memory. This will consume some wait states.
4. The information from DROM memory is sent to the processor. It is also copied into the cache memory and the cache controller updates its directory to reflect the presence of the new information. The information being sent is not just the required instruction, but a block (line) of data is sent to the cache. No performance gain is achieved due to absence of information in cache memory.
5. After executing the first instruction, the processor's prefetched requests a series of memory read bus cycles to fetch the remaining instructions in program loop. But since an entire line was brought earlier, most of the required instructions will be in cache and hence resulting in faster access and performance gain. If the cache is sufficiently large, all instructions in the program loop can become resident in cache memory.
6. The program has loop instruction to jump to the beginning of the loop start over again. The processor then requires the same program again.
7. When the processor requests for the first instruction in the loop, cache controller detects the presence of the instruction in the cache memory and hence provides it to the processor with zero wait states.

5.3.2 Principles of Locality of Reference

1. **Locality of reference** is the term used to explain the characteristics of programs that run in relatively small loops in consecutive memory locations.

2. The locality of reference principle comprises of two components :

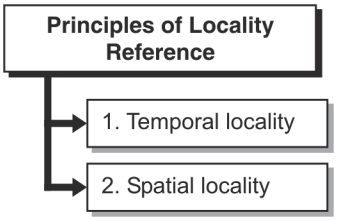


Fig. 5.3.2 : Two components of locality of reference

(1) Temporal locality

(a) Since the programs have loops, the same instructions are required frequently, i.e. the programs tend to use the most recently used information again and again.

(b) If for a long time a information in cache is not used, then it is less likely to be used again.

(c) This is known as the principle of temporal locality.

(2) Spatial locality

(a) Programs and the data accessed by the processor mostly reside in consecutive memory locations.

(b) This means that processor is likely to need code or data that are close to locations already accessed.

(c) This is known as the Principle of Spatial Locality.

3. The performance gains are realized by the use of cache memory subsystem are because of most of the memory accesses that require zero wait states due to principles of locality.

5.3.3 Cache Performance

GTU - W-17

Q. Explain the importance of cache bit and buffer bit. **(W-17, 4 Marks)**

1. Performance of cache subsystems depends on the frequency of cache hits, usually termed as **hit rate**.

$$\% \text{ HIT RATE} = \frac{\text{Cache Hits}}{\text{Total Memory Accesses}} \times 100\ \%$$

2. If a program requires a small area of memory and consists of loops, then maximum cache hits are possible.
3. On the other hand, if the program has non-looping code, many accesses will result in cache misses.
4. Fortunately, most programs run in loops and hence a high percentage of cache hit (85 – 95%) is experienced.
5. Besides locality of reference various other factors contribute to cache hit rates like Cache's architecture, Size of cache memory and Cache memory organization.

5.3.4 Cache Design

Two basic architectures are found in today's systems :

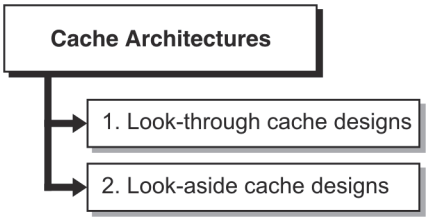


Fig. 5.3.3 : Two basic cache architectures

5.3.4(A) Look-Through Cache Designs

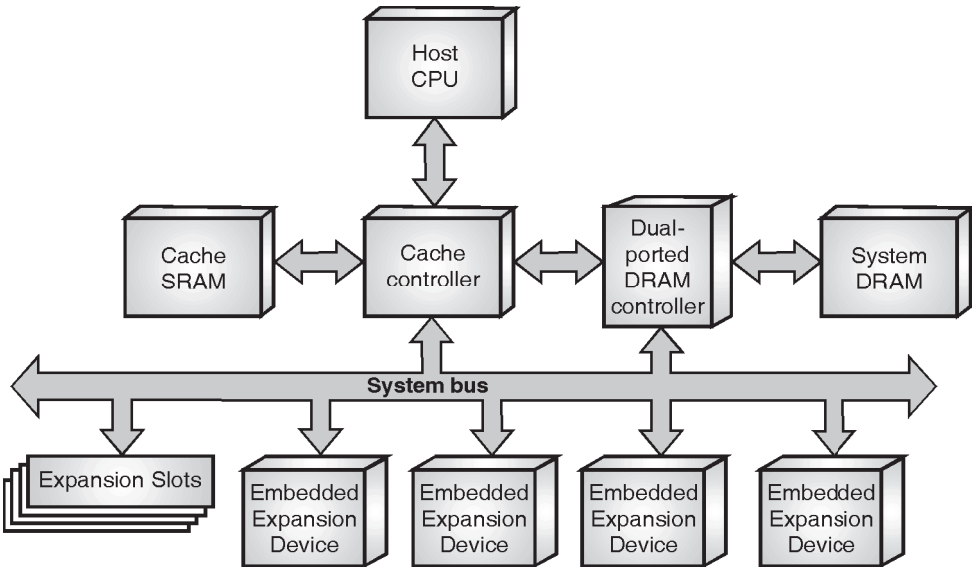


Fig. 5.3.4 : Look-through cache designs

(a) The performance of systems incorporating Look Through Cache is typically higher than that of systems incorporating Look Aside Cache.

(b) Data from main memory (DROM) is not transferred to the processor using system bus hence system bus is free for other bus masters (like DMAC) to access the main memory.

(c) This system isolates the processor's local bus from the system bus hence achieving bus concurrency.

(d) The major advantage is that two bus masters can operate simultaneously. One processor accesses look through cache while another bus master such as DMA can access the system bus is possible.

(e) To expansion devices, a look-through cache controller is like a system processor.

(f) During memory writes, look-through cache provides zero wait state operation (using posted writes) for write misses.

Advantages of Look-Through Cache Designs

(a) It reduces the system and memory bus utilization, leaving them available for use by other bus master.

(b) It allows bus concurrency, where both the processor and another bus master can perform bus cycles at the same time.

(c) It also completes write operations in zero wait states using posted writes.

Disadvantages of Look-Through Cache Designs

(a) In the event that the memory request is a cache miss, the lookup process delays the request to memory. This delay is called as lookup penalty.

(b) It is more complex, costly and difficult to design and implement.

5.3.4(B) Look-Aside Cache Designs

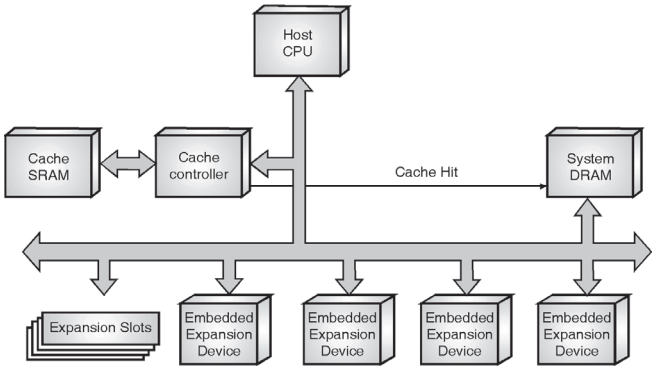


Fig. 5.3.5 : Look-aside cache designs

(a) In this case the processor is directly connected to the system bus or memory bus.

(b) When the processor initiates a bus access, cache controller as well as main memory detects the bus access address.

(c) The cache controller sits aside and monitors each processor memory request to determine if the cache contains the copy of the requested information.

(d) If it is a cache hit, the cache controller terminates the bus cycle by instructing memory subsystem to ignore the request. If it is a cache miss, the bus cycle completes in normal fashion from memory (and wait states are required).

Advantages of Look-Aside Cache Designs

(a) Cache miss cycles complete faster in Look Aside Cache as the bus cycle is already in progress to memory and hence no look up penalty is incurred.

(b) Simplicity of designs because only one address is to be monitored by cache controller form processor and not from I/O devices.

(c) Lower cost of implementation due to their simplicity.

Disadvantages of Look-Aside Cache Designs

(a) The processor requires system bus utilization for its every access, to access both cache subsystem and memory.

(b) Concurrent operations are not possible as all masters reside on the same bus.

5.3.5 Cache Consistency (Also Known as Cache Coherency)

1. In order to work properly for the cache subsystems, the CPU and the other bus masters must be getting the most updated copy of the requested information.

2. There are several cases wherein the data stored in cache or in main memory may be altered whereas the duplicate copy remains unchanged.

Causes of cache consistency problems

1. When the copy of line in cache, no longer matches the contents of line stored in memory, there is loss of cache consistency. It can be either due to cache line being updated while the memory line is not, or the memory line being updated while the cache line is not.
2. In each of these instances the stale data must be updated. It can be a result of cache write hit and hence the caches write policy has to handle this problem for the first case.
3. For the second case the coherency problem is due to some other bus master changing the data in memory. This change is to be updated in cache line by the cache controller, hence the cache controller has to monitor the system bus.

5.3.6 Write Policy

1. When the write hit occurs, the cache memory is updated and it contains the latest data while memory contains stale data.
2. Such a cache line is called as dirty or 'modified' because it has no longer mirrors of its corresponding line in memory.
3. In order to correct this cache consistency problem, the corresponding memory line must be updated to reflect the change made in the cache; else another bus master may get stale data if it reads from these lines.
4. Three write policies are used to prevent this type of consistency problem: Write-through, Buffered or posted write-through and Write-back.

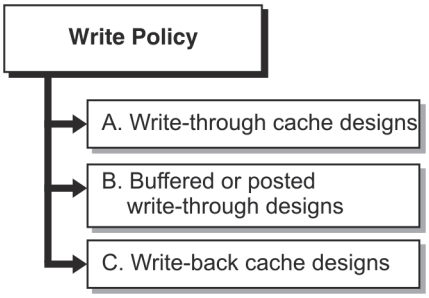


Fig. 5.3.6 : Write policy

A. Write-Through Cache Designs

1. In this write policy, the data is passed to the memory immediately, so that the memory has the updated data.
2. Even on write hit operation, the cache controller updates the line in the cache and the corresponding line in memory, and hence ensuring that consistency is maintained between cache and memory.
3. Very simple and effective implementation.
4. But poor performance due to slow main memory writes operation.
5. Also it doesn't allow bus concurrency.

B. Buffered or Posted Write-Through Designs

1. It has an advantage of providing zero wait state write operation for cache hits as well as cache misses.
2. When a write occurs, buffered write through caches tricks the processor into thinking that the information was written to memory in zero wait states. In fact, the write to main memory has not been performed yet.
3. The look-through cache controller stores the entire write operation in a buffer, and writes to the main memory later. Hence the processor need not perform slow write operation with wait states and hence doesn't impact processor's performance. This is assuming that the posted write buffer is only one transaction deep.
4. But, if there are two back-to-back memory write bus cycles, the cache controller will insert wait states into second bus cycle until the first write to memory has actually been completed. The bus controller will then post the second bus cycle and assert the processor's ready line. But since processor typically writes only one write operation, the memory writes are completed in zero wait states.
5. With this policy, another bus master is not permitted to use the bus until the write-through is completed, thereby ensuring that the bus master will receive the latest information from memory.
   - The write-through operations use either system or memory bus. Hence when write-through to memory is in progress, bus masters are prevented from accessing memory.
   - But actual cache consistency problem occurs only when the bus master reads from a location in memory that is stale. The frequency of this type of occurrence is very less. In fact, the memory line is likely to be updated many times by the processor before another bus master reads from that particular line.
   - As a result the write-through and buffered write-through designs, update memory each time a memory write is performed, although the need for such action may not be required immediately.

C. Write-Back Cache Designs

1. Write-back designs improve the overall system performance by updating a line in main memory only when necessary, thereby keeping the system bus free for use by other processors and bus masters and hence ensuring bus concurrency.
2. The memory is updated only when:
   (a) Another bus master initiates a read operation from a memory line that contains stale data.
   (b) Another bus master initiates a write operation from a memory line that contains stale data.
   (c) The cache line that contains modified information is about to be overwritten in order to store a line newly acquired from memory i.e. during line replacement.
3. Cache controller marks the cache lines as 'modified' in the cache directory when the processor updates them. Hence when read by another master or written into the memory, the cache subsystem checks whether it is marked as 'modified' in cache.
4. They design of such cache controller is COMPLICATED to implement because they must MAKE DECISIONS on when to write 'modified' lines back to memory to ensure consistency.
5. Bus snooping is the method used by cache controller to monitor memory accesses performed by other bus master.
6. The cache controller monitors, or snoops the system bus when another bus master is performing an access to memory so as to maintain cache consistency.
7. A line that is accessed from memory and is also resident in the cache, it is referred to as snoop hit. In such cases either cache or memory line may contain stale data.

5.3.7 Replacement Algorithms

- Replacement algorithm is required to replace a line from the cache memory with the new line as discussed earlier.
- There are various replacement polices available. The widely used ones are LRU, FIFO, LFU and random as shown in Fig. 5.3.7

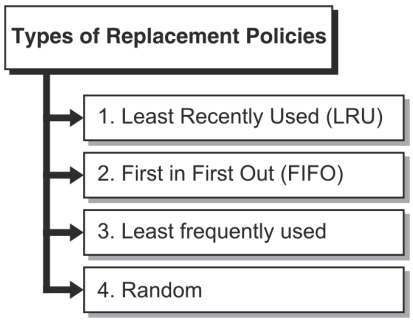


Fig. 5.3.7 : Types of replacement polices

1. **Least Recently used (LRU) :** In this case the line which is least recently used is replaced with the new line. Thus the line which has not been used for longest time is replaced with the new line.
2. **First in first out (FIFO) :** In this case the line which was brought into the cache first is replaced first. Thus the line which has stayed the longest in the cache is replaced.
3. **Least frequently used :** In this case the line which is used for the least number of times is replaced first.
4. **Random :** In this case randomly any line is replaced.

Ex. 5.3.1 : Assume that memory consists of three frames and during execution of a program, following pages are referenced in the sequence : 2 3 2 1 5 2 4 5 3 2 5
Show that behaviour of the page replacement using FIFO strategy.

Soln. :

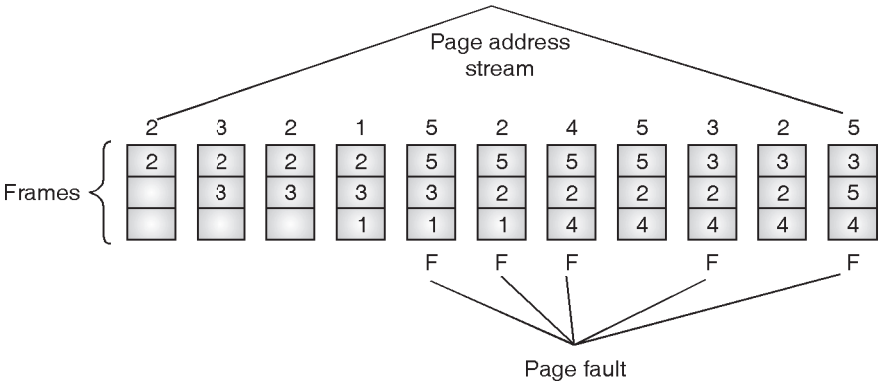


Fig. Ex. 5.3.1 : Behaviour of page replacement using FIFO

Ex. 5.3.2 : Find out page fault for following string using LRU and FIFO method. 6 0 12 0 30 4 2 30 32 1 20 15
(Consider page Frome size = 3)

Soln. :

Page address stream

1. FIFO

	6	0	12	0	30	4	2	30	32	1	20	15
FIFO	6	6	6	6	30	30	30	30	32	32	32	15
		0	0	0	0	4	4	4	4	1	1	1
			12	12	12	12	2	2	2	2	20	20
					F	F	F		F	F	F	F

(a)

2. LRU

LRU	6	6	6	6	30	30	30	30	30	30	20	20
		0	0	0	0	0	2	2	2	1	1	1
			12	12	12	4	4	4	32	32	32	15
					F	F	F		F	F	F	F

(b)

Page faults are indicated by 'F'.

Fig. Ex. 5.3.2

Ex. 5.3.3 : Consider a paging system in which M1 has a capacity of three fROMes. The page address stream formed by executing a program ROM is 2 3 2 1 5 2 4 5 3 2 5 2
Find the page hit using FIFO, LRU and OPT.

Soln. :

1. FIFO

Time	1	2	3	4	5	6	7	8	9	10	11	12
Address space	2	3	2	1	5	2	4	5	3	2	5	2
FIFO	2	2	2	2	5	5	5	5	3	3	3	3
		3	3	3	3	2	2	2	2	2	5	5
				1	1	1	4	4	4	4	4	2
			Hit					Hit		Hit		

2. LRU

LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
			Hit			Hit		Hit			Hit	Hit

3. OPT

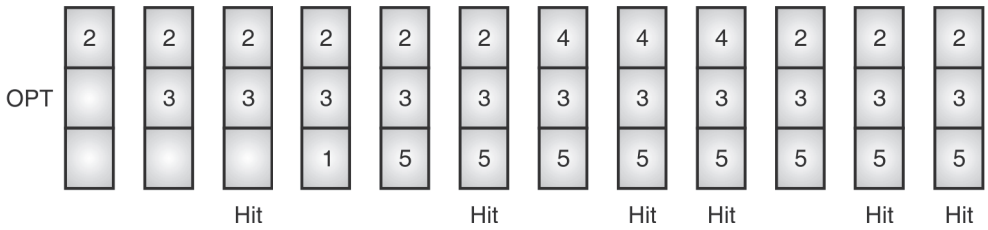


Fig. Ex. 5.3.3

Ex. 5.3.4 : Find out page fault for following string using LRU method. Consider page fROMe size = 3

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1.

Soln. :

Pages accessed	FROMes			
7	7	–	–	
0	7	0	–	
1	7	0	1	
2	2	0	1	F
0	2	0	1	
3	2	0	3	F
0	2	0	3	
4	4	0	3	F
2	4	0	2	F
3	4	3	2	F
0	0	3	2	F
3	0	3	2	
2	0	3	2	
1	1	3	2	F
2	1	3	2	
0	1	0	2	F
1	1	0	2	
7	1	0	7	F
0	1	0	7	
1	1	0	7	

Ex. 5.3.5 : Calculate the hit and miss using various page replacement policies LRU, FIFO,OPT for following sequence (page fROMe size 3) 4, 7, 3, 0, 1, 7, 3, 7, 5, 4, 5, 3, 4, 7. State which one is best for above example ?

Soln. :

1. LRU

4	7	3	0	1	7	3	8	5	4	5	3	4	7
4	4	4	0	0	0	3	3	3	4	4	4	4	4
	7	7	7	1	1	1	8	8	8	8	3	3	3
			3	3	3	7	7	7	5	5	5	5	7
M	M	M	M	M	M	M	M	M	M	H	M	H	H

Fig. Ex. 5.3.5(a)

$$\% \text{ Hit} = \frac{2}{14} \times 100 = 14.3\ \%$$

$$\% \text{ Miss} = \frac{12}{14} \times 100 = 75.7\ \%$$

2. FIFO

4	7	3	0	1	7	3	8	5	4	5	3	4	7
4	4	4	0	0	0	3	3	3	4	4	4	4	4
	7	7	7	1	1	1	8	8	8	8	3	3	3
		3	3	3	7	7	7	5	5	5	5	5	7
												H	H

Fig. Ex. 5.3.5(b)

$$\% \text{ Hit} = \frac{2}{14} \times 100 = 14.3\ \%$$

$$\% \text{ Miss} = \frac{12}{4} \times 100 = 75.7\ \%$$

3. OPT

4	7	3	0	1	7	3	8	5	4	5	3	4	7
4	4	4	0	1	1	1	1	5	5	5	5	5	5
	7	7	7	7	7	7	8	8	4	4	4	4	4
			3	3	3	3	3	3	3	3	3	3	7
					H	H					H	H	H

Fig. Ex. 5.3.5(c)

$$\% \text{ Hit} = \frac{5}{14} \times 100 = 35.7\ \%$$

$$\% \text{ Miss} = \frac{9}{14} \times 100 = 64.3\ \%$$

5.3.8 Multiple Levels of Cache: Cost and Performance Measurement

- Any two level memory has to be analysed with its performance characteristics as per the following set of characteristics.
- The different group of two level memories can be cache memory and main memory, main memory and virtual memory, internal and external cache memory etc.
- Let us see the various parameters to be considered during the performance analysis.

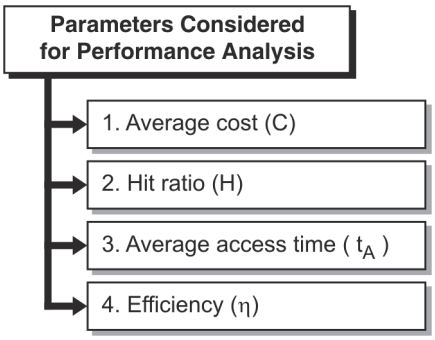


Fig. 5.3.8 : Parameters considered for performance analysis

1. **Average cost (C) =** $\dfrac{C1S1 + C2S2}{S1 + S2}$

   where, C1 and C2 are the costs per bit of memory 1 (faster memory) and memory 2 (slower memory) respectively. S1 and S2 are the sizes of memory 1 and memory 2 respectively.

2. **Hit Ratio (H) =** $\dfrac{N1}{N1 + N2}$

   where N1 is number of hits and N2 is number of misses.

3. **Average access time (t_A) = H t_{A1} + (1 – H) t_{A2}**

   where t_{A1} and t_{A2} are the time taken to access memory 1 (faster memory) and memory 2 (slower memory) respectively.

$$t_A = H\, t_{A1} + (1 - H)\, t_{A2}$$

$$= H\, t_{A1} + (1 - H)\, (t_{A1} + t_B)$$

where $t_{A2} = t_{A1} + t_B = t_{A1} + (1 - H)\, t_B$

4. **Efficiency (η) =** $\dfrac{t_{A1}}{t_A}$

$$= \frac{t_{A1}}{H\, t_{A1} + (1 - H)\, t_{A2}} = \frac{1}{H + (1 - H)\, r}$$

where $r = \dfrac{t_{A2}}{t_{A1}}$ = **Speed Ratio**

5.3.9 Cache Coherency with L1 and L2 Cache

The discussion of this cache coherency may require the concept of the state of a line in cache as given in the Table 5.3.1

Table 5.3.1 : State of a cache line and its implication

State	Description
'modified'	The line in cache has been updated (contains modified data) due to write hit in cache. This alerts the cache subsystem to snoop the system bus and write the modified line back to memory when a snoop hit to the line is detected i.e. only L1 cache has most updated data.
'exclusive'	Indicates that this cache knows that main memory in the system possesses a state copy of this line, therefore it is exclusive to this cache i.e. L1 and L2 cache have updated data.
'shared'	Indicates that the line may be present in several caches and if so, that an exact duplicate of the information (clean data) exists in each source (caches and memory) i.e. L1, L2 and main memory have updated data.
'invalid'	The initial state after reset, indicating that the line is not present in the cache.

1. When L2 cache is used in a processor, care must be taken to ensure cache consistency is maintained between the L1 and L2 caches and memory.
2. To help maintain cache consistency, the L1 cache implements a write-through policy to ensure external cache is notified of all write operations to the L1 cache.
3. The L2 cache can be designed to use either a write-through or write-back policy.
4. If another bus master writes something into the main memory whose copy is present in L1 as well as in L2 cache then,
   (a) When another bus master performs a write operation to memory, the L2 cache controller snoops the address. If a snoop hit is detected by L2 cache, then L2 cache directory entry is invalidated.
   (b) Since the data that is being updated was found in the L2 cache during the snoop, it is quite possible that the internal cache has a copy of the same line and hence in such case L1 cache will contain stale data. To avoid this inconsistency; in case of snoop hit, the address is sent to CPU also and if it detects a L1 cache hit, the directory entry is invalidated.
5. Non cacheable address (NCA) logic monitors each address that CPU outputs and decides whether the data should be cached or not. Data in locations corresponding to memory mapped I/O and dual ported memories are not kept in cache memory.

5.4 Cache Mapping Functions

- Mapping Function and replacement algorithm together decides where a line from the main memory can reside in the cache.
- The different mapping functions are Direct mapping, Fully Associative mapping and Set associative mapping.

5.4.1 Direct Mapping Technique

- In this case each block of main memory can map to only one cache line.
- A given block maps to any **line (i mod j),** where i is the line number of the main memory to be mapped and j is the total number of lines in the cache memory.
- The address is divided into three parts i.e. the word selector, line selector and the tag.

- Least Significant w bits identify unique word of a particular line
- Most Significant s bits specify one memory block to which the cache line corresponds.
- The MSBs are split into a cache line field r and a tag of s-r (most significant)
- **Example**: Let Cache be of 16kByte that is divided into blocks of 4 bytes hence cache is 16k (2^{14}) lines of 4 bytes. And let the main memory size be 16MBytes that requires 24 bit address lines (2^{24}=16M).

Tag (s - r) (8 bits)	Line (r) (14 bits)	Word (w) (2 bits)

- Hence the 24 bit address is divided as 2 bit word identifier (4 byte block), 22 bit block identifier i.e. 7 bit tag 14 bit slot or line(16K lines=2^{14})
- Fig. 5.4.1 shows the organization of Direct Mapping Cache.
- Fig. 5.4.1 shows the method the access the data from the cache implementing direct mapping technique

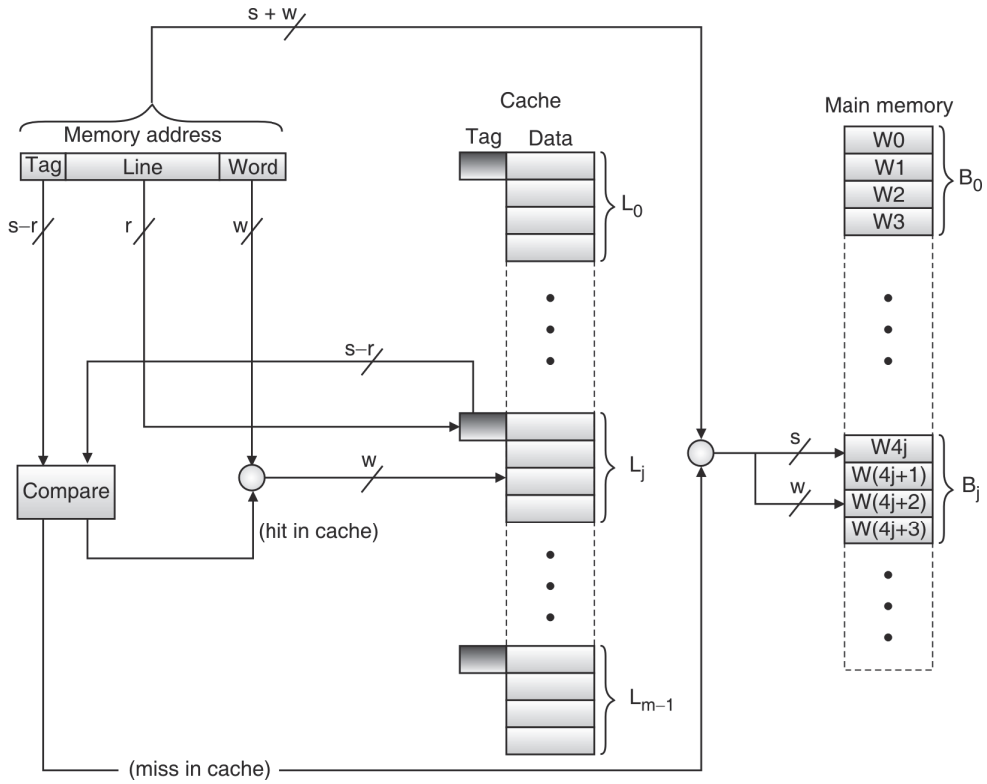


Fig. 5.4.1 : Direct Mapping Technique

- In this case to search a line from the cache memory, the line field selects the particular line, whose tag is to be compared with the tag of the address specified by the processor.
- The advantages of Direct Mapping are:
  1. Simple implementation
  2. Inexpensive
- The disadvantages of Direct mapping are :
  1. Fixed location for given block hence if a program accesses 2 blocks that map to the same line repeatedly, cache misses are very high.

5.4.2 Fully Associative Mapping

- In this case a main memory block can load into any line of cache.
- There are only two fields in the address as tag and word
- The tag uniquely identifies block of memory from where the line has been copied into the cache memory.
- To search a particular data the tag of every line is to be examined for a match. Thus cache searching gets expensive in terms of time required.
- **Example**: Let Cache be of 16k Byte that is divided into blocks of 4 bytes hence cache is 16k (2^{14}) lines of 4 bytes. And let the main memory size be 16M Bytes that requires 24 bit address lines (2^{24} = 16M).
- The associative Mapping Address Structure for this example considered: 22 bit tag stored with each 4-word block of data.
- Compare tag field with tag entry in cache to check for hit. Least significant 2 bits of address identify which word is required from 4-word data block

Tag (22 bits)	Word (2 bits)

- The organization of Fully Associative Cache mapping is shown in the Fig. 5.4.2.

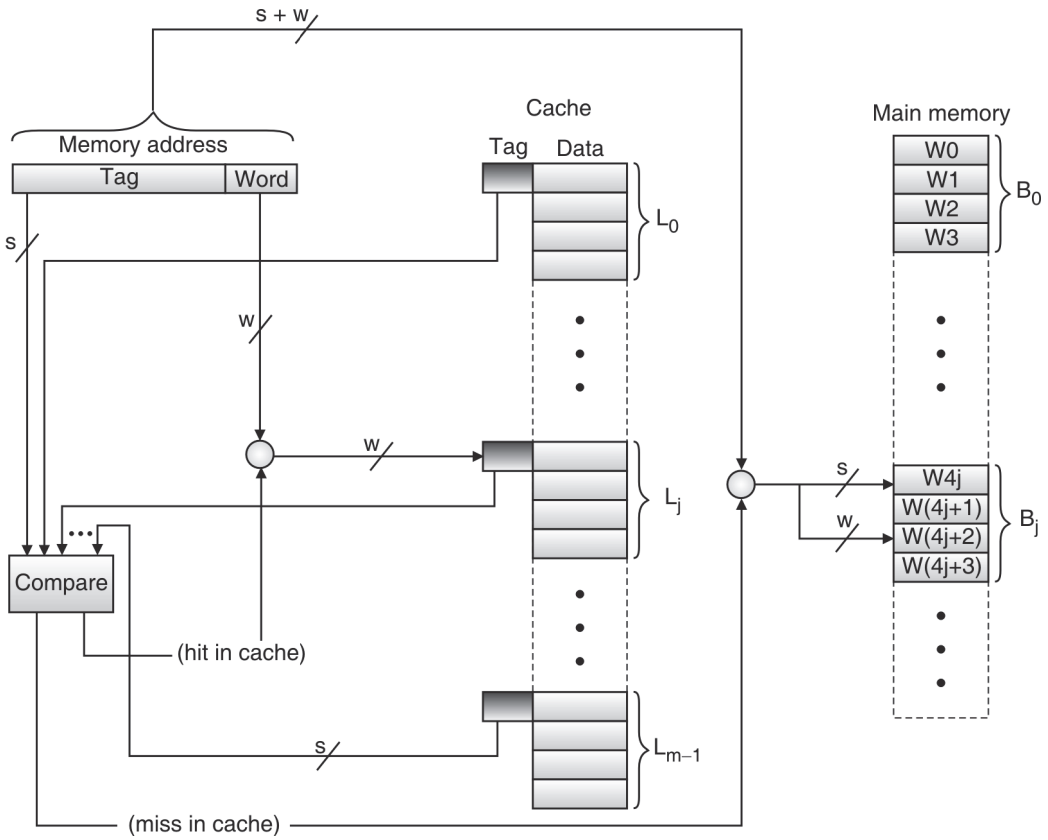


Fig. 5.4.2 : Fully Associative Mapping

Advantages of Associative Mapping are

1. If a program accesses 2 blocks (that would map to the same line in case of Direct mapping) repeatedly, cache misses will not occur

Disadvantages of Associative Mapping are

1. Complex design for many parallel comparisons of tag.
2. Expensive due to implementation of parallel comparator.

5.4.3 Set Associative Mapping

- In this case cache is divided into a number of sets. Each set contains a number of lines.
- A given block maps to any **line in a given set (i mod j),** where i is the line number of the main memory to be mapped and j is the total number of sets in the cache memory.
- For example, if there are 2 lines per set, it is called as 2 way associative mapping i.e. a given block can be in one of 2 lines in only one set.

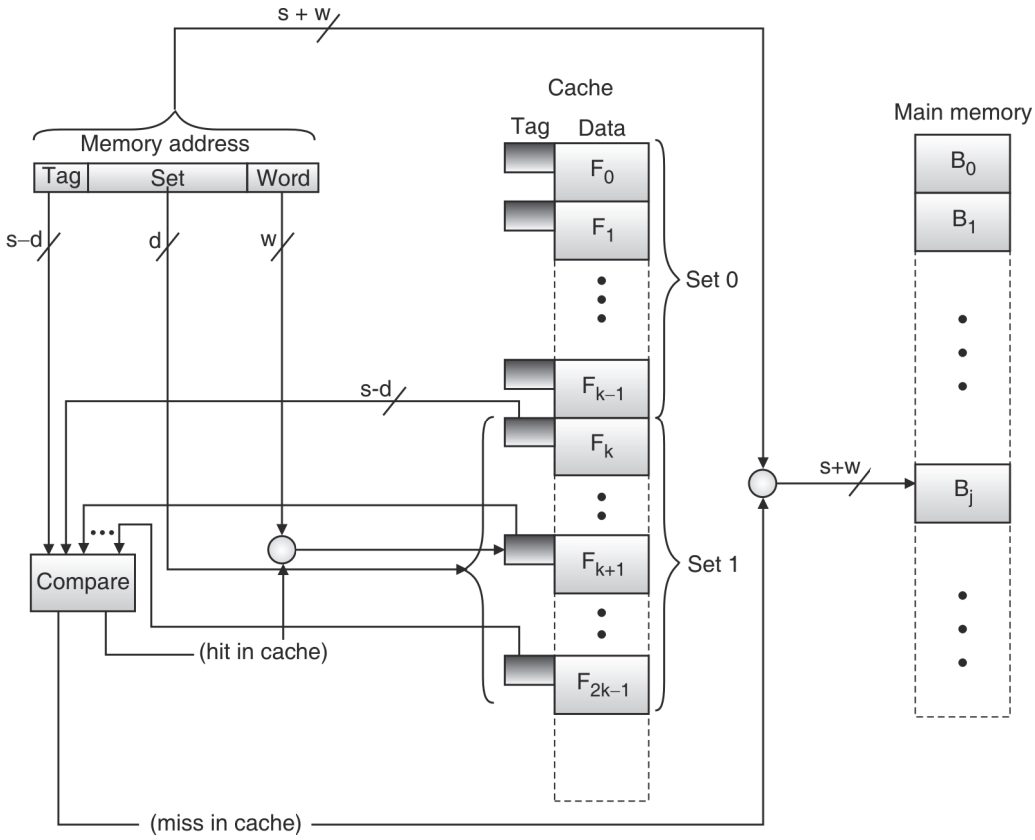


Fig. 5.4.3 : Set Associative Mapping

- **Example:** Let Cache be of 16kByte that is divided into blocks of 4 bytes hence cache is 16k (2^{14}) lines of 4 bytes. And let the main memory size be 16MBytes that requires 24 bit address lines (2^{24}=16M).
- For this example for set associative mapping address structure: 2 bits for one of the 4 words, 7K lines in each of the 2 sets hence 13 bits to select a set (213=8K) and remaining (24 – 13 – 2=) 9 bits for tag.

Tag (9 bits)	Set (13 bits)	Word (2 bits)

- In this case the set field is used to determine cache set to look in and Compare tag field to see if we have a hit.
- Fig. 5.4.3 shows an example of Two Way Set Associative Cache Organization

Advantages of Set Associative Mapping are :

1. If a program accesses 2 blocks that map to the same set repeatedly, cache misses will not occur because they would go into different lines of the set.
2. Not very complex because of just 2, 4 or 7 parallel comparisons.
3. Not much expensive again because of simple implementation.

Ex. 5.4.1 : A block set associative cache consists of 64 blocks divided in 4 block sets. The main memory contains 4096 blocks, each 128 words of 16 bit length (1) How many bits are there in main memory address ? (2) How many bits are there in cache memory address (tag, set and word fields) ?

Soln. :

(1) Main memory size = 4096 blocks x 128 word $=2^{12} \times 2^{7} = 2^{19}$

Thus main memory address lines required is equal to 19.

(2) Cache memory has 64 blocks divided in 4 block sets, thus each set has 16 blocks. Hence $16 = 2^{4}$; 4 address lines for set

Each block has 128 words; hence $128 = 2^{7}$, 7 address lines for word field

Remaining lines i.e. 19 – 4 – 7 = 7 address lines for tag

Tag (7 bits)	Set (4 bits)	Word (7 bits)

5.5 Unified and Split Cache

- When the data and code are stored in the same cache, i.e. they share the same cache memory, it is said to be unified cache.
- If data and code use separate cache, it is split cache
- The instruction fetch unit in a processor fetches code or instructions from the cache memory while the processing units or the ALU access the data from the cache.
- In case of unified cache, the instruction fetch mechanism has to wait if the processing unit is fetching a data from the cache memory. This results in a pipeline starve for one instruction as the entire pipeline has to starve for one clock cycle.
- In case of split cache, there is no pipeline starving for this issue, as there are separate cache memories to store the code and the data

5.6 ARM Memory Hierarchy

The memory hierarchy of ARM is as shown in the Fig. 5.6.1.

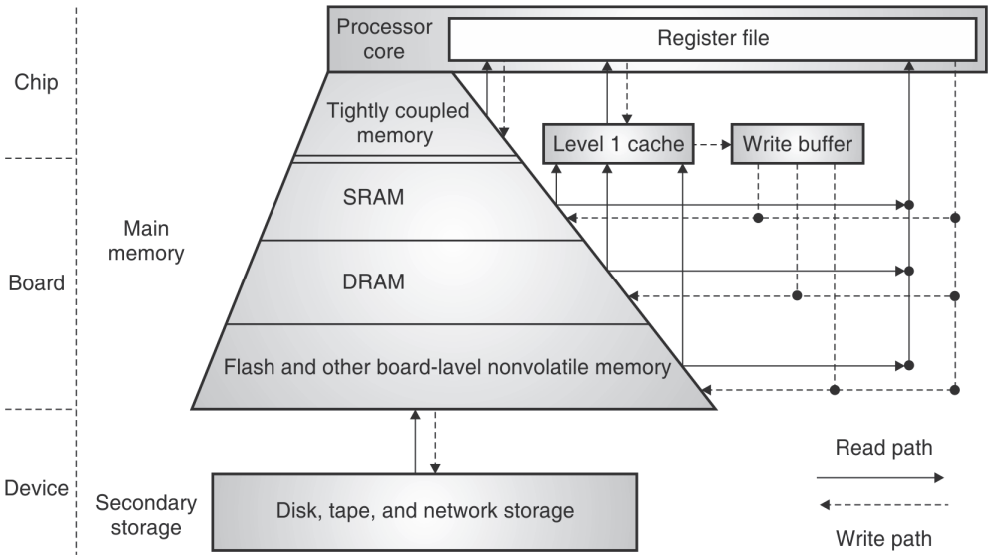


Fig. 5.6.1 : Memory Hierarchy of ARM

- The register file is in the processor itself. The tightly coupled memory has Level 1 cache and write buffer. This is followed by the SRM and DRAM. Then there may be some non-volatile on-board memory and finally the secondary storage.

5.7 ARM Cache Features

The features of ARM related to cache are as listed below:

1. ARM7 to ARM10 have logical caches while ARM11 uses physical cache. The cache located between processor and MMU is logical or virtual cache. The cache between MMU and main memory is called as physical cache.
2. ARM supports both unified as well as Split cache
3. The mapping technique used in set associative mapping technique in ARM processors.
4. Write back as well as Write through policies are supported in ARM processors

5.7.1 Coprocessor 15 for Cache System Control and Various Processes

- Several coprocessor 15 register file are used to configure and control the cache operations in ARM processor.
- The Table 5.7.1 lists the registers used for various operations in coprocessor 15.

Table 5.7.1 : Registers used for various operations in coprocessor 15

Sr. No.	Function/Processes	Primary Register	Secondary Register	Opcode
1	Cache Lockdown	C9	C0	0, 1
2	Drain Write buffer	C7	C10	4
3	Round Robin Replacement	C15	C0	0
4	Clean and flush cache	C7	C5, C6, C7, C10, C13, C14	0, 1, 2

- The lockdown feature permits the code and data to be loaded in cache but exempted from being removed.
- The drain write buffer function ensures all the data in write buffer to be updated in the corresponding memory locations.
- The round robin replacement enables the replacement policy of round robin in the sets of the associative cache memory.
- Flushing a cache means to clear all the contents of the cache memory, so that all the dirty lines in the cache are written back to the memory and then invalidated. Finally all the lines will be invalidated.
- Cleaning the cache means only writing the dirty lines to the main memory
- In both, flushing as well as cleaning the cache, the write back policy is used.

5.8 ARM Memory Map

ARM Memory map is already discussed in Chapter 2. Refer Section 2.5.

5.9 Protected Systems

- When having programs for multi-tasking, there are some separate regions in the memory i.e. regions that are allocated to each task. Such regions need to be protected.
- The regions allocated to each task must be protected from other tasks
- Also each task needs to be allocated its independent or local memory space to store code as well as data.
- The shared memory regions will be used to pass parameters while the local memory spaces will be used by each task to store its task related data and code

5.9.1 Memory Protection Unit (MPU) and ARM Processor with MPU

GTU - W-17

Q. Explain the significance of Memory Protection Unit. **(W-17, 3 Marks)**

- It is required that the working of one task doesn't disrupt another. The process of protecting tasks from one another is called as protection and is implemented by the unit called as protection unit.
- This demands the need of MPU to protect the code and data of a task from another task.
- The hardware MPU protects several software designated regions of the system.
- A region in memory requires two attributes for its definition viz. Starting address and the size (or length).
- The length can be anything between 4KB to 4GB (but obviously powers of 2) i.e. 2^{12} to 2^{32}, with index any integer between 12 to 32
- Hence, the system creates many such boundaries that are to be protected.
- Certain regions in memory are called as protected regions that are controlled by the MPU.
- Overlapping of memory regions allocated to the tasks is allowed in ARM processor.
- Priorities are also set for the regions as to who can access the given region

- To define the size of a region CP15:c6 register is used. The structure of the same is shown in the Fig. 5.9.1.

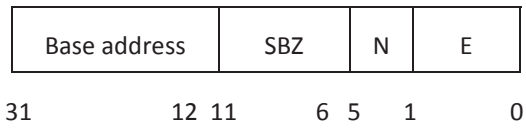


Fig. 5.9.1

- The meaning of these fields are shown in the Table 5.9.1.

Table 5.9.1

Field name	Bit fields	Comments
Base address	[31 : 12]	Address greater than4 KB must be a multiple of the size represented in [5 : 1] value "should be zero"
SBZ	[11 : 6]	Value "should be zero"
N	[5 : 1]	Size of region is 2^{N+1}, where $11 \leq N \leq 31$
E	[0]	Region enable, 1 = enable, 0 = disable

- The CP15 register 5 gives the access permissions for a region. The register structure is shown in the Fig. 5.9.2.

CP 15:c5:c0 stand and instruction region AP

CP 15:c5:c1 standard data region AP

Fig. 5.9.2

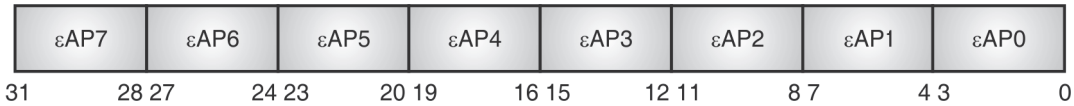


Fig. 5.9.3

- Based on the values of the various fields above, the priority is decided for the region.
- The Table 5.9.2 shows the access permission cases for both the supervisor as well as the user

Table 5.9.2 : Access permission cases for both the supervisor as well as the user

Supervisor	User	Standard AP value encoding	Extended AP value encoding
No access	No access	00	0000
Read/write	No acess	01	0001
Read/write	Read only	10	0010
Read/write	Read/write	11	0011
Unpredictable	Unpredictable	-	0100

Supervisor	User	Standard AP value encoding	Extended AP value encoding
Read only	No access	-	0101
Read only	Read only	-	0110
Unpredictable	Unpredictable	-	0111
Unpredictable	Unpredictable	-	1000 to 1111

5.10 Memory Management Unit (MMU) Advantage

GTU - W-18, W-19

Q. Explain memory management unit in ARM processor. **(W-18, 7 Marks)**

Q. What is Memory Management Unit (MMU)? In which circumstances the processor architecture include MMU in the design ? **(W-19, 4 Marks)**

MPU mode is mainly meant for multi tasking operations. Multiple tasks running simultaneously using separate code, data and stack segment. Besides it also takes care of proper authentication of a task to access a particular segment.

1. Most processors have only two protection levels (user and supervisor), but x86 architecture features four levels of protection, called **Privilege Levels (PL).**
2. They are designed to support the needs of multitasking OS to isolate and protect user programs from each other and the OS from unauthorized access
3. The privilege levels control the use of privileged instructions, I/O instructions, and access to segments and segment descriptors. The x86 architecture offers an additional type of protection on a page basis, when paging is enabled.

   Requested PL ⇒ RPL

   Descriptor PL ⇒ DPL

   Current PL ⇒ CPL

   Effective PL ⇒ EPL

   EPL = max (RPL, CPL)
4. The PLs are numbered 0, 1, 2, and 3. Level 0 is the most privileged level. Level 3 is the least privileged.

5.11 Virtual Memory

GTU - S-16, W-16, S-17, W-17, S-18, W-18, S-19, W-19

Q. Explain the concept of virtual memory. **(S-16, 7 Marks)**

Q. How Virtual Memory works? **(W-16, W-18, 7 Marks)**

Q. Describe the components of a virtual memory system with the help of diagram. **(S-17, 4 Marks)**

Q. What is Memory Management Unit (MMU) ? In which circumstances the processor architecture include MMU in the design? **(W-17, W-19, 4 Marks)**

Q. Describe Page and Page Table with reference to Virtual Memory. **(W-17, 4 Marks)**

Q.	Discuss memory organization in virtual memory system.	**(S-18, 7 Marks)**
Q.	Explain concept of virtual memory for advance microprocessor based system design.	**(S-19, 7 Marks)**
Q.	Write a note on virtual memory.	**(W-19, 4 Marks)**

- Virtual memory is a concept wherein the applications are made to feel that a huge main memory (fast semiconductor memory) is interfaced to the processor, whereas actually a small amount of main memory and a huge external memory (typically slow ROM like magnetic disk) is interfaced.
- The data required by the application is brought from the external slow memory to main memory in blocks (also called as pages) by the mechanism called as Paging.
- Hence now we can say that the entire memory system interfaced to the system looks something as shown in Fig. 5.11.1. The CPU or the processor is connected to the fast memory i.e. cache memory or SRAM which is then connected to the main memory or DRAM and then to the virtual memory or the external memory.
- The Memory Management Unit (MMU) connected to the processor converts the virtual address to the physical address and take care of bringing the pages (block of data) to the main memory from the external memory.

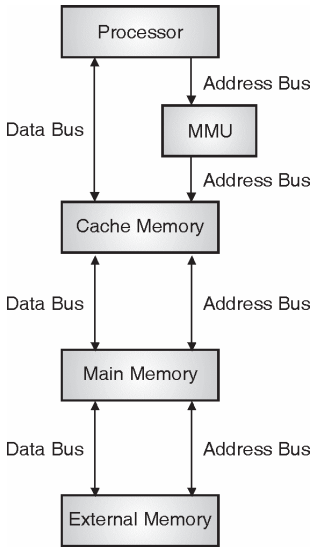


Fig. 5.11.1 : Connection of external or virtual memory to the processor

- The Memory Management Unit (MMU) lies between the processor and the main memory. The main memory is also called as physical memory.
- The external memory works as the virtual memory.
- The processor core generates the virtual address. The MMU translates the virtual address into physical address. This translation of virtual address is explained in the following section.

5.12 Virtual Memory Translation

GTU - W-16, S-17, W-17, W-18, S-19

Q.	Also explain the components of Virtual Memory System.	**(W-16, 7 Marks)**
Q.	How much areas of virtual memory represent by all pave tables except than master level page table ?	**(S-17, 1 Mark)**
Q.	How many levels of page table support by ARM MMU hardware ?	**(S-17, 1 Mark)**

Q.	Describe the requirements of Translation Lookaside Buffer in the implementation of Virtual Memory.	**(W-17, 7 Marks)**
Q.	Explain role of The Translation Look aside Buffer (TLB).	**(W-18, 4 Marks)**
Q.	Explain concept of translation look aside buffer	**(S-19, 4 Marks)**

- The memory management unit or the paging unit is responsible to convert the virtual or the linear address to physical address.
- Fig. 5.12.1 shows how the address translation takes place.
- The address given by the processor i.e. the linear or virtual address, is broken into the page number and the word number in that page.

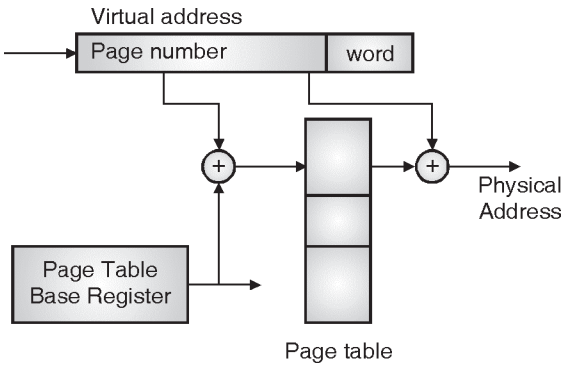


Fig. 5.12.1 : Paging mechanism

- The page number is checked for its presence in the main memory by comparing the entry for each page in the page table. If the page is present the word required is read from the page in the main memory.
- If the page required by the processor is not in the main memory, the page fault (similar to cache miss) occurs and the required page is loaded into the main memory by a special routine called as page fault routine. This technique is called as **Demand Paging** i.e. the page is brought from the external memory to the main memory only when required.
- A Translational Lookaside Buffer (TLB) is implemented in the memory management system, which reduces the memory access time, by translating the linear to physical address without undergoing the paging mechanism.
- The structure of memory management with TLB is as shown in Fig. 5.12.2.

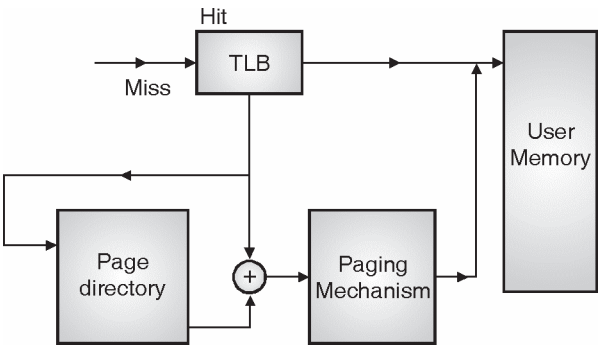


Fig. 5.12.2 : Translation Lookaside Buffer

- As shown in the Fig. 5.12.2, TLB is placed parallely with the paging mechanism and hence if the TLB gives a hit, the paging mechanism doesn't perform the address translation, else the paging mechanism performs the address translation.

5.12.1 Paging Mechanism or Memory Management Unit

- The memory management unit or the paging unit is responsible to convert the virtual or the linear address to physical address.
- Fig. 5.12.3 shows how the address translation takes place.
- The address given by the processor i.e. the linear or virtual address, is broken into the page number and the word number in that page.

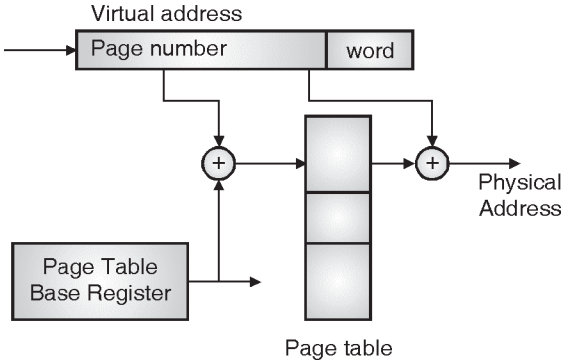


Fig. 5.12.3: Paging mechanism

- The page number is checked for its presence in the main memory by comparing the entry for each page in the page table. If the page is present the word required is read from the page in the main memory.
- If the page required by the processor is not in the main memory, the page fault (similar to cache miss) occurs and the required page is loaded into the main memory by a special routine called as page fault routine. This technique is called as **Demand Paging** i.e. the page is brought from the external memory to the main memory only when required.
- A Translational Look aside Buffer (TLB) is implemented in the memory management system, which reduces the memory access time, by translating the linear to physical address without undergoing the paging mechanism.
- The structure of memory management with TLB is as shown in Fig. 5.12.4.

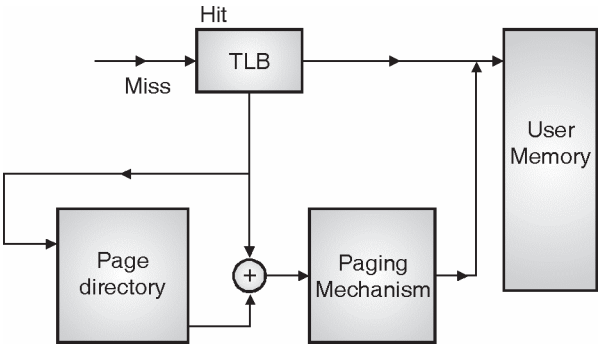


Fig. 5.12.4 : Translation Look aside Buffer

- As shown in the Fig. 5.12.4, TLB is placed parallel with the paging mechanism and hence if the TLB gives a hit, the paging mechanism doesn't perform the address translation, else the paging mechanism performs the address translation.

5.12.2 Segmentation

Segmentation refers to logical division of the main memory so as to give modular storage mechanism and multitasking.

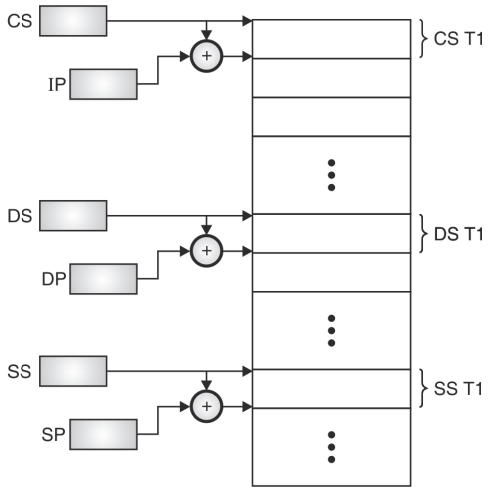


Fig. 5.12.5

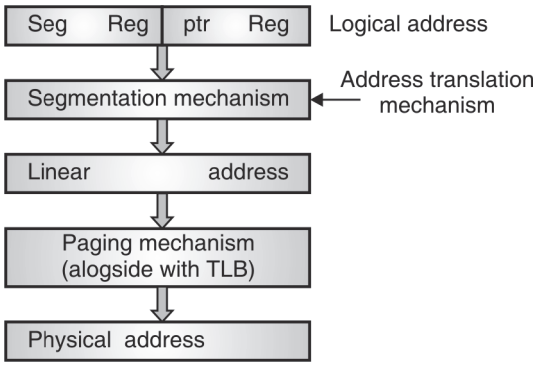


Fig. 5.12.5

5.12.3 Physical Memory vs. Virtual Memory

Table 5.12.1 : Physical Memory vs. Virtual Memory

Sr. No.	Physical Memory	Virtual Memory
1.	Physical memory is on the motherboard and is normally in form of chips (SRAM)	Virtual Memory is inform of disks like Hard disk
2.	When computer is switched ON, physical memory is empty and data is loaded from virtual memory as demanded by the processor i.e. the physical memory is volatile	The data is permanent on virtual memory
3.	Physical memory is costlier compared to virtual memory	Virtual memory is comparatively cheaper
4.	Physical memory is fast i.e. has less access time	Virtual memory is slower i.e. has large access time
5.	Data from physical memory is loaded into cache memory	Data from virtual memory is loaded into physical memory

5.13 Multitasking with MMU

GTU - S-17, W-17, W-18, W-19

Q. Describe how virtual memory works with the help of a diagram showing mapping a task in virtual memory to physical memory using a relocation register. **(S-17, 7 Marks)**

Q. Describe the concept of Multitasking. **(W-17, W-18, W-19, 3 Marks)**

- The concept of Memory protection unit and virtual memory together can ensure multitasking. Each task can be given a different set of pages for storing code, data, stack etc.
- Whenever task switching is to be done the pages being accessed can be changed and accordingly task switching can take place. The usage of the different pages for different tasks is as shown in the Fig. 5.13.1.

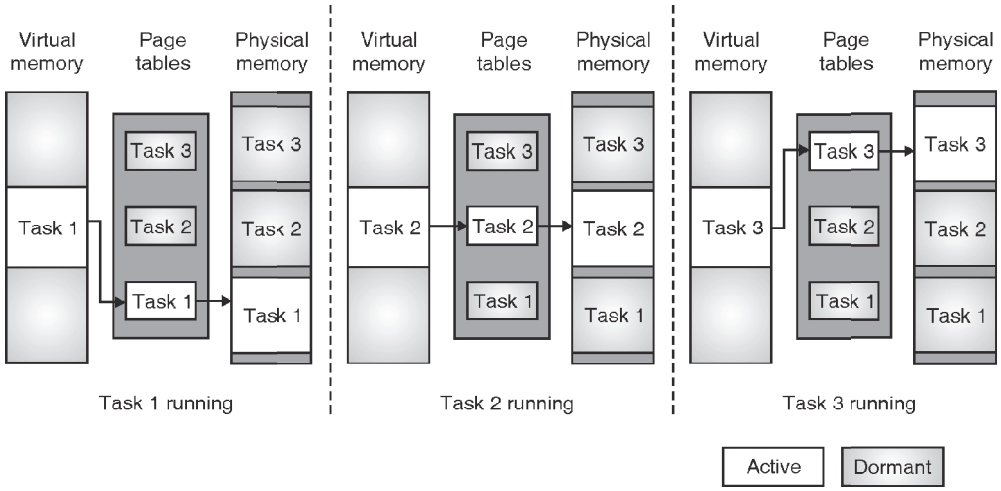


Fig. 5.13.1 : Multitasking using Virtual memory

5.14 MMU Organization in Virtual Memory System

GTU - W-18

Q. What is the use of paging mechanism? Show how paging is done in any of ARM families of processors. **(W-18, 7 Marks)**

- ARM families of processors.
- 80386 / Pentium processor use the virtual memory system with the help of on-chip Memory Management Unit (MMU) i.e. paging mechanism.
- The MMU considers the memory divided into pages of 4KB each. Hence the 4GB memory is divided into 1 M pages of 4 KB each. To select one of these 1 M pages, 20 bits are required.
- CR3 consists of 20-bit page directory base address that selects a page, called as page directory. This page is also of 4 KB that has 32-bit (i.e. 4 Byte) entry, and hence has 1 K entries.
- To select one of these 1 K entries, 10 bits are required. Hence the first 10-bits of the linear address i.e. bit 31 to bit 22, select one of the 1 K entries. This entry in page directory (32-bit entry), has 20-bit that selects another page, called as page table.

- Since there are 1 K entries in the page directory each of them selecting a page table, there can be 1 K page tables.
- The page table size is also 4 KB that has 32-bit (i.e. 4 Byte) entry, and hence has 1 K entries. To select one of these 1 K entries, 10 bits are required. Hence the next 10-bits of the linear address i.e. bit 21 to bit 12, select one of the 1 K entries.
- This entry in a page table (32-bit entry), has 20-bit that selects the user required page. This page is also of 4 KB. The required memory operand from this page is selected by the last 12-bits of the linear address i.e. bit 11 to bit 0. This paging mechanism is shown in the Fig. 5.14.1.

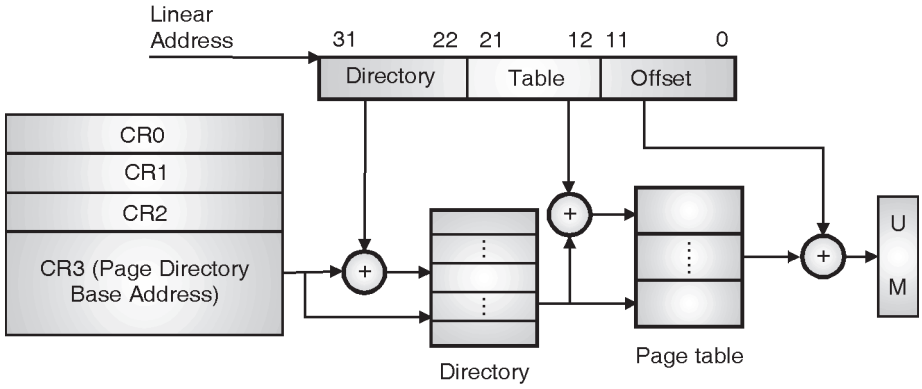


Fig. 5.14.1 : Paging mechanism

5.14.1 Page Tables

The page table or page directory entry is shown in Fig. 5.14.2 The different bits in these entries are discussed below

- Bit 31-12, Page frame address for page table entry, while Page table address for page directory entry.
- Bit 11-9, are reserved for Operating System (OS). The OS can use it for its purpose, storing some information related to this page.
- Bit 8 and 7, are zeroes, are unused.
- Bit 6, D (Dirty); It is set before anything is written page directory. It indicates that this page has some updated data in the physical memory, while the virtual memory has this page not updated or stale data.
- Bit 5, A (Accessed); It indicates that this page is accessed previously or not
- Bit 4, PCD (Page Cache Disable); As in CR0
- Bit 3, PWT (Page Write Through); As in CR0
- Bit 2, U/S (User/Supervisor); This bit differentiates between user and supervisor privileges.
- Bit 1, R/W (Read/Write); It indicates read or write protection for the page.
- Bit 0, P (Present); It indicates whether entry in the PDE and PTE (page Directory Entry and Page Table Entry) can be used. It indicates if the entry is initialized or not.

31 12	11 10	9	8	7	6	5	4	3	2	1	0
Page Frame/Table Add.	OS Reserved		0	0	D	A	PCD	PWT	U/S	R/W	P

Fig. 5.14.2 : Page table/ directory entry

5.14.2 Translation Look Aside Buffer

GTU - W-16

Q.	Define the following : TLB.	**(W-16, 1 Mark)**

Translational Lookaside Buffer (TLB) is implemented in the memory management system, which reduces the memory access time, by giving the physical address without undergoing the paging mechanism. TLB is implemented in the 80386 processor whose structure is as shown in Fig. 5.14.3.

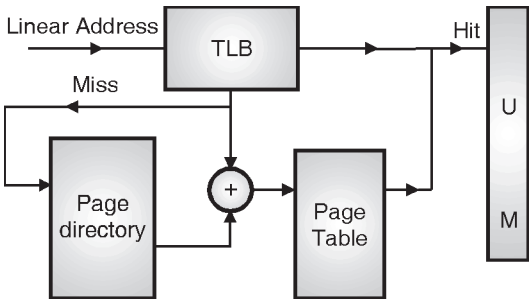


Fig. 5.14.3 : Translational lookaside buffer implementation

5.14.3 ARM MMU

- The ARM MMU besides translating the virtual to physical address, also controls the permission to memory access i.e. the job of protection unit.
- It also determines the behaviour of cache and write buffers
- It consists of page tables, translational lookaside buffer, cache, write buffers, permission details, control registers and fast context switching extension.
- The page tables, TLB, cache, write buffers are similar as explained earlier in this chapter.
- In the next section, we will see in detail about the fast context switching extension.

5.14.4 Fast Context Switch Extension

GTU - W-16, S-17

Q.	Write a note on Fast Context Switch Extension of MMU.	**(W-16, 7 Marks)**
Q.	List out the steps requires to switch between tasks.	**(S-17, 3 Marks)**

1. A **task** is an instance of the execution of a program.
2. A very important feature of any multitasking system is the ability to switch rapidly between tasks.
3. The task switch operation saves the entire state of the machine i.e. all registers, the memory variables and a link to the previous task, loads a new execution state, performs protection checks, and begins the execution of new task.
4. This context is stored in or retrieved from the associated Task State Segment (TSS).
5. The current TSS is identified by a special CPU register called Task Register (TR).

- ARM supports fast context switching. The ARM based processors have a special unit called as Fast Context Switch Extension (FCSE).
- This is a special hardware that resides in the memory management unit and is mainly used to improve the performance of system.

- The FCSE as discussed earlier in this chapter, manages the pages for multiple tasks. Besides the FCSE there is a address translation mechanism that translates the Virtual Address (VA) into the Modified Virtual Address (MVA) that is used to calculate the position in the virtual memory for the context of the task.
- Thus switching between the tasks doesn't requires pushing of all the register contents on the stack or so; instead just changing of the process ID in the FCSE switches the task under execution.
- Some addresses are reserved for each task while some addresses are common for all the tasks, although there is a protection of memory locations shared from other tasks. The process id identifies the task to be accessed.

5.15 Tightly Coupled Memory (TCM)

- ARM9 supports tightly coupled memory interface. In the tightly coupled system, the devices are connected to a common bus i.e. there is no bus hierarchy. The benefit of this system is that there is no latency as all the devices are directly connected to the processor.
- TCM is generally used for critical code or data that requires fast access with zero latency like code or data that have high degree of spatial or temporal locality. Having high degree of temporal locality means has usage for multiple times while spatial locality refers to the nearby sequential code or data.
- ARM supports two separate TCM regions :
  1. Instruction TCM (ITCM)
  2. Data TCM (DTCM)
- The TCM region is physically addressed and is controlled by TCM region register i.e. Control Register “c1”. The size of the TCMs are decided by the IRSIZE register for Instruction TCM and DRSIZE for Data TCM.
- Thus TCM supports memory accesses for ARM processor with zero or more wait states. To insert wait states the IRWAIT and DRWAIT signals are to be used for instruction and data respectively

❑❑❑

CHAPTER 6

UNIT VI

ARM Peripherals and Versions

Syllabus

AMBA Overview, Typical AMBA Based Microcontroller, AHB bus features, AHB Bus transfers, APB bus transfers, APB bridge. Overview of ARM Versions: ARM v5, ARM v6, ARM v7 (Cortex family R: Real time A: Application and M: Microcontroller), ARM big.LITTLE Technology, ARM v8.

6.1 Advanced Microprocessor Bus Architecture (AMBA) Bus System Overview

GTU - S-16, W-16, S-17, W-17, S-18, S-19

Q.	Write a short note on Advanced Microprocessor Bus Architecture (AMBA).	**(S-16, 7 Marks)**
Q.	Define the following : AMBA-APB.	**(W-16, 1 Mark)**
Q.	Describe AMBA arbitration with the help of diagram for a typical AMBA-based system.	**(S-17, 7 Marks)**
Q.	Describe the Bus level support provided by ARM processor for the peripherals.	**(W-17, 7 Marks)**
Q.	Explain advance microprocessor bus architecture (AMBA) and discuss its special features.	**(S-18, 7 Marks)**
Q.	What is AMBA? Explain the AMBA bus architecture.	**(W-16, 4 Mark)**
Q.	Write features of advanced microprocessor bus architecture. (AMBA)	**(S-19, 3 Marks)**

- The busses used for computers and that used for embedded systems are different. In computers based on x86 processors we use busses like PCI, ISA, USB etc.
- The AMBA bus was introduced in 1996 especially for the communication of the peripherals and other units inside the ARM processor. During that time the busses were called as ARM Peripheral Bus (APB) and the ARM System Bus (ASB). Thereafter ARM came up with another bus having higher performance and named it as ARM High Performance Bus (AHB). Using this AMBA, the system designers can make embedded systems with many peripherals on the system. To implement this bus various large data bus widths are available i.e. 64 as well as 128 bits.

The special features of AMBA bus are :

1. There are separate controls for address and data phases
2. Unaligned data accesses are handled using byte strobes
3. It also supports burst transfers giving only the start address
4. It can also handle out of order accesses.
5. Flexibility
6. Wide Adoption
7. Compatibility

- There are various products available in the market that are AMBA based. These products are used in embedded systems. These are various devices available of this type like network device, SPI, UART etc.
- The ARM AMBA is an open bus standard protocol for communication amongst the different blocks of a System on Chip (SoC). The bus standard AMBA 4 ACE also has system-level coherency.
- It also supports cache coherent transfers and distributed virtual memory (DVM) thereby enabling TLB for the virtual memory.
- ACE can cater to a large range of coherent bus masters providing differing capabilities. It also supports bus masters that differ in cache line sizes, cache state models, and masters with write-back or write-through caches.
- ACE also supports coherency between different types of processors like Cortex-A15 and Cortex-A7 processors.
- It also supports I/O coherency.

6.1.1 Advanced High-Performance Bus (AHB)

- The AMBA AHB has the special features like:
  1. High-performance,
  2. High clock frequency system.
  3. Ease of use
  4. Efficient design
- It behaves as the high-performance system backbone bus.
- It not only supports the efficient connection of processors but also supports the on-chip memories as well as the off-chip external memory interfaces that too with low-power peripheral macrocell functions.
- It is also ensures ease of use in an efficient design flow using synthesis and automated test techniques.

6.1.2 Advanced System Bus (ASB)

- The AMBA ASB is used specially for high-performance system modules, although the high performance features are lower compared to the AHB
- It is an alternative system bus suitable in situations where the high-performance feature of AHB is not required.
- Like AHB, ASB also supports not only efficient connection of processors but also on-chip memories as well as off-chip external memory interfaces that too with low-power peripheral macrocell functions.

6.1.3 Comparison of AHB and APB Buses of AMBA Architecture

GTU - S-16, W-17, W-19

Q.	Compare AHB, ASB and APB buses of AMBA bus system of ARM Architecture.	**(S-16, W-19, 7 Marks)**
Q.	List out the differences between ARM System Bus (ASB) and ARM Peripheral Bus (APB).	**(W-17, 3 Marks)**

Sr. No.	AHB or ASB	APB
1.	AHB stands for Advanced High-performance bus and ASB for Advanced system bus are same	APB sands for Advanced Peripheral bus.
2.	AHB uses a full duplex parallel communication.	APB uses massive memory-I/O accesses.
3.	It mainly connects to memories.	It mainly connects to I/O devices.
4.	the APB is simpler than the AHB.	the APB is simpler than the AHB.
5.	AHB has pipelining.	APB has no pipelining.
6.	It is used for high bandwidth.	It is used for low bandwidth.

6.1.4 Advanced Peripheral Bus (APB)

- The AMBA APB is mainly for the applications like low-power peripherals.
- It is optimized such that the power consumption is minimal.
- This reduced interface complexity supports peripheral functions.
- It can be used along with ASB or AHB of the system bus.

6.1.5 A Typical AMBA-based Microcontroller

- An AMBA-based microcontroller typically consists of a high-performance system backbone bus (AMBA AHB or AMBA ASB), able to sustain the external memory bandwidth, on which the CPU, on-chip memory and other Direct Memory Access (DMA) devices reside.
- This bus provides a high-bandwidth interface between the elements that are involved in the majority of transfers. Also located on the high-performance bus is a bridge to the lower bandwidth APB, where most of the peripheral devices in the system are located as shown in the Fig. 6.1.1

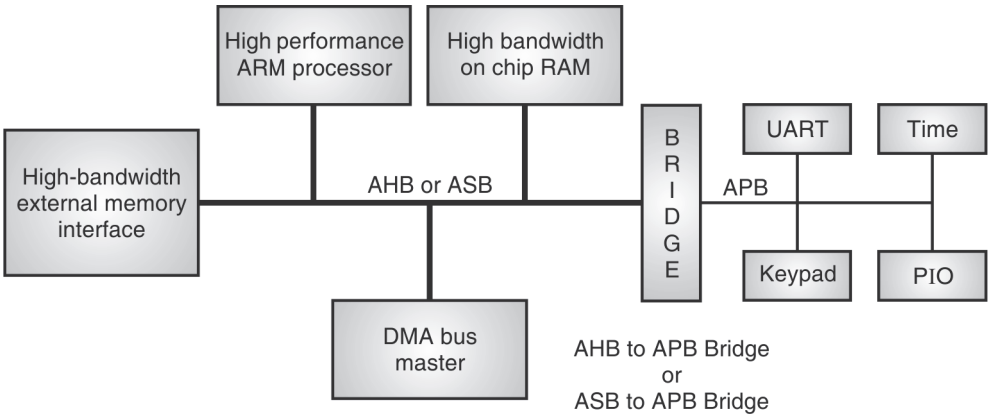


Fig. 6.1.1: Typical AMBA based microcontroller

6.1.6 AHB Bus Transfer and APB Bridge

- AHB is a new generation of AMBA bus which is intended to address the requirements of high performance synthesizable designs.
- It is a high-performance system bus that supports multiple bus masters and provides high-bandwidth operation.
- AMBA AHB implements the features required for high-performance, high clock frequency systems including:
  1. Burst transfers
  2. Split transactions
  3. Single-cycle bus master handover
  4. Single-clock edge operation
  5. Non-tristate implementation
  6. Wider data bus configurations (64/128 bits).
- Bridging between this higher level of bus and the current ASB/APB can be done efficiently to ensure that any existing designs can be easily integrated.
- An AMBA AHB design may contain one or more bus masters, typically a system would contain at least the processor and test interface.
- However, it would also be common for a Direct Memory Access (DMA) or Digital Signal Processor (DSP) to be included as bus masters.
- The external memory interface, APB bridge and any internal memory are the most common AHB slaves. Any other peripheral in the system could also be included as an AHB slave. However, low-bandwidth peripherals typically reside on the APB.

6.1.7 APB Bus Transfer

- APB supports both types of transfers as listed below:
  1. Write Transfer (with and without wait states)
  2. Read transfer (with and without wait states)
- In case of write as well as read transfer it supports both types i.e. with and without wait state. The transfers with wait state are useful for slower devices
- In case of a write transfer without wait states, the address and data are placed on the respective busses in the first clock pulse of the machine cycle. In the second clock pulse of the machine cycle, the enable signal is given so that the data can be stored or given to the respective device
- In case of the write transfer with wait states, the PREADY signal of the APB bus protocol, is disabled (made low) until the device is not ready. Once the device is ready, the PREADY signal is enabled. Hence, the processor inserts wait states until the device is not ready.
- It also supports write strobe mechanism, wherein the PSTRB is asserted high when a byte on a particular data bus is ready. Hence the transfer is done whenever the strobe signal is enabled.

- Similarly, the read transfers are also supported with as well as without wait states. For wait states insertion, the signal PREADY is used.
- APB bus also supports indication of an error in case of faulty transfer in read or write operation. The signal, PSLVERR is used to indicate the error.
- To support the access of protected data or privileged data, a set of three signals (PPROT2, PPROT1 and PPROT0) are used.
- PPROT0 indicates a normal or privileged access. If the bit is '1', it indicates privileged access else normal access.
- PPROT1 indicates whether the access is secure or not. PPROT1 when '1', indicates non-secure access else it is a secure access.
- PPROT2 is used to indicate the access is for data or instruction. If the bit is '1', it indicates instruction access, else data access.

6.2 Overview of ARM Versions: ARM v5, ARM v6, ARM v7, ARM v8

- The first ARM processor was developed at Acorn Computers Limited, England, between 1983 and 1985.
- At that time ARM stood for Acorn RISC Machine, until the formation of Advanced RISC Machines Limited when it was renamed Advanced RISC Machine in 1990.
- The most 32-bit microcontrollers in the embedded industry are ARM microcontrollers, and it has a stake of 75% in this industry.
- With a common architecture, ARM has a very wide range of processor cores and delivers high performance together with low power consumption as well as system cost.
- The ARM processor are of a wide range with solutions for :
  (1) Open platforms for wireless, consumer and imaging applications, that use complicate operating system.
  (2) Mass storage, automotive, industrial and networking with real time embedded applications.
  (3) Smart cards and SIMs requiring high security.
- In the 1980s Apple Computer started working with a company called as Acorn. This work was such crucial that Acorn spun off the design team in 1990 into a new company called Advanced RISC Machines (ARM).
- Later, it became ARM Limited and it was floated on the London Stock Exchange and NASDAQ in 1998.
- The ARM was developed for the first time with its samples in 1985 and was called as **ARM1**, and the first "real" production systems was called as **ARM2** developed in 1986.
- This ARM2 had a 32-bit data bus and 26 bit address bus with 32 bit registers.
- There were 16 such 32-bit registers with one of them serving as a program counter. Some of the bits of program counter were used as control and status flags.
- This ARM2, that used 32000 transistors, was almost the most simple and very useful 32-bit microprocessor in the world.
- The ARM2 was succeeded by **ARM3**, that had a cache memory of 4KB and hence improving the performance.

- Later ARM6 was developed and then the most successful implementation of ARM i.e. the ARM7TDMI which is present in hundreds of millions of cellular phones. Freescale (spun off of Motorola in 2004), IBM, Texas Instruments, Nintendo, Philips, VLSI, Atmel, Sharp, Samsung and STMicroelectronics have also licensed the basic ARM design for various uses.
- The ARM chip has become one of the most used CPU designs in the world, found in everything from hard drives, to mobile phones, to routers, to calculators, to children's toys. Today it accounts for over 75% of all 32-bit embedded CPU's.
- The ARM implementation has five versions of the instructions set that are defined till date. The versions are represented by version numbers 1 to 5.
- In order to specify collections of additional instructions included the instruction set, some of the versions are qualified with variant letters.
- M variant denotes addition of four instructions and T variant denotes additions of entire thumb instruction set.

The five versions are as follows

1. ARM v1

- This version was implemented on ARM 1 processor. It has 26 bit address space.
- It supports branch instructions including a branch and link instruction designed for subroutine calls.
- It supports a software interrupts instruction, for use in making operating system calls.
- It supports byte, word and multi-word load-store instructions.
- It supports the data processing instructions also.
- Version 1 now a days has become obsolete.

2. ARM v2

- This version is a updated version of version 1. It also has 26 bit address space.
- It supports coprocessor supported instructions.
- It supports multiply and multiply-accumulate instructions.
- It includes load and store instructions called SWP and SWPB in a variant called version 2.
- It supports two more banked registers in fast interrupt mode.
- Version 2 is also obsolete.

3. ARM v3

- This version has 32 bit address space.
- In this version with a view to retain the contents of CPSR register in the case of an exception two new registers were added. The program status information which was stored in register R15 is now stored in the CPSR (Current Program Status Register) and the SPSR (Saved Program Status Register).
- In version 3 two new instructions (MRS and MSR) were added to access the CPSR and SPSR registers.
- It supports two new processor modes. The new processor modes help to use Data Abort, undefined instruction exceptions in the code.
- Example implementations are ARM6 processor core and ARM7M

4. ARM v4

- The version 4 is an extension of version 3.
- It includes additional features like :
  1. load / store halfword instructions.
  2. instruction transfer to T state in T variants.
  3. new privileged processor mode using the user mode register.
- Version 4 made a clear distinction of the instructions that should cause the exception to be taken.
- Backwards-compatibility support for 26 bit architecture ceased to be obligatory in version 4.
- Example implementations are ARM7.
- The special features of ARM 7 are as listed below :
  1. It is a 32-bit processor
  2. It has AHB bus interface
  3. It supports Thumb instruction set
  4. It is a RISC processor
  5. Later versions of ARM 7 have DSP instructions
  6. Later versions of ARM7 also have Jazelle instruction set support
  7. Its clock frequency is from 100 to 133 MHz

5. ARM v5

- The version 5 is an upward extension of version 4.
- It includes additional instructions that improve the efficiency of ARM interworking in T variants.
- Version 5 allows same code generation techniques to be used for T and non T variants.
- It supports an additional software breakpoint instruction.
- Version 5 also supports a count leading zeros (CLZ) instruction that allows efficient integer divide and interrupt prioritization.
- Version 5 tightens the definition of how flags are set by multiply instructions.
- It supports co-processor instructions.
- Example implementations are ARM9 and ARM10
- The features of ARM 9 are as listed below :
  1. It is a 32-bit processor
  2. It has AHB bus interface
  3. It supports Thumb instruction set
  4. It is a RISC processor
  5. Later versions of ARM 7 have DSP instructions

6. Later versions of ARM7 also have Jazelle instruction set support
7. Its clock frequency is from 180 to 210 MHz
8. It has 4KB code cache memory
9. It has 4KB data cache memory
10. It has on-chip memory management unit

The variants are as follows

1. The Thumb Instruction Set (T variants)

- It is a subset of the ARM instruction set. The thumb instruction set has 16 bit instructions. i.e. the Thumb instructions are half the size of ARM instructions.
- The Thumb instruction set has greater code density than the ARM instruction set.
- The drawbacks of the Thumb instruction set are that it requires more instructions to complete the same task. Hence the ARM instruction set is best for maximizing the performance.
- The Thumb instruction set does not include instructions required for exception handling.
- Due to these drawbacks the Thumb instruction set is used in conjunction with a suitable version of the ARM instruction set.
- The Thumb instruction set is not valid prior to version 4 of the ARM architecture. It is signified by letter **T**.

2. Thumb instruction set versions

- There are two versions of the Thumb instruction set i.e. version 1 and version 2.
- The version 1 of the Thumb instruction set is used in T variants of the ARM architecture version 4 and version 2 is used in T variants of the ARM architecture version 5.
- The Thumb instruction set version 2 modifies the instructions to improve the efficiency of ARM interworking.
- The Thumb instruction set tightens the definition of how flags are set by multiply instructions.
- The Thumb instruction set includes the breakpoint instruction.
- These modifications are related to the changes between ARM architecture versions 4 and 5.

3. M variants (Long Multiply Instructions)

- The ARM instruction set includes four additional multiply instructions for producing 64 bit multiply and accumulate results.
- These instructions imply the existence of a multiplier that is larger than the minimum. Its presence is denoted by variant letter M.
- The long multiply instructions were first included in the variant of version 3 and exists in all after version 3.

4. E variants (Enhanced DSP instructions)

- The E variants of the ARM instructions set consist of a number of ARM instructions that enhance the performance of an ARM processor for digital signal processing applications.
- The **E** variants support load, store, co-processor register transfer instructions that act on double word data.

- They include a cache preload instruction (PLD).
- They include several new multiply and multiply-accumulate instructions acting on 16 bit data.
- They were first included in the ARM architecture version 5T. They are not valid in non-T or non-M variants of the ARM architecture.

5. ARM v6

- The version 6 is an upward extension of version 5.
- It includes additional instructions for multiprocessor support.
- It also supports unaligned data and mixed endian data.
- It supports multimedia instructions.
- ARM11 is an example implementation of ARMv6.
- The features of ARM 11 are as listed below :
  1. It is a 32-bit processor
  2. It has AHB bus interface
  3. It supports Thumb instruction set
  4. It is a RISC processor
  5. ARM 11 have DSP instructions
  6. ARM 11 also have Jazelle instruction set support
  7. Its clock frequency is from 800 MHz to 1 GHz
  8. It has 4 to 64KB code cache memory
  9. It has 4 to 64 KB data cache memory
  10. It has on-chip memory management unit
  11. It has one to four cores in SMP cluster
  12. It also supports AMBA bus
  13. It has 8 to 9 stages of instruction pipelining
  14. It has separate load-store and arithmetic pipeline
  15. It has branch predication and return stack logic
  16. It has optional memories coupled with DMA for multi-media applications

6. ARM v7

- ARM cortex processors are in ARM v7
- It implements optional floating point (VFP) extension.
- In the subsequent sections we will see in detail about ARM cortex series and their example processor

7. ARM big.little Technology

- It pairs core with high performance like Cortex-A15 processor with a power optimised core like Cortex-A7 processor.

- Since there is a high performance as well as low performance technology mixed, it is called as big.little technology.
- This ensures the increase in performance due to high-performance processor core as well as low power consumption due to low performance processor core. Hence improving battery life.
- ARM CoreLink cache coherent network offers scaling upto 16 processors that links both the processors i.e. the high performance core and low performance core.

8. ARM v8

- It is a 64-bit architecture.
- ARM v8 adds a Cryptographic extension which is optional
- There are two versions of ARMv8 available in market:
  1. ARM v8-A is for high performance market such as mobile and enterprise. These are power efficient implementations
  2. ARM v8-R is for low performance market such as embedded, industrial as well as automotive applications

6.3 ARM v7 : Cortex-A, Cortex-R and Cortex-M

GTU - S-18, S-19

Q.	What is difference between Cortex A, Cortex R and Cortex M ARM processors?	**S-18, 7 Marks**
Q.	Discuss ARM core extensions and architecture revisions.	**S-19, 4 Marks**

- There are various Cortex series of ARM processor based on the special features provided in the processor.
- The Cortex-A series of microcontrollers are for applications that require a lot of computation and controlling. The Cortex-R are especially meant for the real time based applications.
- The Cortes-M processors are meant for the special category of microcontrollers with low power consumption and hence are also called as the future microcontrollers.
- They have a wide range of features that are improved over the previous versions. Each of the microcontrollers have some or the other improvement over its predecessor.

6.3.1 Cortex-A Series

- The ARM cortex-A series processors are a series of application processors used in a wide range of devices that host a rich OS.
- The applications of the Cortex A series ranges from low cost handset to the smart phones. They are used in mobile computing, digital TV and set-top boxes, networking, printers etc.
- Cortex-A processors are used in embedded systems that have high-computational requirements and run on rich operating systems.
- They are also used to deliver interactive media and graphics experience from the latest technological mobile devices that need to have internet access such as handsets and ultra-portable SmartBooks to automotive information and entertainment systems and next generation digital TV systems. Cortex-A processors are also geared towards providing the full internet experience, but are used in a wide variety of applications.

- The following is a list of application areas of Cortex A processors :

1.	Smartbook	2.	Tablet
3.	E-reader	4.	Thin client
5.	Smart Phones	6.	Feature phones
7.	Set top box	8.	Digital TV
9.	Blu Ray player	10.	Gaming consoles
11.	Navigation	12.	Printers
13.	Routers, etc		

6.3.2 Cortex-R Series

- The ARM Cortex-R series is a real time processor series that have high performance and computing solutions for applications that require high reliability, availability, fault tolerance and maintainability.
- Thus the applications that require ARM Cortex-R series processors are mainly such that they require the following attributes:
  1. **High Performance :** The processors are fast in processing as well as have high clock frequency.
  2. **Real-time :** The processors meet the real time deadline constraints in all cases.
  3. **Safe :** The processors are reliable and dependable
  4. **Cost effective :** It requires low power and physical area and hence are cost effective.
- The following is a list of application areas of Cortex R processors :

1.	Vehicle airbag	2.	Vehicle braking
3.	HDD (Hard Disk Drive) Controller	4.	3G, 4G mobiles
5.	Smart phones	6.	Modems
7.	Medical & industrial applications	8.	Networking
9.	Digital camera	10.	Printers, etc

6.3.3 Cortex-M Series

- The Cortex-M processors are meant mainly for mixed signal processing. The ARM Cortex M series processors are very high end processors designed to cater the needs of the future upcoming embedded system applications.
- They are designed for mixed signal processing applications such as smart metering, human interface devices, consumer instruments, medical instruments, etc.
- The main advantages of ARM Cortex-M series are given below:
  1. **Energy efficiency:** These processors have low energy costs and longer battery life
  2. **It requires smaller code:** Since the instruction density is higher, the code is smaller i.e. occupies less memory. Hence less cost of the memory.
  3. **Ease of use:** Since there are lot of tools available, the designing and development is much easy.
  4. **High Performance:** It delivers rich performance at low power.

6.4 Cortex-A Processor Series Survey : Features, Versions and Applications

GTU - S-19

Q. List at least two applications for Cortex A, Cortex M and Cortex R processors. **S-19, 3 Marks**

- Cortex A processors are high performance and low power processors . They have multiple processor cores to increase the performance. For example Cortex-A17 has one to four processors in a single multiprocessor IC. A multiprocessor cortex-A processor normally has the following:

  1. Processor (one or multiple)
  2. Cache memory (one or multi levels)
  3. Generic Timer
  4. Debugging hardware tool
  5. Power management
  6. Peripheral ports etc.

- Each processor in the Cortex-A processor will have its own generic timer, debugger etc. The block diagram of a Cortex-A processor is as shown in the Fig. 6.4.1.

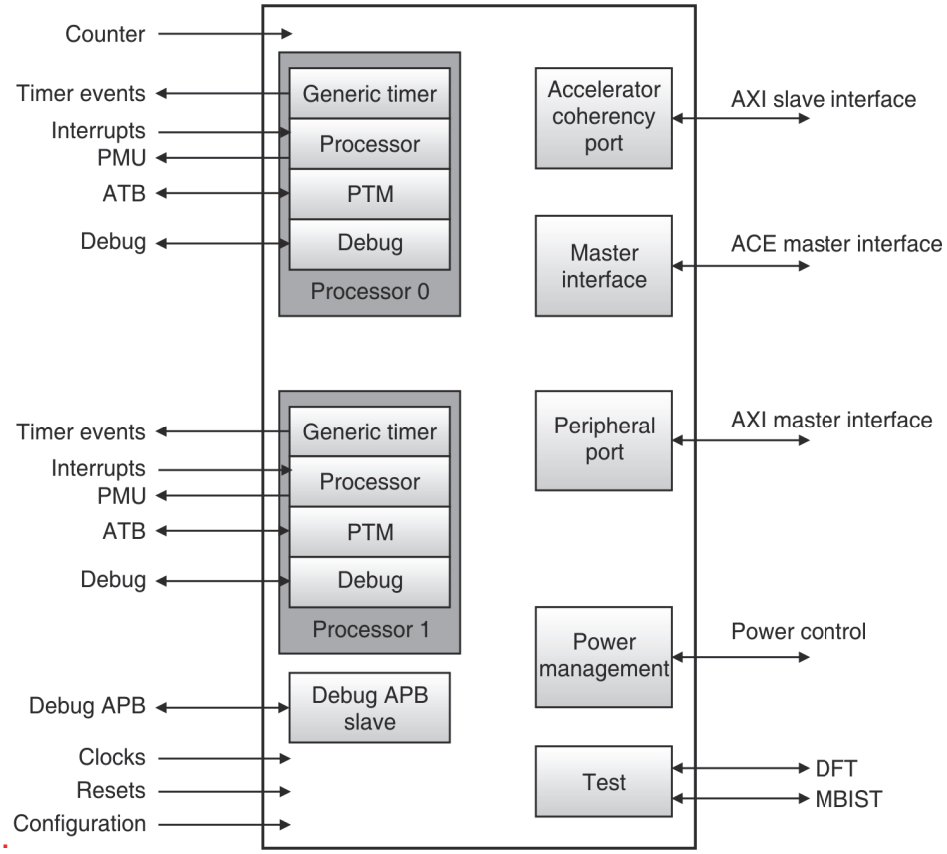


Fig. 6.4.1 : Multiprocessor block diagram

Some of the special and common features of all the versions of the ARM Cortex-A processors are as listed below:

1. The Cortex-A processor support the features and the architecture of ARM7 or ARM8 processor. Thus the processor can support applications of either 64-bit or 32-bit.
2. It supports various instruction sets like that of ARM instruction set, Thumb/Thumb-2 instruction set, Jazelle instruction set, DSP instruction set.
3. The support for various Operating Systems (OS) like Android, Ubuntu, QNX, Symbian, Windows CE etc.

4. It supports special security extensions
5. It has an addressing capacity of upto 1TB
6. It also has a media processing engine
7. It is a RISC processor with 15+ pipeline stages
8. It has an advanced branch prediction logic
9. It has multiple cores (1 to 4 cores)
10. It has L1 cache with full cache coherency. The L1 cache also has a snoop port.
11. It supports pipeline load and store operations and hence increase the performance.

Some of the features of the various versions of the Cortex-A processors is given in the Table 6.4.1:

Table 6.4.1 : Features of Cortex-A processor Series

Core	Cortex-A5	Cortex-A5 MPCore	Cortex-A7	Cortex-A9	Cortex-A9 MPCore	Cortex-A15	Cortex-A17	Cortex-A53	Cortex-A57
Architecture	ARMv7	ARMv7+MP	ARMv7+MP +LPAE	ARMv7	ARMv7+MP	ARMv7+MP +LPAE	ARMv7+MP +LPAE	ARMv8	ARMv8
Interrupt Controller	GIC-390	Integrated-GIC	GIC-400	GIC-390	Integrated-GIC	Integrated-GIC	GIC-400	GIC-500	GIC-500
L2 Cache Controller	L2C-310	L2C-310	Integrated	L2C-310	L2C-310	Integrated	Integrated	Integrated	Integrate

Let us see the special application of the various versions of the processors in the Cortex-A series.

6.4.1 Cortex-A57

- Amongst all the Cortex-A processors, A57 is the best processor in terms of performance. It is capable of supporting future mobiles as well as the computing applications such as tablets.
- The special features of Cortex-A57 processor include:
  1. As discussed earlier it has the highest performance and can be used in the latest smart phones.
  2. It supports 64-bit applications and also supports backward compatibility for the 32-bit applications.
  3. It also has reliability and scalability features for the high-performance applications.

6.4.2 Cortex-A53

- The Cortex-A53 processor has the architecture of ARM 8 and is the most power efficient processor that supports 64-bit applications and is also backward compatible to support 32-bit applications. It is used in applications like basic smart phones.
- The special features of Cortex-A53 processor include:
  1. As discussed earlier it is one of the most power efficient processors and is capable of performing good enough.

2. It supports 64-bit applications and also supports backward compatibility for the 32-bit applications.
3. It also has reliability and scalability features for the high-performance.

6.4.3 Cortex-A17

- The ARM Cortex-A17 processor is a slightly older processor that was targeted for the mobile phones, tablets etc. It is designed with a premium performance and very high set of features.
- The special features of Cortex-A17 processor include:
  1. As discussed earlier it is a processor with premium performance and high end set of features.
  2. It can access a memory of upto 1TB.

6.4.4 Cortex-A15

- The ARM Cortex-A15 processor delivers high performance with low power consumption. It is used in digital applications and has acquired quite a huge section of the market.
- The mobile phones made using Cortex A-15 have a performance much more than the advanced processors and is not only cost effective but also power effective.
- The special features of Cortex-A15 processor include:
  1. As discussed earlier it is a processor with premium performance and consumes less power.
  2. It can access a memory of upto 1TB.
  3. It has an efficient hardware support for data management and arbitration.

6.4.5 Cortex-A9

- The Cortex-A9 series of processors have single to quad core processors with huge MIPS performance per core. It also has a cache coherent memory management and many peripheral controller in the processor. It is also energy efficient processors processor with high performance.
- The special features of Cortex-A9 processor include:
  1. As discussed earlier it is a processor with ideal performance power tradeoff.
  2. It can have one to four cores of processors.

6.4.6 Cortex-A7

- The Cortex-A7 processors are also very energy efficient processor. In fact it was the most energy efficient processor developed till then by ARM. It has better thread performance than its predecessor A5 processor.
- The special features of Cortex-A7 processor include:
  1. As discussed earlier it is a processor with better performance and energy efficient features.
  2. Its architecture and features are almost same as that of A15 or A17 processors in the Cortex-A series
  3. The main aim of the architectural design of the A7 processor was energy efficiency.

6.4.7 Cortex-A5

- Cortex-A5 processor is the smallest of the series with small size constraints and still high end application support. It can be used as a standalone processor to support wide range of applications from smart devices to entry level phones and consumer appliances. It has also been a power efficient processor.
- The special features of Cortex-A5 processor include:
  1. As discussed earlier it is a processor designed with size as the main constraint and hence is a very small sized processor.
  2. It has a wide range of applications that it supports from mobiles to consumer appliances.

  It is application compatible with the later versions of processors.

6.5 Cortex-R Processor Series Survey : Features, Versions and Applications

- Cortex-R processors are used in the embedded applications where real time responses are required. It also ensures high reliability, high availability and maintainability.
- There is also a wide range of processors in the series of Cortex-R. The major common features of Cortex-R series can be listed as below:
  1. High performance i.e. fast processing and high clock frequency. It gives fast execution for complex code and DSP functionality
  2. Real-time i.e. all the deadlines for hard real-time applications are met with. It has a deterministic response to the interrupts.
  3. It supports error correction and parity codes and hence gives a reliable system operation.
  4. It also has a memory protection unit and hence is said to give safe operation.
  5. It is energy efficient processor.
  6. It is also a cost efficient processor.
  7. It has many stages of pipelining and hence is said to be an advanced architecture.
  8. It has a branch prediction logic.
  9. It has a superscalar operation.
  10. There is an option for the floating point unit.

6.5.1 Comparison of Various Cortex-R Processors

- The Cortex-R processors series have various processors like Cortex-R4, Cortex-R5 and Cortex-R7.
- The cortex-R4 is a high performance real time with deterministic response to interrupts.

- The Cortex-R5 has a dual core configuration. It has a space saving floating point unit. It has I/O coherency and fast accessing for the peripheral devices.
- The Cortex-R7 has a large performance increase over its predecessors. It has a high clock frequency and symmetric multiprocessing. It has error management system built-in. It also has a on-chip interrupt controller.

6.5.2 Cortex-R4 Processor

- The Cortex-R4 processor has the following special features:
  1. The performance of Cortex-R4 is 1.68 to 2.45 DMIPS/MHz.
  2. It has a tightly coupled memory. Thus the memory accesses are faster.
  3. It has a two issue super scalar architecture i.e. there are two pipelines that can process two instructions simultaneously.
  4. The processor is 8-staged pipelined processor.
  5. It has a separate cache for code (or instructions) and for data i.e. the I-Cache and D-Cache.
  6. It has a hardware circuit implemented for division.
  7. It has a support for SIMD i.e. Single Instruction Multiple Data operations.
  8. It also has support for DSP (Digital Signal Processing) operations.
  9. It supports IEEE 754 double precision standard for representing floating point data.
  10. It has a vectored Interrupt controller.

6.5.3 Cortex-R5 Processor

- The Cortex-R5 processor has the following special features:
  1. The performance of Cortex-R5 is 1.68 to 2.45 DMIPS/MHz. It has a dual core asymmetric multiprocessing
  2. It has a tightly couple memory. Thus the memory accesses are faster.
  3. The peripheral port access has a lower latency in case of Cortex-R5.
  4. It has a two issue super scalar architecture i.e. there are two pipelines that can process two instructions simultaneously.
  5. The processor is 8-staged pipelined processor.
  6. It has a branch prediction logic.
  7. It has a separate cache for code (or instructions) and for data i.e. the I-Cache and D-Cache.
  8. It has a hardware circuit implemented for division.
  9. It has a support for SIMD i.e. Single Instruction Multiple Data operations.

10. It also has support for DSP (Digital Signal Processing) operations.
11. It supports IEEE 754 double precision standard for representing floating point data.
12. It has a vectored Interrupt controller.
13. It also has a generic interrupt controller.

6.5.4 Cortex-R7 Processor

- The Cortex-R7 processor has the following special features:

1. The performance of Cortex-R7 is 2.5 to 4.35 DMIPS/MHz. It has a dual core asymmetric as well as symmetric multiprocessing
2. It has a tightly couple memory. Thus the memory accesses are faster.
3. The peripheral port access has a lower latency in case of Cortex-R5.
4. It has a two issue super scalar architecture i.e. there are two pipelines that can process two instructions simultaneously.
5. The processor is 11-staged pipelined processor.
6. It has a dynamic as well as static branch prediction logic.
7. It has a separate cache for code (or instructions) and for data i.e. the I-Cache and D-Cache.
8. It has a hardware circuit implemented for division.
9. It has a support for SIMD i.e. Single Instruction Multiple Data operations.
10. It also has support for DSP (Digital Signal Processing) operations.
11. It supports IEEE 754 double precision standard for representing floating point data.
12. It has a vectored Interrupt controller.
13. It also has a generic interrupt controller.

6.5.5 Applications of ARM Cortex-R Processors

The various application sectors where the ARM Cortex-R processors are used are as listed below:

1. Automotive sector: ARM Cortex-R series of processors are used in various units of automotive like airbags, brakes, dashboard display, engine control etc.
2. Data Storage: ARM Cortex-R series of processors are used for storage devices like hard disk controller
3. Mobile Handsets: ARM Cortex-R finds its application in smart phones.
4. It also finds application in various embedded systems like medical instruments and industrial devices.
5. It is used in various other applications like networking instruments, printers, BluRay players, portable media players, digital cameras etc.

6.6 Cortex-M Processor Series Survey : Features, Versions and Applications

- Cortex R processors are used in the embedded applications where mixed signal processing is possible. There are various places where this mixed signal processing is required like various household appliances (refrigerator control, freezers, washing machine, etc), consumer products, medical instrumentation etc.
- The various processors in the series of Cortex-M are M0, M0+, M3, M4 and M7.
- The Cortex-M0 has the lowest area and cost the Cortex-M0+ is the lowest power consuming and hence most energy efficient. The ARM Cortex-M0 processor is suitable for applications like touch screen controller and power management. The ARM Cortex-M0+ processor is suitable for applications like bluetooth smart transceiver, etc.
- ARM Cortex-M3 is performance efficiency and has large set of features for connectivity. It is very efficient in terms of various protocols it supports for connectivity. The applications for which ARM Cortex-M3 processor is most suitable are WiFi transceiver, activity tracker etc.
- ARM Cortex-M4 is a processor with DSP features and also supports SIMD and floating point operations. The ARM Cortex-M4 processor is most suitable for smart metering or other meter applications
- ARM Cortex-M7 is a processor with maximum digital signal computing performance. It has a flexible memory organization. It has cache memory, error coding control etc. It supports single as well as double precision of floating point standards for the data. The ARM Cortex-M7 processors are most suitable for applications like high end audio headset, automotive electronics etc.

6.7 ARM Optimization Techniques

GTU - S-17, W-18

Q.	Describe any three ARM optimization techniques.	**(S-17, 3 Marks)**
Q.	List and explain the various ARM optimization techniques.	**(W-18, 7 Marks)**

There are various things to be considered to optimise the ARM code. The various portability issues for ARM compiler are discussed here :

1. The "int" type data is mostly taken as a 16-bit data format, but for the 32-bit ARM processors it is a 32-bit format. This issue has to be specially taken care in case of signed data.
2. The "char" type data in most of the microcontrollers including ARM is by default unsigned, while for C programmers it is by default signed. This can cause severe issues like infinite loop in case of the condition being checked is i<0. The ARM compiler generates a warning in this case so that the programmer can take care of the same.
3. The two types of endian supported by ARM has to be taken care of. In case if one function is written considering little endian, while the other considering big endian, the system will not work. As discussed earlier, the endian mode can be initialised only once during the reset of the system.

4. The endian problem will also result into issues for the bitfields, as they access a bit in a higher size of data like byte, double byte etc. Thus if the wrong bit is accessed based on endian, the entire system will work in an abrupt manner.
5. The inline assembly feature of the ARM compilers must be used very carefully. It may access the registers that are assigned as another variable by the C program. Hence separate functions must be written using assembly so as to avoid the re-usage of same registers for multiple variables.
6. The alignment of the data pointers also has to be taken care of. The pointers must be aligned according to the data size of the processor being used.

❑❑❑