

SUBJECT CODE : 210254

As per Revised Syllabus of  
**SAVITRIBAI PHULE PUNE UNIVERSITY**  
Choice Based Credit System (CBCS)  
S.E. (Computer) Semester - IV

# MICROPROCESSOR

**Atul P. Godse**

M.S. Software Systems (BITS Pilani)

B.E. Industrial Electronics

Formerly Lecturer in Department of Electronics Engg.

Vishwakarma Institute of Technology

Pune

**Dr. Deepali A. Godse**

M.E., Ph.D. (Computer Engg.)

Head of Information Technology Department,  
Bharati Vidyapeeth's College of Engineering for Women,  
Pune



# MICROPROCESSOR

Subject Code : 210254

S.E. (Computer Engineering) Semester - IV

© Copyright with Authors

All publishing rights (printed and ebook version) reserved with Technical Publications. No part of this book should be reproduced in any form, Electronic, Mechanical, Photocopy or any information storage and retrieval system without prior permission in writing, from Technical Publications, Pune.

Published by :



Amit Residency, Office No.1, 412, Shaniwar Peth,  
Pune - 411030, M.S. INDIA, Ph.: +91-020-24495496/97  
Email : sales@technicalpublications.org Website : www.technicalpublications.org

Printer :

Yogiraj Printers & Binders  
Sr.No. 10/1A,  
Ghule Industrial Estate, Nanded Village Road,  
Tal. - Haveli, Dist. - Pune - 411041.

ISBN 978-81-947993-9-9



9 788194 799399

SPPU 19

# PREFACE

The importance of **Microprocessor** is well known in various engineering fields. Overwhelming response to our books on various subjects inspired us to write this book. The book is structured to cover the key aspects of the subject **Microprocessor**.

The book uses plain, lucid language to explain fundamentals of this subject. The book provides logical method of explaining various complicated concepts and stepwise methods to explain the important topics. Each chapter is well supported with necessary illustrations, practical examples and solved problems. All the chapters in the book are arranged in a proper sequence that permits each topic to build upon earlier studies. All care has been taken to make students comfortable in understanding the basic concepts of the subject.

Representative questions have been added at the end of each section to help the students in picking important points from that section.

The book not only covers the entire scope of the subject but explains the philosophy of the subject. This makes the understanding of this subject more clear and makes it more interesting. The book will be very useful not only to the students but also to the subject teachers. The students have to omit nothing and possibly have to cover nothing more.

We wish to express our profound thanks to all those who helped in making this book a reality. Much needed moral support and encouragement is provided on numerous occasions by our whole family. We wish to thank the **Publisher** and the entire team of **Technical Publications** who have taken immense pain to get this book in time with quality printing.

Any suggestion for the improvement of the book will be acknowledged and well appreciated.

*Authors*  
A. P. Godse  
Dr. D. A. Godse

*Dedicated to Neha & Ruturaj.*

# SYLLABUS

## **Microprocessor - (210254)**

Credit Scheme	Examination Scheme and Marks:
03	Mid_Semester (TH) : 30 Marks
	End_Semester (TH) : 70 Marks

### **Unit I Introduction to 80386**

Brief History of Intel Processors, 80386 DX Features and Architecture, Programmers Model, Operating modes, Addressing modes and data types.

**Applications Instruction Set :** Data Movement Instructions, Binary Arithmetic Instructions, Decimal Arithmetic Instructions, Logical Instructions, Control Transfer Instructions, String and Character Transfer Instructions, Instructions for Block Structured Language, Flag Control Instructions, Coprocessor Interface Instructions, Segment Register Instructions, Miscellaneous Instructions. (**Chapters - 1, 2**)

### **Unit II Bus Cycles and System Architecture**

**Initialization -** Processor State after Reset, Functional pin Diagram, functionality of various pins, I/O Organization, Memory Organization (Memory banks), Basic memory read and writes cycles with timing diagram.

**Systems Architecture -** Systems Registers (Systems flags, Memory Management registers, Control registers, Debug registers, Test registers), System Instructions. (**Chapter - 3**)

### **Unit III Memory Management**

Global Descriptor Table, Local Descriptor Table, Interrupt Descriptor Table, GDTR, LDTR, IDTR. Formats of Descriptors and Selector, Segment Translation, Page Translation, Combining Segment and Page Translation. (**Chapter - 4**)

## **Unit IV Protection**

Need of Protection. Overview of 80386DX Protection Mechanisms : Protection rings and levels, Privileged Instructions, Concept of DPL, CPL, RPL, EPL. Inter privilege level transfers using Call gates, Conforming code segment. Privilege levels and stacks. Page Level Protection, Combining Segment and Page Level Protection. (**Chapter - 5**)

## **Unit V Multitasking and Virtual 8086 Mode**

**Multitasking** - Task State Segment, TSS Descriptor, Task Register, Task Gate Descriptor, Task Switching, Task Linking, Task Address Space.

**Virtual Mode** - Features, Memory management in Virtual Mode , Entering and leaving Virtual mode. (**Chapters - 6, 7**)

## **Unit VI Interrupts, Exceptions, and Introduction to Microcontrollers**

**Interrupts and Exceptions** : Identifying Interrupts, Enabling and Disabling Interrupts, Priority among Simultaneous Interrupts and Exceptions, Interrupt Descriptor Table (IDT), IDT Descriptors, Interrupt Tasks and Interrupt Procedures, Error Code, and Exception Conditions.

**Introduction to Microcontrollers** : Architecture of typical Microcontroller, Difference between Microprocessor and Microcontroller, Characteristics of microcontrollers, Application of Microcontrollers. (**Chapters - 8, 9**)

# TABLE OF CONTENTS

## Unit - I

<b>Chapter - 1      Introduction to 80386</b>	<b>(1 - 1) to (1 - 24)</b>
1.1 Brief History of Intel Processors .....	1 - 2
1.2 80386DX Features.....	1 - 5
1.3 80386DX Architecture.....	1 - 6
1.4 Programmers Model.....	1 - 9
1.4.1 General Purpose / Multipurpose Registers .....	1 - 10
1.4.2 Special - Purpose Registers .....	1 - 11
1.4.3 EFLAGS.....	1 - 11
1.4.4 Segment Registers .....	1 - 14
1.4.4.1 CS (Code Segment) and CS Register .....	1 - 14
1.4.4.2 DS (Data Segment) and DS Register .....	1 - 14
1.4.4.3 ES (Extra Segment) and ES Register .....	1 - 15
1.4.4.4 SS (Stack Segment) and SS Register .....	1 - 15
1.4.4.5 FS and GS .....	1 - 15
1.4.5 Segments and Offsets.....	1 - 15
1.5 Operating Modes.....	1 - 17
1.6 Addressing Modes .....	1 - 17
1.6.1 Immediate Operands .....	1 - 17
1.6.2 Register Operands .....	1 - 18
1.6.3 Memory Operands .....	1 - 18
1.7 Data Types .....	1 - 21
<b>Chapter - 2      Applications Instruction Set</b>	<b>(2 - 1) to (2 - 58)</b>
2.1 Introduction .....	2 - 2
2.1.1 Data Movement Instructions.....	2 - 2

2.1.2 Binary Arithmetic Instructions . . . . .	2 - 2
2.1.3 Decimal Arithmetic Instructions . . . . .	2 - 2
2.1.4 Logical Instructions . . . . .	2 - 3
2.1.5 Control Transfer Instructions . . . . .	2 - 3
2.1.6 String and Character Translation Instructions . . . . .	2 - 3
2.1.7 Instructions for Block-Structured Languages . . . . .	2 - 4
2.1.8 Flag Control Instructions . . . . .	2 - 4
2.1.9 Coprocessor Interface Instructions . . . . .	2 - 4
2.1.10 Segment Register Instructions . . . . .	2 - 4
2.1.11 Miscellaneous Instructions . . . . .	2 - 4
2.2 Instruction Set of 80386.....	2 - 4

## Unit - II

<b>Chapter - 3      Bus Cycles and System Architecture</b>	<b>(3 - 1) to (3 - 36)</b>
3.1 Initialization .....	3 - 2
3.2 Processor State after Reset.....	3 - 2
3.3 Functional Pin Diagram .....	3 - 3
3.3.1 Memory/IO Interface Signals . . . . .	3 - 3
3.3.2 Interrupt Interface Signals . . . . .	3 - 7
3.3.3 DMA Interface Signals . . . . .	3 - 8
3.3.4 Coprocessor Interface Signals . . . . .	3 - 8
3.4 I/O Organization .....	3 - 9
3.4.1 I/O Mapped I/O. . . . .	3 - 10
3.4.2 Memory Mapped I/O . . . . .	3 - 11
3.5 Memory Organization (Memory Banks) .....	3 - 13
3.6 Basic Memory Read and Writes Cycles with Timing Diagram .....	3 - 17
3.6.1 Non-Pipelined Bus Cycles. . . . .	3 - 18
3.6.2 Pipelined Bus Cycles . . . . .	3 - 18
3.6.3 Idle State in Pipelined Bus Cycles . . . . .	3 - 19
3.6.4 Bus Cycle with Wait State . . . . .	3 - 20

3.6.5 Non-Pipelined Read Cycle .....	3 - 20
3.6.6 Non-Pipelined Write Cycle.....	3 - 22
3.6.7 Non-Pipelined Read / Write Cycles.....	3 - 24
3.7 Systems Architecture.....	3 - 26
3.8 Systems Registers .....	3 - 27
3.8.1 System Flags - EFLAGS .....	3 - 27
3.8.2 Memory-Management (System Address) Registers .....	3 - 27
3.8.3 Control Registers.....	3 - 27
3.8.4 Debug Registers .....	3 - 29
3.8.5 Test Registers .....	3 - 31
3.9 Systems Instructions.....	3 - 35

## Unit - III

### **Chapter - 4     Memory Management    (4 - 1) to (4 - 28)**

4.1 Address Translation Overview .....	4 - 2
4.2 Segment Translation.....	4 - 3
4.2.1 Selector .....	4 - 3
4.2.1.1 Index Part .. . . . .	4 - 3
4.2.1.2 Requester's Privilege Level (RPL) . . . . .	4 - 3
4.2.1.3 Table Indicator (TI). . . . .	4 - 4
4.2.2 Global Descriptor Table (GDT) and Local Descriptor Table (LDT) .....	4 - 4
4.2.3 Segment Descriptor .....	4 - 6
4.2.4 General Format of Descriptor .....	4 - 6
4.2.5 Types of Segment Descriptors and Their Formats .....	4 - 8
4.2.5.1 Non-system (Code and Data) Segment Descriptor . . . . .	4 - 8
4.2.5.2 System Segment Descriptors . . . . .	4 - 11
4.2.6 Descriptor Tables - GDT, IDT and LDT .....	4 - 12
4.2.7 Descriptor Registers - GDTR, LDTR and IDTR .....	4 - 14
4.2.7.1 Global Descriptor Table Register (GDTR) . . . . .	4 - 14
4.2.7.2 Interrupt Descriptor Table Register (IDTR) . . . . .	4 - 15

4.2.7.3 Local Descriptor Table Register (LDTR) . . . . .	4 - 15
4.2.8 Segment Registers and Segment Descriptor Cache.....	4 - 17
4.3 Page Translation .....	4 - 20
4.3.1 Page Tables .....	4 - 22
4.3.2 PDE Descriptor .....	4 - 22
4.3.3 PTE Descriptor .....	4 - 24
4.3.4 Translation Lookaside Buffer/Page Translation Cache.....	4 - 25
4.4 Combining Segment and Page Translation .....	4 - 27

## Unit - IV

<b>Chapter - 5     Protection</b>	<b>(5 - 1) to (5 - 20)</b>
5.1 Need of Protection .....	5 - 2
5.2 Overview of 80386DX Protection Mechanisms .....	5 - 2
5.3 Segment Level Protection.....	5 - 3
5.3.1 Type Checking .....	5 - 3
5.3.2 Limit Checking.....	5 - 4
5.3.3 Protection Levels - Privilege Level Protection .....	5 - 4
5.3.4 Concept of DPL, CPL, RPL and EPL.....	5 - 5
5.3.4.1 Restricting Access to Data .. . . . .	5 - 6
5.3.4.2 Accessing Data in Code Segments .. . . . .	5 - 7
5.3.4.3 Restricting Control Transfers .. . . . .	5 - 8
5.3.5 Changing Privilege Levels.....	5 - 8
5.3.5.1 Conforming Code Segment .. . . . .	5 - 8
5.3.5.2 Inter Privileged Level Transfers using Call Gates .. . . . .	5 - 9
5.3.6 Stack Switching .. . . . .	5 - 13
5.4 Page Level Protection .....	5 - 15
5.4.1 Restricting Addressable Domain .. . . . .	5 - 16
5.4.2 Type Checking.....	5 - 16
5.5 Combining Segment and Page Level Protection.....	5 - 16
5.6 I/O Protection .. . . . .	5 - 17

5.6.1 I/O Privilege Level . . . . .	5 - 17
5.6.2 I/O Permission Bit Map . . . . .	5 - 17
5.7 Privilege and I/O Sensitive Instructions . . . . .	5 - 19
5.7.1 Privileged Instructions . . . . .	5 - 19
5.7.2 IOPL Sensitive Instructions . . . . .	5 - 19

## Unit - V

<b>Chapter - 6 Multitasking</b>	<b>(6 - 1) to (6 - 16)</b>
6.1 Introduction . . . . .	6 - 2
6.2 Task State Segment . . . . .	6 - 3
6.3 TSS Descriptor . . . . .	6 - 5
6.4 Task Register . . . . .	6 - 6
6.5 Task Gate Descriptor . . . . .	6 - 8
6.6 Task Switching . . . . .	6 - 9
6.6.1 Task Switching without Task Gate . . . . .	6 - 9
6.6.2 Task Switching with Task Gate . . . . .	6 - 11
6.7 Task Linking . . . . .	6 - 12
6.8 Task Address Space . . . . .	6 - 13
6.8.1 Task Linear-to-Physical Space Mapping . . . . .	6 - 14
6.8.2 Task Logical Address Space . . . . .	6 - 15
<b>Chapter - 7 Virtual 8086 Mode</b>	<b>(7 - 1) to (7 - 10)</b>
7.1 Features . . . . .	7 - 2
7.2 Executing 8086 Code . . . . .	7 - 3
7.2.1 Registers . . . . .	7 - 3
7.2.2 Instructions . . . . .	7 - 3
7.3 Memory Management in Virtual Mode . . . . .	7 - 4
7.3.1 Linear Address Formation . . . . .	7 - 4
7.3.2 Structure of V86 Task . . . . .	7 - 5

7.3.3 Using Paging for V86 Tasks.....	7 - 6
7.3.4 Protection within a V86 Task.....	7 - 6
7.4 Entering and Leaving Virtual 8086 Mode .....	7 - 7
7.4.1 Entering 8086 Virtual Mode.....	7 - 7
7.4.2 Leaving 8086 Virtual Mode .....	7 - 7
7.5 Difference between Real, Protected and Virtual 8086 Modes.....	7 - 8

## Unit - VI

<b>Chapter - 8    Interrupts and Exceptions</b>	<b>(8 - 1) to (8 - 20)</b>
8.1 Introduction .....	8 - 2
8.2 Identifying Interrupts.....	8 - 3
8.3 Enabling and Disabling Interrupts.....	8 - 4
8.3.1 NMI Masks Further NMIs .....	8 - 4
8.3.2 IF Masks INTR .....	8 - 4
8.3.3 RF Masks Debug Faults.....	8 - 4
8.3.4 MOV or POP to SS Masks Some Interrupts and Exceptions .....	8 - 4
8.4 Priority Among Simultaneous Interrupts and Exceptions.....	8 - 5
8.5 Interrupt Descriptor Table .....	8 - 6
8.6 Interrupt Tasks and Interrupt Procedures .....	8 - 8
8.6.1 Interrupt Procedures .....	8 - 8
8.6.1.1 Stack of Interrupt Procedure .....	8 - 9
8.6.1.2 Trap Gate Vs Interrupt Gate .....	8 - 9
8.6.1.3 Returning from an Interrupt Procedure .....	8 - 10
8.6.2 Interrupt Tasks .....	8 - 11
8.7 Error Code.....	8 - 12
8.8 Exception Conditions .....	8 - 13

<b>Chapter - 9    Introduction to Microcontrollers</b>	<b>(9 - 1) to (9 - 32)</b>
--	----------------------------

9.1 Microcontrollers and Embedded Processors.....	9 - 2
---	-------

9.1.1 Comparison between Microprocessor and Microcontroller .....	9 - 3
9.1.2 Different Types of Microcontrollers .....	9 - 4
9.1.2.1 Embedded Microcontrollers . . . . .	9 - 4
9.1.2.2 External Memory Microcontrollers . . . . .	9 - 5
9.1.3 Criteria for Selecting Microcontroller .....	9 - 5
9.1.4 Applications of Microcontroller .....	9 - 6
9.2 Features of 8051 Microcontroller.....	9 - 6
9.3 Block Diagram of 8051 Microcontroller.....	9 - 8
9.4 Register Organization of 8051 Microcontroller .....	9 - 9
9.4.1 A and B Registers .....	9 - 9
9.4.2 Data Pointer (DPTR) .....	9 - 10
9.4.3 Program Counter .....	9 - 10
9.4.4 8051 Flag Bits/PSW Registers .....	9 - 11
9.4.5 Special Function Registers .....	9 - 12
9.4.6 Stack and Stack Pointer .....	9 - 14
9.5 Pin Diagram of 8051 .....	9 - 16
9.6 Memory Organization.....	9 - 18
9.6.1 Internal RAM Organization .....	9 - 19
9.6.1.1 8051 Register Banks (Working Registers) . . . . .	9 - 20
9.6.1.2 Bit / Byte Addressable . . . . .	9 - 21
9.6.1.3 General Purpose RAM . . . . .	9 - 21
9.6.2 ROM Space in the 8051 .....	9 - 21
9.7 External Memory Interfacing.....	9 - 21
9.7.1 External Program Memory.....	9 - 22
9.7.2 External Data Memory .....	9 - 24
9.7.3 Accessing External Data Memory in 8051C .....	9 - 26

---

## **Microprocessor Laboratory**

**(L - 1) to (L - 62)**

---

## **Solved SPPU Question Papers**

**(S - 1) to (S - 10)**

---

## **Solved Model Question Papers**

**(S - 11) to (S - 14)**

## **UNIT - I**

**1**

# **Introduction to 80386**

### **Syllabus**

*Brief History of Intel Processors, 80386 DX Features and Architecture, Programmers Model, Operating modes, Addressing modes and data types.*

### **Contents**

1.1	<i>Brief History of Intel Processors</i>	
1.2	<i>80386DX Features</i>	<i>May-10,13, Marks 4</i>
1.3	<i>80386DX Architecture</i>	<i>May-2000,06,11,12,13, Dec.-05,11,13, Nov.-12, Marks 8</i>
1.4	<i>Programmers Model</i>	<i>Dec.-03,14,19, May-10,13,19, Marks 6</i>
1.5	<i>Operating Modes</i>	
1.6	<i>Addressing Modes</i>	<i>Dec.-14,17, May-18, Marks 3</i>
1.7	<i>Data Types</i>	<i>May-14,19, Marks 5</i>

## 1.1 Brief History of Intel Processors

- The world's first microprocessor, the Intel 4004, was a 4-bit microprocessor. (A bit is a binary digit with a value zero or one and 4-bit microprocessor means the microprocessor can process 4-bit word in one cycle. It has 12-bit address lines to access 4096 4-bit wide memory locations. The 4004 microprocessor has only 45 instructions.
- The Intel released the 4040, as updated version of 4004 with enhancement in speed, and without any improvement in word length and memory size.
- In 1972 announced the 8008, 8-bit and faster version of 4004. This version came up with expanded memory size upto 16 kbytes and additional instructions to make total of 48 instructions. (A byte is 8-bit binary number and a K is 1024).
- In 1974 Intel came out with 8080 was a considerable improvement over its predecessors.
- The 8080 had a much larger instruction set and since NMOS technology used, it is much faster than 8008.
- In 1977, Intel introduced updated version of 8080-8085.
- The Table 1.1.1 shows the improvement of 8080 over 8008 and 8085 over 8080.

Parameter	Processor		
	8008	8080	8085
Speed	Requires 20 $\mu$ s for execution one instruction	Requires 2 $\mu$ s for execution one instruction (10 times faster)	Requires 1.3 $\mu$ s for execution of one instruction
Memory size	16 kbytes	64 kbytes (4 times more)	64 kbytes
TTL compatibility	Not directly compatible	Compatible	Compatible
Interfacing	More costly and complex	Easier and less expensive	More easier and less expensive as it contains internal clock generator and internal system controller

Table 1.1.1

- The next generation was 8086 processor, a 16-bit processor, with advanced architecture and instruction set. At the same time Intel introduced processor 8088. The 8088 is an 8-bit version of the 8086 which has fewer data lines but retains all of the processing features of the 8086. The programs that run on 8088 will also run, without modification on the 8086. The 8086/88 pair were the first members of

iAPX 86, family of microprocessors. This pair has 20 address lines to address upto 1 Mbyte of memory (1 Mbyte = 1024 kbyte) and also supported with 4 or 6 byte instruction queue to implement pipelining feature. (Feature of fetching the next instruction while the current instruction is executing is called pipelining). This pair belongs to CISC (complex instruction set computers) because of the number and complexity of instructions.

- In 1983 the next version was announced, the 80186/88 very similar to 8086/8088 pair. The 80186/88 included many useful peripheral I/O functions as an integral part of the microprocessor. The improved instruction set of 80186/88 supports these peripheral I/O functions. Although the 80186 provided increased functionality, it maintained compatibility with the 8086, ensuring that it could execute 8086 programs.
- After 80186/88, Intel has announced 80286, which is 16-bit processor like 8086. The 80286 was the first family member designed specifically for use as a CPU in a multi-user microcomputer. It contains many advanced modes of operations not supported by 8086. The 80286 boosted a new mode of operation-protected mode. Due to this the entire concept of memory segmentation was changed. The virtual memory management circuitry were included in the 80286, which allow an 80286 to operate in either real address mode or protected virtual address mode.
- In 1986, the next advanced processor, the 80386DX, was introduced. As expected, 80386DX is faster than any of its predecessors, with a minimum operating frequency of 16 MHz. It is an 32-bit processor with 32-bit register set, address bus and data bus.

Internal cache memory	Chip	Introduction	Data bus	Address bus	Number of instructions executed per second
-	4004	1971	4	8	50000
-	8008	1972	8	8	50000
-	8080	1974	8	16	500000
-	8085	1977	8	16	769230
-	8086/88	1978	16/8	20	2.5 million
-	80186/188	1982	16/8	20	

-	80286	1983	16	24	4.0 million
-	80386DX	1986	32	32	25 million
	80386SX	1988	16	24	25 million
8 K	80486DX (With coprocessor)	1989	32	32	50 million
8 K	80486SX (Without coprocessor)	1989	32	32	50 million

**Table 1.1.2 80X86 family tree**

During 1988, an “economy version” of the 80386, called the 80386SX was introduced by Intel. This processor had the same outside connections as the 80286, but inside it was a 386-processor supporting the 386’s expanded instruction set and various operating modes. The Table 1.1.2 shows the 80X86 family tree.

Early in 1989, Intel introduced the 80486DX, the more highly integrated microprocessor with built-in coprocessor. Meanwhile, Intel has also developed step-down version 80486SX (without coprocessor and lower clock speed).

The Pentium, introduced in 1993, was similar to the 80386 and 80486 microprocessors. It contained larger internal cache and data bus width is extended to 64-bit.

The Table 1.1.3 shows the comparison between various pentium processors.

Processor	Released year	Data bus width	Memory size	L1 cache Data-Code	L2 cache	Bus transfer speed
Pentium 60 MHz 66 MHz 120 MHz 133 MHz 233 MHz	1993	64	4 GByte	8 K - 8 K	-	60 - 66 MHz
Pentium Pro 150-166 MHz	1995	64	64 GByte	8 K - 8 K	256 K	60 - 66 MHz
Pentium II 350 MHz 400 MHz 450 MHz	1997	64	64 GByte	16 K - 16 K	512 K	100 MHz
Pentium II Xeon	1998	64	64 GByte	16 K - 16 K	512 K or 1 M	100 MHz

Pentium III 1 GHz Slot 1 version	1998	64	64 GByte	16 K - 16 K	512 K	100 MHz
Pentium III 1 GHz Flip chip version	1998	64	64 GByte	16 K - 16 K	256 K	100 MHz
Pentium III 1 GHz Celeron	1998	64	64 GByte	16 K - 16 K	256 K	66 MHz
Pentium IV 1.3 GHz 1.4 GHz 1.5 GHz	2000	64	64 GByte	16 K - 16 K	256 K	100 MHz

**Table 1.1.3 Comparison between pentium processors**

Pentium IV uses the RAMBUS memory technology in place of SDRAM technology used in other pentium processors.

### Review Questions

1. Explain in brief history of Intel Processors.
2. Give comparison between 8008, 8080 and 8085 Processors.
3. Give comparison between various Intel Processors.

## 1.2 80386DX Features

SPPU : May-10,13

The 80386 processor is available in two different versions, the 80386DX and the 80386SX. The 80386DX has 32-bit address bus and a 32-bit data bus. However, 80386SX has only 24-bit address bus and a 16-bit data bus.

1. The 80386DX is a 32-bit processor. The 32-bit ALU allows to process 32-bit data.
2. It has 32-bit address bus. So it can access up to 4 Gbyte ( $2^{32}$ ) physical memory or 64 terabyte ( $2^{46}$ ) of virtual memory.
3. The 80386DX runs with speed up to 20 MHz instructions per second.
4. The pipelined architecture of the 80386DX, allows simultaneous instruction fetching, decoding, execution and memory management. Instruction pipelining, a high bus bandwidth and on-chip address translation significantly shorten the average instruction execution time of 80386DX. These architectural design features enable the 80386DX to execute 3 to 4 million instructions per second.
5. It allows programmers to switch between different operating systems such as PC-DOS and UNIX.
6. It can operate on 17 different data types.

7. It has built-in virtual memory management circuitry and protection circuitry required to operate an 80386DX in these modes.
8. The 80386DX can operate in real mode, protected mode or a variation of protected mode called virtual 8086 mode. In real mode it functions basically as a fast 8086 or real mode 80286. The protection mode operation provides paging, virtual addressing, multilevel protection and multitasking and debugging capabilities.
9. The 80386DX microprocessor is compatible with their earlier 8086, 8088, 80186, 80188 and 80286 chips. Virtually anything that runs under these microprocessors will also run under the 80386.

### Review Question

1. List the features of 80386DX.

SPPU : May-10,13, Marks 4

### 1.3 80386DX Architecture

SPPU : May-2000,06,11,12,13, Dec.-05,11,13, Nov.-12

The Internal Architecture of 80386DX is divided into 3 sections :

- Central Processing Unit (CPU)
  - Execution unit
  - Instruction decode unit
- Memory Management Unit (MMU)
  - Segmentation unit
  - Paging unit
- Bus Control Unit

The central processing unit is further divided into execution unit and instruction unit. The Memory management unit consists of a segmentation unit and a paging unit. These units operate in parallel. Fetching, decoding, execution, memory management and bus accesses for several instructions are performed simultaneously. This parallel operation is called **pipelined instructions processing**.

#### Execution Unit

The execution unit reads the instruction from the instruction queue and executes the instructions. It consists of three subunits : Control unit, data unit and protection test unit.

**1. Control unit :** It contains microcode and special hardware. The microcode and special hardware allows 80386DX to reduce time required for execution of multiply and divide instructions. It also speeds the effective address calculation.

**2. Data unit :** The data unit contains the ALU, eight 32-bit general purpose registers and a 64-bit barrel shifter. The barrel shifter is used for multiple bit shifts in one clock.

Thus it increases the speed of all shift and rotate operations. The multiply/divide logic implements the bit-shift-rotate algorithms to complete the operations in minimum time. The entire data unit is responsible for data operations requested by the control unit.

**3. Protection test unit :** The protection test unit checks for segmentation violations under the control of the microcode.

### Instruction Decode Unit

The instruction decode unit takes instruction bytes from the code prefetch queue and translates them into microcode. The decoded instructions are then stored in the instruction queue. They are passed to the control section for deriving the necessary control signals.

### Segmentation Unit

The segmentation unit **translates logical addresses into linear addresses** at the request of the execution unit. The segmentation unit compares the effective address for the length limit specified in the segment descriptor. The segment unit adds the segment base and the effective address to generate linear address. Before calculation of linear address it also checks for access rights. It provides a 4 level protection mechanism for protecting and isolating the system code and data from those of the application program.

### Paging Unit

When the 80386DX paging mechanism is enabled, the paging unit **translates linear addresses** generated by the segmentation unit or the code prefetch unit **into physical addresses**. If paging unit is not enabled, the physical address is the same as the linear address, and no translation is necessary. The paging unit gives physical address to the Bus Interface Unit to perform memory and I/O accesses. It organizes the physical memory in terms of pages of 4 kbytes size each.

The control and attribute PLA checks the privileges at the page level. Each of the pages maintains the paging information of the task. The limit and attribute PLA checks segment limits and attributes at segment level to avoid invalid accesses to code and data in the memory segments.

### Bus Control Unit

The Bus Control Unit is the 80386DX's communication with the outside world. It provides a full 32-bit bi-directional data bus and 32-bit address bus. The bus control unit is responsible for following operations :

1. It accepts internal requests for code fetch and for data transfers from the code fetch unit and from the execution unit. It then prioritize the request with the help of prioritizer and generates signals to perform bus cycles.

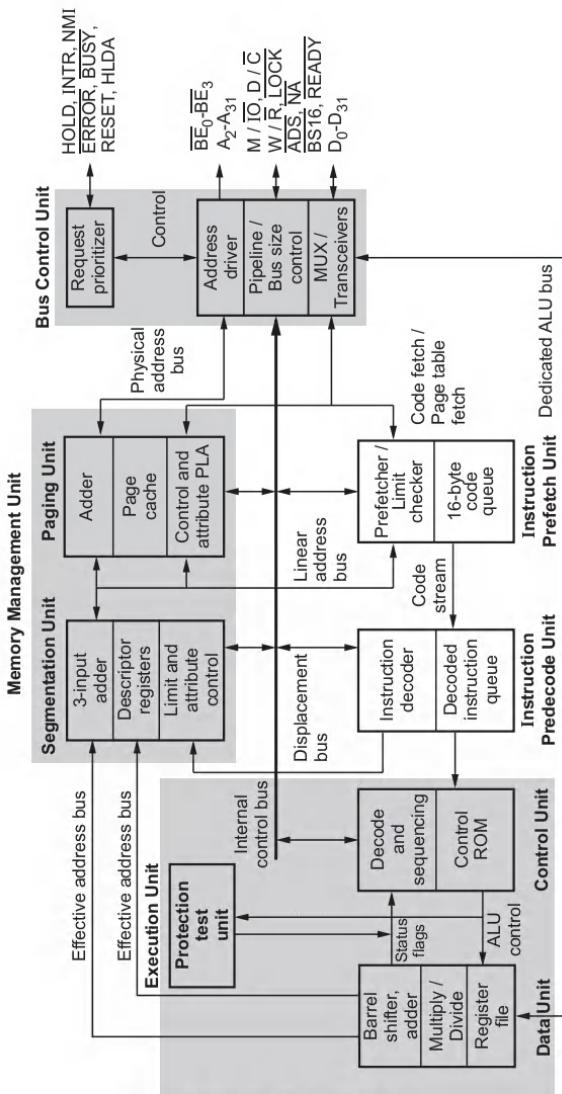


Fig. 1.3.1 80386DX architecture

2. It sends address, data and control signals to communicate with memory and I/O devices. The address driver drives the bus enable and address signal A<sub>0</sub> - A<sub>31</sub> and the transceiver interface the internal data bus with the system bus.

3. It controls the interface to external bus masters and coprocessors.
4. It also provides the address relocation facility.

### Instruction Prefetch Unit

The instruction prefetch unit fetches sequentially the instruction byte stream from the memory. It uses bus control unit to fetch instruction bytes when the bus control unit is not performing bus cycles to execute an instruction. These prefetched instruction bytes are stored in the 16-byte code queue. A 16-byte code queue holds these instructions until the decoder needs them. The prefetcher always fetches instructions in the order in which they appear in the memory. In fact, the prefetcher simply reads code one double word at a time, not caring whether it's bringing in complete instructions or pieces of two instructions with each access. When jump or call instructions are executed, the contents of the prefetched and decode queues are cleared out. In this case, prefetcher again starts filling up its queue.

### Instruction Predecode Unit

The instruction predecode unit takes instruction bytes from the instruction prefetch queue and translates them into microcode. The decoded instructions are then stored in the instruction queue.

#### Review Questions

1. Draw the functional block diagram of 80386DX and explain the main functional units.

SPPU : May-2000, 06, 11, 12, Dec.-05, Nov.-12, Marks 8, Dec.-13, Marks 6

2. Explain the function of central processing unit of 80386DX.

3. Explain the function of memory management unit of 80386DX.

4. What is the necessity of prefetch queue ?

SPPU : May-13, Marks 4

5. What is BIU in 80386 processor ? What are the functions of BIU ?

SPPU : Dec.-11, Marks 8

6. How does queue work in JUMP and CALL instruction execution ?

SPPU : May-13, Marks 4

7. Draw the functional block diagram of 80386DX and explain the main functional units.

SPPU : Dec.-13, Marks 6

### 1.4 Programmers Model

SPPU : Dec.-03, 14, 19, May-10, 13, 19

- The programming model of the 80386DX considered to be program visible because its registers are used during application programming and are specified by the instructions. Other registers, are considered to be program invisible because they

are not addressable directly during applications programming, but may be used indirectly during system programming. Some of them are used to control and operate the protected memory system.

- Fig. 1.4.1 illustrates the programming model of the 80386DX.
- It consists of
  - General purpose / Multi-purpose registers
  - Special purpose registers
  - EFLAGS register
  - Segment registers

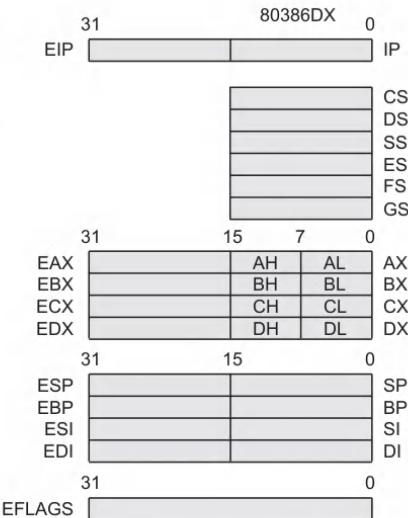


Fig. 1.4.1 80386 register set

### 1.4.1 General Purpose / Multipurpose Registers

- **EAX (accumulator)** : EAX is referenced as a 32-bit register (EAX), as a 16-bit register (AX), or as either of two 8-bit registers (AH and AL). Note that if an 8 - or 16-bit register is addressed, only that portion of the 32-bit register changes without affecting the remaining bits. The accumulator is used for instructions such as multiplication, division, and some of the adjustment instructions. For these instructions, the accumulator has a special purpose, but is generally considered to be a multipurpose register. The EAX register may also hold the offset address of a location in the memory system.
- **EBX (base index)** : EBX is addressable as EBX, BX, BH, or BL. The BX register sometimes holds the offset address of a location in the memory system. The EBX also can address memory data.
- **ECX (count)** : ECX is a general-purpose register that also holds the count for various instructions. The ECX register also can hold the offset address of memory data. Instructions that use a count are the repeated string instructions (REP/REPE/REPNE); and shift, rotate, and LOOP/LOOPD instructions. The shift and rotate instructions use CL as the count, the repeated string instructions use CX, and the LOOP/LOOPD instructions use either CX or ECX.

- **EDX (data)** : EDX is a general-purpose register that also holds a part of the result from a multiplication or part of the dividend before a division. This register can also address memory data.
- **EBP (base pointer)** : EBP points to a memory location for memory data transfers. This register is addressed as either BP or EBP.
- **EDI (destination index)** : EDI often addresses string destination data for the string instructions. It also functions as either a 32-bit (EDI) or 16-bit (DI) general-purpose register.
- **ESI (source index)** : ESI is used as either ESI or SI. The source index register often addresses source string data for the string instructions. Like EDI, ESI also functions as a general-purpose register. As a 16-bit register, it is addressed as SI; as a 32-bit register, it is addressed as ESI.

#### 1.4.2 Special - Purpose Registers

- The special-purpose registers include EIP, ESP, EFLAGS; and the segment registers CS, DS, ES, SS, FS, and GS.
- **EIP (instruction pointer)** : EIP addresses the next instruction in a section of memory defined as a code segment. This register is IP (16 bits) when the microprocessor operates in the real mode and EIP (32 bits) when the 80386 operate in the protected mode. The instruction pointer, which points to the next instruction in a program, is used by the microprocessor to find the next sequential instruction in a program located within the code segment. The instruction pointer can be modified with a jump or a call instruction.
- **ESP (stack pointer)** : ESP addresses an area of memory called the **stack**. The stack memory stores data through this pointer. This register is referred to as SP if used as a 16-bit register and ESP if referred to as a 32-bit register.

#### 1.4.3 EFLAGS

A Flag is a flip-flop which indicates some condition produced by the execution of an instruction or controls certain operations of the EU. The EFLAG register contains thirteen flags. Fig 1.4.2 shows the bit pattern of the EFLAG register.

These flags can be categorized in three different groups.

1. **Status flags** : These flags reflect the state of a particular program.
2. **Control flags** : These flags directly affect the operation of few instructions.
3. **System flags** : These flags reflect the current status of the machine and which are usually used by operating system than by application programs.

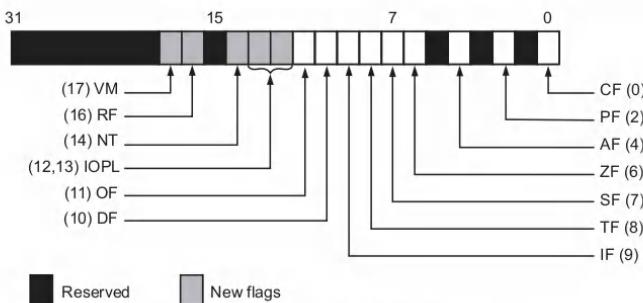


Fig. 1.4.2 Bit pattern of EFLAG register

**Status Flags :** The status flags are : CF (Carry Flag), PF (Parity Flag ), AF (Auxiliary carry Flag), ZF (Zero Flag), SF (Sign Flag), and OF (Overflow Flag). These flags indicate some condition produced by the execution of arithmetic or logical instructions. These flags provide necessary information for arithmetic and logical control decisions.

**CF (Carry flag) :** This bit is set by arithmetic instructions that generate either a carry or a borrow. This bit can also be set, cleared, or inverted with the STC, CLC or CMC instructions, respectively. Carry flag is also used in shift and rotate instructions to contain the bit shifted or rotated out of the register.

**PF (Parity flag) :** The parity bit is set by most instructions if the least significant 8-bit of the result contain even number of one's.

**AF (Auxiliary carry flag) :** This bit is set when there is a carry or borrow after a nibble addition or subtraction, respectively. The programmer can't access this bit directly, but this bit is internally used for BCD arithmetic.

**ZF (Zero flag) :** Zero flag is set to 1, if the result of an operation is zero.

**SF (Sign flag) :** The signed numbers are represented by combination of sign and magnitude. The Most Significant Bit (MSB) indicates sign of the number. For negative number MSB is 1. Sign flag is set to 1, if the result of an operation is negative (MSB = 1).

**OF (Overflow flag) :** In 2's complemented arithmetic, most significant bit is used to represent sign and remaining bits are used to represent magnitude of a number (see Fig. 1.4.3). This flag is set if the result of a signed operation is too large to fit in the number of bits available (7-bits for 8-bit number) to represent it.

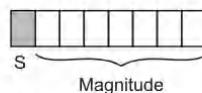


Fig. 1.4.3 Sign and magnitude representation

For example, if you add the 8-bit signed number 01110110 (+118 decimal) and the 8-bit signed number 00110110 (+54 decimal). The result will be 10101100 (+172 decimal),

which is correct binary result. But in this case, it is too large to fit in the 7-bits allowed for the magnitude in an 8-bit signed number. The overflow flag will be set after this operation to indicate that the result of the addition has overflowed into the sign bit.

### Control Flags

**DF ( Direction flag) :** The direction flag controls the direction of string operations. When the D flag is cleared these operations process strings from low memory up towards high memory. This means that offset pointers (usually SI and DI) are incremented by 1 after each operation in the string instructions when D flag is cleared. If the D flag is set, then SI and DI are decremented by 1 after each operation to process strings from high to low memory.

### System Flags

**VM (Virtual Memory) flag :** This flag indicates operating mode of 80386. When VM flag is set, 80386 switches from protected mode to virtual 8086 mode.

**R (Resume) flag/Restart flag :** This flag, when set allows selective masking of some exceptions at the time of debugging.

**NT (Nested flag) :** This flag is set when one system task invokes another task. (i.e. nested task).

**IOPL (I/O Privilege level) :** The two bits in the IOPL are used by the processor and the operating system to determine your application's access to I/O facilities. It holds privilege level, from 0 to 3, at which the current code is running in order to execute any I/O related instruction.

**IF (Interrupt Flag) :** When interrupt flag is set, the 80386 recognizes and handles external hardware interrupts on its INTR pin. If the interrupt flag is cleared, 80386 ignores any inputs on this pin. The IF flag is set and cleared with the STI and CLI instructions, respectively.

**TF (Trap Flag) :** Trap flag allows user to single-step through programs. When an 80386 detects that this flag is set, it executes one instruction and then automatically generates an internal exception 1. After servicing the exception, the processor executes the next instruction and repeats the process. This single stepping continues until program code resets this flag for debugging programs single step facility is used.

### 1.4.4 Segment Registers

- The 80386 has a 1 Mbyte address space in real mode. But all of this memory cannot be active at one time. The 80386 supports six simultaneously accessible memory blocks called **segments**.
- A segment represents an independently accessible block of memory consisting of 64K consecutive byte-wide storage locations. These segments are addressed by 16-bit registers : CS, DS, ES, SS, FS and GS. These registers are called **segment registers**, generate memory addresses when combined with other registers in the microprocessor.
- A segment register functions differently in the real mode when compared to the protected mode operation of the 80386DX.
- Fig. 1.4.4 shows the segment registers.

Bit 15	Bit 0	
	CS	Code segment
	DS	Data segment
	SS	Stack segment
	ES	Extra segment
	FS	Extra segment
	GS	Extra segment

Fig. 1.4.4 Segment registers

#### 1.4.4.1 CS (Code Segment) and CS Register

- The code segment is a section of memory that holds the code (programs and procedures) used by the 80386DX.
- The CS (Code Segment) register defines the starting address of the section of memory holding currently active code segment.
- In real mode operation, it defines the start of a 64K-byte section of memory; in protected mode, it selects a descriptor that describes the starting address and length of a section of memory holding code.
- The code segment is limited to 4G bytes in the 80386 when it operates in the protected mode.

#### 1.4.4.2 DS (Data Segment) and DS Register

- The data segment is a section of memory that contains most data used by a program. Data are accessed in the data segment by an offset address or the contents of other registers that hold the offset address.
- The DS (Data Segment) register is used to hold the address of currently active data segment.
- The data segment is limited to 4G bytes in the 80386 when it operates in the protected mode.

**1.4.4.3 ES (Extra Segment) and ES Register**

- The extra segment is an additional data segment that is used by some of the string instructions to hold destination data.
- The ES (Extra Segment) is used as general data segment register. This register holds the base addresses of memory segment.

**1.4.4.4 SS (Stack Segment) and SS Register**

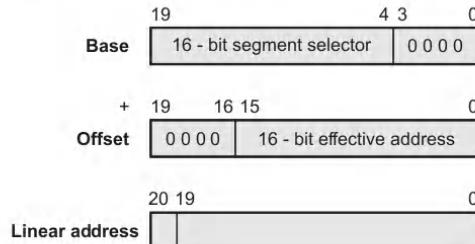
- The stack segment defines the area of memory used for the stack. The stack entry point is determined by the stack segment and stack pointer registers. The BP register also addresses data within the stack segment.

**1.4.4.5 FS and GS**

- The FS and GS segments are supplemental segment registers.
- The FS, and GS registers are used as general data segment registers. These registers hold the base addresses of two different memory segments. These segments are referred as to Extra Segments.

**1.4.5 Segments and Offsets**

- A combination of a segment address and an offset address, access a memory location in the real mode. All real mode memory addresses must consist of a segment address plus an offset address. This is illustrated in Fig. 1.4.5.



**Fig. 1.4.5 Memory addressing in real mode**

- The segment address, located within one of the segment registers, defines the beginning address of any 64K-byte memory segment. The offset address selects any location within the 64K byte memory segment. Segments in the real mode always have a length of 64K bytes.
- Table 1.4.1 and 1.4.2 show the default 16-bit and 32-bit segment and offset address combinations, respectively.

Segment	Offset	Special Purpose
CS	IP	Instruction address
SS	SP or BP	Stack address
DS	BX, DI, SI, an 8-bit number or a 16-bit number	Data address
ES	DI for string instructions	String destination address

**Table 1.4.1 Default 16-bit segment and offset address combinations**

Segment	Offset	Special Purpose
CS	EIP	Instruction address
SS	ESP or EBP	Stack address
DS	EBX, EDI, ESI, EAX, ECX, EDX an 8-bit number or a 32-bit number	Data address
ES	EDI for string instructions	String destination address
FS	No default	General address
GS	No default	General address

**Table 1.4.2 Default 32-bit segment and offset address combinations****Review Questions**

1. Draw and explain the programmer's model of 80386DX.
2. List the different registers in 80386.
3. Describe 80386 flag register with significance of each and every bit in detail. **SPPU : Dec.-03,14, May-10,13, Marks 6**
4. Explain the function of segments and segment registers of 80386.
5. Describe following different flags defined in 80386 processor
  - a) DF
  - b) VM
  - c) NT
  - d) RF**SPPU : May-19, Marks 4**
6. With the help of diagram explain 80386 applications register set. **SPPU : Dec.-19, Marks 4**

## 1.5 Operating Modes

- The operating mode of the 80386 also determines the features that are accessible. The 80386 has three operating modes :
  - Real-Address Mode
  - Protected Mode.
  - Virtual 8086 (V86) Mode
- Real-address mode (often called just "real mode") is the mode of the processor immediately after RESET. In real mode the 80386 appears to programmers as a fast 8086 with some new instructions. Most applications of the 80386 will use real mode for initialization only.
- Protected mode is the natural 32-bit environment of the 80386 processor. In this mode all instructions and features are available.
- Virtual 8086 mode (also called V86 mode) is a dynamic mode in the sense that the processor can switch repeatedly and rapidly between V86 mode and protected mode. The CPU enters V86 mode from protected mode to execute an 8086 program, then leaves V86 mode and enters protected mode to continue executing a native 80386 program.

### Review Question

1. Write a short note on operating modes of 80386DX processor.

## 1.6 Addressing Modes

SPPU : Dec.-14,17, May-18

As a part of programming flexibility, processor provides different ways to access these operands from different locations. The different ways by which processor can access data are referred to as **addressing modes**.

The 80386DX provides a total of 11 addressing modes for instructions to specify operands. These addressing modes can be categorized in three groups :

- Register operand addressing
- Immediate operand addressing
- Memory operand addressing.

### 1.6.1 Immediate Operands

Certain instructions use data from the instruction itself as operands. Such an operand is called an **immediate operand**. The operand may be 32-, 16-, or 8-bits long.

**Example :**

**For 8-bit operand :** MOV AL, 20H : This instruction copies 20H in the lower byte of EAX register.

**For 16-bit operand :** MOV AX, 1020 H : This instruction copies 1020H in the lower word of EAX register

**For 32-bit operand :** MOV EAX, 10B89C20H : This instruction copies 10B89C20H in the EAX register.

### 1.6.2 Register Operands

Operands may be located in one of the 32-bit general registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP), in one of the 16-bit general registers (AX, BX, CX, DX, SI, DI, SP, or BP), or in one of the 8-bit general registers (AH, BH, CH, DH, AL, BL, CL, or DL).

**Examples :**

**For 8-bit operand :** MOV AL, DL : This instruction copies the lower byte contents of the EDX register to the lower byte of the EAX register. Both source and destination operands are the internal registers of 80386DX.

**For 16-bit operand :** MOV AX, DX : This instruction copies the lower word contents of EDX register to the lower word of the EAX register.

**For 32-bit operand :** MOV EAX, EDX : This instruction copies the contents of EDX register to the EAX register.

### 1.6.3 Memory Operands

The remaining 9 addressing modes provide a mechanism for specifying the physical address of an operand. In 80386DX, physical address is calculated before any read or write operation.

The physical address consists of two components : The **segment base address** and an **effective address**. The effective address can be specified in a variety of ways. One way is to encode the effective address of the operand directly in the instruction. This represents direct addressing mode. The effective address can be generated with the combinations of four addressing elements : Base, Index, Scale factor and Displacement.

where,

**Base :** The contents of any general purpose register.

**Index :** The contents of any general purpose register. The index registers are used to access the elements of an array, or a string of characters.

**Scale :** The index register's value can be multiplied by a scale factor either 1, 2, 4 or 8. Scaled index mode is especially useful for accessing arrays or structures.

**Displacement :** An 8, 16 or 32-bit immediate value following the instruction.

The general formula for generating effective address is given as follows :

$$EA = \text{Base} + (\text{Index} \times \text{Scaling factor}) + \text{Displacement}$$

The Fig. 1.6.1 shows the registers that can be used to hold the values of segment base, base, and index.

Physical Address = Segment Base Address + Effective Address

$$(PA) = \text{SBA} + EA$$

$$PA = \text{SBA} : \{\text{Base} + (\text{Index} \times \text{Scale factor}) + \text{Displacement}\}$$

$$PA = \left\{ \begin{array}{l} CS \\ SS \\ DS \\ ES \\ FS \\ FS \\ GS \end{array} \right\} : \left\{ \begin{array}{l} AX \\ BX \\ CX \\ DX \\ SP \\ BP \\ SI \\ DI \end{array} \right\} + \left[ \left\{ \begin{array}{l} AX \\ BX \\ CX \\ DX \\ BP \\ SI \\ DI \end{array} \right\} \times \left\{ \begin{array}{l} 1 \\ 2 \\ 4 \\ 8 \end{array} \right\} \right] + \left\{ \begin{array}{l} 8,16 \text{ or} \\ 32\text{-bit} \\ \text{Displace-} \\ \text{ment} \end{array} \right\}$$

Fig. 1.6.1 Physical address generation

Now we see the different memory operand addressing modes :

**Direct Mode :** In this mode, the instruction is having the effective address of the operand. This effective address is used as an 8, 16 or 32 displacement from the location specified by the current value in the selected segment register is always DS.

**Example :** MOV EBX, [159DH]

$$\text{Here, } PA = DS + 159DH$$

**Register Indirect Mode :** In this mode, the base register gives the effective address of the operand.

**Example :**

$$\text{MOV EBX, [EAX]}$$

$$\text{Here, } PA = DS + EAX$$

**Based Mode :** In this mode, a base register's contents are added to a displacement to form the effective address of the operand.

**Example :** MOV EBX, [ EAX + 24]

$$\text{Here, } PA = DS + EAX + 24$$

**Index Mode :** In this mode, an index register's contents are added to a displacement to form the effective address of the operand.

**Example :**  $MOV EBX, [ SI ] + 159DH$

$$\text{Here, } PA = DS + 159DH + SI$$

**Scaled Index Mode :** In this mode, an index register's contents are multiplied by a scaling factor and then added to displacement to form the effective address of the operand.

**Example :**  $MOV EBX, 159DH + [ SI * 4 ]$

$$\text{Here, } PA = DS + 159DH + ( SI * 4 )$$

**Based Index Mode :** In this mode, the contents of a base register are added to the contents of an index register to form the effective address of the operand.

**Example :**  $MOV EBX, [ ESI ][ EAX ]$

$$\text{Here, } PA = DS + ESI + EAX$$

**Based Scaled Index Mode :** In this mode, the contents of an index register are multiplied by a scaling factor and then added to the base register to obtain the effective address of the operand.

**Example :**  $MOV EBX, [ ESI * 2 ][ EAX ]$

$$\text{Here, } PA = DS + ( ESI \times 2 ) + EAX$$

**Based Index Mode with Displacement :** In this mode, the contents of an index register and the base register and a displacement are all added together to form the effective address of the operand.

**Example :**  $MOV EBX, [ EAX ][ EDI + 24 ]$

$$\text{Here, } PA = DS + EAX + EDI + 24$$

**Based Scaled Index Mode with Displacement :** In this mode, the contents of an index register are multiplied by a scaling factor and result is then added to the contents of a base register and displacement to form the effective address of the operand.

**Example :**  $MOV EBX, [ EAX ][ ESI * 4 ] + 24$

$$\text{Here, } PA = DS + EAX + ( ESI \times 4 ) + 24$$

**Example 1.6.1** Explain where the based scaled indexed addressing with displacement will be useful.

**Solution :** In based scaled indexed addressing with displacement an, effective address is formed by adding 8 bit or 16 bit displacement with the sum of contents of any one of the Base register (BX/BP) and any of the Index register in a default segment. i.e. this addressing is useful whenever one of base register, and one of the index register and displacement is specified.

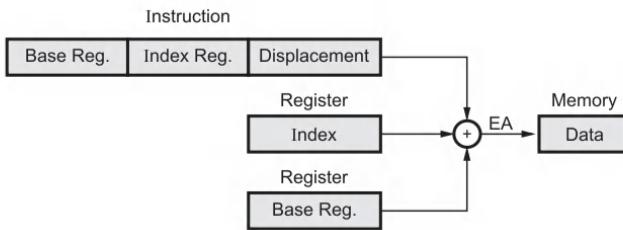


Fig. 1.6.2

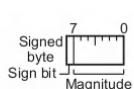
**Review Questions**

1. Discuss the following addressing modes with examples :  
i) Direct    ii) Register indirect    iii) Base plus index    iv) Immediate    v) Scaled indexed.
2. Enlist and explain any three addressing modes of 80386. **SPPU : Dec.-14, Marks 3**
3. Explain immediate and register addressing mode with an examples.

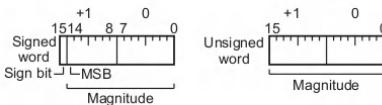
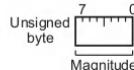
**SPPU : Dec.-17, May-18, Marks 2****1.7 Data Types****SPPU : May-14,19**

- **Bytes, words, and doublewords** are the **fundamental data types**. A byte is eight contiguous bits starting at any logical address. The bits are numbered 0 through 7; bit zero is the least significant bit.
- A word is two contiguous bytes starting at any byte address. A word thus contains 16 bits. The bits of a word are numbered from 0 through 15; bit 0 is the least significant bit. The byte containing bit 0 of the word is called the **low byte**; the byte containing bit 15 is called the **high byte**.
- A doubleword is two contiguous words starting at any byte address. A doubleword thus contains 32 bits. The bits of a doubleword are numbered from 0 through 31; bit 0 is the least significant bit. The word containing bit 0 of the doubleword is called the **low word**; the word containing bit 31 is called the **high word**.
- Although bytes, words, and doublewords are the fundamental types of operands, the 80386DX also supports additional interpretations of these operands. Depending on the instruction referring to the operand, the following additional data types are recognized by 80386DX.
  - **Bit** : A single bit quantity.
  - **Bit Field** : A group of up to 32 contiguous bits, which spans a maximum of four bytes.

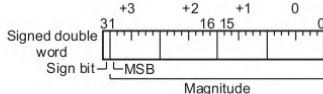
- **Bit String** : A set of contiguous bits, on the Intel386 DX bit strings can be up to 4 gigabits long.
  - **Byte** : A signed 8-bit quantity. (-128 through +127)
  - **Unsigned Byte** : An unsigned 8-bit quantity. (0 through 255)
  - **Integer (Word)** : A signed 16-bit quantity. (-32,768 through +32,767)
  - **Unsigned Integer (Word)** : An unsigned 16-bit quantity. (0 through 65535)
  - **Long Integer (Double Word)** : A signed 32-bit quantity. All operations assume a 2's complement representation. (- $2^{31}$  through  $+2^{31} - 1$ )
  - **Unsigned Long Integer (Double Word)** : An unsigned 32-bit quantity. (0 through  $2^{32} - 1$  )
  - **Signed Quad Word** : A signed 64-bit quantity.
  - **Unsigned Quad Word** : An unsigned 64-bit quantity.
  - **Offset** : A 16 - or 32-bit offset only quantity which indirectly references another memory location.
  - **Near Pointer** : A 32-bit logical address. A near pointer is an offset within a segment. Near pointers are used in either a flat or a segmented model of memory organization.
  - **Far Pointer** : A 48-bit logical address of two components : A 16-bit segment selector component and a 32-bit offset component. Far pointers are used by applications programmers only when systems designers choose a segmented memory organization.
  - **Char** : A byte representation of an ASCII Alphanumeric or control character.
  - **String** : A contiguous sequence of bytes, words or dwords. A string may contain between 1 byte and 4 Gbytes.
  - **BCD** : A byte (unpacked) representation of decimal digits 0 - 9.
  - **Packed BCD** : A byte (packed) representation of two decimal digits 0 - 9 storing one digit in each nibble.
- When the Intel386 DX is coupled with an Intel387 DX Numeric Coprocessor then the following common Floating Point types are supported.
- **Floating Point** : A signed 32-bit, 64-bit, or 80-bit real number representation. Floating point numbers are supported by the Intel387 DX numeric coprocessor.



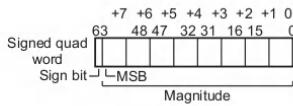
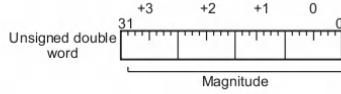
(a) Signed and unsigned byte



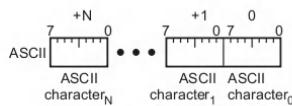
(b) Signed and unsigned word



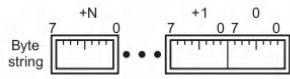
(c) Signed and unsigned double word



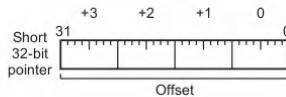
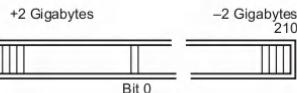
(d) Signed quad word



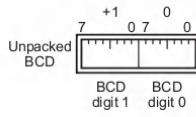
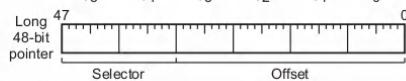
(e) ASCII char



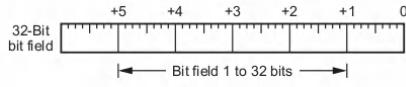
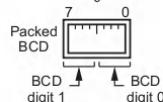
(f) Strings



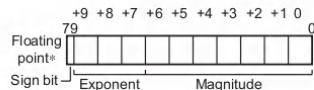
(g) Near and far pointers



(h) Packed and unpacked BCD



(i) Bit field



(j) Floating point (Supported by 80387)

Fig. 1.7.1 Data types supported by 80386DX

- Fig. 1.7.1 illustrates the data types supported by the Intel386 DX and the Intel387 DX numeric coprocessor.

**Review Question**

1. Explain the data types supported by 80386.
2. List fundamental data types of 80386.

**SPPU : May-14, Marks 5****SPPU : May-19, Marks 2**

## **UNIT - I**

# **2**

# **Applications Instruction Set**

### **Syllabus**

**Applications Instruction Set :** Data Movement Instructions, Binary Arithmetic Instructions, Decimal Arithmetic Instructions, Logical Instructions, Control Transfer Instructions, String and Character Transfer Instructions, Instructions for Block Structured Language, Flag Control Instructions, Coprocessor Interface Instructions, Segment Register Instructions, Miscellaneous Instructions.

### **Contents**

2.1	Introduction	
2.2	Instruction Set of 80386	Dec.-13,14,15,17,18,19, May-14,15,16,17,19, · · · Marks 6

## 2.1 Introduction

Instruction set of 80386 can be categorized as data movement instructions, binary arithmetic instructions, decimal arithmetic instructions, logical instructions, control transfer instructions, string and character transfer instructions, instructions for block structured language, flag control instructions, coprocessor interface instructions, segment register instructions, miscellaneous instructions.

### 2.1.1 Data Movement Instructions

These instructions provide convenient methods for moving bytes, words, or doublewords of data between memory and the registers of the base architecture. They fall into the following classes:

1. General-purpose data movement instructions : MOV and XCHG
2. Stack manipulation instructions : PUSH, POP, PUSHA and POPA
3. Type-conversion instructions : CWD, CDQ, CBW, CWDE, MOVSX and MOVZX

### 2.1.2 Binary Arithmetic Instructions

The arithmetic instructions of the 80386 processor simplify the manipulation of numeric data that is encoded in binary. Operations include the standard add, subtract, multiply, and divide as well as increment, decrement, compare, and change sign. Both signed and unsigned binary integers are supported. The binary arithmetic instructions may also be used as one step in the process of performing arithmetic on decimal integers.

1. Addition instructions : ADD, ADC, INC, AAA, and DAA.
2. Subtraction instructions SUB, SBB, DEC, AAS, DAS, CMP, and NEG
3. Comparison and Sign Change Instruction: CMP and NEG
4. Multiplication Instructions : MUL and IMUL
5. Division Instructions : DIV and IDIV

### 2.1.3 Decimal Arithmetic Instructions

The decimal arithmetic instructions are classified as :

1. Packed BCD Adjustment Instructions : DAA and DAS
2. Unpacked BCD Adjustment Instructions : AAA, AAS, AAM and AAD

### 2.1.4 Logical Instructions

The group of logical instructions includes:

1. The Boolean operation instructions : AND, OR, XOR and NOT
2. Bit test and modify instructions : BIT, BTS, BTR and BTC
3. Bit scan instructions : BSF and BSR
4. Shift instructions : SAL, SHL, SAR, SHR, SHLD and SHRD
5. Rotate instructions : ROL, ROR, RCL and RCR
6. Byte set on condition : SETcc
7. Test Instruction : TEST

### 2.1.5 Control Transfer Instructions

The 80386 provides both conditional and unconditional control transfer instructions to direct the flow of execution. Conditional control transfers depend on the results of operations that affect the flag register. Unconditional control transfers are always executed.

1. Jump Instruction : JMP
2. Call Instruction : CALL
3. Return and Return-From-Interrupt Instruction : RET and IRET
4. Unsigned Conditional Transfers : JA/JNBE, JAE/JNB, JB/JNAE, JBE/JNA, JC, JE/JZ, JNE/JNZ, JNP/JPO and JP/JPE
5. Signed Conditional Transfers : JG/JNLE, JGE/JNL, JL/JNGE, JLE/JNG, JNO, JO and JS
6. Loop Instructions : LOOP, LOOPE and LOOPNE
7. Executing a Loop or Repeat Zero Times : JCXZ
8. Software-Generated Interrupts : INT n, INTO, and BOUND

### 2.1.6 String and Character Translation Instructions

1. A set of primitive string operations : MOVS, CMPS, SCAS, LODS and STOS
2. Control flag instructions: CLD and STD
3. Repeat prefixes : REP, REPE/REPZ, REPNE/REPNZ

### 2.1.7 Instructions for Block-Structured Languages

These instructions provide machine-language support for functions normally found in high-level languages. These instructions include : ENTER and LEAVE, which simplify the programming of procedures.

### 2.1.8 Flag Control Instructions

The flag control instructions provide a method for directly changing the state of bits in the flag register.

1. Carry and Direction Flag Control Instructions : STC, CLC, CMC, CLD and STD
2. Flag Transfer Instructions : LAHF, SAHF, PUSHF and POPF

### 2.1.9 Coprocessor Interface Instructions

Coprocessor interface instructions include ESC and WAIT instructions.

### 2.1.10 Segment Register Instructions

The instructions that deal with segment registers are :

1. Segment-register transfer instructions : MOV SegReg, ..., MOV ..., SegReg, PUSH SegReg and POP SegReg
2. Control transfers to another executable segment : JMP far, CALL far and RET far
3. Data pointer instructions : LDS, LES, LFS, LGS and LSS
4. Interrupt-related instructions capable of transferring control to another segment : INT n, INTO, BOUND and IRET

### 2.1.11 Miscellaneous Instructions

1. Address Calculation Instruction : LEA
2. No-Operation Instruction : NOP
3. Translate Instruction : XLAT

## 2.2 Instruction Set of 80386

SPPU : Dec.-13,14,15,17,18,19, May-14,15,16,17,19

### AAA -- ASCII Adjust AL after Addition

Execute AAA only following an ADD instruction that leaves a byte result in the AL register. The AAA adjusts AL to contain the correct decimal digit result. If the addition produced a decimal carry, the AH register is incremented, and the carry and auxiliary carry flags are set to 1. If there was no decimal carry, the carry and auxiliary flags are

set to 0 and AH is unchanged. In either case, AL is left with its top nibble set to 0. To convert AL to an ASCII result, follow the AAA instruction with OR AL, 30H.

**Flags Affected :** AF and CF

**Exceptions :** None

**Examples :**

	;	AL	= 0011 0100 ASCII 4
	;	CL	= 0011 1000 ASCII 8
ADD AL, CL	;	AL	= 0110 1100
	;	6CH	= Incorrect temporary result
AAA	;	AL	= 0000 0010 Unpacked BCD for 2
	;	Carry	= 1 to indicate correct answer is 12 decimal.

### AAD -- ASCII Adjust AX before Division

AAD is used to prepare two unpacked BCD digits (the least-significant digit in AL, the most-significant digit in AH) for a division operation. It converts two unpacked BCD digits to equivalent binary number in AL. This is accomplished by setting AL to AL + (10 \* AH), and then setting AH to 0. AX is then equal to the binary equivalent of the original unpacked two-digit number.

**Flags Affected :** SF, ZF, and PF

**Exceptions :** None

**Examples :**

AAD	;	AX = 0403 unpacked BCD for 43 decimal, CL = 07H
	;	Adjust to binary before division,
	;	AX = 002BH = 2BH = 43 decimal.
DIV CL	;	Divide AX by unpacked BCD in CL.
	;	AL = quotient = 06 unpacked BCD
	;	AH = remainder = 01 unpacked BCD

### AAM -- ASCII Adjust AX after Multiply

After the two unpacked BCD digits are multiplied, the AAM instruction is used to adjust the product to two unpacked BCD digits in AX.

**Flags Affected :** SF, ZF, and PF

**Exceptions :** None

**Examples :**

MUL CL	;	AL = 0000 0100 = Unpacked BCD 4
	;	CL = 0000 0110 = Unpacked BCD 6
	;	AL × CL Result in AX.
	;	AX = 0000 0000 0001 1000 = 0018H

AAM ; AX = 0000 0010 0000 0100 = 0204H  
; Which is unpacked BCD for 24.

Now by adding 3030H in AX register we get the result in ASCII form.

### AAS -- ASCII Adjust AL after Subtraction

Execute AAS only after a SUB instruction that leaves the byte result in the AL register. The AAS adjusts AL so it contains the correct decimal digit result. If the subtraction produced a decimal carry, the AH register is decremented, and the carry and auxiliary carry flags are set to 1. If no decimal carry occurred, the carry and auxiliary carry flags are set to 0, and AH is unchanged. In either case, AL is left with its top nibble set to 0. To convert AL to an ASCII result, follow the AAS with OR AL, 30H.

**Flags Affected :** AF and CF

**Exceptions :** None

**Examples :**

1. ; AL = 0011 1000 ASCII 8  
; CL = 0011 0010 ASCII 2  
SUB AL, CL ; AL = 0000 0110 BCD 06  
; CF = 0
- AAS ; AL = 0000 0010 = BCD 06  
; CF = 0 no borrow required
2. ; AL = 0011 0010 ASCII 2  
; CL = 0011 1000 ASCII 8  
SUB AL, CL ; AL = 1111 1010 = FAH  
; CF = 1
- AAS ; AL = 0000 0110 = BCD 6  
; CF = 1 borrow needed means (- 6)

### ADC -- Add with Carry

ADC performs an integer addition of the two operands DEST and SRC and the carry flag, CF. The result of the addition is assigned to the first operand (DEST), and the flags are set accordingly. ADC is usually executed as part of a multi-byte or multi-word addition operation. The source may be an immediate number, a register, or a memory location. The destination may be a register, or a memory location. The source and destinations in an instruction cannot both be memory locations. The source and destination both must be a doubleword, word or byte. When an immediate byte value is added to a word or doubleword operand, the immediate value is first sign-extended to the size of the word or doubleword operand.

**Flags Affected :** OF, SF, ZF, AF, CF, and PF

#### Exceptions :

Real Mode	Interrupt 13 is generated if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Protected Mode	GP(0) - General protection error is generated if the result is in a nonwritable segment. GP(0) - General protection exception is generated if an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments. SS(0) - Stack fault exception is generated if an illegal address in the SS segment. PF - Page fault condition can also occur.
Virtual 8086 Mode	Same exceptions as in Real Address Mode, PF - Page fault condition can also occur.

#### Examples :

ADC DL, CL	; Add contents of CL to contents of DL with carry ; and store result in DL i.e. DL $\leftarrow$ DL + CL + CY
ADC DX, BX	; Add contents of BX to contents of DX with carry ; and store result in DX i.e. DX $\leftarrow$ DX + BX + CY
ADC EDX, EBX	; Add contents of EBX to the contents of EDX with carry and store ; result in EDX

### ADD -- Add

The instruction is similar to ADC. Only two operands are added.

#### Examples :

ADD AL, 0F0H	; Add immediate number 0F0H to contents of AL.
ADD CL, TOTAL [BX]	; Add byte from effective address ; TOTAL [BX] to contents of CL
ADD CX, TOTAL [BX]	; Add word from effective address ; TOTAL [BX] to contents of CX.
ADD ECX, EAX	; Add contents of ECX to the contents of EAX and store ; result in ECX

### AND -- Logical AND

Each bit of the result of the AND instruction is a 1 if both corresponding bits of the operands are 1; otherwise, it becomes a 0. The destination may be a register, or a memory location. The source and destinations in an instruction cannot both be memory locations. The source and destination both must be a doubleword, word or byte.

**Flags Affected :** CF = 0, OF = 0; PF, SF, and ZF

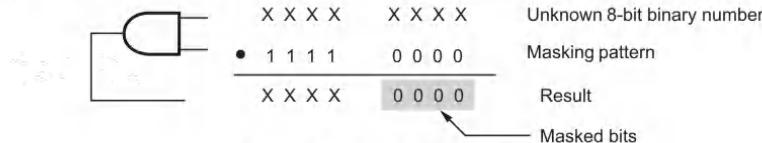
**Exceptions :**

<b>Real Mode</b>	<b>Interrupt 13</b> is generated if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Protected Mode</b>	<b>GP(0)</b> - General protection error is generated if the result is in a nonwritable segment. <b>GP(0)</b> - General protection exception is generated if an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments. <b>SS(0)</b> - Stack fault exception is generated if an illegal address in the SS segment. <b>PF</b> - Page fault condition can also occur.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, <b>PF</b> - Page fault condition can also occur.

**Examples :**

1. AND BL, AL ; AL = 1001 0011 = 93H  
                  ; BL = 0111 0101 = 75H  
     AND BL, AL ; AND byte in AL with byte in BL  
                  ; BL = 0001 0001 = 11H
2. AND CX, 00F0H ; CX = 0110 1011 1001 1110  
                  ; CX = 0000 0000 1001 0000
- AND EDX, EBX ; AND doubleword in EBX with doubleword in EDX and store  
                  ; result in EDX

The AND operation clears bits of a binary number. The task of clearing a bit in a binary number is called **masking**. The Fig. 2.2.1 shows the process of masking.



**Fig. 2.2.1 Masking using AND operation**

### ARPL -- Adjust RPL Field of Selector

The ARPL instruction has two operands. The first operand is a 16-bit memory variable or word register that contains the value of a selector. The second operand is a word register. If the RPL field ("requested privilege level"--bottom two bits) of the first operand is less than the RPL field of the second operand, the zero flag is set to 1 and the RPL field of the first operand is increased to match the second operand. Otherwise, the zero flag is set to 0 and no change is made to the first operand.

ARPL prevents operating system software from accessing subroutines of a more privilege than the caller.

**Flags Affected :** ZF

**Exceptions :**

Real Mode	Interrupt 6 - ARPL is not recognized in Real Address Mode.
Protected Mode	GP(0) - General protection error is generated if the result is in a nonwritable segment. GP(0) - General protection exception is generated if an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments. SS(0) - Stack fault exception is generated if an illegal address in the SS segment. PF - Page fault condition can also occur.
Virtual 8086 Mode	Same exceptions as in Real Address Mode, PF - Page fault condition can also occur.

**Example :**

ARPL memory\_word, BX

### BOUND -- Check Array Index against Bounds

BOUND ensures that a signed array index is within the limits specified by a block of memory consisting of an upper and a lower bound. Each bound uses one word for an operand-size attribute of 16 bits and a doubleword for an operand-size attribute of 32 bits.

**Flags Affected :** None

**Exceptions :**

Real Mode	Interrupt 5 if the bounds test fails. Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. Interrupt 6 if the second operand is a register.
Protected Mode	Interrupt 5 if the bounds test fails. GP(0) - General protection exception is generated if an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments. SS(0) - Stack fault exception is generated if an illegal address in the SS segment. PF - Page fault condition can also occur.
Virtual 8086 Mode	Same exceptions as in Real Address Mode, PF - Page fault condition can also occur.

**Example**

BOUND AX, mem\_word  
BOUND AX, mem\_limits

**BSF -- Bit Scan Forward**

BSF scans the bits in the second word or doubleword operand starting with bit 0. The ZF flag is cleared if the bits are all 0; otherwise, the ZF flag is set and the destination register is loaded with the bit index of the first set bit.

**Flags Affected :** ZF

**Exceptions :**

<b>Real Mode</b>	Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Protected Mode</b>	GP(0) - General protection exception is generated if an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments. SS(0) - Stack fault exception is generated if an illegal address in the SS segment. PF - Page fault condition can also occur.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, PF - Page fault condition can also occur.

**Example**

```
BSF CX, mem_word
BSF ECX, EBX
```

**BSR -- Bit Scan Reverse**

BSR scans the bits in the second word or doubleword operand from the most significant bit to the least significant bit. The ZF flag is cleared if the bits are all 0; otherwise, ZF is set and the destination register is loaded with the bit index of the first set bit found when scanning in the reverse direction.

**Flags Affected :** ZF

**Exceptions :**

<b>Real Mode</b>	Interrupt 13 is generated if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Protected Mode</b>	GP(0) - General protection error is generated if the result is in a nonwritable segment. GP(0) - General protection exception is generated if an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments. SS(0) - Stack fault exception is generated if an illegal address in the SS segment. PF - Page fault condition can also occur.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, PF - Page fault condition can also occur.

**Example**

```
BSR BX, BX
BSR ECX, mem_Dword
```

**BT -- Bit Test**

BT saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag.

**Flags Affected :** CF

**Exceptions :**

Real Mode	Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Protected Mode	GP(0) - General protection exception is generated if an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments. SS(0) - Stack fault exception is generated if an illegal address in the SS segment. PF - Page fault condition can also occur.
Virtual 8086 Mode	Same exceptions as in Real Address Mode, PF - Page fault condition can also occur.

**Examples**

BT BX, CX  
BT EAX, ECX

**BTC -- Bit Test and Complement**

BTC saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag and then complements the bit.

**Flags Affected :** CF

**Exceptions :**

Real Mode	Interrupt 13 is generated if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Protected Mode	GP(0) - General protection error is generated if the result is in a nonwritable segment. GP(0) - General protection exception is generated if an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments. SS(0) - Stack fault exception is generated if an illegal address in the SS segment. PF - Page fault condition can also occur.
Virtual 8086 Mode	Same exceptions as in Real Address Mode, PF - Page fault condition can also occur.

**Examples**

BTC CX, DX  
BTC memory, BX

**BTR -- Bit Test and Reset**

BTR saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag and then stores 0 in the bit.

**Flags Affected :** CF

**Exceptions :**

<b>Real Mode</b>	Interrupt 13 is generated if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Protected Mode</b>	GP(0) - General protection error is generated if the result is in a nonwritable segment. GP(0) - General protection exception is generated if an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments. SS(0) - Stack fault exception is generated if an illegal address in the SS segment. PF - Page fault condition can also occur.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, PF - Page fault condition can also occur.

**Examples**

BTR memory, CX

BTR ECX, EDX

**BTS -- Bit Test and Set**

BTS saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag and then stores 1 in the bit.

**Flags Affected :** CF

**Exceptions :**

<b>Real Mode</b>	Interrupt 13 is generated if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Protected Mode</b>	GP(0) - General protection error is generated if the result is in a nonwritable segment. GP(0) - General protection exception is generated if an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments. SS(0) - Stack fault exception is generated if an illegal address in the SS segment. PF - Page fault condition can also occur.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, PF - Page fault condition can also occur.

**Examples**

BTS ECX, EDX

BTS memory, ECX

## CALL -- Call Procedure

The CALL instruction causes the procedure named in the operand to be executed. When the procedure is complete (a return instruction is executed within the procedure), execution continues at the instruction that follows the CALL instruction.

The action of the different forms of the instruction are described below.

**Near Call** : A call to a procedure which is in the same code segment. The offset of the instruction following CALL is pushed onto the stack. It will be popped by a near RET instruction within the procedure. The CS register is not changed by this form of CALL.

**Far Call** : A call to a procedure which is not in the same code segment. These forms of the instruction push both CS and IP or EIP as a return address.

A call instruction followed by procedure name is known as **direct call** and a call instruction followed by register name is known as **indirect call**. In case of indirect call, register contents are used as an offset of the first instruction of the procedure.

In Protected Mode, both long pointer forms consult the AR byte in the descriptor indexed by the selector part of the long pointer. Depending on the value of the AR byte, the call will perform one of the following types of control transfers :

- A far call to the same protection level
- An inter-protection level far call
- A task switch

**Flags Affected** : All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

### Protected Mode Exceptions

For far calls	GP, NP, SS, and TS
For near direct calls	GP(0) if procedure location is beyond the code segment limits; SS(0) if pushing the return address exceeds the bounds of the stack segment; PF (fault-code) - for a page fault.
For a near indirect call	GP(0) for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments; SS(0) for an illegal address in the SS segment. GP(0) if the indirect offset obtained is beyond the code segment limits. PF(fault-code) for a page fault.
Real Address Mode Exceptions	Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode Exceptions	Same exceptions as in Real Address Mode; PF(fault-code) for a page fault.

**Examples**

CALL Multiply ;	Direct call
CALL CX ;	Indirect call

**CBW/CWDE -- Convert Byte to Word/Convert Word to Doubleword**

CBW converts the signed byte in AL to a signed word in AX by extending the most significant bit of AL (the sign bit) into all of the bits of AH. CWDE converts the signed word in AX to a doubleword in EAX by extending the most significant bit of AX into the two most significant bytes of EAX.

**Flags Affected :** None

**Exceptions :** None

**Example**

```
; AX = 0000 0000 1001 1010
CBW      ; convert signed byte in AL to signed word in AX
          ; Result : AX = 1111 1111 1001 1010
```

**CLC -- Clear Carry Flag**

CLC sets the carry flag to zero.

**Flags Affected :** CF = 0

**Exceptions :** None

**CLD -- Clear Direction Flag**

CLD clears the direction flag.

**Flags Affected :** DF = 0

**Exceptions :** None

**CLI -- Clear Interrupt Flag**

CLI clears the interrupt flag if the current privilege level is at least as privileged as IOPL.

**Flags Affected :** IF = 0

**Exceptions :**

Protected Mode	GP(0) if the current privilege level is greater (has less privilege) than the IOPL in the flags register. IOPL specifies the least privileged level at which I/O can be performed.
Real Address Mode	None.

Virtual 8086 Mode	GP(0) as for Protected Mode.
-------------------	------------------------------

### CLTS -- Clear Task-Switched Flag in CR0

CLTS clears the task-switched (TS) flag in register CR0. This flag is set by the 80386 every time a task switch occurs. It is a privileged instruction that can only be executed at privilege level 0.

**Flags Affected :** TS = 0 (TS is in CR0, not the flag register)

**Exceptions :**

Protected Mode	GP(0) - if CLTS is executed with a current privilege level other than 0.
Real Address Mode	None (valid in Real Address Mode to allow initialization for Protected Mode).
Virtual 8086 Mode	Same exceptions as in Real Address Mode.

### CMC -- Complement Carry Flag

CMC reverses the setting of the carry flag.

**Flags Affected :** CF

**Exceptions :** None

### CMP --Compare Two Operands

CMP subtracts the second operand from the first but, unlike the SUB instruction, does not store the result; only the flags are changed.

**Flags Affected :** OF, SF, ZF, AF, PF, and CF

**Exceptions :**

Protected Mode	GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments. SS(0) - for an illegal address in the SS segment. PF-(fault-code) for a page fault.
Real Address Mode	Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in Real Address Mode. PF(fault-code) for a page fault.

**Examples**

CMP BL, 01H	; Compare immediate number 01H with byte in BL.
CMP CX, BX	; Compare word in BX with word in CX.
CMP CX, TOTAL	; Compare word at displacement TOTAL in DS with word in CX.
CMP EBX, 12345678H	; Compare immediate Dword with EBX

**CMPS/CMPSB/CMPSW/CMPSD -- Compare String Operands**

CMPS compares the byte, word, or doubleword pointed to by the source-index register with the byte, word, or doubleword pointed to by the destination-index register.

If the address-size attribute of this instruction is 16 bits, SI and DI will be used for source- and destination-index registers; otherwise ESI and EDI will be used. The comparison is done by subtracting the operand indexed by the destination-index register from the operand indexed by the source-index register. The result of the subtraction is not stored; only the flags reflect the change.

After the comparison is made, both the source-index register and destination-index register are automatically advanced. If the direction flag is 0 (CLD was executed), the registers increment; if the direction flag is 1 (STD was executed), the registers decrement. The registers increment or decrement by 1 if a byte is compared, by 2 if a word is compared, or by 4 if a doubleword is compared.

CMPSB, CMPSW and CMPSD are synonyms for the byte, word, and doubleword CMPS instructions, respectively.

**Flags Affected :** OF, SF, ZF, AF, PF and CF

**Exceptions :**

<b>Protected Mode</b>	GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, SS(0) - for an illegal address in the SS segment. PF(fault-code) for a page fault.
<b>Real Address Mode</b>	<b>Interrupt 13</b> if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode. PF (fault-code) for a page fault.

**Examples**

CMPSB	;	Compare bytes
CMPSD	;	Compare Dwords

**CWD/CDQ – Convert Word to Doubleword/Convert Doubleword to Quadword**

CWD converts the signed word in AX to a signed doubleword in DX:AX by extending the most significant bit of AX into all the bits of DX. CDQ converts the signed doubleword in EAX to a signed 64-bit integer in the register pair EDX:EAX by extending the most significant bit of EAX (the sign bit) into all the bits of EDX.

**Flags Affected :** None

**Exceptions :** None

#### Example

```
; DX = 0000 0000 0000 0000
; AX = 1001 0000 1001 0001
CWD      ; Convert signed word in AX to signed
          ; doubleword in DX : AX
          ; Result :   DX = 1111 1111 1111 1111
                      AX = 1001 0000 1001 0001
```

### DAA -- Decimal Adjust AL after Addition

Execute DAA only after executing an ADD instruction that leaves a two-BCD-digit byte result in the AL register. The ADD operands should consist of two packed BCD digits. The DAA instruction adjusts AL to contain the correct two-digit packed decimal result.

Instruction works as follows :

1. If the value of the low-order four bits (D-D) in the AL is greater than 9 or if AF is set, the instruction adds 6 (06) to the low-order four bits.
2. If the value of the high-order four bits (D - D) in the AL is greater than 9 or if carry flag is set, the instruction adds 6 (60) to the high-order four bits.

**Flags Affected :** AF, CF, SF, ZF and PF

**Exceptions :** None

#### Examples :

1.	; AL = 0011 1001 = 39 BCD ; CL = 0001 0010 = 12 BCD Add AL, CL DAA ;
	; AL = 0100 1011 = 4BH ; Add 0110 Because 1011 > 9 ; AL = 0101 0001 = 51 BCD
2.	; AL = 1001 0110 = 96 BCD ; BL = 0000 0111 = 07 BCD ADD AL, BL DAA ;
	; AL = 1001 1101 = 9DH ; Add 0110 Because 1101 > 9 ; AL = 1010 0011 = A3H ; 1010 > 9 so add 0110 0000 ; AL = 0000 0011 = 03 BCD, CF = 1. The result is 103.

### DAS -- Decimal Adjust AL after Subtraction

Execute DAS only after a subtraction instruction that leaves a two-BCD-digit byte result in the AL register. The operands should consist of two packed BCD digits. DAS adjusts AL to contain the correct packed two-digit decimal result.

Instruction works as follows :

1. If the value of the low-order four bits (D-D) in the AL is greater than 9 or if AF is set; the instruction subtracts 6 (06) from the low-order four bits.
2. If the value of the high-order four bits (D-D) in the AL is greater than 9 or if carry flag is set, the instruction subtracts 6 (60) from the high-order four bits.

**Flags Affected :** AF, CF, SF, ZF and PF

**Exceptions :** None

**Examples**

1. ; AL = 0011 0010 = 32 BCD  
SUB AL, CL ; CL = 0001 0111 = 17 BCD  
; AL = 0001 1011 = 1BH  
; Subtract 0110 because 1011 > 9  
; AL = 0001 0101 = 15 BCD
2. ; AL = 0010 0011 = 23 BCD  
SUB AL, CL ; CL = 0101 1000 = 58 BCD  
; AL = 1100 1011 = CBH CF = 1  
; Subtract 0110 (6) because 1011 > 9  
; AL = 1100 0101 = C5H  
; Subtract 0110 0000 because 1100 > 9  
; AL = 0110 0101 = 65 BCD CF = 1,  
; CF = 1 means borrow  
; is needed means number is negative (- 65).

## DEC -- Decrement by 1

DEC subtracts 1 from the operand.

**Flags Affected :** OF, SF, ZF, AF, and PF

**Exceptions :**

Protected Mode	GP(0) - if the result is a nonwritable segment. GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments. SS(0) - for an illegal address in the SS segment. PF - (fault-code) for a page fault.
Real Address Mode	Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in Real Address Mode. PF(fault-code) for a page fault.

**DIV -- Unsigned Divide**

DIV performs an unsigned division. The dividend is implicit; only the divisor is given as an operand. The remainder is always less than the divisor. The type of the divisor determines which registers to use as follows :

Size	Dividend	Divisor	Quotient	Remainder
byte	AX	r/m8	AL	AH
word	DX:AX	r/m16	AX	DX
dword	EDX:EAX	r/m32	EAX	EDX

**Flags :** OF, SF, ZF, AR, PF, CF are undefined.

**Exceptions :**

Protected Mode	Interrupt 0 if the quotient is too large to fit in the designated register (AL, AX, or EAX), or if the divisor is 0. GP(0) - for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments. SS(0) - for an illegal address in the SS segment. PF-(fault-code) for a page fault.
Real Address Mode	Interrupt 0 if the quotient is too big to fit in the designated register (AL, AX, or EAX), or if the divisor is 0. Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in Real Address Mode. PF-(fault-code) for a page fault.

**Examples**

DIV CL ; Word in AX/byte in CL, Quotient in AL, remainder in AH.

DIV CX ; Double word in DX and AX/word in CX, Quotient in AX, remainder in DX.

**ENTER -- Make Stack Frame for Procedure Parameters**

ENTER creates the stack frame required by most block-structured high-level languages. The first operand specifies the number of bytes of dynamic storage allocated on the stack for the routine being entered. The second operand gives the lexical nesting level (0 to 31) of the routine within the high-level language source code. It determines the number of stack frame pointers copied into the new stack frame from the preceding frame. BP (or EBP, if the operand-size attribute is 32 bits) is the current stack frame pointer.

**Flags Affected :** None

Protected Mode Exceptions	SS(0)- if SP or ESP would exceed the stack limit at any point during instruction execution. PF- (fault-code) for a page fault.
---------------------------	--

Real Address Mode : None	Virtual 8086 Mode : None
--------------------------	--------------------------

**Examples**

ENTER 0B12H, 0 ; Creates procedure stack frame

ENTER 0534H, 1 ; Creates stack frame for procedure parameter

**ESC (Escape)** is a 5-bit sequence (11011) that begins the opcodes that identify floating point numerical instructions. The ESC tells the 80386 to send the opcode and addresses of operands to the numeric coprocessor.

**HLT -- Halt**

HALT stops instruction execution and places the 80386 in a HALT state.

**Flags Affected :** None

**Exceptions :**

Protected Mode	GP(0)- if the current privilege level is not 0.
Real Address Mode : None	Virtual 8086 Mode : None

**IDIV -- Signed Divide**

IDIV performs a signed division. The dividend, quotient, and remainder are implicitly allocated to fixed registers. Only the divisor is given as an explicit r/m operand. The type of the divisor determines which registers to use as follows :

Size	Divisor	Quotient	Remainder	Dividend
byte	r/m8	AL	AH	AX
word	r/m16	AX	DX	DX:AX
dword	r/m32	EAX	EDX	EDX:EAX

**Flags Affected :** OF, SF, ZF, AR, PF, CF are undefined.

**Exceptions :**

Protected Mode	Interrupt 0 if the quotient is too large to fit in the designated register (AL or AX), or if the divisor is 0. GP(0) - for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments. SS(0) - for an illegal address in the SS segment; PF-(fault-code) for a page fault.
----------------	---

Real Address Mode	<b>Interrupt 0</b> if the quotient is too large to fit in the designated register (AL or AX), or if the divisor is 0. <b>Interrupt 13</b> if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH
Virtual 8086 Mode	Same exceptions as in Real Address Mode. PF-(fault-code) for a page fault.

### IMUL -- Signed Multiply

IMUL performs signed multiplication.

**Flags Affected :** OF and CF; SF, ZF, AF, and PF are undefined.

**Exceptions :**

Protected Mode	GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments. SS(0) - for an illegal address in the SS segment. PF-(fault-code) for a page fault.
Real Address Mode	<b>Interrupt 13</b> if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in Real Address Mode. PF-(fault-code) for a page fault.

**Examples**

IMUL BL	; AL × BL, result in AX
IMUL CX	; AX × CX, high-order word of result in DX and ; low-order word of result in AX.

### IN -- Input from Port

IN transfers a data byte or data word from the port numbered by the second operand into the register (AL, AX, or EAX) specified by the first operand. Access any port from 0 to 65535 by placing the port number in the DX register and using an IN instruction with DX as the second parameter. These I/O instructions can be shortened by using an 8-bit port I/O in the instruction. The upper eight bits of the port address will be 0 when 8-bit port I/O is used.

**Flags Affected :** None

**Exceptions :**

Protected Mode	GP(0) - if the current privilege level is larger (has less privilege) than IOPL and any of the corresponding I/O permission bits in TSS equals 1.
Real Address Mode	None.
Virtual 8086 Mode	GP(0) - fault if any of the corresponding I/O permission bits in TSS equals 1.

**Examples :**

IN AL, 0F8H	; Copy a byte from port 0F8H to AL.
IN AX, 95H	; Copy a word from port 95H to AX.
MOV DX, 30F8H	; Load 16-bit address of the port in DX.
IN AL, DX	; Copy a byte from 8-bit port 30F8H to AL.
IN AX, DX	; Copy a word from 16-bit port 30F8H to AX.
IN EAX, DX	; Copy a Dword from 32-bit part to EAX.

**INC -- Increment by 1**

INC adds 1 to the operand.

**Flags Affected :** OF, SF, ZF, AF and PF

**Exceptions :**

<b>Protected Mode</b>	GP(0) - if the operand is in a nonwritable segment. GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments. SS(0) - for an illegal address in the SS segment. PF-(fault-code) for a page fault.
<b>Real Address Mode</b>	<b>Interrupt 13</b> if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode. PF-(fault-code) for a page fault.

**Examples**

INC AL	; Add 1 to contents of AL.
INC EBX	; Add 1 to contents of EBX.
INC BYTE PTR [BX]	; Increment byte at offset of BX in DS. ; BYTE PTR directive indicates to the assembler ; that the <b>byte from memory</b> is to be incremented.
INC WORD PTR [BX]	; Increment word at offset of BX in DS. ; WORD PTR directive indicates to the assembler ; that the <b>word from memory</b> is to be incremented.

**INS/INSB/INSW/INSD – Input from Port to String**

INS transfers data from the input port numbered by the DX register to the memory byte or word at ES:dest-index. The memory operand must be addressable from ES; no segment override is possible. The destination register is DI if the address-size attribute of the instruction is 16 bits, or EDI if the address-size attribute is 32 bits.

INS does not allow the specification of the port number as an immediate value. The port must be addressed through the DX register value.

The destination address is determined by the contents of the destination index register. Load the correct index into the destination index register before executing INS.

After the transfer is made, DI or EDI advances automatically. If the direction flag is 0 (CLD was executed), DI or EDI increments; if the direction flag is 1 (STD was executed), DI or EDI decrements. DI increments or decrements by 1 if a byte is input, by 2 if a word is input, or by 4 if a doubleword is input.

INSB, INSW and INSD are synonyms of the byte, word, and doubleword INS instructions.

**Flags Affected :** None

**Exceptions :**

Protected Mode	GP(0) - if CPL is numerically greater than IOPL and any of the corresponding I/O permission bits in TSS equals 1. GP(0) - if the destination is in a nonwritable segment. GP(0)- for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments; SS(0) - for an illegal address in the SS segment. PF - (fault-code) for a page fault.
Real Address Mode	Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	GP(0) - fault if any of the corresponding I/O permission bits in TSS equals 1; PF-(fault-code) for a page fault.

### INT/INTO -- Call to Interrupt Procedure

The INT instruction generates via software a call to an interrupt handler. The immediate operand, from 0 to 255, gives the index number into the Interrupt Descriptor Table (IDT) of the interrupt routine to be called. In Protected Mode, the IDT consists of an array of eight-byte descriptors; the descriptor for the interrupt invoked must indicate an interrupt, trap, or task gate. In Real Address Mode, the IDT is an array of four byte-long pointers. In Protected and Real Address Modes, the base linear address of the IDT is defined by the contents of the IDTR.

The INTO conditional software instruction is identical to the INT interrupt instruction except that the interrupt number is implicitly 4, and the interrupt is made only if the 80386 overflow flag is set.

In Real Address Mode, INT n pushes the flags, CS, and the return IP onto the stack, in that order, then jumps to the long pointer indexed by the interrupt number.

**Flags Affected :** None

**Exceptions :**

Protected Mode	GP, NP, SS, and TS as indicated under "Operation" above.
Real Address Mode	None.

<b>Virtual 8086 Mode</b>	GP(0) - fault if IOPL is less than 3, for INT only, to permit emulation. Interrupt 3 (0CCH) generates Interrupt 3. INTO generates <b>Interrupt 4</b> if the overflow flag equals 1.
--------------------------	--

### IRET/IRET D -- Interrupt Return

In Real Address Mode, IRET pops the instruction pointer, CS, and the flags register from the stack and resumes the interrupted routine.

In Protected Mode, the action of IRET depends on the setting of the nested task flag (NT) bit in the flag register. When popping the new flag image from the stack, the IOPL bits in the flag register are changed only when CPL equals 0.

If NT equals 0, IRET returns from an interrupt procedure without a task switch. The code returned to must be equally or less privileged than the interrupt routine (as indicated by the RPL bits of the CS selector popped from the stack). If the destination code is less privileged, IRET also pops the stack pointer and SS from the stack.

If NT equals 1, IRET reverses the operation of a CALL or INT that caused a task switch. The updated state of the task executing IRET is saved in its task state segment. If the task is reentered later, the code that follows IRET is executed.

**Flags Affected :** All; the flags register is popped from stack.

**Exceptions :**

<b>Protected Mode</b>	GP, NP, or SS.
<b>Real Address Mode</b>	<b>Interrupt 13</b> if any part of the operand being popped lies beyond address 0FFFFH.
<b>Virtual 8086 Mode</b>	GP(0) fault if IOPL is less than 3, to permit emulation.

### Jcc --Jump if Condition is Met

Instruction	Description	Condition for Jump
JA	Jump short if above	(CF=0 and ZF=0)
JAE	Jump short if above or equal	(CF=0)
JB	Jump short if below	(CF=1)
JBE	Jump short if below or equal	(CF=1 or ZF=1)
JC	Jump short if carry	(CF=1)
JCXZ	Jump short if CX register is 0	
JECXZ	Jump short if ECX register is 0	

JE/JZ	Jump short if equal	(ZF=1)
JG	Jump short if greater	(ZF=0 and SF=OF)
JGE	Jump short if greater or equal	(SF=OF)
JL	Jump short if less	(SF<OF)
JLE	Jump short if less or equal	(ZF=1 and SF<OF)
JNA	Jump short if not above	(CF=1 or ZF=1)
JNAE	Jump short if not above or equal	(CF=1)
JNB	Jump short if not below	(CF=0)
JNBE	Jump short if not below or equal	(CF=0 and ZF=0)
JNC	Jump short if not carry	(CF=0)
JNE	Jump short if not equal	(ZF=0)
JNG	Jump short if not greater	(ZF=1 or SF<OF)
JNGE	Jump short if not greater or equal	(SF<OF)
JNL	Jump short if not less	(SF>OF)
JNLE	Jump short if not less or equal	(ZF=0 and SF=OF)
JNO	Jump short if not overflow	(OF=0)
JNP	Jump short if not parity	(PF=0)
JNS	Jump short if not sign	(SF=0)
JNZ	Jump short if not zero	(ZF=0)
JO	Jump short if overflow	(OF=1)
JP	Jump short if parity	(PF=1)
JPE	Jump short if parity even	(PF=1)
JPO	Jump short if parity odd	(PF=0)
JS	Jump short if sign	(SF=1)

Conditional jumps (except JCXZ) test the flags which have been set by a previous instruction. The terms "less" and "greater" are used for comparisons of signed integers; "above" and "below" are used for unsigned integers. If the given condition is true, a jump is made to the location provided as the operand.

JCXZ differs from other conditional jumps because it tests the contents of the CX or ECX register for 0, not the flags.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	GP(0) - if the offset jumped to is beyond the limits of the code segment
<b>Real Address Mode : None</b>	<b>Virtual 8086 Mode : None</b>

**JMP -- Jump**

The JMP instruction transfers control to a different point in the instruction stream without recording return information.

**Near Jump (Intra - segment)** : Jump within same code segment.

It does not involve changing the segment register value.

**Far Jump (Inter - segment)** : Jump which is not in same code segment. Intra - segment changes the segment register along with offset.

An intra - segment direct jump uses the offset byte following the instruction. On the other hand, intra-segment indirect jump uses the contents of the location addressed by the bytes following the instruction byte.

The JMP ptr16:16 and ptr16:32 forms of the instruction use a four-byte or six-byte operand as a long pointer to the destination. The JMP and forms fetch the long pointer from the memory location specified (indirection). In Real Address Mode or Virtual 8086 Mode, the long pointer provides 16 bits for the CS register and 16 or 32 bits for the EIP register (depending on the operand-size attribute). In Protected Mode, both long pointer forms consult the Access Rights (AR) byte in the descriptor indexed by the selector part of the long pointer.

**Flags Affected** : All if a task switch takes place; none if no task switch occurs

**Exceptions :**

<b>Protected Mode</b>	<b>Far jumps</b> : GP, NP, SS, and TS <b>Near direct jumps</b> : GP(0) if procedure location is beyond the code segment limits. <b>Near indirect jumps</b> : GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments SS(0) - for an illegal address in the SS segment. GP - if the indirect offset obtained is beyond the code segment limits. PF-(fault-code) for a page fault.
<b>Real Address Mode</b>	<b>Interrupt 13</b> if any part of the operand would be outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as under Real Address Mode. PF-(fault-code) for a page fault.

### LAHF -- Load Flags into AH Register

LAHF transfers the low byte of the flags word to AH. The bits, from MSB to LSB, are sign, zero, indeterminate, auxiliary, carry, indeterminate, parity, indeterminate, and carry.

**Flags Affected :** None

**Exceptions :** None

### LAR -- Load Access Rights Byte

The LAR instruction stores a masked form of the second doubleword of the descriptor for the source selector if the selector is visible at the CPL (modified by the selector's RPL) and is a valid descriptor type. The destination register is loaded with the high-order doubleword of the descriptor masked by 00FxFF00, and ZF is set to 1. The x indicates that the four bits corresponding to the upper four bits of the limit are undefined in the value loaded by LAR. If the selector is invisible or of the wrong type, ZF is cleared.

**Flags Affected :** ZF

**Exceptions :**

Protected Mode	GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments. SS(0) - for an illegal address in the SS segment. PF -(fault-code) for a page fault.
Real Address Mode	Interrupt 6 - LAR is unrecognized in Real Address Mode.
Virtual 8086 Mode	Same exceptions as in Real Address Mode.

#### Examples

LAR BX, mem\_word

LAR DX, EAX

### LEA -- Load Effective Address

LEA calculates the effective address (offset part) and stores it in the specified register. The operand-size attribute of the instruction is determined by the chosen register. The address-size attribute is determined by the USE attribute of the segment containing the second operand. The address-size and operand-size attributes affect the action performed by LEA, as follows :

Operand Size	Address Size	Action Performed
16	16	16-bit effective address is calculated and stored in requested 16-bit register destination.
16	32	32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination.
32	16	16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.
32	32	32-bit effective address is calculated and stored in the requested 32-bit register destination.

**Flags Affected :** None

**Exceptions :**

Protected Mode	UD - if the second operand is a register.
Real Address Mode	Interrupt 6 if the second operand is a register.
Virtual 8086 Mode	Same exceptions as in Real Address Mode.

**Examples**

LEA CX, TOTAL	; Load CX with offset of TOTAL in DS.
LEA BP, SS : STACK_TOP	; Load BP with offset of STACK_TOP in SS.
LEA AX, [BX] [DI]	; Load AX with EA = [BX] + [DI]

### LEAVE -- High Level Procedure Exit

LEAVE reverses the actions of the ENTER instruction. By copying the frame pointer to the stack pointer, LEAVE releases the stack space used by a procedure for its local variables. The old frame pointer is popped into BP or EBP, restoring the caller's frame. A subsequent RET instruction removes any arguments pushed onto the stack of the exiting procedure.

**Flags Affected :** None

**Exceptions :**

Protected Mode	SS(0) - if BP does not point to a location within the limits of the current stack segment.
Real Address Mode	Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in Real Address Mode.

## LGDT/LIDT -- Load Global/Interrupt Descriptor Table Register

The LGDT and LIDT instructions load a linear base address and limit value from a six-byte data operand in memory into the GDTR or IDTR, respectively. If a 16-bit operand is used with LGDT or LIDT, the register is loaded with a 16-bit limit and a 24-bit base, and the high-order eight bits of the six-byte data operand are not used. If a 32-bit operand is used, a 16-bit limit and a 32-bit base is loaded; the high-order eight bits of the six-byte operand are used as high-order base address bits.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	<b>GP(0)</b> - if the current privilege level is not 0, <b>UD</b> - if the source operand is a register, <b>GP(0)</b> - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments. <b>SS(0)</b> - for an illegal address in the SS segment. <b>PF</b> - (fault-code) for a page fault.
<b>Real Address Mode</b>	<b>Interrupt 13</b> if any part of the operand would lie outside of the effective address space from '0' to <b>0FFFFH</b> , <b>Interrupt 6</b> if the source operand is a register.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, <b>PF</b> - (fault-code) for a page fault.

## LGS/LSS/LDS/LES/LFS -- Load Full Pointer

These instructions read a full pointer from memory and store it in the selected segment register:register pair. The full pointer loads 16 bits into the segment register SS, DS, ES, FS, or GS. The other register loads 32 bits if the operand-size attribute is 32 bits, or loads 16 bits if the operand-size attribute is 16 bits.

Instruction	Segment Register Loaded
LGS	GS
LSS	SS
LDS	DS
LES	ES
LFS	FS

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, SS(0) - for an illegal address in the SS segment, the second operand must be a memory operand, not a register. GP(0) - if a null selector is loaded into SS, PF - (fault-code) for a page fault.
<b>Real Address Mode</b>	The second operand must be a memory operand, not a register, Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to OFFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, PF -(fault-code) for a page fault.

**Examples**

LSS SI, myword ; Loads SS : reg 16 with memory pointer  
 LFS EBX, myword ; Loads FS : reg 32 with memory pointer

**LLDT -- Load Local Descriptor Table Register**

LLDT loads the Local Descriptor Table Register (LDTR). The word operand (memory or register) to LLDT should contain a selector to the Global Descriptor Table (GDT). The GDT entry should be a Local Descriptor Table. If so, then the LDTR is loaded from the entry.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	GP(0) - if the current privilege level is not 0, GP - (selector) - if the selector operand does not point into the Global Descriptor Table, or if the entry in the GDT is not a Local Descriptor Table, NP(selector) - if the LDT descriptor is not present, GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, SS(0) - for an illegal address in the SS segment, PF-(fault-code) for a page fault.
<b>Real Address Mode</b>	Interrupt 6 - LLDT is not recognized in Real Address Mode.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode.

**LMSW -- Load Machine Status Word**

LMSW loads the machine status word (part of CR0) from the source operand. This instruction can be used to switch to Protected Mode; if so, it must be followed by an intrasegment jump to flush the instruction queue. LMSW will not switch back to Real Address Mode.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	GP(0) - if the current privilege level is not 0, GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, SS(0) - for an illegal address in the SS segment, PF- (fault-code) for a page fault.
<b>Real Address Mode</b>	<b>Interrupt 13</b> if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, PF-(fault-code) for a page fault.

**Example**

LMSW BX ; Loads MSW with contents of BX

**LOCK -- Assert LOCK# Signal Prefix**

The LOCK prefix causes the LOCK signal of the 80386 to be asserted during execution of the instruction that follows it. In a multiprocessor environment, this signal can be used to ensure that the 80386 has exclusive use of any shared memory while LOCK is asserted.

The LOCK prefix functions only with the following instructions :

BT, BTS, BTR, BTC	memory, register/immediate
XCHG	register, memory
XCHG	memory, register
ADD, OR, ADC, SBB, AND, SUB, XOR	memory, register/immediate
NOT, NEG, INC, DEC	memory

XCHG always asserts LOCK regardless of the presence or absence of the LOCK prefix.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	UD - if LOCK is used with an instruction for which LOCK prefix is not allowed other exceptions can be generated by the subsequent (locked) instruction.
<b>Real Address Mode</b>	<b>Interrupt 6</b> if LOCK is used with an instruction for which LOCK prefix is not allowed exceptions can still be generated by the subsequent (locked) instruction.
<b>Virtual 8086 Mode</b>	UD - if LOCK is used with an instruction for which LOCK prefix is not allowed exceptions can still be generated by the subsequent (locked) instruction.

**LODS/LODSB/LODSW/LODSD -- Load String Operand**

LODS loads the AL, AX, or EAX register with the memory byte, word, or doubleword at the location pointed to by the source-index register. After the transfer is made, the source-index register is automatically advanced. If the direction flag is 0 (CLD was executed), the source index increments; if the direction flag is 1 (STD was executed), it decrements. The increment or decrement is 1 if a byte is loaded, 2 if a word is loaded, or 4 if a doubleword is loaded.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault.
<b>Real Address Mode</b>	Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, PF(fault-code) - for a page fault.

**LOOP/LOOPcond – Loop Control with CX Counter**

LOOP decrements the count register without changing any of the flags. Conditions are then checked for the form of LOOP being used. If the conditions are met, a short jump is made to the label given by the operand to LOOP. If the address-size attribute is 16 bits, the CX register is used as the count register; otherwise the ECX register is used. The operand of LOOP must be in the range from 128 (decimal) bytes before the instruction to 127 bytes ahead of the instruction.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	GP(0) - if the offset jumped to is beyond the limits of the current code segment.
<b>Real Address Mode</b> : None	<b>Virtual 8086 Mode</b> : None

**LSL -- Load Segment Limit**

The LSL instruction loads a register with a segment limit, and sets ZF to 1, provided that the source selector is visible at the CPL and that the descriptor is a type accepted by LSL. Otherwise, ZF is cleared to 0. The segment limit is loaded as a byte granular value. If the descriptor has a page granular segment limit, LSL will translate it to a byte

limit before loading it in the destination register (shift left 12 the 20-bit "raw" limit from descriptor, then OR with 0000FFFF).

**Flags Affected :** ZF as described above

**Exceptions :**

Protected Mode	GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault.
Real Address Mode	Interrupt 6 - LSL is not recognized in Real Address Mode.
Virtual 8086 Mode	Same exceptions as in Real Address Mode.

### LTR -- Load Task Register

LTR loads the task register from the source register or memory location specified by the operand. The loaded task state segment is marked busy. A task switch does not occur.

**Flags Affected :** None

**Exceptions :**

Protected Mode	GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, SS(0) - for an illegal address in the SS segment, GP(0) - if the current privilege level is not 0, GP(selector) if the object named by the source selector is not a TSS or is already busy, NP(selector) - if the TSS is marked "not present", PF(fault-code)- for a page fault.
Real Address Mode	Interrupt 6 - LTR is not recognized in Real Address Mode.
Virtual 8086 Mode	Same exceptions as in Real Address Mode.

### MOV -- Move Data

MOV copies the second operand to the first operand. If the destination operand is a segment register (DS, ES, SS, etc.), then data from a descriptor is also loaded into the register. A MOV into SS inhibits all interrupts until after the execution of the next instruction.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	GP, SS, and NP - if a segment register is being loaded; otherwise, GP(0) - if the destination is in a nonwritable segment; GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, SS(0) - for an illegal address in the SS segment, PF(fault-code) for a page fault.
<b>Real Address Mode</b>	Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, PF(fault-code) - for a page fault.

**Examples**

MOV BX, 592FH ; Load the immediate number 592FH in BX  
 MOV DS, CX ; Copy word from CX register to data segment register.

**MOV -- Move to/from Special Registers**

The above forms of MOV store or load the following special registers in or from a general purpose register :

- Control registers CR0, CR2, and CR3
- Debug Registers DR0, DR1, DR2, DR3, DR6, and DR7
- Test Registers TR6 and TR7

**Flags Affected :** OF, SF, ZF, AF, PF, and CF are undefined

**Exceptions :**

<b>Protected Mode</b>	GP(0) - if the current privilege level is not 0.
<b>Real Address Mode</b>	None.
<b>Virtual 8086 Mode</b>	GP(0) - if instruction execution is attempted.

**Examples**

MOV EAX, CR0 ; Moves control register into EAX register  
 MOV EBX, D0 ; Moves debug register 0 into EBX register  
 MOV CR2, ECX ; Moves ECX into control register 2

**MOVS/MOVSB/MOVSW/MOVSD -- Move Data from String to String**

MOVS copies the byte or word at [(E)SI] to the byte or word at ES:[(E)DI]. The destination operand must be addressable from the ES register; no segment override is possible for the destination. A segment override can be used for the source operand; the default is DS.

The addresses of the source and destination are determined solely by the contents of (E)SI and (E)DI. Load the correct index values into (E)SI and (E)DI before executing the MOVS instruction. MOVS, MOVSW, and MOVSD are synonyms for the byte, word, and doubleword MOVS instructions.

After the data is moved, both (E)SI and (E)DI are advanced automatically. If the direction flag is 0 (CLD was executed), the registers are incremented; if the direction flag is 1 (STD was executed), the registers are decremented. The registers are incremented or decremented by 1 if a byte was moved, 2 if a word was moved, or 4 if a doubleword was moved.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	GP(0) - if the result is in a nonwritable segment, GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault.
<b>Real Address Mode</b>	Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, PF(fault-code) - for a page fault.

### MOVsx -- Move with Sign-Extend

MOVsx reads the contents of the effective address or register as a byte or a word, sign-extends the value to the operand-size attribute of the instruction (16 or 32 bits), and stores the result in the destination register.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS or GS segments, SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault.
<b>Real Address Mode</b>	Interrupt 13 - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, PF(fault-code) - for a page fault.

### Example

MOVsx AX, CL ; Moves sign extended contents of CL in AX

### MOVZX -- Move with Zero-Extend

MOVZX reads the contents of the effective address or register as a byte or a word, zero extends the value to the operand-size attribute of the instruction (16 or 32 bits), and stores the result in the destination register.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	<b>GP(0)</b> - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, <b>SS(0)</b> - for an illegal address in the SS segment, <b>PF(fault-code)</b> - for a page fault.
<b>Real Address Mode</b>	<b>Interrupt 13</b> - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, <b>PF(fault-code)</b> - for a page fault.

**Example**

MOVZX AX, CL ; Moves zero extended contents of CL into AX

### MUL -- Unsigned Multiplication of AL or AX

MUL performs unsigned multiplication. Its actions depend on the size of its operand, as follows :

- A byte operand is multiplied by AL; the result is left in AX. The carry and overflow flags are set to 0 if AH is 0; otherwise, they are set to 1.
- A word operand is multiplied by AX; the result is left in DX:AX. DX contains the high-order 16 bits of the product. The carry and overflow flags are set to 0 if DX is 0; otherwise, they are set to 1.
- A doubleword operand is multiplied by EAX and the result is left in EDX:EAX. EDX contains the high-order 32 bits of the product. The carry and overflow flags are set to 0 if EDX is 0; otherwise, they are set to 1.

**Flags Affected :** OF and CF as described above; SF, ZF, AF, PF, and CF are undefined

**Exceptions :**

<b>Protected Mode</b>	<b>GP(0)</b> - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, <b>SS(0)</b> - for an illegal address in the SS segment, <b>PF(fault-code)</b> - for a page fault.
<b>Real Address Mode</b>	<b>Interrupt 13</b> - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, <b>PF(fault-code)</b> - for a page fault.



**Examples**

MUL BL

; AL  $\times$  BL, result in AX.

MUL BX

; AX  $\times$  BX, result high word in DX low word in AX.**NEG -- Two's Complement Negation**

NEG replaces the value of a register or memory operand with its two's complement. The operand is subtracted from zero and the result is placed in the operand. The carry flag is set to 1, unless the operand is zero, in which case the carry flag is cleared to 0.

**Flags Affected :** CF, OF, SF, ZF and PF

**Exceptions :**

Real Mode	GP(0) - if the result is in a nonwritable segment, GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS or GS segments; SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault.
Protected Mode	Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in real-address mode, PF(fault-code) - for a page fault.

**NOP -- No Operation**

NOP performs no operation.

**Flags Affected :** None

**Exceptions :** None**NOT -- One's Complement Negation**

NOT inverts the operand; every 1 becomes a 0 and vice versa.

**Flags Affected :** None

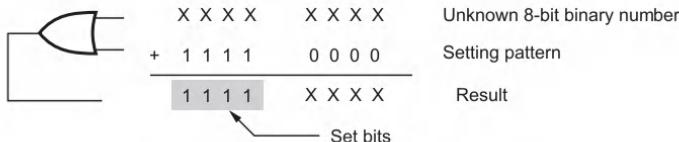
**Exceptions :**

Protected Mode	GP(0) - if the result is in a nonwritable segment, GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS or GS segments, SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault.
Real Address Mode	Interrupt 13 - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in real-address mode; PF(fault-code) - for a page fault.

## OR -- Logical Inclusive OR

OR computes the inclusive OR of its two operands and places the result in the first operand. Each bit of the result is 0 if both corresponding bits of the operands are 0; otherwise, each bit is 1.

The OR instruction is used to set (make one) any bit in the binary number. This is illustrated in Fig. 2.2.2.



**Fig. 2.2.2 Setting bit/s using OR operation**

**Flags Affected :** OF = 0, CF = 0, SF, ZF, and PF is undefined

**Exceptions :**

Protected Mode	GP(0) - if the result is in a nonwritable segment, GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS or GS segments; SS(0) - for an illegal address in the SS segment; PF(fault-code) - for a page fault.
Real Address Mode	Interrupt 13 - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in real-address mode, PF(fault-code) - for a page fault.

### Examples

1. ; AL = 1001 0011 = 93H  
; BL = 0111 0101 = 75H  
OR BL, AL  
; OR byte in AL with byte in BL.  
; BL = 1111 0111 = F7H
2. ; CX = 0110 1011 1001 1110  
OR CX, 00F0H  
; CX = 0110 1011 1111 1110

## OUT -- Output to Port

OUT transfers a data byte or data word from the register (AL, AX or EAX) given as the second operand to the output port numbered by the first operand. Output to any port from 0 to 65535 is performed by placing the port number in the DX register and then using an OUT instruction with DX as the first operand. If the instruction contains an eight-bit port ID, that value is zero-extended to 16 bits.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	<b>GP(0)</b> - if the current privilege level is higher (has less privilege) than IOPL and any of the corresponding I/O permission bits in TSS equals 1.
<b>Real Address Mode</b>	None
<b>Virtual 8086 Mode</b>	<b>GP(0)</b> - fault if any of the corresponding I/O permission bits in TSS equals 1.

**Examples**

OUT 0F8H, AL	; Copy contents of AL to 8-bit port 0F8H.
OUT 0FBH, AX	; Copy contents of AX to 16-bit port 0FBH.
MOV DX, 30F8H	; Load 16-bit address of the port in DX.
OUT DX, AL	; Copy the contents of AL to port 30F8H.
OUT DX, AX	; Copy the contents of AX to port 30F8H.

**OUTS/OUTSB/OUTSW/OUTSD -- Output String to Port**

OUTS transfers data from the memory byte, word or doubleword at the source-index register to the output port addressed by the DX register. If the address-size attribute for this instruction is 16 bits, SI is used for the source-index register; otherwise, the address-size attribute is 32 bits and ESI is used for the source-index register.

OUTS does not allow specification of the port number as an immediate value. The port must be addressed through the DX register value. Load the correct value into DX before executing the OUTS instruction.

The address of the source data is determined by the contents of source-index register. Load the correct index value into SI or ESI before executing the OUTS instruction.

After the transfer, source-index register is advanced automatically. If the direction flag is 0 (CLD was executed), the source-index register is incremented; if the direction flag is 1 (STD was executed), it is decremented. The amount of the increment or decrement is 1 if a byte is output, 2 if a word is output or 4 if a doubleword is output. OUTSB, OUTSW and OUTSD are synonyms for the byte, word and doubleword OUTS instructions.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	<b>GP(0)</b> - if CPL is greater than IOPL and any of the corresponding I/O permission bits in TSS equals 1, <b>GP(0)</b> - for an illegal memory operand with an effective address in the CS, DS or ES segments, <b>SS(0)</b> - for an illegal address in the SS segment, <b>PF(fault-code)</b> - for a page fault.
<b>Real Address Mode</b>	<b>Interrupt 13</b> - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

<b>Virtual 8086 Mode</b>	GP(0) - fault if any of the corresponding I/O permission bits in TSS equals 1, PF(fault-code) - for a page fault.
--------------------------	--

### POP -- Pop a Word from the Stack

POP replaces the previous contents of the memory, the register or the segment register operand with the word on the top of the 80386 stack, addressed by SS:SP (address-size attribute of 16 bits) or SS:ESP (address size attribute of 32 bits). The stack pointer SP is incremented by 2 for an operand-size of 16 bits or by 4 for an operand-size of 32 bits. It then points to the new top of stack.

POP CS is not an 80386 instruction. Popping from the stack into the CS register is accomplished with a RET instruction.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	GP, SS and NP - if a segment register is being loaded; SS(0) - if the current top of stack is not within the stack segment, GP(0) - if the result is in a nonwritable segment, GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS or GS segments, SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault.
<b>Real Address Mode</b>	Interrupt 13 - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in real-address mode; PF(fault-code) - for a page fault.

### POPA/POPAD -- Pop all General Registers

POPA pops the eight 16-bit general registers. However, the SP value is discarded instead of loaded into SP. POPA reverses a previous PUSHAD, restoring the general registers to their values before PUSHAD was executed. The first register popped is DI.

POPAD pops the eight 32-bit general registers. The ESP value is discarded instead of loaded into ESP. POPAD reverses the previous PUSHAD, restoring the general registers to their values before PUSHAD was executed. The first register popped is EDI.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	SS(0) - if the starting or ending stack address is not within the stack segment, PF(fault-code) - for a page fault.
<b>Real Address Mode</b>	Interrupt 13 - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in real-address mode, PF(fault-code) - for a page fault.

## POPF/POPFD -- Pop Stack into FLAGS or EFLAGS Register

POPF/POPFD pops the word or doubleword on the top of the stack and stores the value in the flags register. If the operand-size attribute of the instruction is 16 bits, then a word is popped and the value is stored in FLAGS. If the operand-size attribute is 32 bits, then a doubleword is popped and the value is stored in EFLAGS.

**Flags Affected :** All flags except VM and RF

**Exceptions :**

Protected Mode	SS(0) - if the top of stack is not within the stack segment.
Real Address Mode	Interrupt 13 - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	GP(0) - fault if IOPL is less than 3, to permit emulation.

## PUSH -- Push Operand onto the Stack

PUSH decrements the stack pointer by 2 if the operand-size attribute of the instruction is 16 bits; otherwise, it decrements the stack pointer by 4. PUSH then places the operand on the new top of stack, which is pointed to by the stack pointer.

The 80386 PUSH ESP instruction pushes the value of ESP as it existed before the instruction. This differs from the 8086, where PUSH SP pushes the new value (decremented by 2).

**Flags Affected :** None

**Exceptions**

Protected Mode	SS(0) - if the new value of SP or ESP is outside the stack segment limit, GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault.
Real Address Mode	None; if SP or ESP is 1, the 80386 shuts down due to a lack of stack space.
Virtual 8086 Mode	Same exceptions as in real-address mode, PF(fault-code) - for a page fault.

### Example

PUSH EAX ; Pushes EAX

### PUSHA/PUSHAD – Push all General Registers

PUSHA and PUSHAD save the 16-bit or 32-bit general registers, respectively, on the 80386 stack. PUSHA decrements the stack pointer (SP) by 16 to hold the eight word values. PUSHAD decrements the stack pointer (ESP) by 32 to hold the eight doubleword values.

**Flags Affected :** None

**Exceptions :**

Protected Mode	SS(0) - if the starting or ending stack address is outside the stack segment limit, PF(fault-code) - for a page fault.
Real Address Mode	Before executing PUSHA or PUSHAD, the 80386 shuts down if SP or ESP equals 1, 3, or 5; if SP or ESP equals 7, 9, 11, 13, or 15, exception 13 occurs.
Virtual 8086 Mode	Same exceptions as in real-address mode, PF(fault-code) - for a page fault.

### PUSHF/PUSHFD -- Push Flags Register onto the Stack

PUSHF decrements the stack pointer by 2 and copies the FLAGS register to the new top of stack; PUSHFD decrements the stack pointer by 4, and the 80386 EFLAGS register is copied to the new top of stack which is pointed to by SS:ESP.

**Flags Affected :** None

**Exceptions :**

Protected Mode	SS(0) - if the new value of ESP is outside the stack segment boundaries
Real Address Mode	None
Virtual 8086 Mode	GP(0) - fault if IOPL is less than 3, to permit emulation

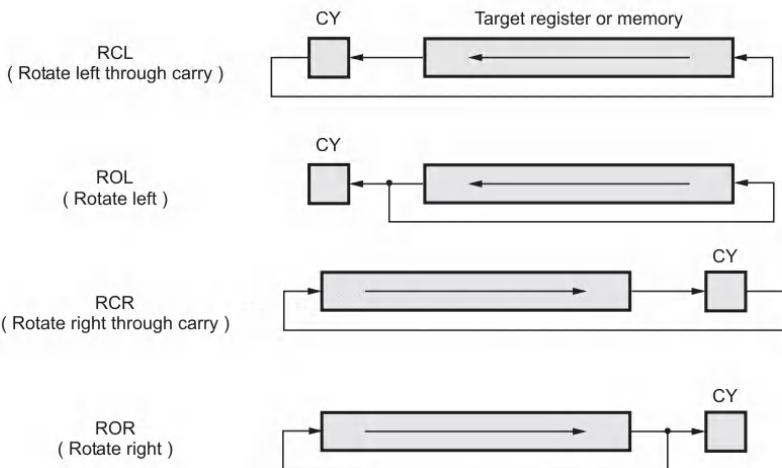
### RCL/RCR/ROL/ROR – Rotate

Each rotate instruction shifts the bits of the register or memory operand given. The ROL (rotate left) instructions shift all the bits left, except for the MSB, which is returned to the LSB. The ROR (rotate right) instructions do the reverse.

For the RCL and RCR instructions, the carry flag is part of the rotated quantity. RCL shifts the carry flag into the LSB and shifts the MSB into the carry flag; RCR shifts the carry flag into the MSB and shifts the LSB into the carry flag. For the ROL and ROR instructions, the original value of the carry flag is not a part of the result, but the carry flag receives a copy of the bit that was shifted from one end to the other.

The rotate is repeated the number of times indicated by the second operand, which is either an immediate number or the contents of the CL register. To reduce the maximum instruction execution time, the 80386 does not allow rotation counts greater than 31. If a rotation count greater than 31 is attempted, only the bottom five bits of the rotation are used. The 8086 does not mask rotation counts. The 80386 in Virtual 8086 Mode does mask rotation counts.

The overflow flag is defined only for the single-rotate forms of the instructions (second operand = 1). It is undefined in all other cases. For left shifts/rotates, the CF bit after the shift is XORed with the high-order result bit. For right shifts/rotates, the high-order two bits of the result are XORed to get OF.



**Fig. 2.2.3 Rotate operations**

**Flags Affected :** OF (only for single rotates) and CF

**Exceptions :**

Protected Mode	GP(0) - if the result is in a nonwritable segment, GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, SS(0) - for an illegal address in the SS segment; PF(fault-code) - for a page fault.
Real Address Mode	Interrupt 13 - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in Real Address Mode, PF(fault-code) - for a page fault.

**Examples**

ROL CX, 1	; Word in CX one bit position left, MSB to LSB and CF
MOV CL, 03H	; Load number of bits to rotate in CL.
ROR BL, CL	; Rotate BL right three positions.

**REP/REPE/REPZ/REPNE/REPNZ -- Repeat Following String Operation**

REP, REPE (repeat while equal) and REPNE (repeat while not equal) are prefix that are applied to string operation. Each prefix cause the string instruction that follows to be repeated the number of times indicated in the count register or (for REPE and REPNE) until the indicated condition in the zero flag is no longer met.

Synonymous forms of REPE and REPNE are REPZ and REPNZ, respectively.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct.

**Flags Affected :** ZF by REP CMPS and REP SCAS as described above

**Exceptions :**

<b>Protected Mode</b>	UD - if a repeat prefix is used before an instruction that is not in the list above; further exceptions can be generated when the string operation is executed; refer to the descriptions of the string instructions themselves.
<b>Real Address Mode</b>	<b>Interrupt 6</b> - if a repeat prefix is used before an instruction that is not in the list above; further exceptions can be generated when the string operation is executed; refer to the descriptions of the string instructions themselves.
<b>Virtual 8086 Mode</b>	UD - if a repeat prefix is used before an instruction that is not in the list above; further exceptions can be generated when the string operation is executed; refer to the descriptions of the string instructions themselves.

**Example**

REPZ CMPSB	; Compare string bytes until CX = 0
	; or until string bytes not equal.

**RET -- Return from Procedure**

RET transfers control to a return address located on the stack. The address is usually placed on the stack by a CALL instruction and the return is made to the instruction that follows the CALL.

For the intrasegment (near) return, the address on the stack is a segment offset, which is popped into the instruction pointer. The CS register is unchanged. For the intersegment (far) return, the address on the stack is a long pointer. The offset is popped first, followed by the selector.

In real mode, CS and IP are loaded directly. In Protected Mode, an intersegment return causes the processor to check the descriptor addressed by the return selector. The AR byte of the descriptor must indicate a code segment of equal or lesser privilege (or greater or equal numeric value) than the current privilege level. Returns to a lesser privilege level cause the stack to be reloaded from the value saved beyond the parameter block.

**Flags Affected :** None

**Exceptions :**

Protected Mode	GP, NP, or SS, PF(fault-code) - for a page fault.
Real Address Mode	Interrupt 13 - if any part of the operand would be outside the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in Real Address Mode, PF(fault-code) - for a page fault.

### SAHF -- Store AH into Flags

SAHF loads the flags listed above with values from the AH register, from bits 7, 6, 4, 2, and 0, respectively.

**Flags Affected :** SF, ZF, AF, PF and CF

**Exceptions :** None

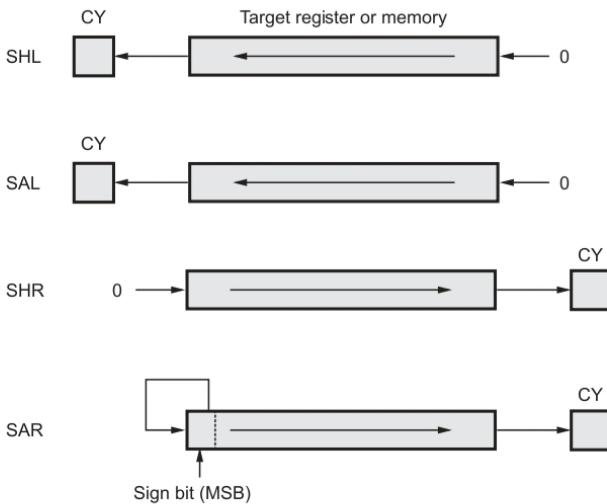
### SAL/SAR/SHL/SHR -- Shift Instructions

SAL (or its synonym, SHL) shifts the bits of the operand towards left. The MSB is shifted into the carry flag, and the LSB is set to 0.

SAR and SHR shift the bits of the operand towards right. The LSB is shifted into the carry flag. The effect is to divide the operand by 2.

The shift is repeated the number of times indicated by the second operand, which is either an immediate number or the contents of the CL register. To reduce the maximum execution time, the 80386 does not allow shift counts greater than 31. If a shift count greater than 31 is attempted, only the bottom five bits of the shift count are used. (The 8086 uses all eight bits of the shift count.)

The overflow flag is set only if the single-shift forms of the instructions are used. For left shifts, OF is set to 0 if the MSB of the answer is the same as the result of the carry flag (i.e., the top two bits of the original operand were the same); OF is set to 1 if they are different. For SAR, OF is set to 0 for all single shifts. For SHR, OF is set to the MSB of the original operand.

**Fig. 2.2.4 Shift operations**

**Flags Affected :** OF for single shifts; OF is undefined for multiple shifts; CF, ZF, PF, and SF.

#### Exceptions :

<b>Protected Mode</b>	GP(0) - if the result is in a nonwritable segment, GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS or GS segments; SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault.
<b>Real Address Mode</b>	Interrupt 13 - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, PF(fault-code) - for a page fault.

#### Examples

```

SAL CX, 1           ; Shift word in CX 1 bit position left, 0 in LSB
MOV CL, 05H         ; Load desired number of shifts in CL
SAL AX, CL          ; Shift word in AX left 5 times 0s in 5 least-significant bits.

```

### SBB -- Integer Subtraction with Borrow

SBB adds the second operand (DEST) to the carry flag (CF) and subtracts the result from the first operand (SRC). The result of the subtraction is assigned to the first operand (DEST), and the flags are set accordingly.

When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended.

**Flags Affected :** OF, SF, ZF, AF, PF and CF

**Exceptions :**

Protected Mode	GP(0) - if the result is in a nonwritable segment, GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, SS(0) for an illegal address in the SS segment, PF(fault-code) - for a page fault.
Real Address Mode	Interrupt 13 - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in Real Address Mode, PF(fault-code) - for a page fault.

**Example**

SBB DL, CL	; Subtract contents of CL and status of carry flag ; from the contents of DL and store result in DL. ; i.e. DL ← DL – CL – CY
SBB DX, BX	; Subtract contents of BX and status of carry ; flag from the DX and store result in DX. ; i.e. DX ← DX – BX – CY

### SCAS/SCASB/SCASW/SCASD -- Compare String Data

SCAS subtracts the memory byte or word at the destination register from the AL, AX or EAX register. The result is discarded; only the flags are set. The operand must be addressable from the ES segment; no segment override is possible.

If the address-size attribute for this instruction is 16 bits, DI is used as the destination register; otherwise, the address-size attribute is 32 bits and EDI is used.

The address of the memory data being compared is determined solely by the contents of the destination register, not by the operand to SCAS. The operand validates ES segment addressability and determines the data type. Load the correct index value into DI or EDI before executing SCAS.

After the comparison is made, the destination register is automatically updated. If the direction flag is 0 (CLD was executed), the destination register is incremented; if the direction flag is 1 (STD was executed), it is decremented. The increments or decrements are by 1 if bytes are compared, by 2 if words are compared, or by 4 if doublewords are compared.

SCASB, SCASW and SCASD are synonyms for the byte, word and doubleword SCAS instructions that don't require operands. They are simpler to code, but provide no type or segment checking.

**Flags Affected :** OF, SF, ZF, AF, PF and CF.

**Exceptions :**

Protected Mode	GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, SS(0) - for an illegal address in the SS segment; PF(fault-code) - for a page fault.
Real Address Mode	Interrupt 13 - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in Real Address Mode, PF(fault-code) for a page fault.

### SETcc -- Byte Set on Condition

SETcc stores a byte at the destination specified by the effective address or register if the condition is met or a 0 byte if the condition is not met.

**Flags Affected :** None

**Exceptions :**

Protected Mode	GP(0) - if the result is in a non-writable segment, GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS or GS segments, SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault.
Real Address Mode	Interrupt 13 - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in Real Address Mode, PF(fault-code) - for a page fault.

**Examples**

```
SETO mem_byte ; Sets bytes if OF = 1
SETB AL        ; Sets AL if below (CF = 0)
```

### SGDT/SIDT -- Store Global/Interrupt Descriptor Table Register

SGDT/SIDT copies the contents of the descriptor table register the six bytes of memory indicated by the operand. The LIMIT field of the register is assigned to the first word at the effective address. If the operand-size attribute is 32 bits, the next three bytes are assigned the BASE field of the register and the fourth byte is written with zero. The last byte is undefined. Otherwise, if the operand-size attribute is 16 bits, the next four bytes are assigned the 32-bit BASE field of the register.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	<b>Interrupt 6</b> - if the destination operand is a register, <b>GP(0)</b> - if the destination is in a nonwritable segment, <b>GP(0)</b> - for an illegal memory operand with an effective address in the CS, DS, ES, FS or GS segments, <b>SS(0)</b> - for an illegal address in the SS segment, <b>PF(fault-code)</b> - for a page fault
<b>Real Address Mode</b>	<b>Interrupt 6</b> - if the destination operand is a register, <b>Interrupt 13</b> - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, <b>PF(fault-code)</b> - for a page fault.

**Examples**

SGDT memory ; Stores GDTR into memory  
 SIDT memory ; Stores IDTR into memory

**SHLD -- Double Precision Shift Left**

SHLD shifts the first operand provided by the r/m field to the left as many bits as specified by the count operand. The second operand (r16 or r32) provides the bits to shift in from the right (starting with bit 0). The result is stored back into the r/m operand. The register remains unaltered.

**Flags Affected :** OF, SF, ZF, PF and CF. AF and OF are undefined.

**Exception :**

<b>Protected Mode</b>	<b>GP(0)</b> - if the result is in a nonwritable segment, <b>GP(0)</b> - for an illegal memory operand with an effective address in the CS, DS, ES, FS or GS segments; <b>SS(0)</b> - for an illegal address in the SS segment, <b>PF(fault-code)</b> - for a page fault.
<b>Real Address Mode</b>	<b>Interrupt 13</b> - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, <b>PF(fault-code)</b> - for a page fault.

**Example**

SHLD EAX, EBX, 05H ; EAX  $\leftarrow$  SHL (EAX concatenated with EBX) 5 times —

**SHRD -- Double Precision Shift Right**

SHRD shifts the first operand provided by the r/m field to the right as many bits as specified by the count operand. The second operand (r16 or r32) provides the bits to shift in from the left (starting with bit 31). The result is stored back into the r/m operand. The register remains unaltered.

**Flags Affected :** OF, SF, ZF, PF and CF. AF and OF are undefined.

**Exceptions :**

<b>Protected Mode</b>	GP(0) - if the result is in a nonwritable segment, GP(0) - for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments, SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault
<b>Real Address Mode</b>	<b>Interrupt 13</b> - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, PF(fault-code) - for a page fault.

**Example**

SHRD EAX, EBX, 06H ;  
EAX ← SHR (EAX concatenated with EBX) 6 times

**SLDT -- Store Local Descriptor Table Register**

SLDT stores the Local Descriptor Table Register (LDTR) in the two-byte register or memory location indicated by the effective address operand. This register is a selector that points into the Global Descriptor Table.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	GP(0) - if the result is in a nonwritable segment, GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS or GS segments, SS(0) - for an illegal address in the SS segment; PF(fault-code) - for a page fault.
<b>Real Address Mode</b>	<b>Interrupt 6</b> - SLDT is not recognized in Real Address Mode.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, PF(fault-code) - for a page fault.

**Example**

SLDT BX ; Stores LDTR in BX register

**SMSW -- Store Machine Status Word**

SMSW stores the machine status word (part of CR0) in the two-byte register or memory location indicated by the effective address operand.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	GP(0) - if the result is in a nonwritable segment, GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS or GS segments, SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault.
-----------------------	---

Real Address Mode	<b>Interrupt 13</b> - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in Real Address Mode, <b>PF(fault-code)</b> - for a page fault.

**Example**

SMSW BX ; Stores MSW in BX register

**STC -- Set Carry Flag**

STC sets the carry flag to 1.

**Flags Affected :** CF = 1

**Exceptions :** None

**STD -- Set Direction Flag**

STD sets the direction flag to 1, causing all subsequent string operations to decrement the index registers, (E)SI and/or (E)DI, on which they operate.

**Flags Affected :** DF = 1

**Exceptions :** None

**STI -- Set Interrupt Flag**

STI sets the interrupt flag to 1. The 80386 then responds to external interrupts after executing the next instruction if the next instruction allows the interrupt flag to remain enabled.

**Flags Affected :** IF = 1

**Exceptions :**

Protected Mode	GP(0) - if the current privilege level is greater (has less privilege) than the I/O privilege level.
Real Address Mode : None	Virtual 8086 Mode : None

**STOS/STOSB/STOSW/STOSD -- Store String Data**

STOS transfers the contents of all AL, AX, or EAX register to the memory byte or word given by the destination register relative to the ES segment. The destination register is DI for an address-size attribute of 16 bits or EDI for an address-size attribute of 32 bits.

After the transfer is made, DI is automatically updated. If the direction flag is 0 (CLD was executed), DI is incremented; if the direction flag is 1 (STD was executed), DI is decremented. DI is incremented or decremented by 1 if a byte is stored, by 2 if a word is stored or by 4 if a doubleword is stored.

STOSB, STOSW and STOSD are synonyms for the byte, word and doubleword STOS instructions, that do not require an operand. They are simpler to use, but provide no type or segment checking.

**Flags Affected :** None

**Exceptions :**

Protected Mode	GP(0) - if the result is in a nonwritable segment; GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS or GS segments; SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault
Real Address Mode	Interrupt 13 - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in Real Address Mode; PF(fault-code) - for a page fault.

**STR** - Store task register

The contents of the task register are copied to the two-byte register or memory location indicated by the effective address operand.

**Flags Affected :** None

**Exceptions :**

Protected Mode	GP(0) - if the result is in a nonwritable segment, GP(0) for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault.
Real Address Mode	Interrupt 6 - STR is not recognized in Real Address Mode.
Virtual 8086 Mode	Same exceptions as in Real Address Mode.

**Example**

STR DX ; Stores TR in DX

### SUB -- Integer Subtraction

SUB subtracts the second operand (SRC) from the first operand (DEST). The first operand is assigned the result of the subtraction and the flags are set accordingly.

When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended to the size of the destination operand.

**Flags Affected :** OF, SF, ZF, AF, PF and CF

**Exceptions :**

Protected Mode	GP(0) - if the result is in a nonwritable segment, GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS or GS segments, SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault.
Real Address Mode	Interrupt 13 - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in Real Address Mode, PF(fault-code) - for a page fault.

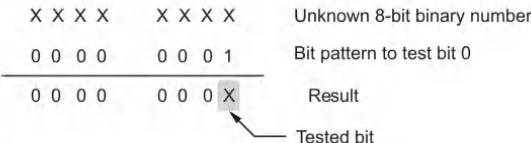
**Examples**

SUB AL, 0F0H	; Subtract immediate number 0F0H
SUB CL, TOTAL [BX]	; Subtract byte from effective address TOTAL [BX] ; from the contents of CL and store result in CL

### TEST -- Logical Compare

TEST computes the bit-wise logical AND of its two operands. Each bit of the result is 1 if both of the corresponding bits of the operands are 1; otherwise, each bit is 0. The result of the operation is discarded and only the flags are modified.

The TEST instruction functions in the similar manner as a CMP instruction. The difference is that the TEST instruction normally tests a single bit (or occasionally multiple bits), while the CMP instruction tests the entire byte or word. The Fig. 2.2.5 shows the bit pattern and test operation for testing of bit 0. If zero flag is set ( $Z = 1$ ) after this operation, the bit under test bit-0 is zero ; otherwise bit-0 is 1.



**Fig. 2.2.5 TEST operation**

The zero flag is usually tested by JZ or JNZ instructions. Therefore, the TEST instruction is usually followed by either the JZ or JNZ instruction.

**Flags Affected :** OF = 0, CF = 0; SF, ZF and PF.

**Exceptions :**

<b>Protected Mode</b>	GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS or GS segments, SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault.
<b>Real Address Mode</b>	<b>Interrupt 13</b> - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, PF(fault-code) - for a page fault.

**Examples**

```
TEST AL, CL      ; AND CL with AL.  
                  ; Update flags, result is not stored.  
TEST BX, CX      ; AND CX with BX  
                  ; Update flags, result is not stored.
```

**VERR, VERW -- Verify a Segment for Reading or Writing**

The two-byte register or memory operand of VERR and VERW contains the value of a selector. VERR and VERW determine whether the segment denoted by the selector is reachable from the current privilege level and whether the segment is readable (VERR) or writable (VERW). If the segment is accessible, the zero flag is set to 1; if the segment is not accessible, the zero flag is set to 0.

**Flags Affected :** ZF

**Exceptions :**

<b>Protected Mode</b>	GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS or GS segments, SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault.
<b>Real Address Mode</b>	<b>Interrupt 6</b> - VERR and VERW are not recognized in Real Address Mode.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, PF(fault-code) - for a page fault.

**Examples**

```
VERR CX          ; ZF ← 1, if segment can be read, selector in CX register  
VERW mem_word    ; ZF ← 1, if segment can be written, selector in memory word
```

**WAIT -- Wait until BUSY# Pin is Inactive (HIGH)**

WAIT suspends execution of 80386 instructions until the BUSY# pin is inactive (high). The BUSY# pin is driven by the 80287 numeric processor extension.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	NM - if the task-switched flag in the machine status word (the lower 16 bits of register CR0) is set, MF - if the ERROR input pin is asserted.
<b>Real Address Mode</b>	Same exceptions as in Protected Mode.
<b>Virtual 8086 Mode</b>	Same exceptions as in Protected Mode.

**XCHG -- Exchange Register/Memory with Register**

XCHG exchanges two operands. If a memory operand is involved, BUS LOCK is asserted for the duration of the exchange, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL.

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	GP(0) - if either operand is in a nonwritable segment, GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments; SS(0) - for an illegal address in the SS segment; PF(fault-code) - for a page fault.
<b>Real Address Mode</b>	Interrupt 13 - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
<b>Virtual 8086 Mode</b>	Same exceptions as in Real Address Mode, PF(fault-code) - for a page fault.

**Examples**

XCHG BX, CX	; Exchange word in BX with word in CX.
XCHG AL, CL	; Exchange byte in AL with byte in CL.
XCHG AL, SUM [BX]	; Exchange byte in AL with byte in memory at ; EA = SUM + [BX]. PA = EA + DS.

**XLAT/XLATB -- Table Look-up Translation**

XLAT changes the AL register from the table index to the table entry. AL should be the unsigned index into a table addressed by DS:BX (for an address-size attribute of 16 bits) or DS:EBX (for an address-size attribute of 32 bits).

**Flags Affected :** None

**Exceptions :**

<b>Protected Mode</b>	GP(0) - for an illegal memory operand with an effective address in the CS, DS, ES, FS, or GS segments, SS(0) - for an illegal address in the SS segment, PF(fault-code) - for a page fault.
-----------------------	---

Real Address Mode	<b>Interrupt 13</b> - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in Real Address Mode, <b>PF(fault-code)</b> - for a page fault.

### XOR -- Logical Exclusive OR

XOR computes the exclusive OR of the two operands. Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same. The answer replaces the first operand.

The XOR instruction is used if some bits of a register or memory location must be inverted. This instruction allows part of a number to be inverted or complemented. This is illustrated in Fig. 2.2.6.

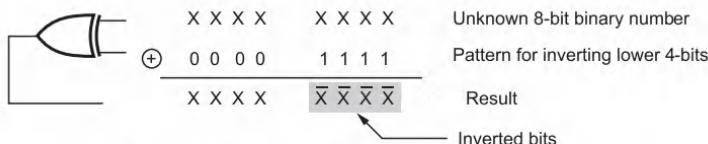


Fig. 2.2.6 Inversion of part of a number using XOR operation

#### Flags Affected

CF = 0, OF = 0; SF, ZF, and PF. AF is undefined.

#### Exceptions :

Protected Mode	<b>GP(0)</b> - if the result is in a nonwritable segment, <b>GP(0)</b> - for an illegal memory operand with an effective address in the CS, DS, ES, FS or GS segments, <b>SS(0)</b> - for an illegal address in the SS segment, <b>PF(fault-code)</b> - for a page fault.
Real Address Mode	<b>Interrupt 13</b> - if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
Virtual 8086 Mode	Same exceptions as in Real Address Mode, <b>PF(fault-code)</b> - for a page fault.

#### Examples

1. ; AL = 1010 1111 = AFH

; BL = 1111 0000 = F0H

XOR BL, AL ; XOR byte in AL with byte in BL

; BL = 0101 1111 = 5FH

**Review Question**

1. What is the use of bit test and modify instructions.
2. How IMUL is different from MUL ?
3. Enlist the program flow control instructions.
4. Explain the following instructions of 80386 :
  - i) SLDT ii) LEA iii) OUT iv) CMP v) XCHG
5. List and explain iteration control instructions of 80386 DX microprocessor.
6. Explain the following instruction with an example :
  - (i) ENTER      (ii) LEAVE      (iii) BOUND
7. Explain the following instructions with examples :
  - (i) BSR      (ii) CLTS      (iii) DAA
8. Differentiate between DIV and IDIV instruction.
9. What is the use of following instructions ?
  - i) Wait ii) Lock
10. Explain the following instructions, mention flags affected :
  - i) CWD      ii) BT      iii) LAHF
11. Explain with example SHL and ROL instructions.
12. Explain the following instructions, mention flags affected :
  - i) LIDT      ii) CLD      iii) MOVS
13. Explain any three control transfer instructions of 80386.
14. What is the use of following instructions in 80386 ? Mention which flags gets effected with each instruction : ADC, DIV, CMP
15. Explain shift and rotate instructions of 80386.
16. Explain LEA and XLAT instructions.

**SPPU : Dec.-13, Marks 3****SPPU : Dec.-13, Marks 3****SPPU : May-14, Marks 3****SPPU : Dec.-14, Marks 5****SPPU : Dec.-15, Marks 4****SPPU : May-15, Marks 3****SPPU : May-16, Marks 3****SPPU : May-16, Marks 2****SPPU : May-17, Marks 2****SPPU : May-17, Marks 6****SPPU : Dec.-17, Marks 4****SPPU : Dec.-17, Marks 6****SPPU : Dec.-18, Marks 6****SPPU : Dec.-18, Marks 6****SPPU : May-19, Marks 6****SPPU : Dec.-19, Marks 2**

**Notes**

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

## **UNIT - II**

# **3**

# **Bus Cycles and System Architecture**

### **Syllabus**

**Initialization** - Processor State after Reset. Functional pin Diagram, functionality of various pins, I/O Organization, Memory Organization (Memory banks), Basic memory read and writes cycles with timing diagram.

**Systems Architecture** - Systems Registers (Systems flags, Memory Management registers, Control registers, Debug registers, Test registers), System Instructions.

### **Contents**

3.1 Initialization	
3.2 Processor State after Reset . . . . .	<b>Dec.-17,19, May-18,19,</b> . . . . . Marks 4
3.3 Functional Pin Diagram . . . . .	<b>May-01,02,03,09,10,12,14,16,17,18,</b> . . . . . <b>Dec.-02,03,11,14,15,17,18,19,</b> . . . . . Marks 8
3.4 I/O Organization . . . . .	<b>May-17,19, Dec.-17,</b> . . . . . Marks 6
3.5 Memory Organization (Memory Banks)	
3.6 Basic Memory Read and Writes Cycles with Timing Diagram	
	. . . . . <b>Dec.-02,03,10,13,14,15,17,18,19,</b>
	. . . . . <b>May-02,09,10,11,13,14,17,18,19,</b> . . . Marks 16
3.7 Systems Architecture	
3.8 Systems Registers . . . . .	<b>Dec.-03,10,11,14,17,18,19,</b> . . . . . <b>May-02,04,09,10,11,12,13,16,17,18,19,</b> . . . . . <b>Nov.-12,</b> . . . . . Marks 10
3.9 Systems Instructions	

### 3.1 Initialization

After a signal on the RESET pin, certain registers of the 80386 are set to predefined values. These values are adequate to enable execution of a bootstrap program, but additional initialization must be performed by software before all the features of the processor can be utilized.

### 3.2 Processor State after Reset

SPPU : Dec.-17.19, May-18.19

**EAX Register :** The contents of EAX depend upon the results of the power-up self test. The self-test may be requested externally by assertion of BUSY# at the end of RESET. The EAX register holds zero if the 80386 passed the test. A nonzero value in EAX after self-test indicates that the particular 80386 unit is faulty. If the self-test is not requested, the contents of EAX after RESET is undefined.

**DX Register :** DX holds a component identifier and revision number after RESET as shown in Fig. 3.2.1. DH contains 3, which indicates an 80386 component. DL contains a unique identifier of the revision level.

**CR0 Register :** Control register zero (CR0) contains the values shown in Fig. 3.2.2. The ET bit of CR0 is set if an 80387 is present in the configuration (according to the state of the ERROR# pin after RESET). If ET is reset, the configuration either contains an 80287 or does not contain a coprocessor. A software test is required to distinguish between these latter two possibilities.

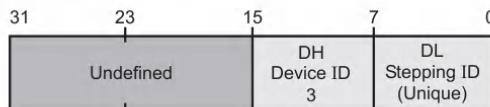


Fig. 3.2.1 Contents of EDX after Reset

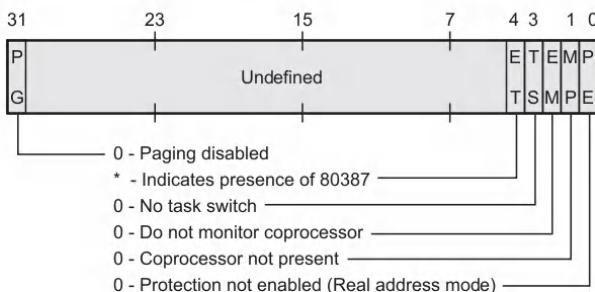


Fig. 3.2.2 Initial contents of CR0

### Remaining registers and flags :

EFLAGS = 00000002H	SS selector = 0000H
IP = 0000FFF0H	FS selector = 0000H
CS selector = 0000H	GS selector = 0000H
DS selector = 0000H	IDTR: base = 0 limit = 03FFH
ES selector = 0000H	All registers not mentioned above are undefined.

**Table 3.2.1**

These settings imply that the processor begins in real-address mode with interrupts disabled.

### Review Questions

- What is the status of the various registers of 80386 processor after reset.
- What are the contents of various registers of processor 80386 after reset ?

**SPPU : Dec.-17, May-18, Marks 3**

- Explain 80386 processor state after RESET.

**SPPU : May-19, Dec.-19, Marks 4**

## 3.3 Functional Pin Diagram

**SPPU : May-01,02,03,09,10,12,14,16,17,18, Dec.-02,03,11,14,15,17,18,19**

The Fig. 3.3.1 (a) and (b) shows functional pin diagram and pin layout of 80386DX. (Refer Fig. 3.3.1 on next page)

These signals are separated in four major groups :

- |                        |                          |
|------------------------|--------------------------|
| 1. Memory/IO interface | 2. Interrupt interface   |
| 3. DMA interface       | 4. Coprocessor interface |

### 3.3.1 Memory/IO Interface Signals

It includes data bus, separate address bus, five bus status signals and three bus control signals.

**Data Bus :** The data bus consists of 32 pins ( $D_{31} - D_0$ ). These lines are used to transfer 8, 16, 24, or 32-bit data at one time.

### Address Bus :

The 80386DX generates 32-bit address. The higher 30-bits of address are sent on the  $A_{31}-A_2$ . The lower 2-bits, select one of four bytes of the 32-bit data bus. These two bits are internally decoded and sent on the four byte enable pins ( $\overline{BE}_3 - \overline{BE}_0$ ). Each byte enable pin corresponds to one of four bytes of the 32-bit data bus.

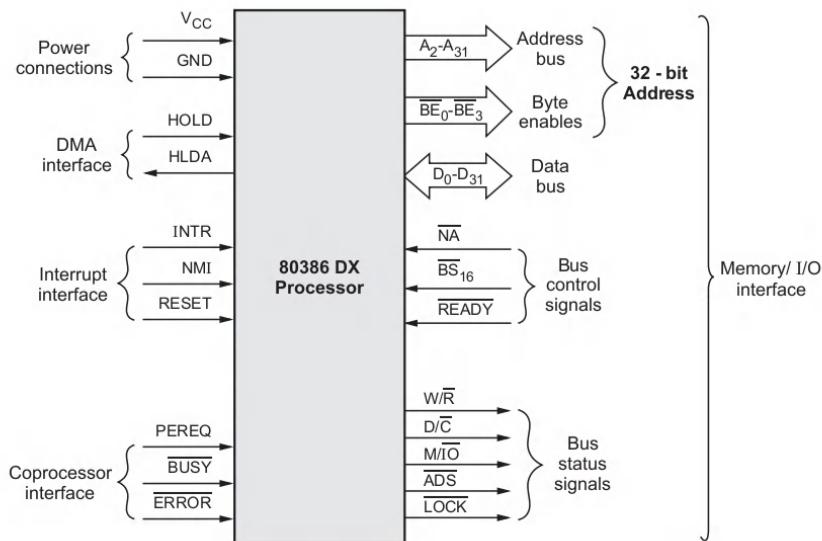


Fig. 3.3.1 (a) Functional pin diagram of 80386DX

BE <sub>3</sub>	BE <sub>2</sub>	BE <sub>1</sub>	BE <sub>0</sub>	D <sub>31</sub> - D <sub>24</sub>	D <sub>23</sub> - D <sub>16</sub>	D <sub>15</sub> - D <sub>8</sub>	D <sub>7</sub> - D <sub>0</sub>
1	1	1	0				XXXXXXXXXX
1	1	0	1			XXXXXXXXXX	
1	0	1	1		XXXXXXXXXX		
0	1	1	1	XXXXXXXXXX			
1	1	0	0			XXXXXXXXXX	XXXXXXXXXX
1	0	0	1		XXXXXXXXXX	XXXXXXXXXX	
0	0	1	1	XXXXXXXXXX	XXXXXXXXXX		
1	0	0	0		XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX
0	0	0	1	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX	
0	0	0	0	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX

Fig. 3.3.2 (a) Types of data transfers for various byte enable combinations

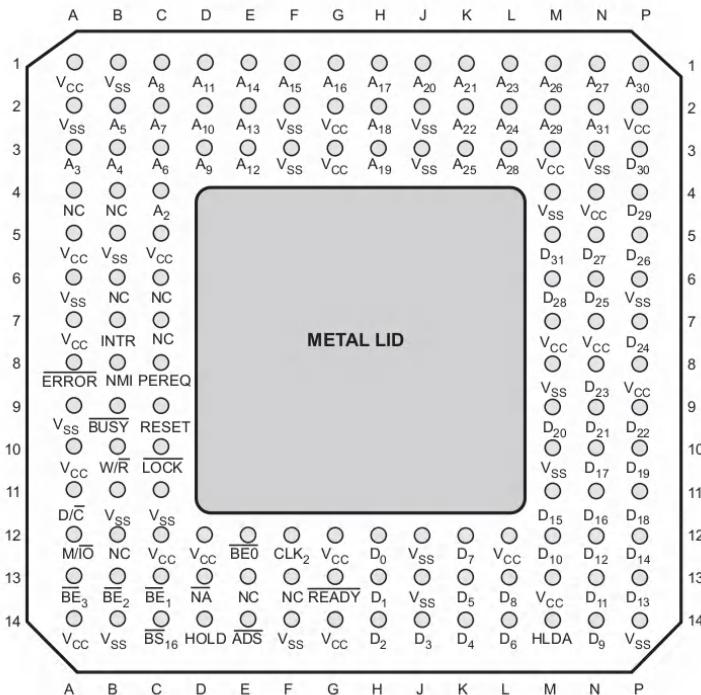


Fig. 3.3.1 (b) Pin layout of 80386DX

$\overline{BE}_3$	$\overline{BE}_2$	$\overline{BE}_1$	$\overline{BE}_0$	$D_{31} - D_{24}$	$D_{23} - D_{16}$	$D_{15} - D_8$	$D_7 - D_0$
1	1	1	0				XXXXXXXXXX
1	1	0	1			XXXXXXXXXX	
1	0	1	1		XXXXXXXXXX	DDDDDDDD	
0	1	1	1	XXXXXXXXXX		DDDDDDDD	
1	1	0	0			XXXXXXXXXX	XXXXXXXXXX
1	0	0	1		XXXXXXXXXX	XXXXXXXXXX	
0	0	1	1	XXXXXXXXXX	XXXXXXXXXX	DDDDDDDD	DDDDDDDD
1	0	0	0		XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX
0	0	0	1	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX
0	0	0	0	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX

Fig. 3.3.2 (b) Data transfers including duplication

Fig. 3.3.2 (a) and (b) show the data transfers for all the possible variations of the byte enable outputs with and without duplication of data. The 80386DX performs data duplication during certain types of write cycles. When the data is transferred over high-order of data bus, same data is available on the low-order of data bus. This duplication is indicated by "DDDDDDDD"

in Fig. 3.3.2 (b). For an instruction acquisition bus cycle  $\overline{BE}_0 - \overline{BE}_3$  signals are 0000. This is because code is always fetched as 32-bit words (aligned double words).

**Bus Status Signals :** The bus status signals decide the type of bus cycle to be performed. These signals are :

1. Address status
2. Write/Read
3. Memory/IO
4. Data/Control
5. LOCK.

**Address status (ADS) :** A low on this pin indicates that the valid address is present on the address bus.

**Write/Read (W/R) :** This signal decides the specific type of memory or I/O operation that will occur during a bus cycle. A low on this pin indicates that, the data is to be read from memory or an I/O port, on the other hand a high on this pin indicates that the data is to be written into memory or an I/O port.

**Memory/I/O (M/IO) :** This signal identifies whether the current bus cycle is memory cycle or I/O cycle. Logic 0 on this pin indicates that the current bus cycle is I/O cycle whereas logic 1 indicates that the current cycle is memory cycle.

**Data/Control (D/C) :** This signal identifies whether the current bus cycle is data or control cycle. This signal is logic 0 for instruction fetch, interrupt acknowledge, and halt/shut down operations and logic 1 for memory and I/O data read and write operations.

It is important to note that using  $M/IO$ ,  $D/C$  and  $W/R$ , we can identify the type of bus cycle that 80386 is currently executing.

The Table 3.3.1 shows the list of all possible machine cycles with their definition signals.

Bus enable signal	Data lines
$\overline{BE}_0$	$D_7 - D_0$
$\overline{BE}_1$	$D_{15} - D_8$
$\overline{BE}_2$	$D_{23} - D_{16}$
$\overline{BE}_3$	$D_{31} - D_{24}$

M/IO	D/C	W/R	Type of Bus Cycle
0	0	0	Interrupt acknowledge
0	0	1	Idle
0	1	0	I/O data read
0	1	1	I/O data write
1	0	0	Memory code read
1	0	1	Halt/shutdown
1	1	0	Memory data read
1	1	1	Memory data write

**Table 3.3.1 Machine cycle definition signals and types of machine cycles**

**LOCK :** This signal is used in multiprocessor systems. In multiprocessor systems resources are shared, such as global memory. The LOCK signal is used to ensure that the 80386DX has uninterrupted control of the system bus and the shared resource. By making LOCK output 0, the 80386DX can lock up the shared resource for the other masters in the system.

**Bus Control Signals :** The three bus control signals allow external logic to control the bus cycle. These signals are 1. READY 2. Next Address (NA) 3. Bus Size 16 (BS16).

**READY :** It is used primarily to synchronize slower peripherals with the microprocessor. This signal is produced by the microcomputer's memory or I/O subsystem. When READY signal is logic 0, slow memory or I/O devices tell 80386DX that they are ready for next data transfer. If ready is logic 1 then processor enters wait state since logic 1 on ready pin indicates that, the data transfer of current cycle is not yet completed.

**Next Address Request (NA) :** The external bus control logic control this signal. It activates pipelining by making next address request input low. Pipelining increases the address to data access time. By increasing the address to data access time, same level of performance can be obtained with slower, memory devices.

**Bus Size 16 (BS16) :** This signal activates 16-bit data bus operation; data is transferred on the low-order 16-bits of the data bus, and an extra cycle is provided for transfers of more than 16-bits.

### 3.3.2 Interrupt Interface Signals

**There are three interrupt interface signals :**

1. Interrupt request (INTR)
2. Nonmaskable interrupt request (NMI)
3. System reset (RESET).

**INTR :** The INTR input of the 80386 allows external devices to interrupt 80386 program execution. This input is sampled at the beginning of each instruction cycle. To ensure recognition of interrupt by the 80386, the INTR input must be held high until the 80386 acknowledges the interrupt by performing the interrupt acknowledge cycles. Thus it must be high at least eight CLK2 periods prior to the instruction to guarantee recognition as a valid interrupt. This specific requirement reduces the false triggering.

**Nonmaskable Interrupt (NMI) :** As name indicates this interrupt input is nonmaskable. This input is edge-triggered. A valid interrupt on this pin causes 80386 to execute interrupt service routine. The 80386 will not service subsequent NMI requests until the current request has been serviced.

**Reset :** Reset input forces 80386 to go into the reset state. This is an active high signal. When this signal is high it resets system resources, such as I/O ports, and the interrupt flag. After reset 80386 starts execution of program from memory address FFFFFFFF0H in real mode.

### 3.3.3 DMA Interface Signals

**HOLD and HLDA :** These pins are used to interface DMA controller. The DMA controller can request for bus access by asserting HOLD pin and 80386DX tells the DMA controller that the buses are available by asserting HLDA signal. The 80386DX activates its HLDA signal after completion of current bus cycle and it enters in HOLD state. In HOLD state, its local bus signals are in high impedance state.

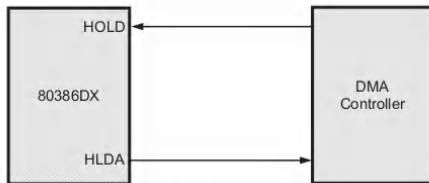


Fig. 3.3.3 DMA interface

### 3.3.4 Coprocessor Interface Signals

These signals are used to interface either 80287 or 80387 to the 80386DX. This group includes three signals (1) BUSY (2) ERROR (3) PEREQ (Coprocessor request)

**BUSY :** BUSY and ERROR are status signals from the coprocessor. BUSY signal is used to tell 80386 that the coprocessor is executing a numeric instruction. Thus when BUSY is logic 0, 80386 does not request the numeric coprocessor to perform another calculation until BUSY returns to logic 1.

**ERROR :** ERROR signal is used to indicate occurrence of error in the calculation. If an error occurs in a calculation performed by the numeric coprocessor, it informs 80386 that error has occurred by activating ERROR signal.

**PEREQ :** The coprocessor cannot transfer data over the data bus by itself. Whenever the coprocessor needs to read or write data from memory, it signals 80386 to initiate data transfer. The coprocessor does this by making PEREQ signal high.

#### Review Questions

1. Draw the functional pin diagram of 80386DX.
2. Explain the significance of the following signals with relevance to 80386
  - a.  $\overline{BE}_0$  to  $\overline{BE}_3$
  - b.  $\overline{BS}_{16}$  to  $\overline{NA}$  with their interdependency

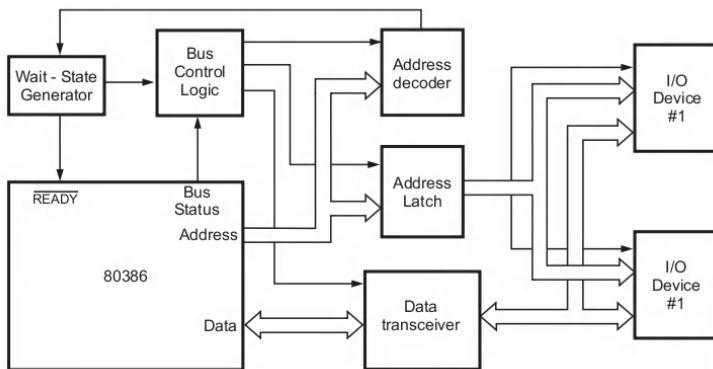
SPPU : May-02,03,12,16, Dec.-02,14,15, Marks 2

c. <u>PREQ</u>	<b>SPPU : Dec.-02, Marks 2</b>
d. <u>ERROR</u>	<b>SPPU : Dec.-02, May-10, Marks 2</b>
e. <u>READY</u>	<b>SPPU : Dec.-02, May-10,16, Marks 2</b>
3. Explain various memory / IO interface signals of 80386DX.	<b>SPPU : May-16, Marks 4</b>
4. Explain any two interrupt signals of 80386DX.	<b>SPPU : May-01, Marks 4</b>
5. Explain coprocessor interface signals of 80386DX.	<b>SPPU : May-11, Marks 8</b>
6. List and explain the hardware interrupts pins of 80386 processor ?	<b>SPPU : Dec.-11, Marks 8</b>
7. Explain the significance of the following signal of 80386 : NMI.	<b>SPPU : May-10, Dec.-14, Marks 2</b>
8. What is reset address of 80386 microprocessor ?	<b>SPPU : Dec.-03, Marks 2</b>
9. What is physical address of 80386 on 'POWER ON'. Explain.	<b>SPPU : May-09, Marks 2</b>
10. Explain the significance of the following signal of 80386 : D/C.	<b>SPPU : May-10, Marks 2</b>
11. What is the use of HOLD and HLDA signals ?	<b>SPPU : May-14, Dec.-15, Marks 3</b>
12. Explain the function of <u>LOCK</u> signal.	<b>SPPU : Dec.-15, Mark 1</b>
13. Explain following signals. i) ADS#      ii) READY#      iii) NA#	<b>SPPU : May-17, Marks 3</b>
14. Explain following signals BEO# through BE3#.	<b>SPPU : May-17, Marks 3</b>
15. Explain HOLD and HLDA signals of 80386DX.	<b>SPPU : Dec.-17,19, Marks 3</b>
16. Explain the following signals : i) NMI   ii) INTR   iii) RESET	<b>SPPU : Dec.-17, May-18, Marks 3</b>
17. Explain the following signals : i) W/R#   ii) D/C#   iii) M/IO#	<b>SPPU : May-18, Marks 3</b>
18. Explain the following signals of 80386 : M/IO#, W/R#, READY#	<b>SPPU : Dec.-18, Marks 6</b>

### 3.4 I/O Organization

**SPPU : May-17,19, Dec.-17**

- The 80386 supports 8-bit, 16-bit, and 32 bit I/O devices that can be mapped into either the 64-kilobyte I/O address space or the 4-gigabyte physical memory address space.
- Fig. 3.4.1 shows the basic I/O interface. It consists of Bus control logic, address decoder, address latch, data transceiver, and wait state generator along with 80386 processor.



**Fig. 3.4.1 Basic I/O interface block diagram**

- Input/Output devices can be interfaced with 80386 systems in two ways.
  1. I/O mapped I/O
  2. Memory mapped I/O

### 3.4.1 I/O Mapped I/O

- In I/O mapped I/O, the I/O devices are treated separate from memory.
- The 80386 supports software and hardware features for separate memory and I/O address spaces. Fig. 3.4.2 shows the memory and I/O spaces in real mode.
- The 80386 has four special instructions IN, INS, OUT, and OUTS to transfer data through input/output ports in I/O mapped I/O system.
- $M/\overline{IO}$  signal is always low when 80386 is executing these instructions. So  $M/\overline{IO}$  signal is used to generate separate addresses for Input/Output.
- Only  $2^8$  (2<sup>8</sup>) I/O addresses can be generated when direct addressing method is used. By using indirect addressing method this range can be extended upto 65536 (2<sup>16</sup>) addresses.

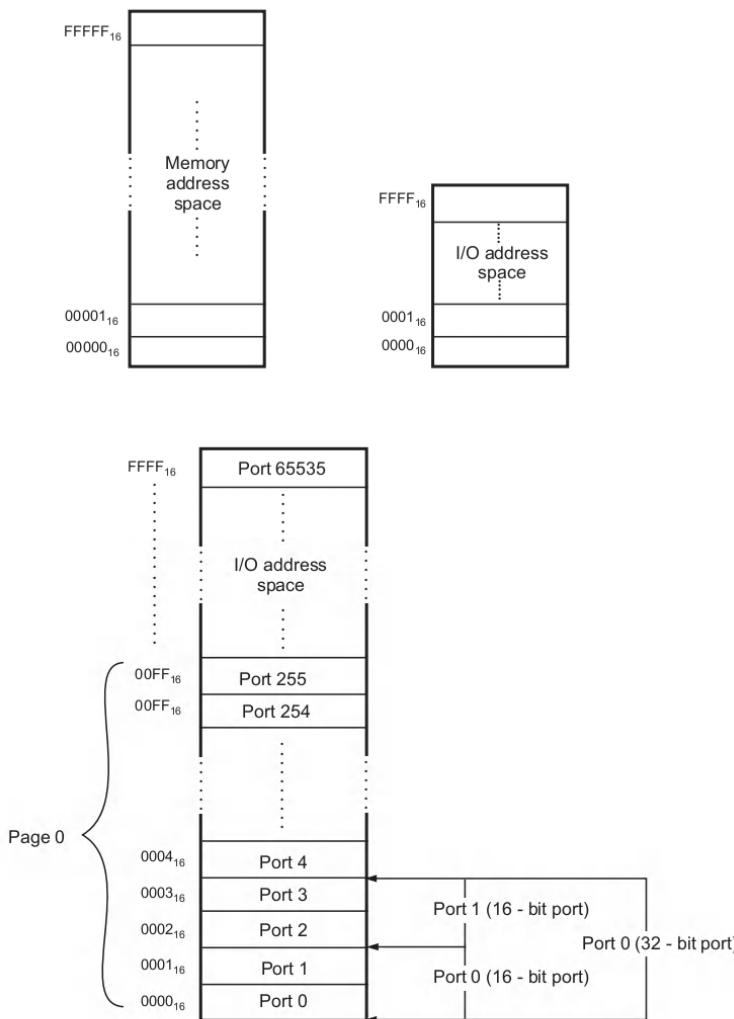
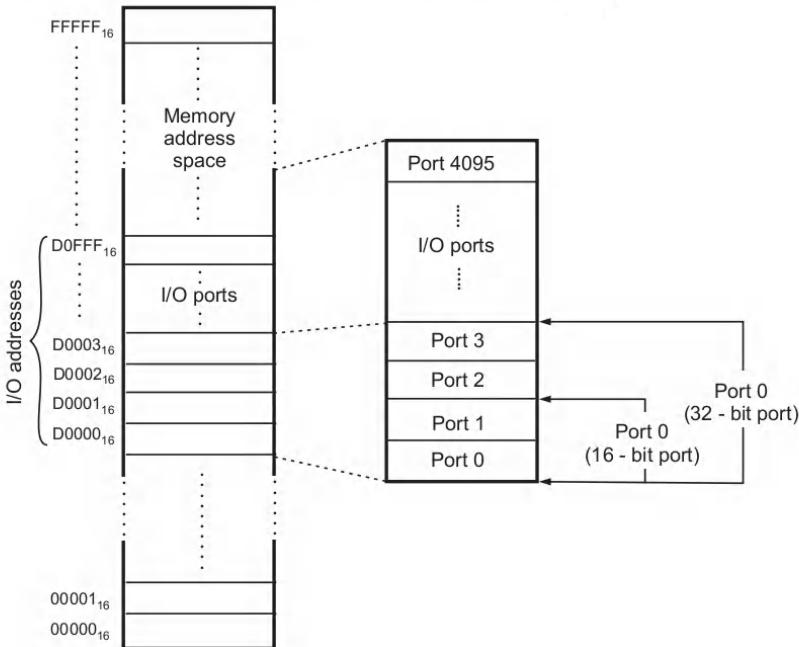


Fig. 3.4.2 Memory and I/O spaces in real mode

### 3.4.2 Memory Mapped I/O

- In memory mapped I/O, I/O device is placed in the memory address space of the microcomputer system. I/O device is connected as if it is a memory location. For this reason, the method is known as **memory mapped I/O**.

- In memory-mapped I/O, some of the memory address spaces are dedicated to the I/O system.
- Fig. 3.4.3 shows memory mapped I/O devices in the 80386DX memory address space. Here, 4096 memory addresses from D0000H through D0FFFH are assigned to I/O devices. The contents of D0000H represents byte wide I/O port 0; contents of D0001H and D0002H represent word-wide port 0; and contents of D0000H through D0003H represent double word wide port 0.



**Fig. 3.4.3 Memory mapped I/O devices**

- In memory mapped I/O, any instruction that references memory may be used to access an I/O port located in the memory space.
- For example, the MOV instruction can transfer data between any register and a port; and the AND, OR, and TEST instructions may be used to manipulate bits in the internal registers of a device.
- Memory-mapped I/O allows to use additional addressing modes for selecting the desired I/O device (e.g., direct address, indirect address, base register, index register, scaling).

- Memory-mapped I/O, like any other memory reference, is subject to access protection and control when executing in protected mode.

### Differentiate between memory mapped I/O and I/O mapped I/O

Sr. No.	Memory mapped I/O	I/O Mapped I/O
1.	I/O devices are placed in the memory address space of the processor.	Provides separate I/O space, distinct from physical memory.
2.	All memory related instructions can be used to access I/O device.	Only I/O related instructions are used to transfer data through the I/O ports.
3.	Since memory address space is more than I/O address space, the maximum number of I/O device can be connected is more.	Maximum number of I/O devices can be connected is less than that in memory mapped I/O.
4.	Since more number of address lines used more decoding hardware is required.	Comparatively less decoding hardware is required.
5.	More flexibility in using addressing modes.	Less flexibility in using addressing modes.

### Review Questions

1. What is I/O mapped I/O and memory mapped I/O ?
2. Show the memory and I/O spaces in I/O mapped I/O and real mode for 80386.
3. Show the memory and I/O spaces in memory mapped I/O for 80386.
4. Differentiate between memory mapped I/O and I/O mapped I/O.

**SPPU : May-17, Dec.-17, Marks 4**

5. Write the two mechanisms that provide protection for I/O functions.

**SPPU : May-17, Marks 2**

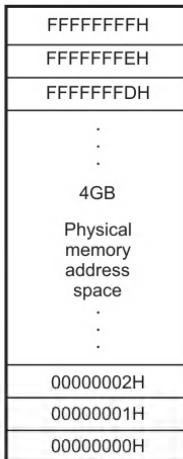
6. By which two ways, 80386 allows input/output to be performed ? Explain each in details.

**SPPU : May-19, Marks 6**

### 3.5 Memory Organization (Memory Banks)

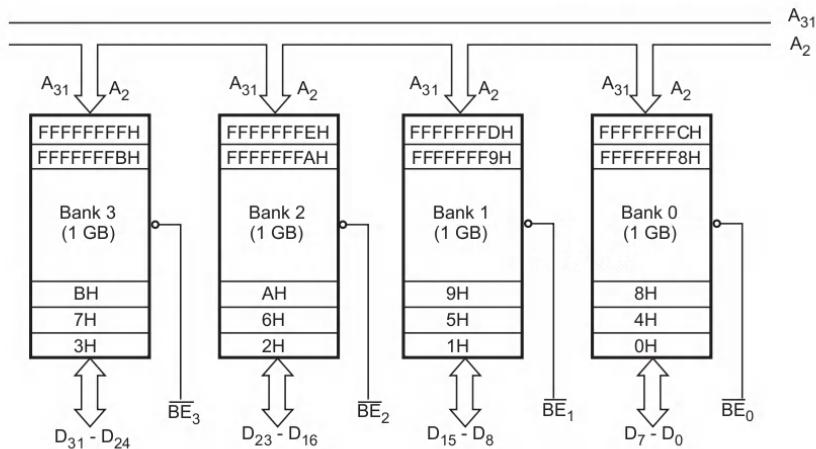
- The 80386DX has 32-bit address bus so it can access upto 4G-bytes ( $2^{32}$ ) of memory locations.
- Fig. 3.5.1 shows the physical address space. From the software point of view this memory is organized over the address range from 00000000H through FFFFFFFFH and 80386DX can access data in this memory as byte, word or double words.

- The words are accessed from two consecutive memory locations whereas double words are accessed from four consecutive memory locations.
- To implement this entire memory is divided into four independent byte-wide memory banks: Bank0–Bank3. Each bank is 1G-byte in size.



**Fig. 3.5.1 Physical address space**

- Fig. 3.5.1 shows the organization of four memory banks.
- As shown in figure bank 0, bank 1, bank 2 and bank 3 are selected using byte enable signals  $\overline{BE}_0$ ,  $\overline{BE}_1$ ,  $\overline{BE}_2$  and  $\overline{BE}_3$  signals, respectively.
- Address lines A31–A2 are connected in parallel to all memory banks which make it possible to access 1G-byte of memory. But the 32-bit data bus is distributed over four memory banks, 8-bit each.
- The 80386 uses byte enable signals instead of the two least significant address bits, because 80386 has problems involved in addressing more than one byte of memory at a time. When 80386 accesses word from even address, it uses two consecutive memory locations for example,
  - MOV WORD PTR DS : [2000H], 5678H** This instruction writes 78 to address 2000H and 56 to address 2001H.
  - Similarly, when 80386 accesses Dword from address divisible by 4, it uses four consecutive memory locations. This works fine when 80386 accesses even byte in case of word access and address divisible by 4 in case of Dword access, since address on the A31–A2 lines is same for word as well as Dword access.



**Fig. 3.5.2 Organization of four memory**

- The data transfers using such addresses are called **Aligned transfers**. But to access word from odd address or to access Dwords from address not divisible by 4 unaligned 80386 faces problem in placing the correct address on the address bus. This problem is solved by replacing two address pins A0 and A1 with four byte enable pins.
- Without the two least significant address pins, the 80386 produces only addresses that are even multiples of 4. To distinguish between four addresses (four banks) byte enable signals are used.
- Table 3.5.1 shows the use of the four byte enables pins with the address pins.

Desired Address	A31 through A02	BE0	BE1	BE2	BE3
Single-Byte Transfers					
00002000	00002000	On	Off	Off	Off
00002001	00002000	Off	On	Off	Off
00002002	00002000	Off	Off	On	Off
00002003	00002000	Off	Off	Off	On
00002004	00002004	On	Off	Off	Off

00002005	00002004	Off	On	Off	Off
	Double-Byte (Word) Transfers				
00002000	00002000	On	On	Off	Off
00002001	00002000	Off	On	On	Off
00002002	00002000	Off	Off	On	On
00002004	00002004	On	On	Off	Off
00002005	00002004	Off	On	On	Off
	Quad-Byte (Dword) Transfers				
00002000	00002000	On	On	On	On
00002004	00002004	On	On	On	On

**Table 3.5.1****Unaligned transfers**

Desired Address	A31 Through A 02	BE0	BE1	BE2	BE3
	Double-Byte (Word) Transfers				
00002003	00002004	On	Off	Off	Off
	00002000	Off	Off	Off	On
	Quad-Byte (Dword) Transfers				
00002001	00002004	On	Off	Off	Off
	00002000	Off	On	On	On
00002002	00002004	On	On	Off	Off
	00002000	Off	Off	On	On
00002003	00002004	On	On	On	Off
	00002000	Off	Off	Off	On
00002005	00002008	On	Off	Off	Off
	00002004	Off	On	On	On

**Review Questions**

1. Draw and explain the memory organization of 80386.
2. Explain the significance of byte enable signals.
3. Explain terms - aligned transfer and unaligned transfer.

**3.6 Basic Memory Read and Writes Cycles with Timing Diagram****SPPU : Dec.-02,03,10,13,14,15,17,18,19, May-02,09,10,11,13,14,17,18,19**

- The internal and external bus operations of 80386 are synchronized by the clock signal.
- The 80386 perform variety of machine (bus) cycles in response to internal requirements and external requirements.
- There are seven types of machine (bus) cycles/operations.
  - 1) Memory read
  - 2) Memory write
  - 3) I/O read
  - 4) I/O write
  - 5) Instruction fetch
  - 6) Interrupt acknowledge
  - 7) Halt/Shut down

M/IO	D/C	W/R	Type of Bus Cycle
0	0	0	Interrupt acknowledge
0	0	1	Idle
0	1	0	I/O data read
0	1	1	I/O data write
1	0	0	memory code read
1	0	1	Halt/shutdown
1	1	0	Memory data read
1	1	1	Memory data write

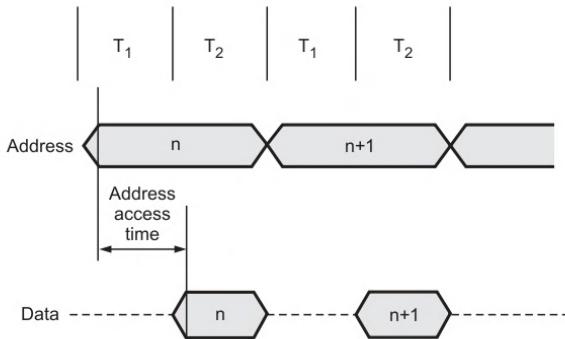
**Table 3.6.1 Machine cycle definition signals and types of machine cycles**

- In each machine cycles corresponding status signals are activated.
- Table 3.6.1 shows the status signals along with the machine cycles.

- The memory read and memory write machine cycles can be locked to prevent another bus master from using the bus.

### 3.6.1 Non-Pipelined Bus Cycles

- Fig. 3.6.1 shows typical nonpipelined microprocessor machine cycle.

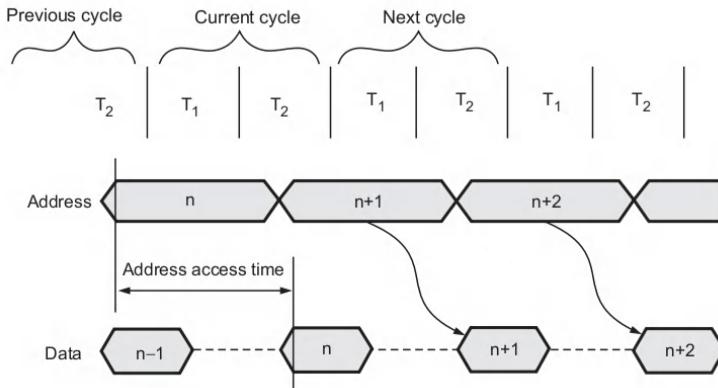


**Fig. 3.6.1 Typical nonpipelined microprocessor bus cycle**

- During  $T_1$ , the 80386DX sends the address, bus status signal and control signals. In case of write cycle, data to be output is also send on the data bus, during  $T_1$ .
- As shown in Fig. 3.6.2, after address access time read or write data transfer takes place over the data bus. This activity is carried out in  $T_2$ .

### 3.6.2 Pipelined Bus Cycles

- Pipelining allows machine cycles to be overlapped. The main advantage of pipelining is that it increases the amount of time required for the memory or I/O device to respond. This time is also referred as **access time**.
- The 80386DX implements pipelining by overlapping addressing of the next bus cycle with the data transfer of previous bus cycle.
- Fig. 3.6.2 shows the pipelined bus cycles of 80386DX.
- Fig. 3.6.2 shows that address becomes valid in  $T_2$  state of the previous bus cycle, and the data transfer for address takes place in  $T_2$ -state of the current bus cycle.
- It is important to note that the address  $A_n + 1$  becomes valid during  $T_2$  of the current bus cycle and actual data transfer for address  $A_n + 1$  takes place in  $T_2$  state of the next bus cycle.

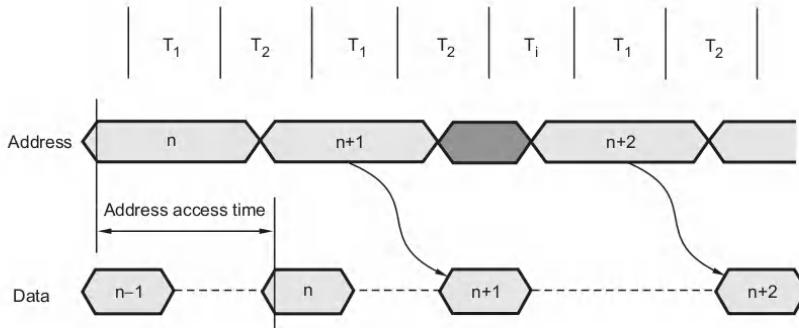


**Fig. 3.6.2 Typical pipelined bus cycle of 80386DX**

- If the processor is 80386DX-20 then one T-state time is 50 ns. In pipelined bus cycle the access time for memory and I/O device is 100 ns whereas access time for memory and I/O device in nonpipelined bus cycle is approximately 50 ns.

### 3.6.3 Idle State in Pipelined Bus Cycles

- Fig. 3.6.3 shows the idle bus state.



**Fig. 3.6.3 Bus idle state**

- In the pipelined bus cycle, we have seen that addressing of next cycle is overlapped with the data transfer of the current cycle. But in some situations such as prefetch queue is full and the instruction that is currently being executed does not require to access operands in memory or I/O device, no bus activity will take place. In such situations bus enters into a state called **idle state**.

### 3.6.4 Bus Cycle with Wait State

- Wait states can be inserted to extend the duration of 80386DX bus cycle with the help of external input signal  $\overline{\text{READY}}$ .
- This signal is sampled in the later part of the  $T_2$  state of every bus cycle to ensure that the data transfer is completed. As shown in Fig. 3.6.4, logic 1 at this input indicates that the data transfer is not completed.
- As long as  $\overline{\text{READY}}$  is held at the 1 level, the read or write data transfer does not take place and the current  $T_2$  state becomes a wait state ( $T_w$ ) to extend the bus cycle. The bus cycle is not completed until external hardware returns  $\overline{\text{READY}}$  back to logic 0.

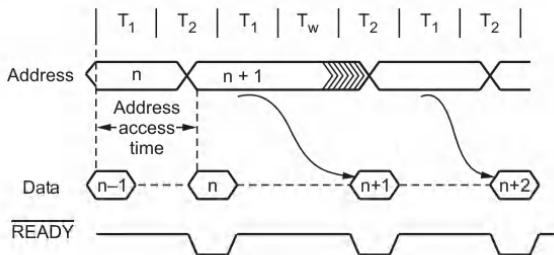


Fig. 3.6.4 Bus cycle with wait states

### 3.6.5 Non-Pipelined Read Cycle

- Fig. 3.6.5 (See Fig. 3.6.5 on next page) shows the timings for two nonpipelined read cycles (with and without a wait state). First read cycle is without wait state and second cycle is with wait state.
- The sequence of events for the nonpipelined read cycle is as follows :*
  - The read operation starts at the beginning of phase in the  $T_1$  state of the bus cycle. In this phase, 80386DX sends the address on the address bus and enables signals ( $\overline{\text{BE}0}$  -  $\overline{\text{BE}3}$ ) according to data transfer type. After sending the address, in the same phase, 80386 DX activates its  $\overline{\text{ADS}}$  signal to indicate valid address is placed on the address bus. In phase 1 of  $T_1$  - state 80386DX also activates the bus cycle definition signals :  $\overline{\text{M/I/O}}$ ,  $\overline{\text{D/C}}$  and  $\overline{\text{W/R}}$ . For read cycle  $\overline{\text{W/R}}$  is low.  $\overline{\text{M/I/O}}$  is high for memory read and low for an I/O read.  $\overline{\text{D/C}}$  signal differentiate between data and instruction code. This signal is high if data is to be read and low if an instruction code is to be read. At the end of phase 2 of  $T_1$  - state,  $\overline{\text{ADS}}$  is returned to its inactive logic 1 state. The address bus, byte enable pins, and bus status pins remain active through the end of the read cycle.

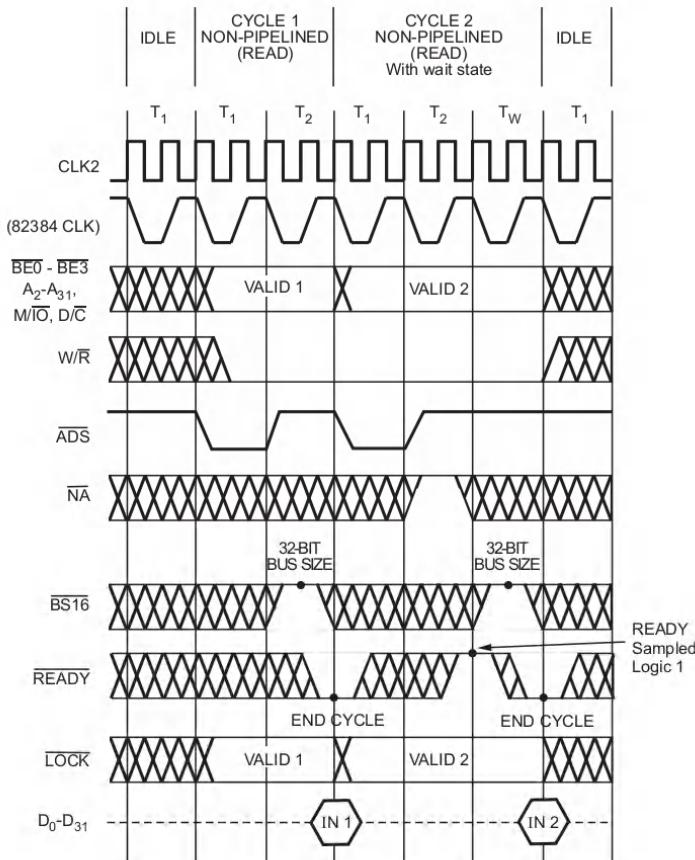


Fig. 3.6.5 Non pipelined read cycle

- At the beginning of phase 1 of T2 state external device activates BS16 signal. The 80386DX samples this signal in the middle of phase 1 of T2-state. If this signal is high, 80386DX does the 32-bit data transfer; otherwise 80386 DX performs 16-bit data transfer. The 80386 DX does this data transfer in phase 2 of T2-state.
- At the end of phase 2 of T2-state the **READY** signal is sampled by the 80386DX. The logic 1 on this signal inserts wait state in the current bus cycle to extend the bus cycle. In wait state ( $T_w$ ), the signals from T2-state are

maintained throughout the wait state period. It's just a repetition of T2-state. Thus the period of one wait state ( $T_w = T_2$ ) is equal to 50 ns of 20 MHz clock operation. If this signal is low, 80386 DX proceeds with next bus cycle.

- The  $\overline{LOCK}$  signal low indicates it is bus locked cycle. If bus cycles are locked the other bus master is not allowed to take control of the bus between two locked bus cycles.

### 3.6.6 Non-Pipelined Write Cycle

- Fig. 3.6.6 (see Fig. 3.6.6 on next page) shows the timings for two nonpipelined write cycles (with and without a wait state) first write cycle is without wait state and second cycle is with wait state.
- *The sequence of events for the nonpipelined write cycle is as follows :*
  - The nonpipelined write cycle is similar to nonpipelined read cycle. The write operation starts at the beginning of phase 1 in the T1 state of the bus cycle. In this phase, 80386DX sends the address on the address bus and enables signals  $\overline{BE}_0$  -  $\overline{BE}_3$  according to data transfer type. After sending address in the same phase, 80386DX activates its  $\overline{ADS}$  signal to indicate valid address is placed on the address bus. In phase 1 of T1-state 80386DX also activates the bus cycle definition signals :  $M\overline{I/O}$ ,  $D\overline{/C}$  and  $W\overline{/R}$ . For write cycle  $W\overline{/R}$  is high.  $M\overline{I/O}$  is high for memory and low for I/O write.  $D\overline{/C}$  signal is high.
  - At the beginning of phase 2 of T1-state, 80386DX sends data on the data bus. This data remains valid until the start of phase 2 of the T1-state of the next bus cycle.
  - At the end of phase 2 of T1 - state,  $\overline{ADS}$  is returned to its inactive logic 1 states. The address bus, byte enable pins, and bus status pins remain active through the end of the write cycle.
  - In the middle of phase 1 of T2-State, 80386DX samples  $\overline{BS16}$  input. If this signal is high, 80386 DX does the 32-bit data transfer; otherwise 80386DX performs 16-bit data transfer.
  - At the end of phase 2 of T2-state the  $\overline{READY}$  signal is sampled by the 80386DX. The logic 1 on this signal inserts wait state in the current bus cycle to extend the bus cycle. In wait state ( $T_w$ ), the signals from T2-state are maintained throughout the wait state period. It's just a repetition of T2-state. Thus the

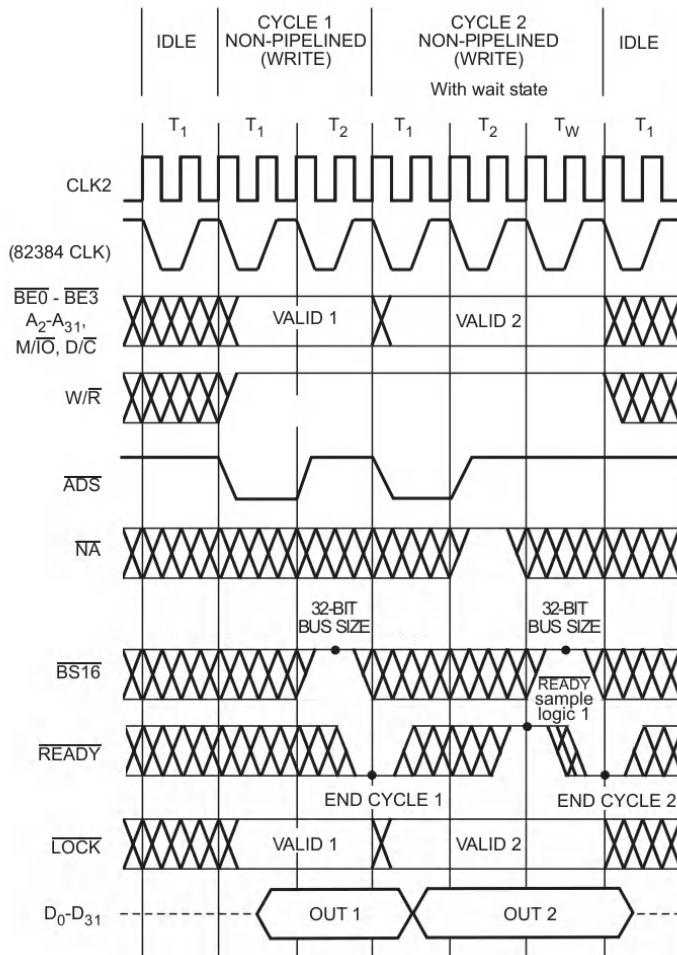


Fig. 3.6.6 Non-pipelined write cycle

period of one wait state ( $T_W - T_2$ ) is equal to 50 ns of 20 MHz clock operation. If this signal is low, 80386DX proceeds with next bus cycle.

### 3.6.7 Non-Pipelined Read / Write Cycles

- As mentioned earlier, address pipelining allows bus cycles to be overlapped, increasing the amount of time available for the memory or I/O device to respond.
- Fig. 3.6.7 shows both nonpipelined and pipelined read and write cycles. The cycle 1 and cycle 2 in the diagram show nonpipelined write and read cycles, respectively, whereas cycle 3 and cycle 4 in the diagram show pipelined write and read cycles, respectively. This diagram also shows how wait state can be avoided using pipelined bus cycle.

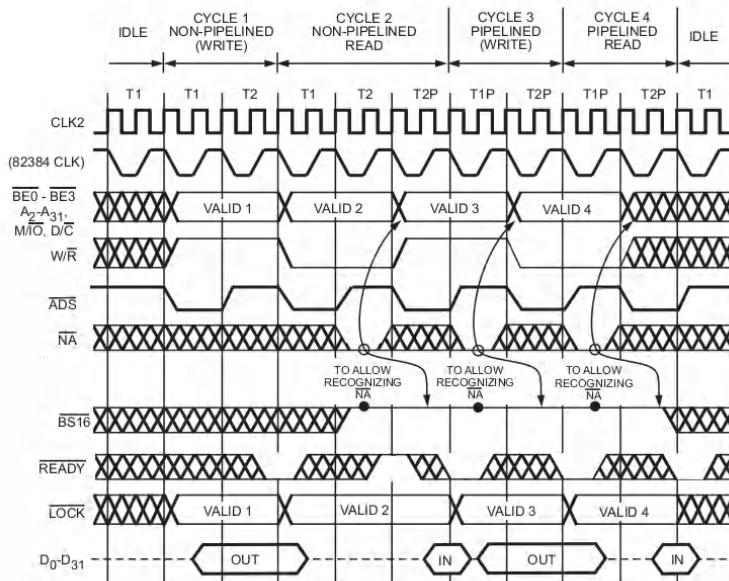


Fig. 3.6.7 Pipelined Read/Write Cycle

- In the pipelined bus cycle the address for the next bus cycle is sent during the T2 - state of the current cycle. In 80386DX, NA (next address) signal initiates address pipelining. The 80386DX samples NA signal at the beginning of phase 2 of any T state in which ADS is not active, specifically.
  - In the second T-state of a non-pipelined address cycle
  - In the first T-state of a pipelined address cycle
  - In any wait state of a non-pipelined address or pipelined address cycle unless NA has already been sampled active.

- In Fig. 3.6.7 NA is tested as 0 (active) during T2 of cycle 2 which ensures that 80386DX has to execute next cycle as pipelined bus cycle. The cycle 2 (nonpipelined read cycle) is also extended with one wait state because READY pin is not active, in wait state, the valid address for the next bus cycle is sent on the address bus as next bus cycle is pipelined bus cycle.
- The next cycle (cycle 3) is pipelined write cycle. In this, data is sent on the data bus in phase 2 of T1p-state and remains valid for the rest of the cycle. The READY signal is sampled at the end of T2p - state. As it is low, write cycle is completed without wait state. Fig. 3.6.7 shows NA is active during T1p of cycle 3, which ensures that 80386DX has to execute next cycle as pipelined bus cycle.
- The next cycle (cycle 4) is pipelined read cycle. In this, READY signal is tested 0 at the end of phase 2 of T2p - state. This means that read cycle is completed without wait state. It is important to note that due to pipelined address cycle access time is extended and one state (T-wait) of read cycle is saved.

### Review Questions

- Draw and explain the typical nonpipelined microprocessor bus cycle.*
- What do you understand by nonpipelined bus cycles in 80386 system ? Explain with the help of timing diagram.* **SPPU : Dec.-02,03, Marks 10**
- Draw and explain the typical pipelined bus cycle of 80386DX.*
- What do you understand by pipelined bus cycles in 80386 system ? What is the criteria for choosing between pipelined bus cycles and non-pipelined bus cycles ? Explain with the help of timing diagram.* **SPPU : Dec.-02, Marks 10**
- With the help of suitable timing diagram explain the pipelined bus cycles in 80386 system.* **SPPU : Dec.-03, Marks 8**
- What is idle state ? Explain it with the help of waveform.*
- Explain the bus cycle with wait state with the help of waveform.*
- Draw the timing diagram for read cycle with non pipelined address.* **SPPU : May-14, Marks 6**
- Draw neat timing diagram [consisting of CLK2, BE0 to BE3, A2-A31, M/IO, D/C, W/R, NA, BS16, ADS, LOCK, D<sub>0</sub> – D<sub>31</sub>] for the bus activity of one non-pipelined memory read cycle of 80386 processor.* **SPPU : May-02, Marks 8**
- Draw non-pipelined read cycle for 80386. Explain.* **SPPU : May-09, Dec.-14,15, Marks 6**
- Draw and explain nonpipelined write cycle of 80386DX.*
- Draw timing diagram of write machine cycle for 80386. Show status of important signals and list activities carried out in sequence.* **SPPU : May-10, 13, 14, Marks 8**

13. Enlist the differences between pipelined and non pipelined machine cycle.

**SPPU : May-14, Marks 4**

14. Draw the timing diagram of non-pipelined read cycle followed by pipelined write cycle and explain.

**SPPU : Dec.-10, May-11, Marks 16**

15. Draw the timing diagram for write cycle with pipelined address.

**SPPU : Dec.-13,14, Marks 5**

16. Draw the timing diagram for read cycle with pipelined address.

**SPPU : Dec.-13, Marks 5**

17. Draw read cycle with pipelined address timing.

**SPPU : May-17, Marks 6**

18. Draw read cycle with non - pipelined address timing.

**SPPU : Dec.-17, May-18,19, Marks 7**

19. Draw 'write cycle with pipelined address timing'.

**SPPU : Dec.-17, Marks 6**

20. With the help of neat diagram, explain the pipelined read bus cycle.

**SPPU : Dec.-18, Marks 6**

21. When WAIT state is required in 80386 read bus cycle ? Explain with neat diagram.

**SPPU : Dec.-18, Marks 6**

22. Draw and explain read cycle with non-pipelined address timing

**SPPU : May-19, Marks 8**

23. Draw and explain bus states and transitions when address pipelining is not used.

**SPPU : Dec.-19, Marks 5**

24. List various bus states when address pipelining is used.

**SPPU : Dec.-19, Marks 4**

25. Draw write cycle with non-pipelined address timing.

**SPPU : Dec.-19, Marks 5**

### 3.7 Systems Architecture

Many of the architectural features of the 80386 are used only by systems programmers. This chapter presents an overview of these aspects of the architecture.

The systems-level features of the 80386 architecture include :

- Memory Management
- Protection
- Multitasking
- Input/Output
- Exceptions and Interrupts
- Initialization
- Coprocessing and Multiprocessing
- Debugging

These features are implemented by registers and instructions.

### 3.8 Systems Registers

SPPU : Dec.-03,10,11,14,17,18,19, Nov.-12, May-02,04,09,10,11,12,13,16,17,18,19

The registers designed for use by systems programmers fall into these categories :

- EFLAGS
- Memory - Management Registers
- Control Registers
- Debug Registers
- Test Registers.

#### 3.8.1 System Flags - EFLAGS

Refer section 1.4.3.

#### 3.8.2 Memory-Management (System Address) Registers

There are four system address register : TR (Task Register), IDTR (Interrupt Descriptor Table Register), GDTR (Global Descriptor Table Register) and LDTR (Local Descriptor Table Register). Fig. 3.8.1 shows these special registers which are used in protected mode.

These registers hold the addresses for the four special descriptor table segments.

- The TR (Task Register) points to the Task state segment
- The IDTR (Interrupt Descriptor Table Register) points to the Interrupt Descriptor Table (IDT)
- The GDTR (Global Descriptor Table Register) points to the Global Descriptor Table (GDT)
- The LDTR (Local Descriptor Table Register) points to the Local Descriptor Table (LDT)

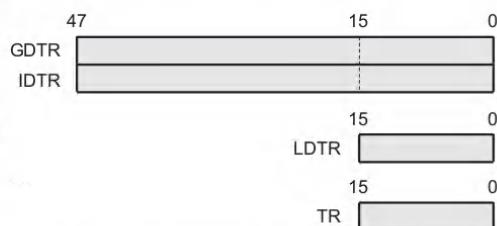


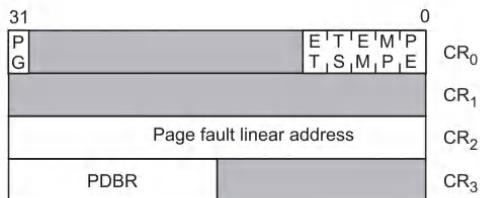
Fig. 3.8.1 Protected mode registers

#### 3.8.3 Control Registers

There are four control registers : CR0, CR1, CR2 and CR3. Fig. 3.8.2 shows control registers. These registers define the machine state that affects all the tasks in the systems.

### Control Register 0 (CR<sub>0</sub>)

The CR<sub>0</sub> holds the MSW (Machine Status Word). It contains six status bits : PE (Protection Enable), MP (Math Present), EM (Emulate Coprocessor), TS (Task Switched), ET (Extension Type), and PG (Paging).



**Fig. 3.8.2 Control registers**

**PE (Protection Enable)** : This bit is

similar to the VM bit in EFLAGS in that it controls the 80386's mode of operation. When PE is set, it is in protection mode otherwise it operates in Real Mode.

**MP (Math Present)** : When this bit is set, the 80386 assumes that real floating point hardware (80287 or 80387) is present in the system. When this bit is clear, the 80386 assumes that no such coprocessor exists, and will not attempt to use real floating point hardware.

**EM (Emulate Coprocessor)** : When this bit is set, the 80386 will generate an exception 11 (device not available) whenever it attempts to execute a floating point instruction. Programmer can use this exception handler to emulate floating point hardware in software.

**TS (Task Switched)** : The 80386 sets the bit automatically every time it performs a task switch. It will never clear this bit on its own. But programmer can clear this bit using CLTS instruction.

**ET (Extension Type)** : When power is applied, 80386 detects whether numeric processor connected is 80287 or 80387 and sets ET to logic 1, if numeric processor is 80387. This is necessary because the 80387 uses a slightly different protocol than 80287.

**PG (Paging)** : This bit enables or disables paging mechanism in Memory Management Unit (MMU). If bit is set, paging is enabled.

### Control Register 1 (CR<sub>1</sub>)

This is reserved by Intel.

### Control Register 2 (CR<sub>2</sub>)

CR<sub>2</sub> is read-only register. The 80386, itself writes the last 32-bit linear address of page fault routine in this register. When page fault occurs, the 80386 generates exception 14 (page fault). This address is important for writing page fault routine. The page fault routine helps programmer to find cause of the fault.

**Control Register 3 (CR<sub>3</sub>)** Control register 3 holds the physical address of the root of the two-level paging tables used when paging is enabled. It is also called **Page Directory Base Register (PDBR)**.

### 3.8.4 Debug Registers

Debug registers allow 80386 to provide debugging feature. The DR<sub>0</sub> to DR<sub>7</sub> registers are used to control debug feature. The debug registers DR<sub>0</sub> to DR<sub>3</sub> contain addresses associated with one of four breakpoints defined by certain bits in debug register 7 (DR<sub>7</sub>). Fig. 3.8.3 shows debug registers. The software debugger can load breakpoint addresses in these registers to aid in debugging.

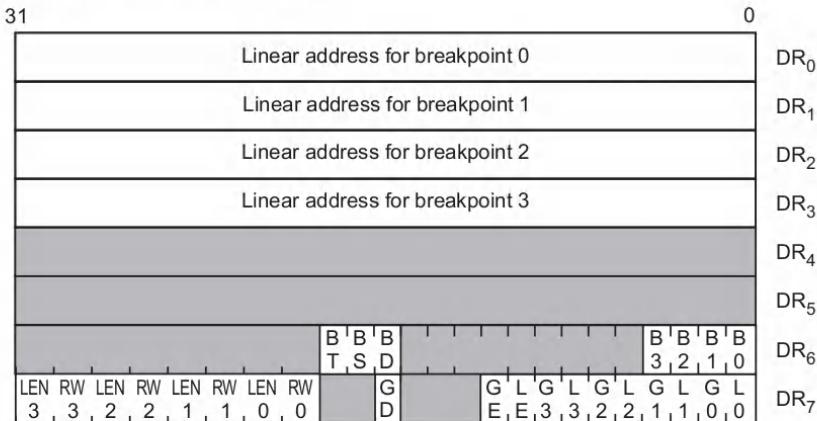


Fig. 3.8.3 Debug registers

#### Debug Registers 0 through 3

The first four debug registers (DR<sub>0</sub> - DR<sub>3</sub>) hold four linear addresses for breakpoints. The addresses in these registers are compared with address of the each instruction at the time of instruction execution and if a match is found, an exception 1 (debug fault) is generated. This allows 80386 to monitor upto four different addresses in the system.

#### Debug Registers 4 and 5

Registers 4 and 5 are undefined.

## Debug Register 6

Debug register 6 is also called **debug status register**. The 80386 sets the appropriate bits in this register which gives information of the probable causes for the last debug fault (Exception 1). The 80386 never clears these bits. Programmer must clear these status bits by writing into DR<sub>6</sub>.

The status bits are :

**B0 (Breakpoint 0) :** The 80386 sets this bit when it references the linear address contained in DR<sub>0</sub>, according to the conditions set by the LEN<sub>0</sub>, RW<sub>0</sub>, L<sub>0</sub>, G<sub>0</sub>, LE, and GE fields in DR<sub>7</sub>.

**B1-3 ( Breakpoint 1-3 ) :** These bits are similar to B<sub>0</sub>. In each case the linear address is referred from the respective debug register. For example : B<sub>2</sub> refers DR<sub>2</sub>.

**BD (Break for debug register access) :** The access for the debug registers can be locked by setting GD bit in DR<sub>7</sub>. The BD bit, if set, allows to invoke exception 1 handler, if processor tries to access debug register even though the access is locked.

**BS (Break for single step) :** This bit is set if the 80386 has invoked exception 1 since trace bit is set (TF bit is set in EFLAGS)

**BT (Break for task switch) :** When a task is initiated whose trace bit is set, the 80386 invokes an exception 1 if BT bit is set.

## Debug Register 7

It controls the debug feature. By programming bits in this register, programmer can configure the debug operation of the four linear address breakpoints. Each breakpoint is controlled by a set of four fields. These are :

**L0 (Local Enable) :** When this bit is set, the breakpoint address in DR<sub>0</sub> is monitored as long as 80386 is executing current task. When a task switch occurs, this bit is cleared by the 80386 and it must be re-enabled by writing into DR<sub>7</sub> required.

**G0 (Global Enable) :** When this bit is set, the breakpoint address in DR<sub>0</sub> is monitored all times, regardless of task. This bit must be cleared by writing into DR<sub>7</sub>.

**RW0 (Read/Write access) :** These bits decides the type of access that must occur at the address in DR<sub>0</sub>. Table 3.8.1 gives the list of different access types.

RW	RW bits in register DR <sub>7</sub>
00	Break on Code fetch only
01	Break on Data write only
10	Reserved
11	Break on Data Read or write only

Table 3.8.1 RW bits

**LEN0 (Breakpoint length) :** The breakpoints are further distinguished by its size. Each address register from DR0 through DR3 also has an associated length (LEN) field that specifies the length of the data item being monitored. The Table 3.8.2 shows the different sizes of the breakpoints.

There are in all four such fields ( L, G, RW and LEN ) for four breakpoints (B0-B3). The DR<sub>7</sub> contains three more bits. These are :

**LE (Local Exact) :** The pipelined architecture of 80386 fetches, decodes next instruction before the current one completes. Due to this, 80386 may not set status bit in DR<sub>6</sub> at the instant breakpoint occurs. If you set local exact bit, 80386 sets, corresponding status bit at the same instant at which breakpoint occurs, when 80386 is running the current task. When a task switch occurs this bit is cleared. This bit applies to all four linear breakpoints.

**GE (Global Exact) :** This is similar to the LE bit. If this bit is set 80386 informs about breakpoint at the instant it occurs regardless of task.

**GD (Global debug access) :**

When this bit is set, the 80386 denies the further access to any of the debug registers, either for reading or writing.

### 3.8.5 Test Registers

Among the eight test registers (TR<sub>0</sub>-TR<sub>7</sub>), only two test registers (TR<sub>6</sub>-TR<sub>7</sub>) are currently defined. The Fig. 3.8.4 shows the bit pattern of test registers.

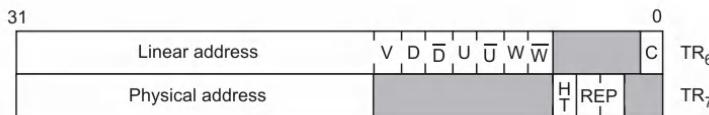


Fig. 3.8.4 Test registers

These registers are used to check Translation Lookaside Buffer (TLB) of the paging unit.

**Test Register 6 :** This is the TLB testing command register. By writing into this register, it is possible to either initiate a write directly into the 80386's TLB or to perform TLB lookups. TR<sub>6</sub> is divided into fields as follows :

C	:	This is a command bit. When this bit is cleared, a write to the TLB is performed. If it is set, the processor performs a TLB lookup.
		The next 7-bits are used as tag attributes for the TLB cache, either when writing a new entry or when performing a TLB lookup.
W (bit 5)	:	Not writable
W (bit 6)	:	Writable
Ū (bit 7)	:	Not user
U (bit 8)	:	User
D̄ (bit 9)	:	Not dirty
D (bit 10)	:	Dirty
V (bit 11)	:	Valid
Linear address (bits 12-31)	:	This is the tag field of the TLB. This field serves as the upper 20-bits of a linear address to be used for TLB references.

**Test Register 7 :** This register is the data testing register of the TLB. When a program is performing writes, the entry to be stored is contained in this register, along with cache set information

TR<sub>7</sub> is divided into fields as follows :

REP	:	This is replacement pointer. This field indicates which set of the TLB's four-way set associative cache to write to.
HT	:	This is pointer location. If this bit is set, the REP field determines which cache set to write to. If it is cleared, the set is determined with an internal algorithm.
Physical address (bits 12-31)	:	This is the data field of the TLB. This field contains either the physical address to be written into the TLB or the result of a valid TLB hit.

**Registers**

<b>General purpose</b>	:	EAX, EBX, ECX, EDX, ESP, EBP, ESI and EDI
Scratch pad	:	EAX, EBX, ECX and EDX
Index	:	ESI and EDI
Pointer	:	ESP and EBP
<b>Segment</b>	:	CS, DS, SS, ES, FS and GS. Store the base addresses of the currently active segments. In Real Mode and in Protected Mode these registers are used to store selectors which point the descriptors for segments.
<b>Flag</b>	:	
Status	:	CF, PF, AF, ZF, SF and OF
Control	:	DF
System	:	VM, R, NT, IOPL, IF, TF
<b>System Address Registers</b>	:	TR, IDTR, GDTR and LDTR
<b>Control Registers</b>	:	CR <sub>0</sub> - CR <sub>3</sub>
CR <sub>0</sub>	:	Stores machine status word
CR <sub>1</sub>	:	Reserved by Intel
CR <sub>2</sub>	:	Stores address of page fault routine
CR <sub>3</sub> (PDBR)	:	Stores address of two level paging tables
<b>Debug</b>	:	DR <sub>0</sub> - DR <sub>7</sub>
DR <sub>0</sub> -DR <sub>3</sub>	:	DR <sub>0-3</sub> holds four linear addresses for breakpoints
DR <sub>4</sub>	:	Stores information of last debug fault
DR <sub>7</sub>	:	Decides the access for the debug register controls the debug feature
<b>Test Registers</b>	:	TR <sub>0</sub> - TR <sub>7</sub>
TR <sub>0</sub> - TR <sub>5</sub>	:	Undefined
TR <sub>6</sub> - TR <sub>7</sub>	:	Checks translation look-aside buffer of the paging unit

**Review Questions**

1. List the different registers in 80386.
2. Explain the function of segment registers of 80386.
3. Explain the function of index, pointer and base registers.
4. List the different registers in 80386.
5. Describe 80386 flag register with significance of each and every bit in detail.

**SPPU : Dec.-03,14, May-10,13,18, Marks 6**

6. Write machine status word ( $CR_0$ ) with significance of each and every bit in it.
7. Define the purpose of each debug register in 80386.
8. Define the purpose of each control register in 80386.
9. What is the function of test registers ?
10. Explain the function of the segment registers in 80386DX.
11. What is MSW (Machine Status Word) in 80386 ? Draw its format.

**SPPU : May-02,12,16, Dec.-17, Marks 6**

12. Explain all control registers ( $CR_0$  -  $CR_3$ ) of 80386DX. Is there any bit in any of the control register to identify whether 80287/80387 math-coprocessor is in the system.

**SPPU : May-04, Marks 8**

13. Explain control registers used in 80386.

**SPPU : Nov.-12, Dec.-17, May-09, 16, Marks 10**

14. Explain control register set of 80386 with their formats. **Dec.-10, Marks 10**
15. Explain the debug registers of 80386 processor with their formats.

**SPPU : Dec.-11, May-13, Marks 10**

16. What is significance of debug registers ? Explain  $DR_6$  and  $DR_7$ . **SPPU : May-13, Marks 6**
17. Give the significance of ET. **SPPU : May-11,13, Marks 2**
18. Give the significance of EM. **SPPU : May-11, Marks 2**
19. Give the significance of MP. **SPPU : May-11, Marks 2**
20. Give the significance of VM. **SPPU : May-11,13, Marks 2**
21. What is the use of direction flag ? **SPPU : May-17, Marks 2**
22. Draw and explain the system address and system segment registers. **SPPU : May-17, Marks 4**
23. What is the use of interrupt flag. **SPPU : May-18, Marks 2**

24. What is the role of DR0 to DR3 registers in debugging ? Explain task switch breakpoint.

SPPU : Dec.-18, Marks 4

25. With the help of neat diagram explain format of DR6 register.

SPPU : Dec.-18, Marks 6

26. List and explain control registers of 80386.

SPPU : May-19, Marks 4

27. Draw and explain EFLAGS register of 80386.

SPPU : Dec.-19, Marks 4

### 3.9 Systems Instructions

Systems instructions deal with such functions as :

#### 1. Verification of pointer parameters :

ARPL - Adjust RPL

LAR - Load Access Rights

LSL - Load Segment Limit

VERR - Verify for Reading

VERW - Verify for Writing

#### 2. Addressing descriptor tables :

LLDT - Load LDT Register

SLDT - Store LDT Register

LGDT - Load GDT Register

SGDT - Store GDT Register

#### 3. Multitasking :

LTR - Load Task Register

STR - Store Task Register

#### 4. Coprocessing and multiprocessing :

CLTS - Clear Task-Switched Flag

ESC - Escape instructions

WAIT - Wait until coprocessor not Busy

LOCK - Assert Bus-Lock Signal

#### 5. Input and output:

IN - Input

OUT - Output

INS - Input String

OUTS - Output String

#### **6. Interrupt control :**

CLI - Clear Interrupt-Enable Flag

STI - Set Interrupt-Enable Flag

LIDT - Load IDT Register

SIDT - Store IDT Register

#### **7. Debugging :**

MOV - Move to and from debug registers

#### **8. TLB testing :**

MOV - Move to and from test registers

#### **9. System control:**

SMSW - Set MSW

LMSW - Load MSW

HLT - Halt Processor

MOV - Move to and from control registers.

The instructions SMSW and LMSW are provided for compatibility with the 80286 processor. 80386 programs access the MSW in CR0 via variants of the MOV instruction. HLT stops the processor until receipt of an INTR or RESET signal.

#### **Review Question**

1. *List various system instructions.*



## **UNIT - III**

**4**

# **Memory Management**

### **Syllabus**

*Global Descriptor Table, Local Descriptor Table, Interrupt Descriptor Table, GDTR, LDTR, IDTR. Formats of Descriptors and Selector, Segment Translation, Page Translation, Combining Segment and Page Translation.*

### **Contents**

4.1	Address Translation Overview	
4.2	Segment Translation	..... <b>May-01,06,07,08,09,10, 11,12,13,14,17,18,19, Dec.-2000,02,10,11,13,14,18, Nov.-12 .....</b> Marks 10
4.3	Page Translation	..... <b>Dec.-2000,05,06,07,10,17,19, May-2000,01,02,04,05,06,07, 10,11,12,18, Nov.-12 .....</b> Marks 18
4.4	Combining Segment and Page Translation	..... <b>May-02, 12, Dec.-10, 18, .....</b> Marks 8

## 4.1 Address Translation Overview

- In real mode, 80386's segmentation unit shifts the selector left four bits and adds the result to the offset to form the physical address.
- In protected mode, the 80386 transforms logical addresses (i.e., addresses as viewed by programmers) into physical address (i.e., actual addresses in physical memory) in two steps :
  - Segment translation**, in which a logical address (consisting of a segment selector and segment offset) are converted to a linear address.
  - Page translation**, in which a linear address is converted to a physical address. This step is optional, at the discretion of systems-software designers.
- In protected mode every segment selector contents of segment register has a linear base address associated with it, and it is stored in the segment descriptor.

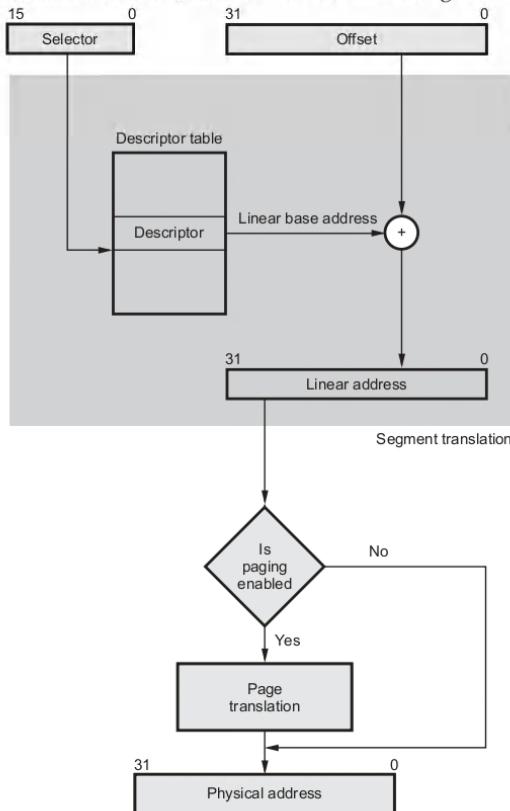


Fig. 4.1.1 Address translation overview

- A selector is used to point a descriptor for the segment in a table of descriptors.
- The linear base address from the descriptor is then added to the 32-bit offset to generate the 32-bit linear address. This process is known as **segmentation** or **segment translation**.
- If paging unit is not enabled then the 32-bit linear address corresponds to the physical address. But if paging unit is enabled, paging mechanism translates the linear address space into the physical address space by **paging translation**. This is illustrated in Fig. 4.1.1.
- The following sections describe the segmentation and paging mechanism in detail.

### Review Question

1. Draw and explain the protected mode addressing mechanism.

## 4.2 Segment Translation

GTU : May-01,06,07,08,09,10,11,12,13,14,17,18,19 Dec.-2000,02,10,11,13,14,18, Nov.-12

- Segment translation is a process of converting logical address into a linear address.

### 4.2.1 Selector

- Fig. 4.2.1 shows the segmentation mechanism. It shows how selector is used to access a descriptor in a descriptor table.

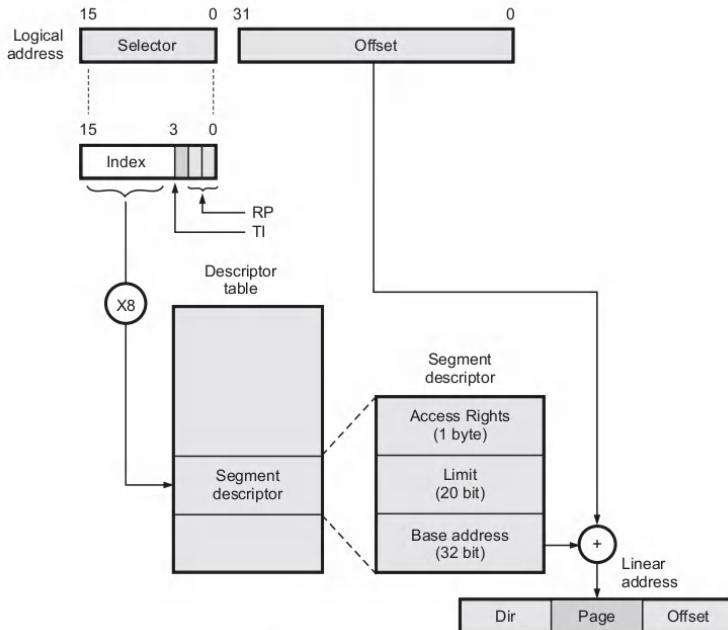
#### 4.2.1.1 Index Part

- The 13-bit index part of selector is multiplied by 8 and used as a pointer to the desired descriptor in a descriptor table.
- The index value is multiplied by 8 because each descriptor requires 8 bytes in the descriptor table.
- The descriptor in the descriptor table contains mainly **base address**, **segment limit** and **access right byte**.
- The 80386 adds the base address from the descriptor to the effective address or offset to generate a linear address.

#### 4.2.1.2 Requester's Privilege Level (RPL)

- As shown in the Fig. 4.2.1, the selector component of each logical address contains 2 bits which represent the privilege level of the program section requesting access to a segment.
- Level 0 is the most privileged and level 3 is the least privileged. More privileged levels are numerically smaller than less privileged levels.

- The descriptor of each segment contains 2 bits which represent the privilege level of that segment.
- When an executing program attempts to access a segment, the memory management unit compares the privilege level in the selector with the privilege level in the descriptor.
- If the segment selector has the same or greater privilege level, then the memory management unit allows the segment to be accessed.
- If the selector privilege level is lower than the privilege level of the segment, the memory management unit denies the access and sends an interrupt signal to the CPU indicating a privilege level violation.



**Fig. 4.2.1 Segmentation mechanism**

#### 4.2.1.3 Table Indicator (TI)

- Table Indicator (TI) bit decodes which descriptor table should be referred by the selector.

#### 4.2.2 Global Descriptor Table (GDT) and Local Descriptor Table (LDT)

- There are two major categories of descriptor table in a 80386 system :

- Global and
- Local.
- The Global Descriptor Table (GDT) is a general purpose table of descriptors, can be used by all programs to reference segments of memory. Whereas a Local Descriptor Table (LDT) are set up in the system for individual task or closely related group of tasks.
- Table Indicator (TI) bit in the selector decides which descriptor table should be referred by the selector. When TI bit is 0, the index portion of the selector refers to a descriptor in the GDT. When TI bit is 1, it refers to descriptor in the current LDT. This is illustrated in Fig. 4.2.2.
- Fig. 4.2.2 shows that the first entry in the GDT is reserved by the processor and should be all zeros. This is known as the **NULL descriptor**.
- The processor does not cause an exception when a segment register (other than CS or SS) is loaded with a null selector. However, it will cause an exception when the

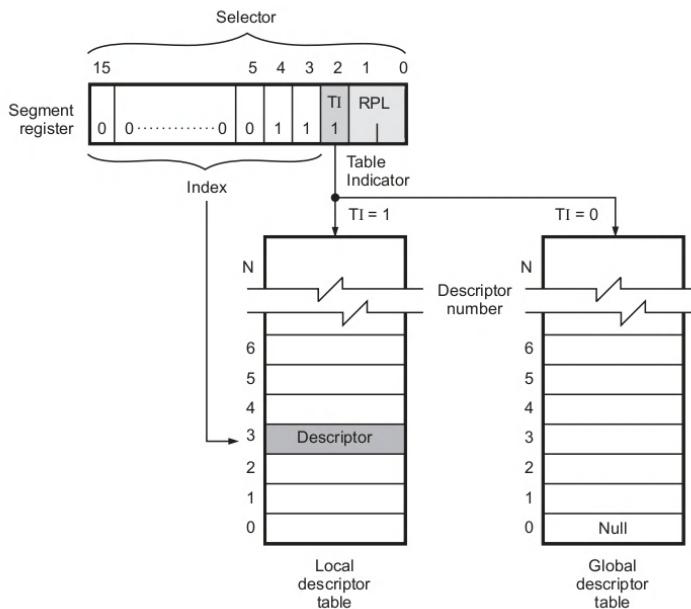


Fig. 4.2.2 Selector and descriptor tables

segment register is used to access memory. This feature is useful for initialising unused segment registers so as to trap accidental references.

#### 4.2.3 Segment Descriptor

- In protected mode, Memory Management Unit (MMU) uses the segment selector to access a descriptor for the desired segment in a table of descriptors in memory.
- Segment descriptor is a special structure which describes the segment. Exactly one segment descriptor must be defined for each segment of the memory.
- Descriptors are eight type quantities which contain attributes about a given region of linear address space (i.e. a segment). These attributes include the **32-bit base linear address** of the segment, the **20-bit length** and **granularity** of the segment, the **protection level**, read, write or execute **privileges**, the **default size of the operands** (16-bit or 32-bit), and the **type of segment**.

#### 4.2.4 General Format of Descriptor

- Fig. 4.2.3 shows the general format of a descriptor. As shown in Fig. 4.2.3, segment descriptor has following fields.
- Base** : It contains the 32-bit base address for a segment. Thus defines the location of the segment within the 4 gigabyte linear address space. The 80386

Bytes	31	23	15	Access Right Byte		0
4	SEGMENT BASE 15 .... 0				SEGMENT LIMIT 15 .... 0	
+ 4	BASE 31 .... 24	G	X	0	A V L	LIMIT 19.... 16
8	P	DPL 7	S 6	Type 3 2 1 A 0	BASE 23 .... 16	4

BASE	Base address of the segment
LIMIT	The length of the segment
P	Present bit : 1 = Present 0 = Not present
DPL	Descriptor privilege Level 0 - 3
S	Segment descriptor : 0 = System descriptor 1 = Code or Data segment descriptor
TYPE	Type of segment
A	Accessed bit
G	Granularity bit : 1 = Segment length is page granular 0 = Segment length is byte granular
0	Bit must be zero (0) for compatibility with future processors
AVL	Available field for user or OS

**Note :**

In a maximum - size segment (i.e. a segment with G = 1 and segment limit 19 .... 0 = FFFFFH), the lowest 12 bits of the segment base should be zero. (i.e. segment base 11 .... 000 = 000H).

Fig. 4.2.3 General segment descriptor format

concatenates the three fragments of the base address to form a single 32-bit address.

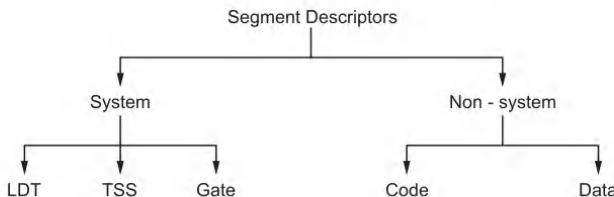
- **Limit :** It defines the size of the segment. The 80386 concatenates the two fragments of the limit field to form a 20 bit value. The 80386 interprets this 20-bit value in two ways, depending on the setting of the granularity bit (G) :
  - If G bit 0 : In units of one byte, to define a limit of up to 1 M byte ( $2^{20}$ )
  - If G bit 1 : In units of 4 kilobytes, to define a limit of up to 4 gigabytes.
- **Granularity Bit :** It specifies the units with which the limit field is interpreted. When bit is 0, the limit is interpreted in units of one byte; otherwise limit is interpreted in units of 4 Kbytes.
- **0 (Reserved by Intel) :** It neither can be defined nor can be used by user. This bit must be zero for compatibility with future processors.
- **AVL/U (User Bit) :** This bit is completely undefined, and 80386 ignores it. This is available field/bit for user or operating system.
- **Access rights byte :**
- **P (Present Bit) :** The present P bit is 1 if the segment is loaded in the physical memory, if P = 0 then any attempt to access this segment causes a not present exception (exception 11).
- **DPL (Descriptor Privilege Level) :** It is a 2-bit field defines the level of privilege associated with the memory space that the descriptor defines - DPL<sub>0</sub> is the most privileged whereas DPL<sub>3</sub> is the least privileged.
- **S (System Bit) :** The segment S bit in the segment descriptor determines if a given segment is a system segment or a code or a data segment. If the S bit is 1 then the segment is either a code or data segment, if it is 0 then the segment is system segment.
- **Type :** This specifies the specific descriptors among various kinds of descriptors. (Detail explanation is given in the following sections).
- **A (Accessed Bit) :** The 80386 automatically sets this bit when a selector for the descriptor is loaded into a segment register. This means that 80386 sets accessed bit whenever a memory reference is made by accessing the segment.

#### Points to remember about segment descriptor

- Describes a segment.
- Must be created for every segment.
- Is created by the programmer.
- Determines a base address of the segment (32-bit)
- Determines a size of the segment using limit field (20-bit)
- Determines a type of the segment. (4-bits)
- Determines a privilege level of the segment. (2-bits)
- Whether segment is physically present (1 bit)
- Whether segment has been accessed before (1 bit)
- Granularity of limit field (1 bit)
- Size of operands within segment (1 bit)
- Decides default operand size (1 bit)

#### 4.2.5 Types of Segment Descriptors and Their Formats

- Fig. 4.2.4 shows the types of segment descriptors. As shown in the Fig. 4.2.4, there are two main categories of segments. **System segments** and **non-system segment**. These two basic types are further categorised into five types.



**Fig. 4.2.4 Types of segment descriptors**

##### 4.2.5.1 Non-system (Code and Data) Segment Descriptor

- The code and data segment descriptors are the non-system segment descriptors.
- Fig. 4.2.5 shows the general format for code and data segment descriptor and Table 4.2.1 illustrate how the specific bits in the access right byte are interpreted in data and code segment descriptors.
  - **B-Bit (Big) :** The B-bit of **data segment descriptor** determines the size of stack pointer. B = 0 : size : 16-bit (SP) B = 1: 32-bit (ESP).

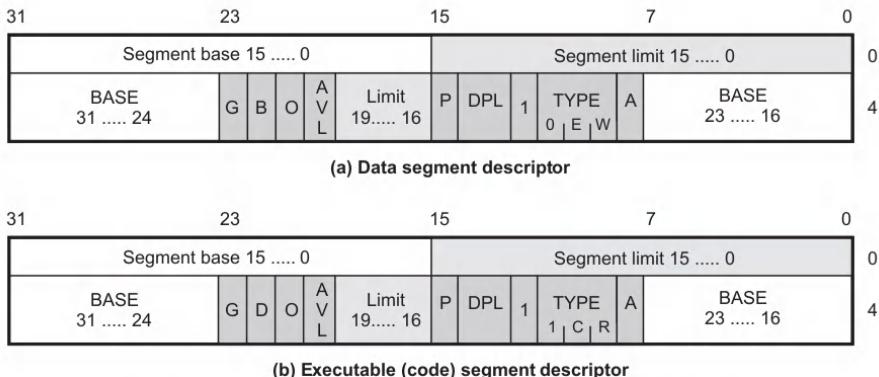


Fig. 4.2.5 General format for data and code segment descriptors

- D-Bit (Default size)** : This bit of **code segment descriptor** determines the default size of operand and default size of address for the instructions of a code segment. D = 0 : size : 16-bits D = 1 : size : 32-bits.

Bit Position	Name	Function	
		P = 1	Segment is mapped into physical memory.
7	Present (P)	P = 0	No mapping to physical memory exists, base and limit are not used.
			Segment privilege attribute used in privilege tests.
6-5	Descriptor Privilege Level (DPL)	S = 1	Code or Data (includes stacks) segment descriptor
		S = 0	System segment descriptor or Gate descriptor
4	Segment Descriptor (S)	E = 0	Descriptor type is data segment;
		ED = 0	Expand up segment, offsets must be $\leq$ limit.
3	Executable (E)	ED = 1	Expand down segment, offsets must be $>$ limit.
		W = 0	Data segment may not be written into.
2	Expansion Direction (ED)	W = 1	Data segment may be written into.
			Data segment
1	Writeable (W)		

Note : If data segment (S = 1, E = 0)				
3	Executable (E)	E = 1	Descriptor type is code segment;	Code segment
2	Conforming (C)	C = 1	Code segment may only be executed when CPL $\geq$ DPL and CPL remains unchanged.	
1	Readable (R)	R = 0	Code segment may not be read.	
		R = 1	Code segment may be read.	
Note : If code segment (S = 1, E = 1)				
0	Accessed (A)	A = 0	Segment has not been accessed.	
		A = 1	Segment selector has been loaded into segment register or used by selector test instructions.	

Table 4.2.1 Access rights for segments

- The **Executable (E) bit** indicates whether segment is code or data segment. If E bit is 1, segment is code segment otherwise segment is data segment.
- The code segment may be executable, or executable and read. This is determined by **Readable (R) bit**. If R bit is 1, code segment is executable and readable otherwise it is only executable.
- If **Conforming Bit (C)** is 1, code segment can be executed and shared by programs at different privilege levels.
- In case of stack segment, segment starts at the base linear address plus the maximum segment limit, whereas data segment start at the base linear address and expand to the base linear address plus limit as shown in Fig. 4.2.6.

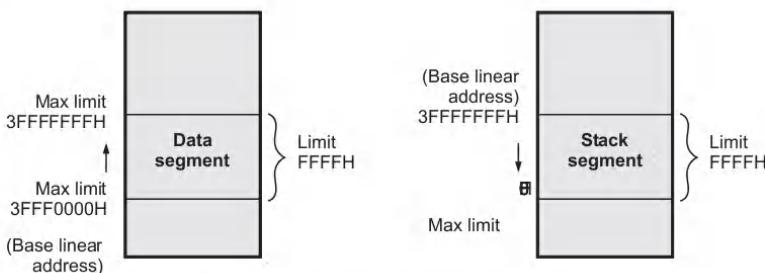


Fig. 4.2.6 Expansion direction for data and stack segment

- The **Expansion Direction (ED)** bit specifies expansion direction for the segment. If ED = 0, expansion direction is upward which is data segment and if ED = 1, expansion direction is downwards which is stack segment.

- The Write (W) bit for data segment indicates whether the data segment is read only, or read and write. If W bit 0, data segment is read only; otherwise it is read/write segment. For stack segment W bit must be logic 1.

#### 4.2.5.2 System Segment Descriptors

- System segments gives the information of operating system tables, tasks and gates. Fig. 4.2.7 shows the general format of system segment descriptor.
- From Fig. 4.2.7 it can be seen that several descriptor fields (Base address, limit, Granularity bit G and Present bit P) are similar to the general segment descriptor.
- Fig. 4.2.7 also shows the various types of system segment descriptors. Let us discuss the various system segment descriptors.

The diagram illustrates the structure of a System Segment Descriptor. It consists of two main parts: a bit-field layout and a corresponding table of type definitions.

**Bit Field Layout:**

31	Segment Base 15 .... 0								Segment Limit 15 .... 0								0
31 .... 24	G	X	0	A	V	L	Limit 19.... 16	P	DPL	0	Type	Base 23 .... 16	+4				

**Type Definitions:**

Type	Defines	Type	Defines
0	Reserved by Intel	8	Reserved by Intel
1	Available 80286 TSS	9	Available intel 80386DX TSS
2	LDT	A	Undefined (Intel reserved)
3	Busy 80286 TSS	B	Busy intel 80386DX TSS
4	80286 call gate	C	Intel 80386DX call gate
5	Task gate (for 80286 or Intel 80386DX task)	D	Undefined (Intel reserved)
6	80286 Interrupt gate	E	Intel 80386DX interrupt gate
7	80286 Trap gate	F	Intel 80386DX trap gate

Fig. 4.2.7 System segment descriptor

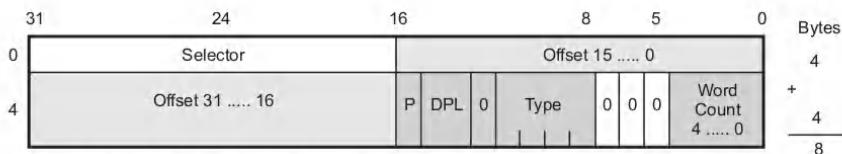
**a) LDT descriptors (S = 0, Type = 2) :** The LDT descriptors are present only in the Global Descriptor Table (GDT). They contain the information about the local descriptor tables. The local descriptor tables contains the segment descriptors which are unique to a particular task. The DPL (Descriptor Privilege field) of this descriptor is ignored because it can be accessed with only privilege level 0.

**b) TSS descriptor (S = 0, Type = 1, 3, 9, B) :** In a multitasking environment computer performs more than one task at a time, and it also switch between the task. A task can be a single program, or it can be a group of related programs. When it switches from task 1 to task 2, it stores all the information necessary to restart the task1 later in time exactly as it was left. It involves saving the contents of all of the processor registers as well as any read/write memory variables and the address of next instruction to be executed. Such information is called **state of the task** or **context of the task**.

- The 80386 uses a special segment called **Task State Segment** (TSS) to store the state/context of the task. This segment can be addressed with the help of Task State Segment (TSS) descriptor. The TSS descriptor contains information about the location, size and privilege level of a TSS.
- Along with the context of the task, the TSS also contains the **linkage field** for the next task which allows the nesting of tasks. The TSS descriptor gives base address and limit for TSS. Its TYPE field is used to indicate whether task is currently BUSY (i.e. on a chain of active tasks) or the TSS is available. The TYPE field also indicates if the segment contains a 80286 or an 80386DX TSS.

**c) Gate descriptors (S = 0, TYPE = 4 – 7, C, F) :** A gate is a special type of descriptor. It allows the 80386 to automatically perform protection checks. There are four types of gate descriptors :

- Call gate
- Task gate
- Interrupt gate
- Trap gate
- Call gates are used to change privilege levels. Task gates are used to perform a task switch and interrupt and trap gates are used to specify interrupt service routines. Fig. 4.2.8 shows the format of the four types of gate descriptors.



D word count : The number of double words to copy from caller's stack to the called procedure's stack. Only used with call gate.

Destination Selector : (16 - bit)      Selector to the target code segment or  
    Selector to the target task state segment for task gate

Destination Offset : (32 - bit)      Entry point within the target code segment

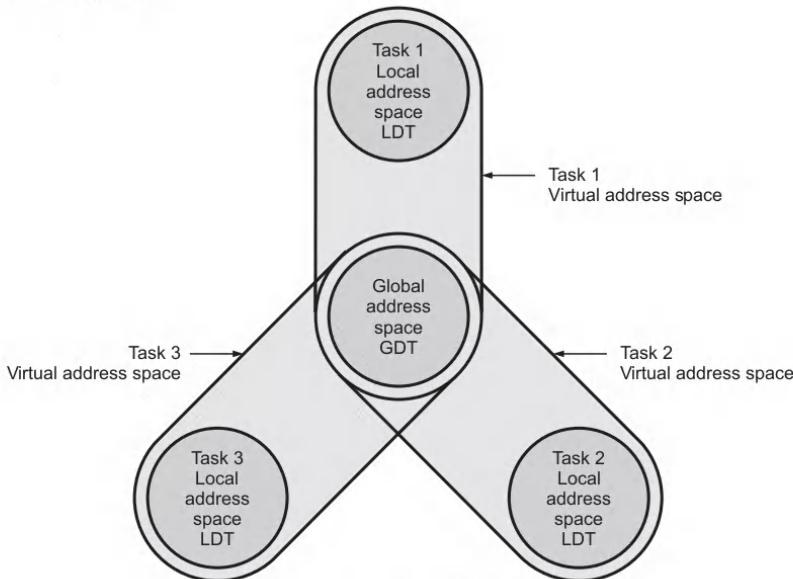
**Fig. 4.2.8 Gate descriptor formats**

#### 4.2.6 Descriptor Tables - GDT, IDT and LDT

- As mentioned earlier, segment descriptors are grouped and placed one after the other in contiguous memory locations. This group arrangement is known as a **descriptor table**.
- The maximum limit for the length of descriptor table is 64 kbytes and we know that each descriptor takes 8 bytes to store the information of a particular segment.

So descriptor table can have as many as 8192 descriptors. The upper 13 bits of a selector are used as an index into the descriptor table.

- There are three types of descriptor tables :
  - **The Global Descriptor Table (GDT)** : It is a general purpose table of descriptors, can be used by all programs to reference segments of memory. The GDT can have any type of segment descriptor except for descriptors which are used for serving interrupts.
  - **The Interrupt Descriptor Table (IDT)** : It holds the segment descriptors that define interrupt or exception handling routines. The IDT is a direct replacement for the interrupt vector table used in 8086 system.
  - **A Local Descriptor Tables (LDT)** : They are set up in the system for individual task or closely related group of tasks.
- Fig. 4.2.9 shows how tasks use its individual memory area defined by the descriptors from the corresponding local descriptor table and how it shares the memory area defined by the descriptors from the global descriptor table.
- As we know, the descriptors are stored in the descriptor tables. But it is important to know that where these tables are stored ? It is possible to place descriptor tables anywhere in the processor's address space and it is not necessary to keep them together.



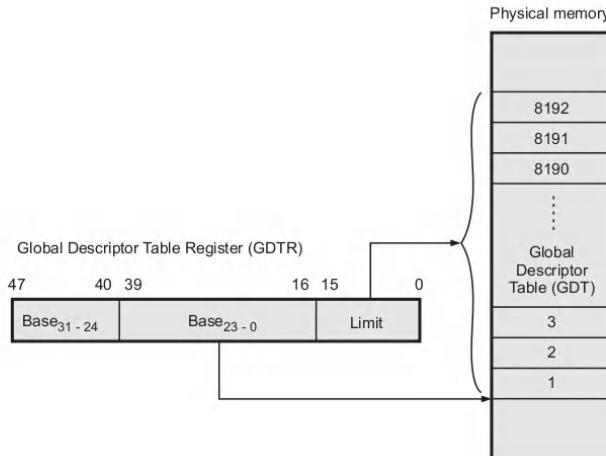
**Fig. 4.2.9 Memory area shared by different tasks**

## 4.2.7 Descriptor Registers - GDTR, LDTR and IDTR

- Each of the tables has a register associated with it the GDTR, the LDTR and the IDTR.
- Each of these register contains the **32-bit linear address** of the base of its descriptor table and the **table's limit**. The base address of a descriptor table is the linear address of the first byte of the first descriptor in the table. The limit specifies how long the table is and therefore how many descriptors it has.

### 4.2.7.1 Global Descriptor Table Register (GDTR)

- Fig. 4.2.10 shows how the contents of the global descriptor table register are used to define a Global descriptor table in the 80386DX physical memory address space.
- GDTR is a 48-bit register located inside the 80386DX.



**Fig. 4.2.10 GDTR and GDT**

- The lower two bytes of this register specifies the LIMIT, (in bytes) for the GDT. The value of limit is one less than the actual size of the table. For example, if LIMIT is 03FFH then the table is 1024 (1023 + 1) bytes in length ( $03FFH = 1023_{10}$ ). Since the LIMIT field is 16 bit long, the GDT can grow up to 65,536 bytes long.
- The upper four bytes of GDTR specifies the 32-bit linear address of the base of the Global Descriptor Table (GDT).

#### 4.2.7.2 Interrupt Descriptor Table Register (IDTR)

- Like global descriptor table register, interrupt descriptor table register holds the 16-bit limit and 32-bit linear address of the base of the Interrupt Descriptor Table (IDT).
- Fig. 4.2.11 shows how the contents of the Interrupt Descriptor Table Register are used to define a Interrupt Descriptor Table (IDT) in the 80386DX physical memory address space.

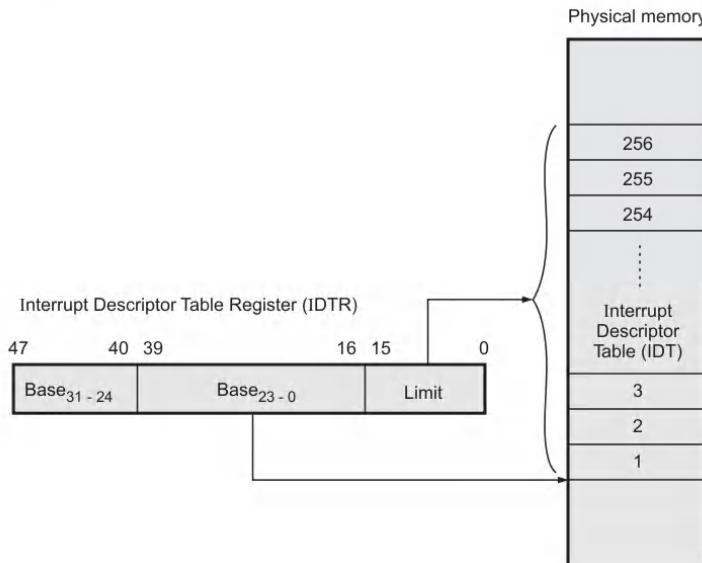


Fig. 4.2.11 IDTR and IDT

- Like GDTR, the IDTR is also 48 bit in length, with lower two bytes defines Limits and upper 4 bytes defines the base address. Since limit field is two bytes, the IDT can also be up to 65,536 bytes long. But the 80386DX only supports upto 256 interrupts or exceptions; therefore, the size of the IDT should not be set to support more than 256 interrupts.

#### 4.2.7.3 Local Descriptor Table Register (LDTR)

- Unlike GDTR and IDTR, the **LDTR is a 16-bit register**. It does not specify any limit or base address for the segment but it specifies the address of the LDT descriptor stored in the Global Descriptor Table (GDT).
- Fig. 4.2.12 shows LDTR, GDT and LDT.

- It shows how contents of LDTR are used indirectly to define a Local Descriptor Table.
- LDTR holds a selector that points to an LDT descriptor in the GDT. Whenever a

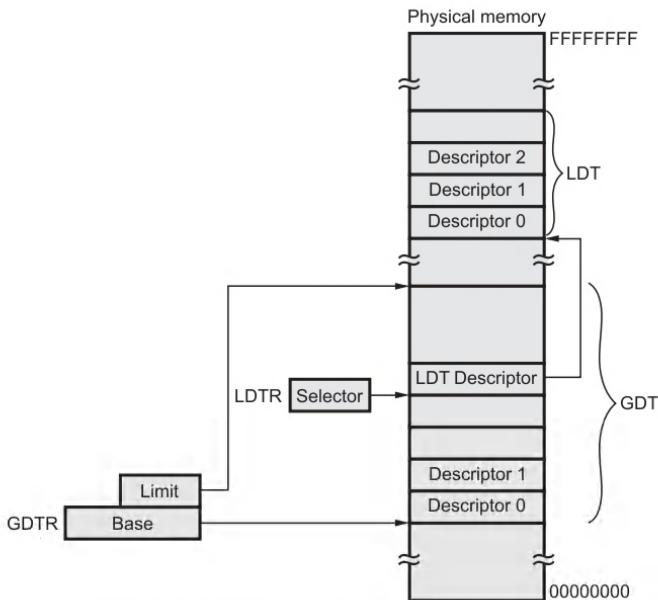


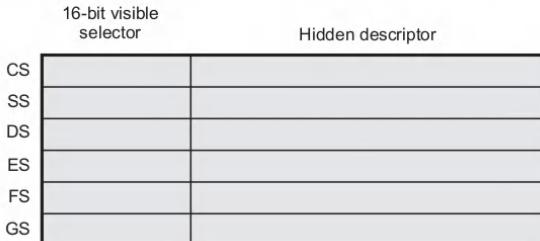
Fig. 4.2.12 Global and local descriptor tables

selector is loaded into the LDTR, the corresponding descriptor is located in the global descriptor table.

- The contents of this descriptor defines the local descriptor table. The 32-bit base value defines starting point of the table in the 80386DX physical memory address space and 16-bit limit specifies the size of the table.
- The GDT can contain many LDT descriptors. To put particular LDT in service, it is necessary to load the LDTR with corresponding selector.
- For loading the values in GDTR, IDTR and LDTR registers, 80386DX provides LGDT, LLDT, and LIDT instructions. It also provides SGDT, SLDT and SIDT instructions. These (48 bits) instructions copy the contents of the descriptor table registers into the six bytes of memory pointed by the destination operand.
- These tables are manipulated by the operating system. Thus, the instructions used for loading the descriptor tables are privileged instructions.

#### 4.2.8 Segment Registers and Segment Descriptor Cache

- The segment register contents are used as a selector to select specific descriptor from the descriptor table. This part of the segment register is visible to programmer.
- Fig. 4.2.13 shows complete segment register with visible and hidden part of it. The hidden part is referred to as **segment descriptor cache register**.



**Fig. 4.2.13 Segment register and segment descriptor cache**

- Using these registers 80386DX stores information from descriptor, thereby avoiding the need to consult a descriptor table every time it accesses memory.
- Segment register (visible portion) contents are manipulated by programs whereas segment descriptor cache register (hidden portion) contents are manipulated by processor.
- Once the descriptors are cached, subsequent references to them are performed without any overhead for loading of the descriptor. This is the biggest advantage of segment descriptor cache registers.

**Example 4.2.1** Assume  $(DS) = 0204H$   $[ESI] = 00002000H$ . Paging is disabled and mode is protected mode.

- From which of the three descriptors (IDT, LDT, GDT) the descriptor will be considered ? Give the descriptor number.
- Assume appropriate values in the descriptor selected and explain how the address translation takes place when the following instruction is executed.  
`MOV AX, [ESI]`

**Solution :** 1. Here, DS register is used as a selector. Fig. 4.2.14 shows the definitions of the selector bits and the contents of DS are 0204H.

**Fig. 4.2.14**

From the figure we can see that

$$\text{RPL} = 00$$

$$\text{TI} = 1$$

Since TI (Table Indicator) bit is set, the descriptor from the current LDT will be referred.

2. The descriptor gives the segment base address and segment limit. Let us assume segment base address = 0000 0000 H and limit = FFFFFH. As paging is disabled, the physical address of memory is given by

$$\text{PA} = \text{Base address} + \text{Offset}$$

**Note :** Offset < segment limit.

In our case offset is given by ESI (0000 2000H), which is within the limit i.e. less than segment limit. Therefore the physical address of memory,

$$\begin{aligned}\text{PA} &= 0000\ 0000\ \text{H} + 0000\ 2000\ \text{H} \\ &= 0000\ 2000\ \text{H}\end{aligned}$$

When MOV AX, [ESI] instruction is executed the contents from memory location 0000 2000H are copied into AL register and contents from memory location 0000 2001H are copied into AH register.

### Review Questions

1. Explain segment translation in 80386DX.
2. What do you mean by RPL ?
3. How virtual address is translated into physical address if paging is disable in 80386.
4. Draw the general segment descriptor format. Explain the significance of each field in it.

5. What is the function of granularity bit ?
6. State how the granularity bit affects the limit field.
7. What is the function of access right byte.
8. What are the different types of descriptors ?
9. Draw the general format for data and code segment descriptors and explain the significance of each field in it.
10. Explain the meaning and usage of 'Expand down' segments. How are the base and limit fields interpreted for these segments ?
11. Draw and explain the general format of system segment descriptor.
12. Draw and explain the gate descriptor format.
13. What is the purpose of word count bit in call gate descriptor ?
14. How many global description can be stored in GDT ? Justify.
15. Explain the purpose, structure and locations of various descriptor tables used in 80386.
16. What do you mean by segment descriptor cache ? Explain the use of it.

**SPPU : Dec.-13, Marks 3**

17. Explain memory segmentation of 80386 microprocessor. **SPPU : Nov.-12, Marks 8**
18. How CS, SS, DS, ES differ from real mode functionally ? **SPPU : Dec.-02, Marks 4**
19. Write difference between real and protected mode of 80386 with respect to : Memory segmentation. **SPPU : May-12, 13, Marks 4**
20. Explain the working of segment selector in protected mode operation of 80386 processor. **SPPU : Dec.-11, Marks 8**
21. How are descriptors used to access the operand ? **SPPU : May-06, Marks 4**
22. What are various types of descriptors ? **SPPU : May-06, Marks 2**
23. What is LDT descriptor ? **SPPU : May-14, Marks 4**
24. Explain the code segment descriptor format in 80386 processor. **SPPU : Dec.-11, Marks 8**
25. What is an interrupt descriptor ? **SPPU : May-07, Marks 2**
26. What do you understand by descriptor table ? Which are different types of descriptor tables in 80386 ? Explain. **SPPU : May-08, Marks 8**
27. What is IDT ? Which are different descriptors present in IDT ? **SPPU : May-09, 11, Marks 6**
28. Explain IDT of 80386 in detail with diagram and format. **SPPU : Dec.-10, Marks 8**
29. What is the difference between GDT and LDT. **SPPU : Dec.-14, Marks 10**

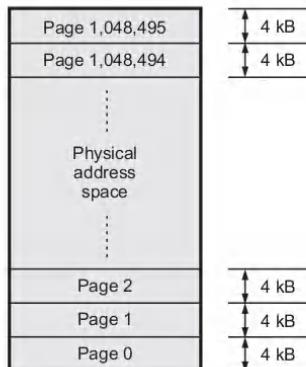
30. List and explain the different descriptor table register used in protected mode of 80386 processor. **SPPU : Dec.-11, Marks 10**
31. What is significance of global descriptor table registers ? Explain with diagram. **SPPU : May-13, Marks 4**
32. What is an IDTR ? What are its contents following reset of 80386 ? **SPPU : May-01, Marks 3**
33. Specify size and function of IDTR. **SPPU : May-10, Marks 2**
34. What is the significance of interrupt descriptor table registers ? Explain with diagram. **SPPU : May-13, Marks 4**
35. The GDTR, IDTR are 48 bits with base address and limits specified but the LDTR is only 16 bits. How does this then access the memory ? **SPPU : Dec.-2000, Marks 3**
36. Specify size and function of LDTR. **SPPU : May-10, Marks 2**
37. What is the significance of local descriptor table registers ? Explain with diagram. **SPPU : May-13, Marks 4**
38. How to convert logical address to linear address when 80386 is operating in PM mode. Explain necessary registers used for the same. **SPPU : May-09, Marks 10**
39. Draw and explain how 80386 microprocessor translates logical address into linear address. **SPPU : Nov.-12, Marks 10**
40. Explain segment address translation in detail. **SPPU : May-17, Marks 4**
41. Draw and explain segment descriptor. **SPPU : May-17,18, Marks 6**
42. With the help of neat diagram explain how logical address is converted into physical address ? Assume paging mechanism is disabled. **SPPU : Dec.-18, Marks 6**
43. Draw and explain the format of a selector. **SPPU : May-19, Marks 2**
44. With help of diagram explain the 80386 mechanism to translate logical address to linear address. **SPPU : May-19, Marks 6**

### 4.3 Page Translation

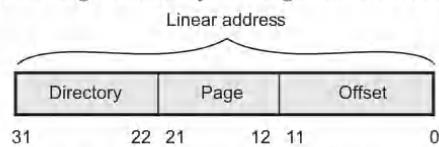
**SPPU : Dec.-2000,05,06,07,10,17,19, May-2000,01,02,04,05,06,07,10,11,12,18, Nov.-12**

- Page translation is the second phase of address translation. In this phase 80386 transforms a linear address generated by segment translation into a physical address.
- The page translation step is optional. Page translation is in effect only when the PG bit of CR0 is set.
- Page translation is must if the operating system is to implement multiple virtual 8086 tasks, page-oriented protection, or page oriented virtual memory.

- When paging is enabled, the paging unit arranges the physical address space into 1,048,496 pages that are each 4096 bytes long. Fig. 4.3.1 shows organization of physical address space using paging.
- There are three components to the paging mechanism of the 80386DX.
- Page directory, page tables, and page itself (page frame or page).
- Like segmentation, paging depends on special memory resident tables. Out of three components, page directory and page tables are in the table form. Both are made up of 32-bit descriptors.
- Unlike tables of segment descriptors, each page directory or page table must contain exactly 1024 descriptors, making each directory or table exactly 4096 bytes (4 kB) long.
- A page frame is a 4 kbyte unit of contiguous addresses of physical memory.
- When paging is enabled the linear address generated by the segment translation process is not used as a physical address. The 80386DX uses two levels of tables to translate the linear address (from the segment translation) into a physical address.
- Fig. 4.3.2 shows the format of linear address. Processor internally divides a linear address into three fields : Two fields of 10 bits each and one field of 12 bits.
- The most significant 10 bits (**DIR field**) of the linear address are used as an index into a page directory. The next most significant 10 bits (**PAGE field**) of the linear address are used as an index into the page table determined by the page directory. The least significant 12 bits (**OFFSET**) select one of 4096 bytes of memory from the page frame determined by the page table.
- The physical address of the current page directory is stored in the control register (CR3) which is also referred to as **Page Directory Base Register (PDBR)**.
- Fig. 4.3.3 shows how the 80386DX converts the DIR, PAGE and OFFSET fields of a linear address into the physical address by consulting two levels of page tables.
- The descriptor in a page directory is referred to as a **Page Directory Entry (PDE)** and descriptor in the page table is referred to as **Page Table Entry (PTE)**.



**Fig. 4.3.1 Paged organization of the physical address space**



**Fig. 4.3.2 Linear address format**

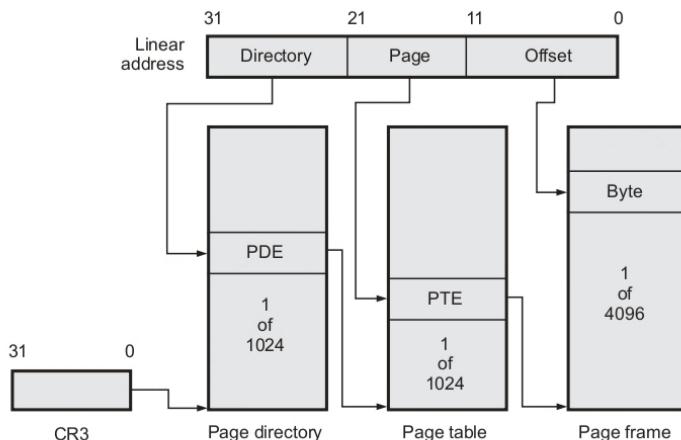


Fig. 4.3.3 Linear to physical address translation

### 4.3.1 Page Tables

- A page table is simply an array of 32-bit page specifiers. A page table is itself a page, and therefore contains 4 Kilobytes of memory or at most 1K 32-bit entries.
- Two levels of tables are used to address a page of memory.
- At the higher level is a **page directory**. The page directory addresses up to 1K **page tables** of the second level.
- A page table of the second level addresses up to 1K pages.
- All the tables addressed by one page directory, therefore, can address 1M pages ( $2^{20}$ ). Because each page contains 4K bytes ( $2^{12}$  bytes), the tables of one page directory can span the entire physical address space of the 80386 ( $2^{20}$  times  $2^{12} = 2^{32}$ ).

### 4.3.2 PDE Descriptor

- Fig. 4.3.4 shows format for page directory entry. A page directory entry is having six fields.

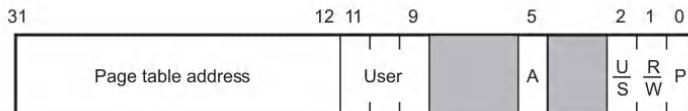


Fig. 4.3.4 Page directory entry

**Page Table Address :** The page table address specifies the physical starting address of the base of a page table. This field (page table address) specifies 20 most significant bits and remaining 12 bits are all 0's. This locates all page tables on 4 K boundaries.

**User/Avail :** Bits 9, 10, and 11 are not used by the 80386. Users are free to use them.

**Accessed Bit :** The 80386 automatically sets accessed bit whenever PDE is used in address translation or another page related function. It is never cleared unless you write code to do it manually.

**User/ Supervisor and Read/ Write Bits :** These bits are not used for address translation, but are used for page-level protection which the 80386 performs at the same time as address translation.

If User/ Supervisor bit is set, the memory pages covered by this PDE are accessible from all privilege levels. If it is cleared, the pages are accessible only by PL0, 1 and 2.

If User/ Supervisor bit is cleared Read/ Write bit has no effect. But if User/ Supervisor bit is set and read/ Write bit is 1, memory pages covered by this PDE are write protected. If Read/ Write bit is set, write privileges are allowed from PL3 code. The access rights just discussed are summarized in Table 4.3.1.

U/S	R/W	Permitted Level 3	Permitted Access Levels 0, 1, or 2
0	0	None	Read/Write
0	1	None	Read/Write
1	0	Read-Only	Read/Write
1	1	Read/Write	Read/Write

Table 4.3.1 Protection provided by R/W and U/S

#### Present :

The present bit indicates whether a page table entry can be used in address translation. P = 1 indicates that the entry can be used and page table pointed by PDE is present in the physical memory. If P = 0, the page table referred to is not present. Fig. 4.3.5 shows the format of a not present page descriptor.



Fig. 4.3.5 Not present page descriptor

### 4.3.3 PTE Descriptor

- Fig. 4.3.6 shows format for page table entry. A page table entry has seven fields.

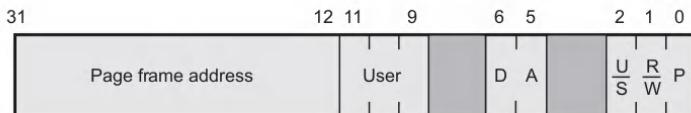


Fig. 4.3.6 Page table entry

**Page Frame Address :** The page frame address specifies the physical starting address of a 4 kB page frame or a page. This field (page frame address) specifies 20 most significant bits and remaining 12 bits are all 0's. This locates all pages on 4 K boundaries.

**User/Avail Bits :** Bits 9, 10, 11 are not used by the 80386. Users are free to use them.

**Accessed Bit :** Accessed bit is set by the 80386 whenever this PTE is used in a paging related function. The 80386 never clears this bit. User can keep track of the most often used pages of memory by periodically testing and clearing this bit in all PTEs.

**Dirty Bit :** The dirty bit is automatically set by the 80386 whenever page frame covered by PTE is written into. The 80386 never clears this bit. User can keep track of the most often written page of memory by periodically testing and clearing this bit.

**User/ Supervisor and Read/ Write Bits :** These bits are not used for address translation, but are used for page-level protection which the 80386 performs at the same time as address translation.

If User/Supervisor bit is set, the memory pages covered by this PTE are accessible from all privilege levels. If it is cleared, the pages are accessible only by PL0, 1 and 2.

If User/Supervisor bit is cleared Read/Write bit has no effect. But if User/Supervisor bit is set and Read/Write bit is 1, memory pages covered by this PTE are write protected. If Read/Write bit is set, write privileges are allowed from PL3 code.

**Present :** The present bit indicates whether a page table entry can be used in address translation. P = 1 indicates that the entry can be used and page table pointed by PTE is present in the physical memory. If P = 0, the page table referred to is not present.

#### 4.3.4 Translation Lookaside Buffer / Page Translation Cache

- The 80386DX paging mechanism is designed to support demand paged virtual memory systems. However, performance would degrade substantially if the processor was required to access two levels of tables (Page directory and page table) for every memory access.
- To solve this problem, the 80386 DX stores the most recently used page table entries in an on-chip cache. This cache is called the **Translation Lookaside Buffer (TLB)**.
- The TLB holds up to 32 page table entries. The 32-entry TLB coupled with a 4 K page size, results in coverage of 128 K bytes of memory addresses.
- Whenever program generates linear address that maps to a Page Table Entry (PTE) already in the cache, the 80386DX can use the cached information it has internally. This saves two outside memory references, improving performance in address translation.
- For many common multi-tasking systems, the TLB will have a hit rate of about 98 %. This means that the processor will only have to access the two-level page structure on 2 % of all memory accesses.
- Fig. 4.3.7 illustrates how the TLB supports the 80386DX paging mechanism.

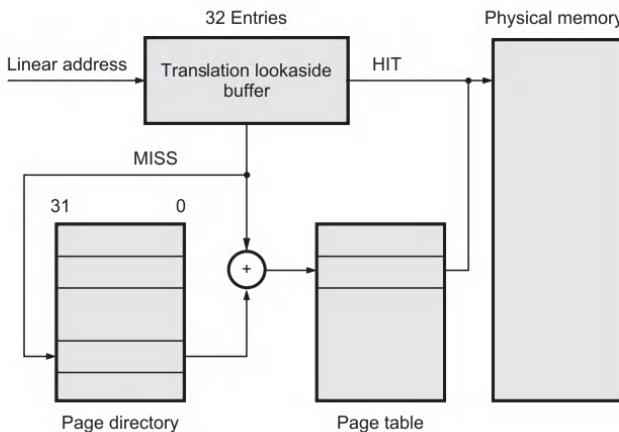


Fig. 4.3.7 Translation lookaside buffer

**Review Questions**

1. Explain with example, how 32-bit linear address is converted to physical address when paging is enabled ? Illustrate with proper example.

**SPPU : May-02, 12, Dec.-10, Marks 8**

2. What are the various fields in page directory entry and page table entry ? What are their uses ?

3. Explain the process by which 80386 translates a logical address into a physical address into protected mode.

**SPPU : Dec.-2000, Marks 4, Nov.-12, Marks 10**

4. What is logical address, physical address and linear address in 80386 ?

**SPPU : Dec.-2000, Marks 4**

5. What is the significance of segment register, global descriptor table and global descriptor table register in conversion of linear address to physical address of protected mode of 80386. Explain in detail with format of each.

**SPPU : May-11, Marks 12**

6. 80386 is operating in paged mode and has accessed some pages. You now make changes in the PTE and pages in the memory. What will happen ? Does the 80386 follow these changes ?

**SPPU : May-2000, Marks 3**

7. How 80386 translates the logical address to physical address, when paging is enables in protected mode ? Explain with the help of necessary formats of descriptors and diagrams.

**SPPU : May-02, Marks 16**

8. How linear address produced by the segmentation process is handled in paging process to get an operand ? Explain with the help of suitable diagram.

**SPPU : May-04, Marks 6**

9. With the help of suitable diagram explain the process of conversion of virtual address to physical address. Assume paging is enabled. Also explain the page translation process of 80386.

**SPPU : May-05, 10, Marks 10**

10. What do you mean by paging ? Assume that paging is enabled. Explain the virtual address translation process to access operand from the memory.

**SPPU : May-06, 07, Dec.-06, Marks 10**

11. How paging gets selected ?

**SPPU : Dec.-07, Marks 4**

12. Explain logical to physical address conversion when 80386 operating in protected mode. Draw necessary diagrams and formats.

**SPPU : Dec.-10, Marks 18**

13. What are the different registers used for paging ? Explain.

**SPPU : May-11, Marks 8**

14. What is the basic difference between segmentation and paging ?

**SPPU : May-04, Dec.-07, Marks 4**

15. Explain the translation process in protected mode from a logical address into a physical address with paging implemented. Show preferably with diagrams the translations process and data structures, tables involved. What protections and privilege checks are made ? Can you have paging without segmentation ? **SPPU : May-01, Marks 7**
16. Explain the process of bringing a page from virtual memory to physical memory. **SPPU : Dec.-07, Marks 8**
17. What is the purpose of TLB and descriptor cache ? How do they reduce system overheads ?
18. Explain the role of TLB in the process. **SPPU : May-04, Marks 2**
19. What is TLB ? How is this useful while accessing physical memory ? Draw the necessary diagram to explain your concepts. **SPPU : May-05,07, Dec.-05,06,07, Nov.-12, Marks 8**
20. Explain paging mechanism. **SPPU : Dec.-17, Marks 4**
21. Explain paging mechanism. **SPPU : May-18, Marks 4**
22. Draw and explain the 80386 address translation mechanism considering PG bit in CRO in set. **SPPU : May-18, Marks 6**
23. Explain how linear address 0080400A H will be translated into physical address using paging mechanism. Whether the address generated will be the same to linear address ? **SPPU : Dec.-19, Marks 6**
24. With the help of diagram explain the 80386 mechanism to translate logical address to linear address and linear to physical address. **SPPU : Dec.-19, Marks 6**

#### 4.4 Combining Segment and Page Translation

**SPPU : May-02, 12, Dec.-10**

- Fig. 4.4.1 (See Fig. 4.4.1 on next page) shows both the phases of address translation. It shows how logically address is converted into physical address when paging is enabled.

#### Review Questions

- Explain with example, how 32-bit linear address is converted to physical address when paging is enabled ? Illustrate with proper example. **SPPU : May-02, 12, Dec.-10, Marks 8**
- Explain how linear address is converted into physical address by 80386 memory management. **SPPU : Dec.-18, Marks 6**

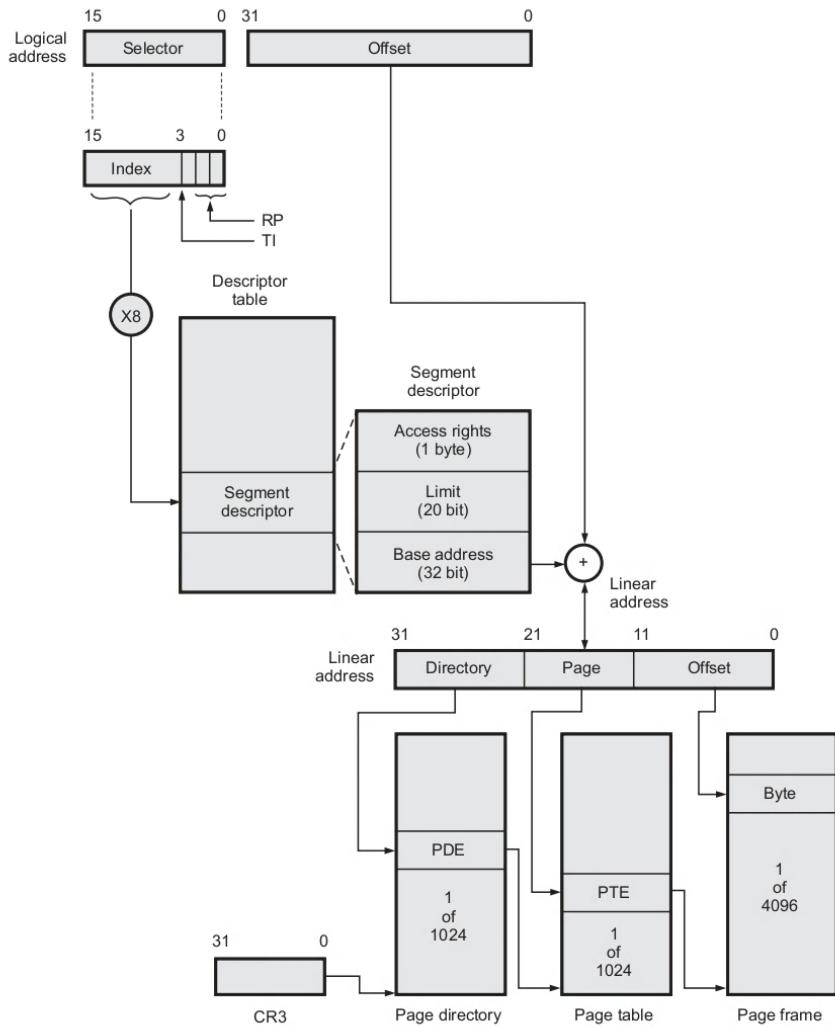


Fig. 4.4.1 Protected mode address translation



## **UNIT - IV**

**5**

# **Protection**

### **Syllabus**

*Need of Protection, Overview of 80386DX Protection Mechanisms : Protection rings and levels, Privileged Instructions, Concept of DPL, CPL, RPL, EPL. Inter privilege level transfers using Call gates, Conforming code segment, Privilege levels and stacks. Page Level Protection, Combining Segment and Page Level Protection.*

### **Contents**

5.1	<i>Need of Protection</i>	
5.2	<i>Overview of 80386DX Protection Mechanisms</i>	<b>May-18,</b> ..... Marks 2
5.3	<i>Segment Level Protection</i>	<b>May-2000,02,04,05,06,08,09,19,</b> ..... <b>10,11,12,13,15,18, Nov.-12,</b> ..... <b>Dec.-2000,02,06,</b> ..... <b>07,08,11,15,17,18,</b> ..... Marks 6
5.4	<i>Page Level Protection</i>	<b>May-11,19, Dec.-19,</b> ..... Marks 4
5.5	<i>Combining Segment and Page Level Protection</i>	
5.6	<i>I/O Protection</i>	<b>May-13, 18, Dec.-19,</b> ..... Marks 6
5.7	<i>Privilege and I/O Sensitive Instructions</i>	<b>May-2000, 02, 05, 13,</b> ..... <b>Dec.-08, 10, 17,</b> ..... Marks 8

## 5.1 Need of Protection

- Problem may occur in a multitasking operating systems or multi-user systems when two or more users attempt to read and change the contents of a memory location at the same time. The section of a program where the value of a variable is being read and changed (critical section) must be protected from access by other tasks until the operation is complete.
- Another region that requires protection is the operating system code. The incorrect address in a user program may cause program to write over the critical sections of the operating system corrupting the operating system code and data areas.
- The system then 'locks-up' and the only way to get control again is to reboot the system. In a multitasking system this is intolerable, so several methods are used to protect the operating system.

### Review Question

1. *What is the need of protection ?*

## 5.2 Overview of 80386DX Protection Mechanisms

SPPU : May-18

- The 80386 uses **segment level protection** and **page level protection** mechanisms to protect critical sections.
- Protection in the 80386 has five aspects :
  1. Type checking
  2. Limit checking
  3. Restriction of addressable domain
  4. Restriction of procedure entry points
  5. Restriction of instruction set.
- Each reference to memory is checked by the hardware to verify that it satisfies the protection criteria.
- All these checks are made before the memory cycle is started; any violation prevents that cycle from starting and results in an exception.
- Since the checks are performed concurrently with address formation, there is no performance penalty.

### Review Questions

1. *Give the overview of 80386DX protection mechanisms.*

2. *List five aspects of protection in the 80386.*

SPPU : May-18, Marks 2

## 5.3 Segment Level Protection

**SPPU : May-2000,02,04,05,06,08,09,10,11,12,13,15,18,19, Nov.-12,  
Dec.-2000,02,06,07,08,11,15,17,18**

- All five aspects of protection apply to segment translation:
  1. Type checking
  2. Limit checking
  3. Restriction of addressable domain
  4. Restriction of procedure entry points
  5. Restriction of instruction set.
- The segment is the unit of protection, and segment descriptors store protection parameters.
- Protection checks are performed automatically by the CPU when the selector of a segment descriptor is loaded into a segment register and with every segment access.
- Segment registers hold the protection parameters of the currently addressable segments.
- When an attempt is made to access a segment first of all, the 80386 checks to see if the descriptor table indexed by the selector contains a valid descriptor for that selector. If the selector attempts to access a location outside the limit of the descriptor table or the location indexed by the selector in the descriptor table does not contain a valid descriptor, then an exception is produced.
- The 80386 also checks to see if the segment descriptor is of the right type to be loaded into the specified segment register cache. The descriptor for a read-only data segment, for example cannot be loaded into the SS register, because a stack must be able to be written to. A selector for a code segment which has been marked "execute only" cannot be loaded into the DS register to allow reading the contents of the segment.
- If all above protection conditions are met, the limit, base, and access rights bytes of the segment descriptor are copied into the hidden part of the segment register. The 80386 then checks the P (Present) bit of the access byte to see if the segment for that descriptor is present, a type 11 exception is generated.
- After a segment selector and descriptor are loaded into a segment register, further checks are made each time a location in the actual segment is accessed. These checks are **type checking** and **limit checking**.

### 5.3.1 Type Checking

- Type field of the descriptor specifies.
  1. Type of the descriptor and
  2. Intended usage of the segment.

- As mentioned in the previous section, W (Writable), R (Readable), C (Conforming), A (Accessed) and, E (Expanded-Down) bits from type field specify the usage of the segment and restrict segment for particular use only.
- For example, if R bit 1, the segment is read only segment. Its accessed is limited to only reading purpose.
- Type checking is used to detect whether any program is attempting to use segments in ways not intended by the programmer.

### 5.3.2 Limit Checking

- The 80386DX uses limit field of a segment descriptor to prevent programs from addressing outside the segments.
- It interprets limit field depending on the setting of the G (granularity) bit, which specifies whether limit value counts 1 byte or 4 kbytes.
- In case of data segments processor also checks ED (Expansion Direction) bit and B (Big) bit.
- For all types of segments expand-down data segment, the value of the limit is one less than the size (expressed in bytes) of the segments.
- The 80386DX causes a general protection exception when program attempts to
  - Access memory byte at an address  $>$  limit
  - Access memory word at an address  $\geq$  limit
  - Access memory Dword at an address  $\geq$  (limit-2)
- For expand-down data segments, the limit is interpreted differently. In these cases the range of valid addresses is from limit + 1 to either 64 K or  $2^{31} - 1$  (4 Gbyte) depending on the B-bit.

### 5.3.3 Protection Levels - Privilege Level Protection

- The 80386 has four levels of protection which are optimized to support the needs of a multi-tasking operating system to isolate and protect user programs from each other and the operating system.
- The four level of protections are four privilege levels, numbered from 0 to 3. The value zero represents highest privilege level and value 3 represents lowest privilege level.
- Fig. 5.3.1 shows how a 80386DX protected mode system can be set up with four privilege levels.
- It shows that operating system kernel is assigned with the highest privilege level, which is privilege level 0 (PL0). The system services such as BIOS procedures are

assigned with PL1, whereas custom device drivers are assigned with PL2 and finally application programs are assigned with PL3.

### 5.3.4 Concept of DPL, CPL, RPL and EPL

- The 80386DX assigns these levels to different objects such as descriptors and selectors.
- The assigned privilege levels are stored in the respective fields as given below.

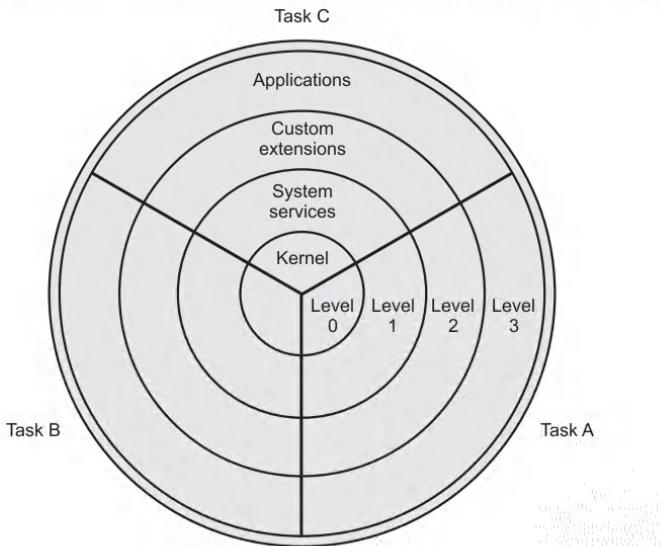


Fig. 5.3.1 Assignment of privilege levels

#### Descriptor Privilege Level (DPL)

- Descriptors contain field called the Descriptor Privilege Level (DPL). It is the least privileged level at which a task may access that descriptor and the segment associated with that descriptor.
- It is contained in the access right byte of the descriptor of the segment.

#### Current Privilege Level (CPL)

- The 80386DX stores the descriptors in the internal cache (hidden portion of segment registers) for currently executing segments. Privilege levels for such descriptors are referred to as Current Privilege Level (CPL).
- This privilege level is also called Task Privilege Level.
- It specifies privilege level of currently executing task.

- A task's CPL can **only be changed** by control transfers through **gate descriptors** to a code segment with a different privilege level.
- For example, an application program running at PL = 3 may call an OS routine at PL = 1 (via a gate) which would cause the task's CPL to be set to 1 until the OS routine is finished.
- **Normally, CPL = DPL** of the segment that the processor is currently executing.
- **CPL changes** as control is transferred to segments with **differing DPLs**.

### Requestor Privilege Level (RPL)

- Selectors contain field called the **Requester's Privilege Level (RPL)**. The RPL is intended to represent the privilege level of the procedure that originates a selector.
- RPL is the **two least significant bits of selector**.

### Effective Privilege Level (EPL)

- When access to a new memory segment is desired, an **Effective Privilege Level (EPL)** is computed. This is the greater (least privileged) of CPL and RPL.
- EPL is defined as
 
$$\text{EPL} = \max \{ \text{RPL}, \text{CPL} \} \text{ (numerically)}$$
  - Thus the task becomes less privileged
- For example, if RPL = 2 and CPL = 1, EPL = 2 → task became less privileged.

#### 5.3.4.1 Restricting Access to Data

- Program can load a data segment register only if the DPL of the target segment is numerically greater than or equal to the maximum of the CPL and the selector's RPL.
- In other words, a procedure can only access data that is at the same or less privileged level. Following Table 5.3.1 gives exact idea about data access.

No.	Privilege levels			Access
	DPL	CPL	RPL	
1	2	0	1	Valid
2	3	1	2	Valid
3	1	1	0	Valid
4	1	2	0	Invalid
5	2	2	3	Invalid

Table 5.3.1 Data accesses

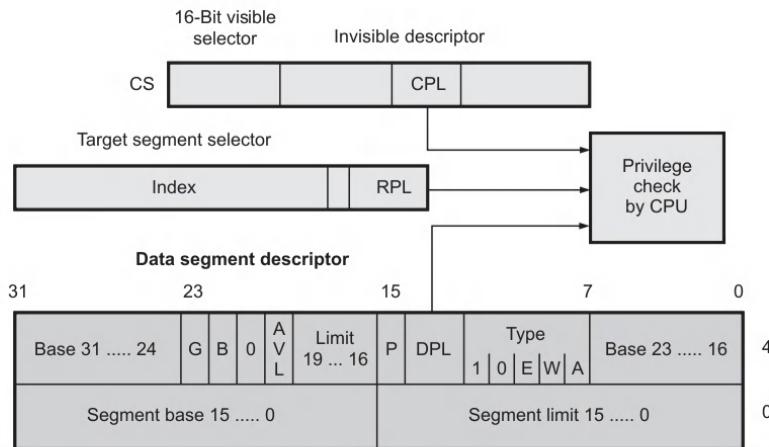


Fig. 5.3.2 Privilege check for data access

### 5.3.4.2 Accessing Data in Code Segments

- It is possible to read data from code segment. There are three ways of reading data from code segments.
  - Load a data segment register with a selector of a non conforming, readable, executable segment.
  - Load a data segment register with a selector of a conforming, readable, executable segment.

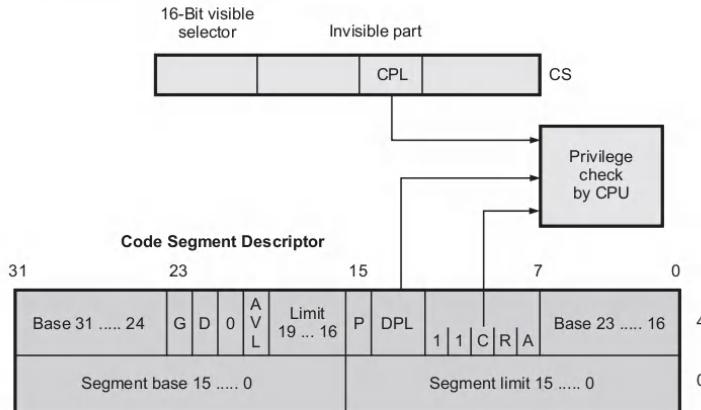


Fig. 5.3.3 Privilege check for control transfer

- 3. Use a CS override prefix to read a readable, executable segment whose selector is already loaded in the CS register.
- In case 1, procedure can only access data that is at the same or less privileged level.
- Case 2 is always valid because the privilege level of segment whose conforming bit is set.
- Case 3 is also always valid because the DPL of the code segment in CS is by definition, equal to CPL.

#### 5.3.4.3 Restricting Control Transfers

- The 80386DX can transfer program control with the help of JMP, CALL, RET, INT and IRET instructions. The “near” forms of JMP, CALL and RET transfer control within the current segment so these are subjected to only limit checking. But in case of far JMP, CALL and RET transfers, control is transferred to other segment. In such cases 80386DX performs privilege checking.
- To successfully transfer the control to other segment, both the RPL and the CPL must be a number less than or equal to the DPL of the segment.
- In other words, the privilege level of the requesting selector and current privilege level must both be greater than or equal to the privilege level of the desired segment.

$$\text{Max (CPL, RPL)} \leq \text{DPL}$$

#### 5.3.5 Changing Privilege Levels

- After looking all these restrictions the question that might come to mind at this point is, if a task cannot access a segment with a more privileged (numerically less) DPL, how can user programs access the operating system kernel, BIOS, or utility procedures in segments which have more privileged (numerically less) DPLs ?
- There are two ways to access a procedure located in a segment which has a higher privilege level.
  1. The first option has a restriction that the segment which has a higher privilege level must be a **conforming code segment**.
  2. The second option is more complex, but allows to access the segment which has a higher privilege level using special structure known as **call gate**.

##### 5.3.5.1 Conforming Code Segment

- A code segment is considered conforming if bit 2 of the access rights byte of its descriptor is set. (C = 1 for conforming and C = 0 for non-conforming).

- Conforming code segments differ from the **non-conforming code** segment in that they have no inherent privilege level of their own; they conform to that level of the code that CALLs them or JMPs to them.
- For example, if a program in a PL3 segment transfers control to a conforming code segment, then the conforming code runs with CPL equal to 3. If the same segment is invoked by PL0 code, it runs with a CPL of 0.
- When the control is transferred to a conforming code segment, the RPL bits of register CS are not changed to match the DPL of segment, as they normally would be. Instead, they still reflect the correct CPL the DPL of the last non-conforming code segment that was executed. This is the only time that the RPL bits in the CS register might not match the DPL bits in the currently executing segment.
- Even though conforming code segments do not have any particular privilege level associated with them, there is still one restriction regarding when a conforming segment can be used. The DPL of the conforming descriptor must always be less than or equal to the current CPL. You can never transfer control to a segment whose DPL is greater (less privileged) than the current segment. This is done because at the time of transferring control back to the original segment from conforming code segment there is change in privilege level. Here, conforming code segment must have higher or same privilege level than original segment to allow control to return back to the original segment.
- Table 5.3.2 gives exact idea about access of conforming code segment.

No.	Current Privilege Level (CPL)	DPL of conforming code segment	Access
1	3	2	Valid
2	2	0	Valid
3	1	1	Valid
4	1	2	Invalid
5	2	3	Invalid

**Table 5.3.2 Accessing of conforming code segment**

### 5.3.5.2 Inter Privileged Level Transfers using Call Gates

- A call gate is simply a special type of descriptor as shown in Fig. 5.3.4. Unlike code, data, or stack descriptors or the system type LDT descriptor, call gate descriptors **do not define any memory space**. They have **no base address or limit fields**. Actually, they are not descriptors at all, but it is convenient to place them in descriptor tables.
- It acts as an interface layer between code segments at different privilege levels.

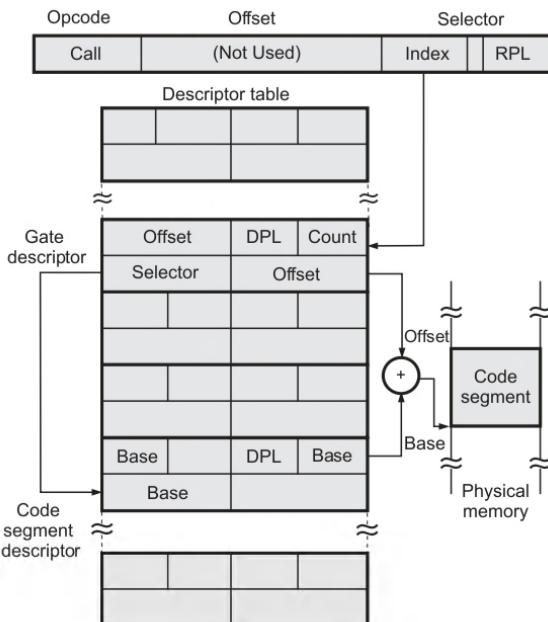
- The **call gate** is the only mechanism that allows to call a procedure located in any segment (conforming or non-conforming) which has a higher privilege level.
- JMPs are not allowed. Hence the name **call gate**.
- It is important that the CALL must refer a call gate, not the destination code segment.
- The call gate defines the code segment and the exact offset where the control is to be transferred. Users are not allowed to specify the desired offset in their programs. Because any wrong offset may corrupt the procedure if control is transferred into the middle of a subroutine or, worse yet, into the middle of an instruction.
- Call gate descriptor is put in the GDT or in LDT, just as segment and other descriptors.
- The call gate descriptor contains two important things :
  1. Selector which points to the descriptor for the segment where the procedure is actually loaded.
  2. Offset of the called procedure in its segment.

#### Call Gates

31	23	15	7	0	
Offset 31 ..... 16	P	DPL	Type 0 1 1 0 0	0 0 0	(WC) DWORD count
Selector			Offset 15 ..... 0		4 0

**Fig. 5.3.4 Format of 80386 call gate**

- If the call is valid, the selector from the call gate (points to the descriptor for the segment where the procedure is actually loaded) is placed in the visible portion of CS register and the corresponding segment descriptor is loaded into the hidden portion of CS register.
- The 80386DX then uses the base address from the segment descriptor and the offset from the call gate descriptor to calculate the physical address of the called procedure as shown in Fig. 5.3.5. (See Fig. 5.3.5 on next page).



**Fig. 5.3.5 Indirect transfer via call gate**

- During this process the validity of control transfer is checked using four different privilege levels :
  - The CPL (Current Privilege Level).
  - The RPL (Requester's Privilege Level) of the selector used to specify the call gate
  - The DPL of the gate descriptor
  - The DPL of the descriptor of the target executable segment.
- For valid control transfer, the transfer must satisfy the following privilege rules for CALL instruction as shown in Fig. 5.3.6 (a).

**Target DPL  $\leq$  Max (RPL, CPL)  $\leq$  Call Gate DPL**

- For example, if you are running in a PL2 code segment (CPL=2), and you want to call a PL0 procedure (target DPL=0), you must use a gate to that procedure with a DPL of 2 or 3.
- Fig. 5.3.6 (b) shows some valid accesses to higher privileged levels.

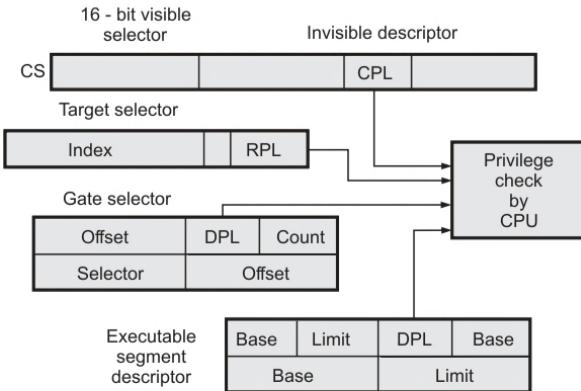


Fig. 5.3.6 (a) Privilege check via call gate

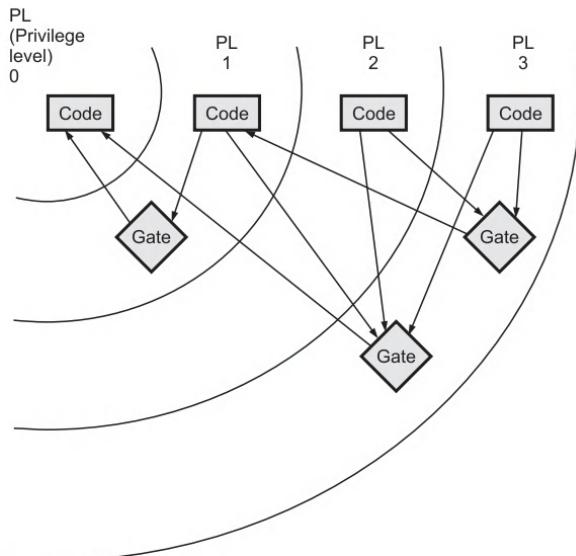


Fig. 5.3.6 (b) Some valid accesses to higher privileged levels using call gates

**Privilege requirements to use a call gate :**

- Call gate DPL must be numerically greater than or equal to the current privilege level.
  - Call gate DPL must be numerically greater than or equal to the RPL of the gate selector.
  - Call gate DPL must be numerically greater than or equal to the target code segment DPL.
  - Target code segment DPL must be numerically less than or equal to the current privilege level.
- 
- In call gates, the procedure is accessed indirectly, through the call gate descriptor, rather than directly through a segment descriptor. This indirect access has two major advantages.
    1. This approach permits another level of privilege checking before access to the procedure in the higher privileged segment. The privilege level of the calling program (CPL) is compared with DPL of the call gate. If the privilege level of the calling program (CPL) is numerically greater than the DPL of call gate, the access will not be allowed.
    2. User programs cannot accidentally enter higher privileged segments at just any old point. If they are going to enter at all, they must enter at the specific offset contained in the call gate descriptors.

**5.3.6 Stack Switching**

- The change in privilege level changes the address domain of the program. The 80386 also changes stacks in case of change in privilege level.
- When call gate causes a change in privilege, stack segment and pointer are saved, and a new stack is used that corresponds to the new, inner privilege level.
- When controls returned to outer level code, the use of the original stack is restored.
- If there is a valid call through gate, the 80386 uses a new stack. It takes segment selector and the pointer for this stack from the TSS. If user is calling procedure with privilege level 1 (PL1), the new stack selector and stack pointer are taken from SS1 and ESP1, respectively.
- The old stack selector and stack pointer are immediately pushed onto this new stack. Then 80386 finds the number of double word (32-bit) entries to be pushed from old stack to new stack from WC (Word Count) field from the call gate descriptor. This means that WC field decides number of passing parameters to the

new stack. After this, old CS selector and EIP offset are pushed onto the new stack.

- Finally, CS is loaded from the selector field of the call gate descriptor, EIP is loaded from the offset field, and execution starts at the new address.

### Review Questions

- Explain in detail segment level protection using privilege mechanism.
- Write a note on privilege level protection.
- State the privilege rules for
  - Accessing data in code segment
  - Control transfer.
- What is a difference between conforming code segment and non-conforming code segment and rule to access it ? **SPPU : Dec.-2000, 02, Marks 6**
- How many privilege levels do 80386 supports ? **SPPU : May-05, Marks 3**
- What do you mean by privilege levels ? What is the need ? **SPPU : May-05, Dec.-07, Marks 4**
- Explain the four levels of protection mechanism implemented by 80386. **SPPU : May-13, Dec.-15, Marks 12**
- What is DPL, RPL and CPL ? **SPPU : May-05,10,12, Marks 5**
- What are the privilege checks made if accessed area is code, data or stack ? **SPPU : Dec.-2000, Marks 4**
- Write privilege checks performed by 80386 while accessing code or data with protection mechanism. **SPPU : May-10,12, Marks 5**
- Explain CPL, RPL and DPL concepts while accessing the code or data with protection mechanism of 80386. Also draw the data segment descriptor format and code segment descriptor format to demonstrate mechanism of privilege level checks. **SPPU : May-05, Marks 5**
- What are different methods to call a function with higher privilege level ? Explain with the help of suitable example. **SPPU : May-05,06, Marks 10**
- How will you access a function from higher privilege level ? Explain. **SPPU : Dec.-07, Marks 4**
- 80386 is currently executing program from code segment having PL as 2. If it needs to access the code from PLO, is it possible ? If yes, which are the methods used for the same ? If no, justify your answer. **SPPU : May-11, Marks 18**
- What is the use of DPL field of conforming code segment descriptor ? How does it differ from DPL field of non-conforming code segment descriptor ? Explain with suitable example. **SPPU : May-02,04, Marks 8**

16. What do you mean by conforming code ? Explain. Is there any alternative for this mechanism ? Explain. SPPU : Dec.-06, Marks 10
17. Explain conforming code segment and non-conforming code segment. SPPU : Dec.-08, May-10, Marks 6
18. Explain the working of conforming code segment. SPPU : May-13, Marks 6
19. What is call gate ? SPPU : May-02, Marks 5
20. What are different methods to call a function with higher privilege level ? Explain with the help of suitable example. SPPU : Dec.-11, May-05,06,08,12, Marks 10
21. What is call gate descriptor ? Give its significance in detail. SPPU : Dec.-08, Marks 10
22. What is call gate ? Explain its role in changing PL. SPPU : May-09, Marks 10
23. CALL gate acts as an interface layer to a code with different privilege levels. Justify the statement with the help of CALL gate descriptor. SPPU : May-10, Marks 8
24. Explain CALL gate mechanism in detail. SPPU : Nov.-12, Marks 6
25. How does 80386 handle stack access when you change the privileged level. SPPU : May-2000, Marks 3
26. Explain how stacks are handled when privilege level is changed through call gate. SPPU : May-02, Marks 5
27. Write stack related steps performed by 80386 processor in executing an inter-level CALL. SPPU : May-10, Marks 4
28. Explain conforming code segment and non-conforming code segment. SPPU : Dec.-2000, 02, 08, May-10, Marks 4
29. What is CPL and RPL ? SPPU : Dec.-17, May-18, Marks 2
30. With the help of suitable diagram, explain how call gate descriptor is used to change the privilege levels in protected mode ? SPPU : Dec.-18, Marks 6
31. Define DPL, RPL and CPL. SPPU : Dec.-18, Marks 2
32. With appropriate diagram explain the concept of privilege levels in 80386. SPPU : May-19, Marks 4
33. How call gate descriptor is used to locate the procedure in another code segment ? How protection is provided ? SPPU : May-19, Marks 6

#### 5.4 Page Level Protection

SPPU : May-11, Dec.-19

- Page level protection involves two kinds of protections
  1. Restriction of addressable domain
  2. Type checking
- The U/S and R/W fields of PDEs and PTEs are used to control access to pages.

### 5.4.1 Restricting Addressable Domain

- The U/S bit is 0 for the operating system and other system software and related data. It is a supervisor level. When the 80386DX is executing at supervisor level, all pages are addressable. If U/S bit is 1, 80386DX is executing at user level. In this case, only pages that belongs to the user level are addressable.

### 5.4.2 Type Checking

- At the level of page addressing two types of accessing are defined.
  - Read only access ( $R/\bar{W} = 1$ )
  - Read/ $\overline{Write}$  access ( $R/\bar{W} = 0$ )
- When 80386 is executing at supervisor level, all pages are assigned with Read/write access, whereas at user level page access depends on  $R/\bar{W}$  bit in the PDE and PTE fields.
- If  $R/\bar{W}$  bit is 1 pages are only readable and if  $R/\bar{W}$  bit is 0 pages are both readable and writeable.
- When 80386DX is executing at user level, it cannot access page belongs to supervisor level.

#### Review Questions

1. Differentiate between segment level protection and page level protection.

SPPU : May-11, Marks 4

2. List aspects of protection related to pages.

SPPU : May-19, Dec.-19, Marks 2

### 5.5 Combining Segment and Page Level Protection

- When paging is enabled, the 80386 first evaluates segment protection, then evaluates page protection.
- If the processor detects a protection violation at either the segment or the page level, the requested operation cannot proceed; a protection exception occurs instead.
- It is possible to define a large data segment which has some subunits that are read-only and other subunits that are read-write. In this case, the page directory (or page table) entries for the read-only subunits would have the U/S and R/W bits set to X0, indicating no write rights for all the pages described by that directory entry (or for individual pages).

**Review Question**

1. Explain how segment and page level protections function in combination.

**5.6 I/O Protection****SPPU : May-13, 18, Dec.-19**

- The 80386 supports two mechanisms for protecting I/O ports in protected mode :
  1. The IOPL field in the EFLAG register defines the right to use I/O related instructions (I/O privilege level).
  2. The I/O permission bit map of a 80386 TSS segment defines the right to use ports in the I/O address space.

**5.6.1 I/O Privilege Level**

- In this mechanism, for execution of IN, INS, OUT, OUTS, CLI and STI instructions, the CPL of a procedure or task must be the same or a lower number than the number represented by the IOPL bits. (CPL ≤ IOPL).
- Any attempt by a less privileged procedure to use these instructions result in a general protection exception.
- The instructions IN, INS, OUT, OUTS, CLI and STI are called **sensitive instructions** because they are sensitive to IOPL.
- Each task has its own unique copy of the flag register. Thus each task can have a different IOPL. The task can change IOPL. But only procedure executing at privilege level 0 can change IOPL; otherwise IOPL remains unaltered. It does not result in an exception.

**5.6.2 I/O Permission Bit Map**

- The second mechanism for protecting I/O ports from unauthorized access is an optional I/O permission bit map which allows ports to be associated only with specific task.
- Fig. 5.6.1 shows the I/O address bit map. (See Fig. 5.6.1 on next page.)
- The I/O permission bit map is a bit vector. The size of the map and its location in the TSS (Task State Segment) are variable. The 80386 locates the I/O permission map. By means of the I/O map base field which is in the fixed portion of the TSS. Each bit in the map corresponds to an I/O port byte address. Thus 16-bit ports use 2-bits each and 32-bit ports use 4-bit each. To access I/O port the corresponding bit in the I/O bitmap must be 0.
- When program attempts to access a port, the 80386 first compares the CPL of the task with the IOPL. If the access passes the IOPL test and an I/O bit map is compulsory, the 80386 checks the map bit corresponding to the addressed port. If corresponding bit is 0, access is granted.

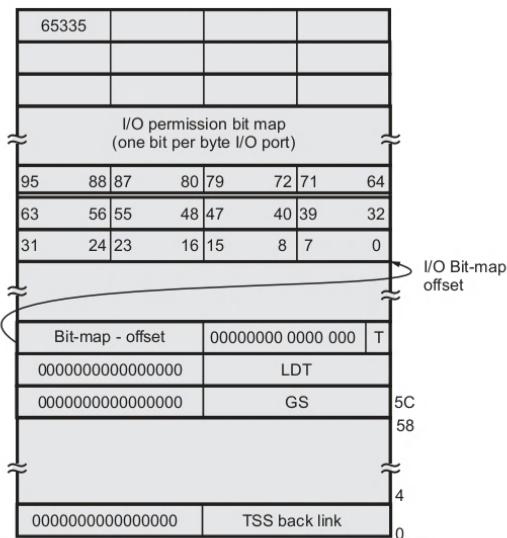


Fig. 5.6.1 I/O address bit map

**Example 5.6.1** An 80386 system has 256 I/O ports with addresses from 00H to FFH. All these ports except 21H to 2FH are to be made accessible to a user at PL3. Show how the I/O permission bit map look like.

**Solution : i) I/O permission bit map :**

FF	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	F0
EF	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	E0
DF	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	D0
CF	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	C0
BF	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	B0
AF	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A0
9F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	90
8F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	80
7F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	70
6F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	60
5F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50
4F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	40
3F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	30
2F	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	20
1F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10
0F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00

**Review Questions**

1. Explain the significance of IOPL in 80386 processor.

**SPPU : May-13, Marks 2**

2. Explain the function of I/O permission bit map.

**SPPU : May-18, Marks 3**

3. Write a short note on 'I/O permission bit map'.

4. List mechanism which provide protection for I/O functions and explain the role of IOPL in providing protection for I/O functions.

**SPPU : Dec.-19, Marks 6**

## 5.7 Privilege and I/O Sensitive Instructions

**SPPU : May-2000,02,05,13, Dec.-08,10,17**

- There are 19 privileged instructions supported by 80386. Privileged instructions are those that affect the segmentation and protection mechanism, alter the interrupt flag, or perform peripheral I/O. These instructions are divided in two groups.

- Privileged Instructions (Group I)
- IOPL - Sensitive Instructions (Group II)

### 5.7.1 Privileged Instructions

- The instructions that affect the system data structures are come under first group. The instructions under this group must be executed when CPL is 0; otherwise 80386 generates general protection exception.
- Table 5.7.1 shows the instructions from group I (Privileged Instructions).

Instruction	Action
HLT	Halts the processor
CLTS	Clears task-switched flag
LGDT, LIDT, LLDT	Loads GDT, IDT, LDT registers
LTR	Loads task register
LMSW	Loads machine status word
MOV CRn, REG/MOV REG, CRn	Moves to/from control registers
MOV DRn, REG/MOV REG, DRn	Moves to/from debug registers
MOV TRn, REG/MOV REG, TRn	Moves to/from test registers

Table 5.7.1 Privileged instructions

### 5.7.2 IOPL Sensitive Instructions

- Here, the IOPL field in the FLAG register defines the right to use I/O related instructions. Hence the instructions from this group are called **sensitive instructions**.

- Table 5.7.2 shows the IOPL sensitive instructions.
- In order to execute these instructions, the CPL of a procedure or task must be the same or a lower number than the number represented by the IOPL bits ( $CPL \leq IOPL$ ).

Instruction	Action
CLI	Disables interrupts
STI	Enables interrupts
IN, INS	Inputs data from I/O port
OUT, OUTS	Outputs data to I/O port

**Table 5.7.2 IOPL - sensitive instructions****Review Questions**

1. What are privileged instructions in 80386 and the response if used ?

**SPPU : May-2000, Marks 3**

2. What are different privileged and IOPL sensitivity instructions of 80386 processor ? Explain in short each instruction.

**SPPU : May-02, Marks 8**

3. Explain any two privilege level instruction.

**SPPU : May-05, Marks 3**

4. What is the meaning of 'privileged instructions' ?

**SPPU : Dec.-08, Marks 2**

5. What is privileged instructions ? Explain two examples of privileged instructions.

**SPPU : Dec.-10, Marks 8**

6. What is privileged instruction ? Explain its significance with examples.

**SPPU : May-13, Marks 6**

7. Explain any two I/O privilege instructions.

**SPPU : Dec.-17, Marks 4**

## **UNIT - V**

**6**

# **Multitasking**

### **Syllabus**

*Task State Segment, TSS Descriptor, Task Register, Task Gate Descriptor, Task Switching, Task Linking, Task Address Space.*

### **Contents**

6.1	<i>Introduction</i>	.....	<b>May-17, Dec.-19,</b> .....	<b>Marks 4</b>	
6.2	<i>Task State Segment</i>	.....	<b>May-01,02,08,12,13,18,</b> ..... .....	<b>Dec.-2000,02,08,10,17,18,</b> <b>Nov.-12,</b> .....	<b>Marks 8</b>
6.3	<i>TSS Descriptor</i>	.....	<b>May-10,12,13, Dec.-02,13,</b> ..	<b>Marks 5</b>	
6.4	<i>Task Register</i>	.....	<b>May-10,18, Dec.-08,</b> .....	<b>Marks 6</b>	
6.5	<i>Task Gate Descriptor</i>	.....	<b>May-02,03,04,07,13,14,</b> .....	<b>Dec.-03,</b> .....	<b>Marks 8</b>
6.6	<i>Task Switching</i>	.....	<b>May-01,02,03,07,09,10,12,</b> .....	<b>Dec.-2000,02,03,07,08,</b> ..	<b>Marks 8</b>
6.7	<i>Task Linking</i>	.....	<b>May-08,09, Dec.-02,08,19,</b> ..	<b>Marks 6</b>	
6.8	<i>Task Address Space</i>				

## 6.1 Introduction

SPPU : May-17, Dec.-19

- Multitasking is the ability of a computer to run more than one program or task at the same time.
- On a single processor multitasking system, multiple processes do not actually run at the same time since there is only one processor. Instead, the processor switches among the processes that are active at any given time. Because of this action, it appears to the user as though the processor is executing all of the tasks at once.
- The 80386 has special registers, **special data structures** to support efficient and protected multitasking system. These are :
  - Task State Segment (TSS)
  - Task State Segment Descriptor
  - Task Register
  - Task Gate Descriptor.
- With these registers and data structures the 80386 switches execution from one task to another task, saving the environment of the current task. Thus task can be continued later.
- Apart from simple task switch, the 80386 supports two other **task management features** :
  - Interrupts and
  - Exceptions
- They can cause task switches. The 80386 not only switches automatically to the task that handles the interrupt or exception, but it automatically switches back to the interrupted task when the interrupt or exception has been serviced. Interrupt task may interrupt lower priority interrupt tasks to any depth.
- As each task can have separate LDT and page directory, it can have a different logical to linear mapping and a different linear-to-physical mapping. Due to this tasks can be isolated and prevented from interfering with one another.

### Review Question

1. *What is multitasking ?*
2. *List various data structures provided by 80386 to support multitasking.*
3. *List the registers and data structures that are used in multitasking.*
4. *Write a short note on "Multitasking" feature of 80386.*

SPPU : May-17, Marks 2

SPPU : Dec.-19, Marks 4

## 6.2 Task State Segment

SPPU : May-01,02,08,12,13,18, Dec.-2000,02,08,10,17,18, Nov.-12

- Fig. 6.2.1 shows the format of a TSS. It is a special type of segment, used to manage the task. The 80386 uses TSS like a scratch-pad. It stores everything it needs to know about a task in TSS. This means that task environment (context) is stored in the TSS.
- TSS is not accessible to the general user program or program even at privilege level 0. The fields within TSS are accessible to only 80386.
- The fields of a TSS are divided into two sets : **Dynamic set** and **Static set**.

31		0
	Bit Map Offset	0000000000000000 T
0000000000000000	LDT	64
0000000000000000	GS	60
0000000000000000	FS	5C
0000000000000000	DS	58
0000000000000000	SS	54
0000000000000000	CS	50
0000000000000000	ES	4C
	EDI	48
	ESI	44
	EBP	40
	ESP	3C
	EBX	38
	EDX	34
	ECX	30
	EAX	2C
	EFLAGS	28
	EIP	24
	CR3	20
0000000000000000	SS2	1C
	EIP2	18
0000000000000000	SS1	14
	EIP1	10
0000000000000000	SS0	0C
	EIP0	8
0000000000000000	Back link	4
		0

Fig. 6.2.1 Task state segment

**1. Dynamic Set :** The 80386 updates dynamic set when it switches from one task to another task. This set includes :

- The general registers (EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI )
  - The segment registers (CS, SS, DS, ES, FS, GS)
  - The flag registers (EFLAGS)
  - The instruction pointer (EIP)
  - Back link.
- The first four fields (general registers, segment registers/selectors, flags and instruction pointer) save the state of the microprocessor, 80386. Saving EIP guarantees that the task can be restarted at the point at which it was stopped and saving EFLAGS allows 80386 to execute conditional instructions properly, when the task is restarted.
  - The **back link** is used by the 80386 to keep track of a previous task. By executing a return instruction at the end of the new task, the back link selector for the previous TSS is automatically loaded into task register. This activates the previous task and restores the prior program environment.

**2. Static Set :** The 80386 only reads fields from this set. This set includes :

- The selector for the task's LDT
- The register (PDBR) that contains the base address of the task's page directory
- Pointers to the stacks for privilege levels 0-2
- The T-bit (debug trap bit) which causes the 80386 to raise a debug exception when a task switch occurs.
- The I/O map offset.

**Note** TSS static set saves the selector for the task's LDT. This means that TSS descriptors must appear only in the GDT.

- Task switching may change the privilege level changing the addressable domain of the program. As rule says the privilege level of the stack segment must exactly match the privilege level of the code segment at all times, the 80386 has to change stack when there is change in privilege level. Due to this previous stack segment and pointer are abandoned, and a new stack is used that corresponds to the new privilege level.
- When control is returned to previous level, the previous stack is restored. To store stack pointer and stack selector of the previous task fields ESP0, ESP1, ESP2, SS0, SS1, SS2 hold the stack segment pointers and stack selectors for privilege levels 0, 1 and 2.

The I/O map base holds the 16-bit offset of the beginning of the I/O permission bit map. It is implemented on a task-by-task basis and affects the hardware privilege checking only for I/O instructions.

### Review Questions

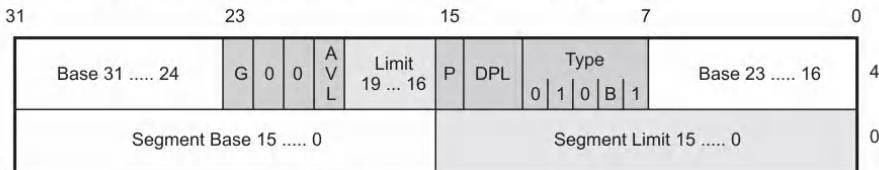
1. *What is multitasking ?*
2. *What is a Task State Segment (TSS) and its contents ?* SPPU : Dec.-2000, Marks 4
3. *What is a TSS ? What is its size and contents ? Explain clearly indicating checks and its usage by 80386.* SPPU : May-01, Marks 7
4. *What is back-link field of TSS ? What is its offset in TSS ?* SPPU : May-02, Marks 4
5. *Explain the significance of the TSS.* SPPU : Dec.-02, May-12, Marks 5
6. *Explain significance of backlink field.* SPPU : May-08, Marks 5
7. *Write short note on TSS.* SPPU : Dec.-10, May-08, Marks 6
8. *Explain TSS with the help of diagram in detail.* SPPU : Dec.-08, May-13, Marks 8
9. *What is TSS ? What are the contents of it ? Discuss its use in multitasking ?* SPPU : Nov.-12, Marks 8
10. *Why is it necessary to abandon previous stack segment and pointer when privilege level occurs during task switch ?*
11. *Draw and briefly explain task state segment.* SPPU : Dec.-17, Marks 6
12. *Draw and explain TSS.* SPPU : May-18, Marks 7
13. *What is the role of TSS in multitasking ? Explain I/O permission bitmap in TSS.* SPPU : Dec.-18, Marks 6

### 6.3 TSS Descriptor

SPPU : May-10,12,13, Dec.-02,13

- Like other segments, the task state segment is defined by descriptor called **TSS descriptor**.
- Fig. 6.3.1 shows the task state segment descriptor. It contains fields like other segments.
- The **B-bit** in the type field indicates whether the task is busy. Tasks are not re-entrant, The B-bit allows 80386 to detect an attempt to switch to a task that is already busy.
- The **BASE**, **LIMIT**, and **DPL** fields and the **G-bit** and the **P-bit** have functions similar to other descriptors.
- The limit field, however must have a value equal to or greater than 103 (104-1), because 80386 requires minimum 104 bytes of storage in order to perform a

context save. A larger limit is permissible and it is required if an I/O permission map is present. The maximum limit for TSS is 4 Gbyte.



**Fig. 6.3.1 Task state segment descriptor**

- To access TSS descriptor, the procedure must have privilege level less than or equal to (numerically) privilege level specified by DPL field of the TSS descriptor. Usually this access is restricted for only trusted softwares, whose privilege level is zero. This can be done by setting DPL fields of TSS descriptor to zero. Thus only trusted softwares has the right to perform task switching.

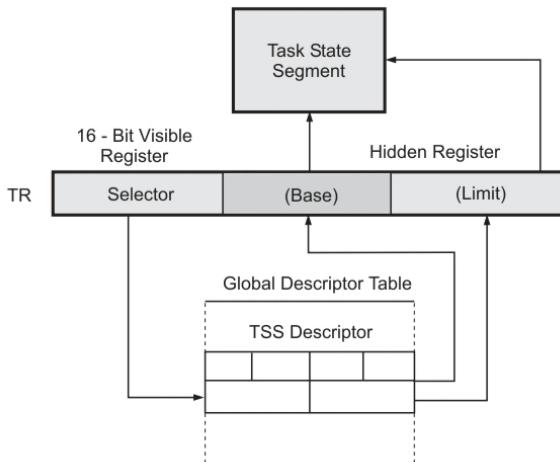
#### Review Questions

- What is TSS descriptor ? SPPU : May-10, Dec.-13, Marks 3
- What purpose the DPL of the TSS descriptor serves ?
- What is a Task State Segment (TSS) ? Give the format of TSS descriptor. How does it differ from gate descriptor ?
- Explain the significance and format of the TSS descriptor. SPPU : Dec.-02, May-12, Marks 5
- Draw neat diagram to explain TSS descriptor. SPPU : May-13, Marks 3

#### 6.4 Task Register

**SPPU : May-10,18, Dec.-08**

- The Task Register (TR) specifies the currently executing task by pointing to the TSS.
- Fig. 6.4.1 shows the path by which 80386 accesses the current task. Task Register is a selector for the TSS. [Refer Fig. 6.4.1 on next page]
- It has both **visible portion** which can be read and changed by instructions and **invisible portion** (maintained by the 80386 to correspond to the visible portion which can not be read by any instruction).
- The selector in the visible portion is used to specify a TSS descriptor in the GDT and invisible portion is used to cache the base and limit values from the TSS descriptor.
- Holding the base and limit in the invisible portion of the Task Register makes execution of the task more efficient, because the processor does not need to



**Fig. 6.4.1 Task register**

repeatedly fetch these values from memory when it references the TSS of the current task.

- The 80386 gives two instructions to read and modify the visible portion of the task :
  - LTR (Load Task Register) and
  - STR (Store Task Register)

**LTR (Load Task Register)** : It loads the visible portion of the task register with the selector and invisible portion with information from the TSS descriptor selected by selector. LTR is a privileged instruction. Thus it is executed only when CPL is zero.

**STR (Store Task Register)** : It stores the visible portion of the task register in a general register or memory word. STR is not a privilege instruction.

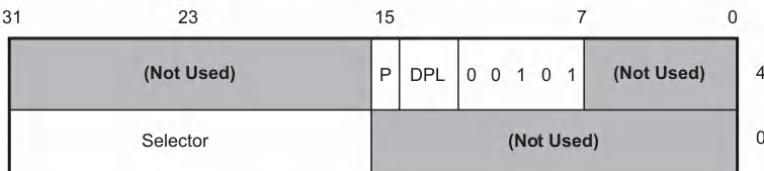
### Review Questions

1. Explain task register. **SPPU : Dec.-08, Marks 2**
2. Specify size and function of TR. **SPPU : May-10, Marks 2**
3. Explain instructions used to read and modify the visible portion of the task.
4. Explain the role of task register in multitasking and the instructions used to modify and read TR. **SPPU : May-18, Marks 6**

## 6.5 Task Gate Descriptor

SPPU : May-02,03,04,07,13,14, Dec.-03

- Task gates, like call gates, are special system gates. It has its own descriptor. A task gate descriptor does not define a memory segment but instead acts as an interface point between user code and a task state segment.
- It provides an indirect and protected reference to a TSS.
- Fig. 6.5.1 shows the format of a task gate descriptor. A task gate descriptor defines a selector to a TSS descriptor which uniquely identifies a task.
- Like the selector to a call gate, the selector to a task gate can be used in place of a selector to a code segment in FAR JMP and FAR CALL instructions.



**Fig. 6.5.1 Task gate descriptor**

- As mentioned earlier, the DPL field of a task gate controls the right to use the descriptor to cause a task switch.
- Procedure selects a task gate descriptor only when the maximum of selector's RPL and the CPL of the procedure is numerically less than or equal to the DPL of the descriptor.

$$\text{MAX (CPL, RPL)} \leq \text{task gate DPL}$$

- Now if DPL privilege level is 0. Then privilege constraint prevents untrusted procedures (procedures having privilege level from 1 to 3) from causing task switch. But through task gates we can switch from lower privilege to higher privilege because when a task gate is used, the DPL of the target TSS descriptor is not used for privilege checking. Thus a procedure that has access to a task gate has the power to cause a task switch.

### Review Questions

1. *What is a task ?*

SPPU : Dec.-03, Marks 2; May-07,13, Marks 3

2. *What is the use of task gate ?*

SPPU : May-14 Marks 3

3. *Draw and explain the format of task gate descriptor.*

4. *What is the difference between TSS descriptor and task gate descriptor ? Explain with respect to multitasking.*

SPPU : May-02,03,04, Marks 8

## 6.6 Task Switching

SPPU : May-01,02,03,07,09,10,12, Dec.-2000,02,03,07,08

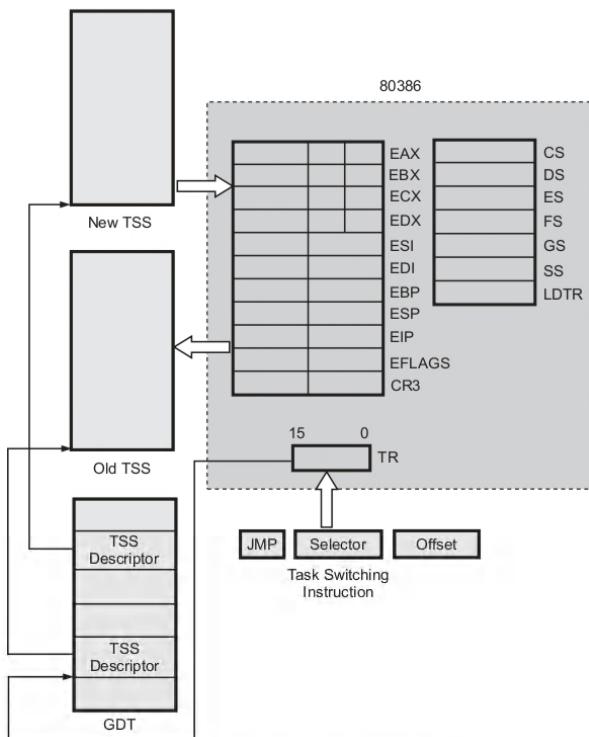
- It is important to note that after every task switch i.e. after loading a new context from a TSS and updating TR, the 80386 marks the new TSS as "busy". It does this by setting bit 41 in the currently running TSS descriptor. Therefore currently running task is always a busy task. **The 80386 cannot do task switch into a task which is busy. Tasks are not reentrant and task switches therefore cannot be recursive.**
- The 80386 does task switching in any of four cases :
  1. A long jump or call instruction contains a selector which refers to a TSS descriptor. This is the simplest method and can be easily implemented by the operating system Kernel at the end of a time slice.
  2. The selector in a long jump or call instruction refers to a task gate. In this case the selector for the destination TSS is in the task gate. This indirect method has advantages regarding privilege levels and protection.
  3. The interrupt selector refers to a task gate in the interrupt descriptor table. The task gate contains the selector for the new TSS. If the access passes all the privilege level tests, the selector and descriptor for the interrupt task will be loaded into the task register. The Nested Task (NT) bit in the EFLAGS register will be set.
  4. An IRET instruction is executed with the NT bit in the EFLAGS register set. The IRET instruction uses the back link selector in the TSS to return execution to the interrupted task.

### 6.6.1 Task Switching without Task Gate

- Fig. 6.6.1 shows task switch operation.

#### Steps involved in Task Switching (Without Task Gate) :

1. **Privilege Check :** The current task is checked to see whether it is allowed to switch to the designated task. This is done by checking DPL of the designated TSS with RPL and CPL of the current task. If the DPL of the TSS descriptor is numerically greater than or equal to the maximum of CPL and the RPL of the selector then only the current task is allowed to switch to the designated task.
2. **Limit and Present Bit Checking :** The TSS descriptor for the designated task is checked for its limits and presence.
3. **Saving the State of the Current Task :** The 80386DX finds the base address of the current TSS cached in the task register. It copies the registers into the current TSS (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, ES, CS, DS, SS, FS, GS, the flag register and EIP). The EIP field of the TSS points to the instruction after the one that



**Fig. 6.6.1 Task switch operation**

caused the task switch. The selector for the current task is saved as a back link selector in the new task.

- Loading of Task Register :** The visible portion of the task register is loaded with the selector of the designated task's TSS descriptor. This sets the TS (Task switch) bit in the Machine Status Word (MSW). This TS bit is useful to systems software when a coprocessor is present. The TS bit signals that the context of the coprocessor may not correspond to the current 80386DX task. The B bit in the new task's descriptor is marked busy. Then the corresponding task state descriptor is read from the GDT and loaded into the task register cache (hidden portion of task register).
- Resuming Execution :** Finally, 80386 starts execution of designated task, with the instruction pointed by the new contents of the code segment selector (CS) and instruction pointer (EIP).

- The old program environment is preserved by saving the selector for the old TSS as the back link selector in the new TSS. By executing a return instruction at the end of the new task, the back link selector for the old TSS is automatically reloaded into TR and then program execution resumes at the point where it left off in the old task.

### 6.6.2 Task Switching with Task Gate

- In this, the indirect method is used for task switching. Task switching is done by jumping to or calling a task gate. Fig. 6.6.2 shows task switching through a task gate.

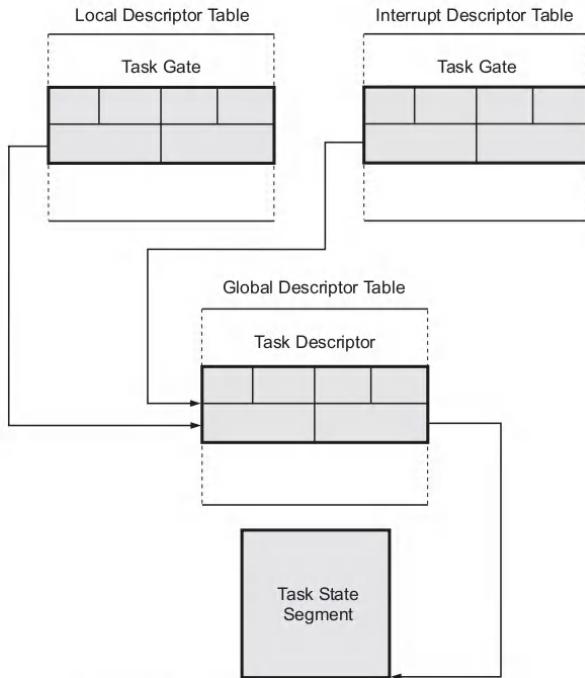


Fig. 6.6.2 Task switching through task gate

#### Steps involved in Task Switching (Using Task Gate) :

- Privilege Check :** When task gate is used, the DPL of the new TSS descriptor is not used for privilege checking. The DPL of the task gate is compared with the CPL and RPL of the gate selector. If the DPL of the task gate is numerically greater than or equal to the maximum of CPL and the RPL of the gate selector, the current task is allowed to switch to the designated/new task. The remaining steps

are similar excepts that for loading selector for TSS descriptor into TR task gate is referred instead of CALL or JMP instruction.

- In case of exceptions, interrupts and IRETs regardless of the DPL of the new task gate or TSS descriptor, the current task is allowed to switch to the new task.

### Review Questions

- What do you mean by task switching and context of the task ?*
- Explain the methods by which task switch is forced.*
- Explain the steps involved in task switching without task gate.*
- Explain the steps involved in task switching with task gate.*
- Explain the function and reaction of the 80386 when the task switch occurs.*

SPPU : Dec.-2000, May-10, Marks 5

- How does TSS support task switching ?*
- Explain the process of task switching in processor 80386.*
- Write short note on task switching in 80386.*
- Explain i) Busy bit ii) Ts (Task switch) bit.*

SPPU : May-02,03,07,12 Dec.-02,03,07, Marks 8

SPPU : May-09, Marks 6

SPPU : Dec.-08, Marks 4

### 6.7 Task Linking

SPPU : May-08,09, Dec.-02,08,19

- Nested tasks are analogous to nested subroutines. If task switch was caused by a FAR CALL instruction or by an exception, fault or trap, the new task is considered to be nested within the old task that invoked it.
- In any of these cases, when the task executes an IRET instruction, the 80386 automatically task-switches back to the task that invoked it. To do so, there is a **mechanism of linking the tasks**, which is equivalent of a call/return stack.
- The task linking mechanism consists of **Back Link** and **NT (Nested Task)** flag.
- The Back Link is used to keep a track of a previous task. By executing a IRET instruction at the end of the new task, the back link selector for the previous TSS is automatically loaded into task register. This activates the previous task and restores the prior program environment.
- The 80386 sets the NT (Nested Task) flag in the EFLAGS register, when one system task invokes another task. The 80386 uses NT as a flag so that it can tell whether the Back Link field in the current TSS is valid. Should it encounter an IRET instruction ? This is the only means by which 80386 determines whether it should perform a task switch or a normal IRET.

- RET instruction does not ‘unnest’ tasks, even if they were nested by CALL instructions. Only IRET can ‘unnest’ tasks.

### Nested Task Switches

- Nested tasks act like subroutines.
- CALL instruction to task gate will nest tasks.
- Interrupt or exception to task gate will nest tasks.
- JMP instruction will not nest tasks.
- New TSS gets old TSS selector in Back Link field.
- New task gets nested task bit set in EFLAGS register.
- New task must return to old task with IRET instruction.

### Review Questions

1. Write a note on nested tasks.
  2. When is the ‘back link’ entry in TSS valid ? What is the purpose of this entry ?
  3. How tasks are nested ?
  4. What is nested task ?
  5. Explain : NT (Nested Task) bit.
  6. What is ‘Nested task’ ? How are they handled in 80386 ?
  7. Write a short note on “Task Linking”.
- SPPU : Dec.-02, Marks 4
- SPPU : May-08, Marks 6
- SPPU : Dec.-08, Marks 2
- SPPU : May-09, Marks 6
- SPPU : Dec.-19, Marks 4

### 6.8 Task Address Space

- The LDT selector and PDBR fields of the TSS give software systems designers flexibility in utilization of segment and page mapping features of the 80386. By appropriate choice of the segment and page mappings for each task, tasks may share address spaces, may have address spaces that are largely distinct from one another, or may have any degree of sharing between these two extremes.
- The ability for tasks to have distinct address spaces is an important aspect of 80386 protection. A module in one task cannot interfere with a module in another task if the modules do not have access to the same address spaces. The flexible memory management features of the 80386 allow systems designers to assign areas of shared address space to those modules of different tasks that are designed to cooperate with each other.

### 6.8.1 Task Linear-to-Physical Space Mapping

- The choices for arranging the linear-to-physical mappings of tasks fall into two general classes :

#### 1. One linear-to-physical mapping shared among all tasks

When paging is not enabled, this is the only possibility. Without page tables, all linear addresses map to the same physical addresses. When paging is enabled, this style of linear-to-physical mapping results from using one page directory for all tasks. The linear space utilized may exceed the physical space available if the operating system also implements page-level virtual memory.

#### 2. Several partially overlapping linear-to-physical mappings

- This style is implemented by using a different page directory for each task. Because the PDBR (Page Directory Base Register) is loaded from the TSS with each task switch, each task may have a different page directory.
- In theory, the linear address spaces of different tasks may map to completely distinct physical addresses. If the entries of different page directories point to different page tables and the page tables point to different pages of physical memory, then the tasks do not share any physical addresses.

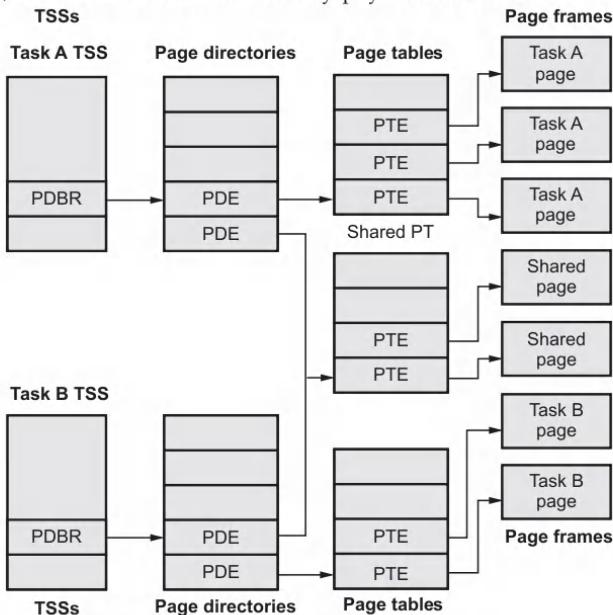


Fig. 6.8.1 Partially-overlapping linear spaces

- In practice, some portion of the linear address spaces of all tasks must map to the same physical addresses. The task state segments must lie in a common space so that the mapping of TSS addresses does not change while the processor is reading and updating the TSSs during a task switch. The linear space mapped by the GDT should also be mapped to a common physical space; otherwise, the purpose of the GDT is defeated. Fig. 6.8.1 shows how the linear spaces of two tasks can overlap in the physical space by sharing page tables.

### 6.8.2 Task Logical Address Space

- By itself, a common linear-to-physical space mapping does not enable sharing of data among tasks. To share data, tasks must also have a common logical-to-linear space mapping; i.e., they must also have access to descriptors that point into a shared linear address space. There are three ways to create common logical-to-physical address-space mappings :
  1. **Via the GDT :** All tasks have access to the descriptors in the GDT. If those descriptors point into a linear-address space that is mapped to a common physical-address space for all tasks, then the tasks can share data and instructions.
  2. **By sharing LDTs :** Two or more tasks can use the same LDT if the LDT selectors in their TSSs select the same LDT segment. Those LDT-resident descriptors that point into a linear space that is mapped to a common physical space permit the tasks to share physical memory. This method of sharing is more selective than sharing by the GDT; the sharing can be limited to specific tasks. Other tasks in the system may have different LDTs that do not give them access to the shared areas.
  3. **By descriptor aliases in LDTs :** It is possible for certain descriptors of different LDTs to point to the same linear address space. If that linear address space is mapped to the same physical space by the page mapping of the tasks involved, these descriptors permit the tasks to share the common space. Such descriptors are commonly called **aliases**. This method of sharing is even more selective than the prior two; other descriptors in the LDTs may point to distinct linear addresses or to linear addresses that are not shared.

#### Review Questions

1. Write a note on task address space.
2. Explain the two ways of arranging the linear-to-physical mappings of tasks.
3. Write a note on task logical address space.



## **Notes**

## **UNIT - V**

**7**

# **Virtual 8086 Mode**

### **Syllabus**

*Features, Memory management in Virtual Mode , Entering and leaving Virtual mode.*

### **Contents**

7.1 Features .....	<b>May-04,05,06,07,08,09,12,16,18,19,</b>	
.....	<b>Dec.-05,06,18,19,</b>	Marks 6
7.2 Executing 8086 Code .....	<b>May-17, Dec.-17,</b>	Marks 6
7.3 Memory Management in Virtual Mode .....	<b>May-17, Dec.-18,</b>	Marks 6
7.4 Entering and Leaving Virtual 8086 Mode .....	<b>Dec.-02,17,</b>	
.....	<b>May-08,09,12,16,18,19,</b>	Marks 6
7.5 Difference between Real, Protected and Virtual 8086 Modes .....	<b>Dec.-05,10,11,12,</b>	
.....	<b>May-10,13,14,15,</b>	Marks 6

## 7.1 Features

SPPU : May-04,05,06,07,08,09,12,16,18,19, Dec.-05,06,18,19

- In multitasking system, it is necessary to switch back and forth between real and protected mode. Because in multitasking system, there is a mixture of tasks, some use segment-offset addressing (Real mode addressing) and some use descriptors (Protected mode addressing). The 8086 virtual mode solves this problem.

### Features of Virtual 8086 Mode

- A 80386 operating in protected mode can easily switch to virtual 8086 mode to execute a time slice of an 8086 program and then easily switch back to protected mode to execute a time slice of protected mode task.
- The 80386 allows execution of one or more 8086, 8088, 80186 or 80188 programs in an 80386 protected mode environment, as different tasks in the virtual 8086 mode.
- In 8086 virtual mode, the 80386 treats the segment registers exactly the same way as it does in real mode. Therefore, the **address range of a virtual 8086 mode task is 1 Mbyte**.
- The segment and offset registers together give the linear address instead of physical address.
- The physical address is generated from the linear address with the help of page translation. Thus the physical address may be anywhere in the 4 gigabyte memory addressable by the 80386.
- In 8086 virtual mode, the 80386 provides mechanism to selectively trap and manage input/output and interrupt activity. Using software it is possible to determine the Input/Output Privilege Level (IOPL) that selectively controls input/output transfer and it is also possible to use the input/output port permission map to selectively control access to input/output ports.

### Review Questions

- What is the necessity of virtual 8086 mode ?
- What is the address space in virtual 8086 mode ?
- Explain the advantages of virtual 86 mode.
- What is virtual 8086 mode ?
- List any three differences between Virtual 86 mode and 8086.
- List the features of virtual mode.
- Write short note on virtual 8086 mode.
- Explain features of "Virtual 8086 mode".

SPPU : May-05, Dec.-05, Marks 3

SPPU : May-04,05,06,07,08,09,12,16, Dec.-05,06, Marks 6

SPPU : Dec.-18, Marks 3

SPPU : May-18, Dec.-19, Marks 3

SPPU : May-19, Marks 3

## 7.2 Executing 8086 Code

SPPU : May-17, Dec.-17

- The processor executes in V86 mode when the VM (virtual machine) bit in the EFLAGS register is set. The processor tests this flag under two general conditions :
  1. When loading segment registers to know whether to use 8086-style address formation.
  2. When decoding instructions to determine which instructions are sensitive to IOPL.
- Except for these two modifications to its normal operations, the 80386 in V86 mode operated much as in protected mode.

### 7.2.1 Registers

Virtual 8086 mode register set includes :

1. All the registers defined for the 8086 plus
2. The new registers introduced by the 80386 : FS, GS, debug registers, test registers and control registers.

### 7.2.2 Instructions

- In virtual mode, 80386 can execute normal 8086 instructions as well as new instructions introduced by 80186/80188, 80286 and 80386 as listed below. For execution of new instructions and new override prefixes use of FS and GS segment registers is allowed. Instructions can utilize 32-bit operands through the use of the operand size prefix.
1. New instructions introduced by 80186/80188 and 80286
    - PUSH immediate data
    - PUSH ALL and POP ALL ( PUSH A and POP A )
    - Multiply immediate data
    - Shift and rotate by immediate count
    - String I/O
    - ENTER and LEAVE
    - BOUND
  2. New instructions introduced by 80386
    - LSS, LFS, LGS instructions
    - Long displacement conditional jumps
    - Single bit instruction
    - Bit scan

- Byte set on condition
- Double shift instruction
- Move with sign/zero extension
- Generalized multiply

**Key Point** To access these instructions only 8086 addressing modes can be used.

### Review Questions

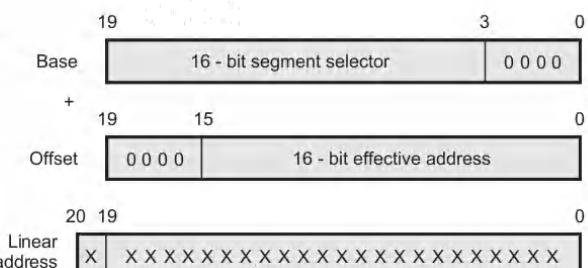
1. List the instructions in virtual 8086 mode.
2. Which bit of EFLAGS indicates V86 mode ? Explain, how hardware and software cooperate with each other to emulate V86 mode ? **SPPU : May-17, Marks 4**
3. With neat diagram explain the process of linear address formation in V86 mode. **SPPU : Dec.-17, Marks 6**

## 7.3 Memory Management in Virtual Mode

**SPPU : May-17, Dec.-18**

### 7.3.1 Linear Address Formation

- In virtual 8086 mode, the contents of segment registers are not used as a selector to point the descriptor. But the segment register contents are used to generate linear address with the help of offset.



**Fig. 7.3.1 Virtual 8086 mode address generation**

- The linear address is generated by adding the contents of the appropriate segment register which are shifted left by 4 bit to an effective address/offset. Fig. 7.3.1 shows virtual 8086 mode address generation.
- If there is a carry generated after addition of shifted segment register contents and effective address, unlike 8086, resulting 21 bit address is a linear address. An 80386 in virtual 8086 mode is allowed to generate linear addresses anywhere in the range 0 to 10FFEH (one megabyte plus approximately 64 kbytes) of the task's linear address space.

- Virtual 8086 tasks generate 32-bit linear addresses. While an 8086 program can only utilize the lower order 21 bits of a linear address, the linear address can be mapped via page tables to any 32-bit physical address.
- Unlike 8086 and 80286, the 80386 can generate 32-bit effective address with the address size command prefix. This address should not exceed beyond 65535 to maintain compatibility with 80286 real mode; otherwise 80386 generate pseudo-protection faults (INT 12 OR INT 13 with no error code).

### 7.3.2 Structure of V86 Task

- The processor enters V86 mode to execute the 8086 program and returns to protected mode to execute the monitor or other 80386 tasks.
- To run successfully in V86 mode, an existing 8086 program needs the following:
  - A V86 monitor.
  - Operating-system services.
- The V86 monitor is 80386 protected-mode code that executes at privilege-level zero. The monitor consists primarily of initialization and exception-handling procedures. As for any other 80386 program, executable-segment descriptors for the monitor must exist in the GDT or in the task's LDT. The linear addresses above 10FFEFH are available for the V86 monitor, the operating system, and other systems software. The monitor may also need data-segment descriptors so that it can examine the interrupt vector table or other parts of the 8086 program in the first megabyte of the address space.
- In general, there are two options for implementing the 8086 operating system :
  1. The 8086 operating system may run as part of the 8086 code. This approach is desirable for any of the following reasons :
    - The 8086 applications code modifies the operating system.
    - There is not sufficient development time to re-implement the 8086 operating system as 80386 code.
  2. The 8086 operating system may be implemented or emulated in the V86 monitor. This approach is desirable for any of the following reasons:
    - Operating system functions can be more easily coordinated among several V86 tasks.
    - The functions of the 8086 operating system can be easily emulated by calls to the 80386 operating system.

### 7.3.3 Using Paging for V86 Tasks

- Paging is not necessary for a single V86 task; however, paging is useful or necessary for any of the following reasons :
  - To create multiple V86 tasks. Each task must map the lower megabyte of linear addresses to different physical locations.
  - To emulate the megabyte wrap. Wrapping truncates the high-order bit to give addresses only up to 20 bits long.
  - To create a virtual address space larger than the physical address space.
  - To share 8086 OS code or ROM code that is common to several 8086 programs that are executing simultaneously.
  - To redirect or trap references to memory-mapped I/O devices.

### 7.3.4 Protection within a V86 Task

- In V86 mode, protection mechanisms offered by descriptors is not available. Thus in V86 mode, to protect the systems software designers may follow either of these approaches :
  - Reserve the first megabyte (plus 64 kilobytes) of each task's linear address space for the 8086 program. An 8086 task cannot generate addresses outside this range.
  - Use the U/S bit of page-table entries to protect the virtual-machine monitor and other systems software in each virtual 8086 task's space. When the processor is in V86 mode, CPL is 3. Therefore, an 8086 program has only user privileges. If the pages of the virtual-machine monitor have supervisor privilege, they cannot be accessed by the 8086 program.

#### Review Questions

- Write a note on structure of V86 task
- Explain the two options for implementing the 8086 operating system in V86 mode.
- State the reasons why paging is useful for V86 mode ?
- How is protection achieved in V86 mode?
- Describe how physical address is obtained in virtual 8086 mode.
- Write short note on "Protection within a V86 task".
- Explain linear address formation in virtual mode of 80386.

SPPU : May-17, Marks 4

SPPU : Dec.-18, Marks 6

## 7.4 Entering and Leaving Virtual 8086 Mode

SPPU : Dec.-02,17, May-08,09,12,16,18,19

- The 80386 enters or leaves 8086 virtual mode due to any of the three reasons as shown in Fig. 7.4.1.
  - An interrupt that vectors to a task gate.
  - An action of the schedule of the 80386 operating system.
  - An IRET when the NT (Nested Task) flag is set.

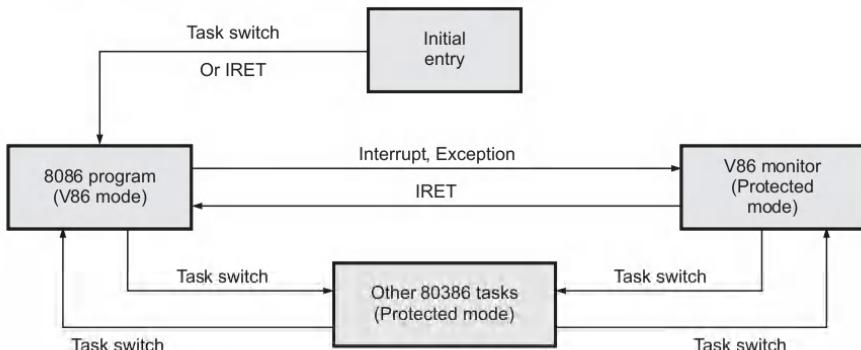


Fig. 7.4.1 Entering and leaving an 8086 program

### 7.4.1 Entering 8086 Virtual Mode

- The 80386 can enter 8086 virtual mode by either of two means :
  - A task switch to an 80386 task loads the image of EFLAGS from the new TSS. If the VM bit in EFLAGS register is set, the 80386 enters virtual 8086 mode to execute the new task. If the VM bit is not set, the 80386 executes the new task as a normal protected mode task.
  - An IRET from a procedure that loads the EFLAGS image changes the VM bit if the Current Privilege Level (CPL) at the time of IRET is zero. If changed status of the VM bit is 1 then 80386 enters in 8086 virtual mode.

### 7.4.2 Leaving 8086 Virtual Mode

- The 80386 leaves the 8086 virtual mode when an interrupt or exception occurs.
  - A task switching from a 8086 virtual task to any other task caused by interrupt or exception loads EFLAGS from the TSS of the new task. If the new TSS is an 80386 TSS and the VM bit is zero, or if the TSS is an 80286 TSS, the 80386 clears the VM bit of EFLAGS. It then loads the segment registers as defined by the new TSS and begins executing the instructions of the new task according to 80386 protected mode description.

2. The interrupt or exception which vectors to a privilege-level zero procedure, stores the current setting of EFLAGS on the stack, then clears the VM bit. As VM bit is zero, the 80386 starts executing the instructions in its protected mode environment.

### Review Questions

1. Write down the steps to enter and leave the virtual 8086 mode.
2. What is virtual-86 mode of 80386 ? Is it possible to switch back and forth between protected mode and virtual-86 mode ? If yes. How ? SPPU : Dec.-02, Marks 6
3. How to switch from real mode to VM-86 mode of 80386. SPPU : May-08,09,16, Marks 2
4. How to switch from protected mode to virtual 86 mode. SPPU : May-12, Marks 4
5. With neat diagram explain "entering and leaving V86 mode". SPPU : Dec.-17, Marks 6
6. Explain entering and leaving V86 mode. SPPU : May-18, Marks 4
7. With the necessary diagrams explain entering and leaving V86 mode ? SPPU : May-19, Marks 6

### 7.5 Difference between Real, Protected and Virtual 8086 Modes

SPPU : Dec.-05,10,11,12, May-10,13,14,15

Sr. No.	Real mode	Protected mode	Virtual 8086 mode
1.	In this mode 80386 maintains the compatibility of the object code with 8086. It supports same architecture as 8086, but it can access the 32-bit register set of 80386DX.	This mode provides a sophisticated memory management and the hardware-assisted protection mechanism.	This mode allows the execution of real mode programs that are incapable of running directly in protected mode.
2.	Memory size is limited to 1 Mbyte.	It increases the linear address space to 4 GB and allows the running of virtual memory programs of 64 terabytes.	Memory address range of virtual 8086 mode task is limited by 1 Mbyte.
3.	Only A <sub>2</sub> -A <sub>19</sub> address lines are active and A <sub>20</sub> -A <sub>31</sub> address lines are normally held high.	All address lines are active	All address lines are active.
4.	Paging mechanism is in-active	Paging mechanism is active.	Paging mechanism is active.
5.	Linear address is same as physical address.	Physical address is generated from linear address with the help of page translation.	Physical address is generated from linear address with the help of page translation.

6.	Protection mechanism is not available	Protection mechanism is available	Provides mechanism to selectively trap and manage I/O and interrupt activity.
7.	Multitasking is not supported	Multitasking is supported	Multitasking is supported.
8.	When 80386 is reset or powered up it is initialized in real mode.  It is possible to enter into real mode from protected mode by resetting the PE bit of the CR0 register.	80386 enters into protected mode by setting PE bit in the CR0 register to logic 1.	By setting VM bit in EFLAGS register to logic 1, the 80386 enters in virtual 8086 mode.

**Review Questions**

1. Compare RM, VM and PM modes of 80386.

**SPPU : Dec.-10,11, Marks 8**

2. What is difference between real mode and protected mode ?

**SPPU : May-10,13,15, Dec.-05, Marks 8**

3. Differentiate between real mode and virtual 8086 mode.

**SPPU : Dec.-12, Marks 4**

4. Explain the different operating modes of 80386.

**SPPU : May-14, Marks 6**



## *Notes*

## **UNIT - VI**

**8**

# **Interrupts and Exceptions**

### **Syllabus**

*Identifying Interrupts, Enabling and Disabling Interrupts, Priority among Simultaneous Interrupts and Exceptions, Interrupt Descriptor Table (IDT), IDT Descriptors, Interrupt Tasks and Interrupt Procedures, Error Code, and Exception Conditions.*

### **Contents**

8.1 <i>Introduction</i> . . . . .	<b>May-17, 19, Dec.-19,</b> . . . . . Marks 6
8.2 <i>Identifying Interrupts</i> . . . . .	<b>May-17, 19,</b> . . . . . Marks 4
8.3 <i>Enabling and Disabling Interrupts</i> . . . . .	<b>Dec.-19,</b> . . . . . Marks 3
8.4 <i>Priority Among Simultaneous Interrupts and Exceptions</i>	
8.5 <i>Interrupt Descriptor Table</i> . . . . .	<b>May-17, Dec.-18,</b> . . . . . Marks 6
8.6 <i>Interrupt Tasks and Interrupt Procedures</i> . . . . .	<b>May-17, Dec.-17, 18,</b> . . . . . Marks 6
8.7 <i>Error Code</i>	
8.8 <i>Exception Conditions</i> . . . . .	<b>May-18,</b> . . . . . Marks 4

## 8.1 Introduction

SPPU : May-17,19, Dec.-19

- Sources for External Interrupts
  - Maskable interrupts (INTR)
  - Nonmaskable interrupt (NMI)
- Sources for Exceptions
  - Processor detected : These are further classified as faults, traps and aborts.
  - Programmed : The instructions INTO, INT 3, INT n and BOUND can trigger exceptions. These instructions are often called "software interrupts", but the processor handles them as exceptions.
- Exceptions are classified as **faults**, **traps** or **aborts** depending on the way they are reported and whether restart of the instruction that caused the exception is supported.
  - **Faults** : Faults are exceptions that are reported "before" the instruction causing the exception. Faults are either detected before the instruction begins to execute or during execution of the instruction. If detected during the instruction, the fault is reported with the machine restored to a state that permits the instruction to be restarted.
  - **Traps** : A trap is an exception that is reported at the instruction boundary immediately after the instruction in which the exception was detected.
  - **Aborts** : An abort is an exception that permits neither precise location of the instruction causing the exception nor restart of the program that caused the exception. Aborts are used to report severe errors.
- Such as hardware errors and inconsistent or illegal values in system tables.

### Review Questions

1. Explain the sources of interrupt and exception.
2. Explain the different exception conditions-Faults, Traps and Aborts.
3. Define "Faults".
4. List different sources of interrupts and explain different ways by which 80386 can enable and disable interrupts.

SPPU : May-17, Marks 6

SPPU : May-19, Marks 2

SPPU : Dec.-19, Marks 3

## 8.2 Identifying Interrupts

SPPU : May-17,19

- The processor associates an identifying number with each different type of interrupt or exception.
- Table 8.2.1 shows the assignment of interrupt and exception identifiers.

Interrupt number	Cause of exception	Description
0	DIV, IDIV	Divide error
1	All	Debug exceptions
3	INT	Breakpoint
4	INTO	Overflow
5	BOUND	Bounds check
6	Any undefined opcode or LOCK used with wrong instruction	Invalid opcode
7	ESC or WAIT	Coprocessor not available
8	INT vector is not within IDTR limit	Interrupt table limit too small
9-11		Reserved
12	Memory operand crosses offset 0 or OFFFFH	Stack fault
13	Memory operand crosses offset OFFFFH or attempt to execute past offset OFFFFH or instruction longer than 15 bytes	Pseudo-protection exception
14,15		Reserved
16	ESC or WAIT	Coprocessor error
17-31	Reserved	
32-255	Available for external	Interrupts via INTR pin

Table 8.2.1 Interrupts and exception ID assignment

**Review Questions**

1. What is the use of direction flag ?
2. Explain "How 80386 identifies interrupts ?"

**SPPU : May-17, Marks 2****SPPU : May-19, Marks 4****8.3 Enabling and Disabling Interrupts****SPPU : Dec.-19**

- Certain conditions and flag settings cause the processor to inhibit certain interrupts and exceptions at instruction boundaries. These conditions and settings are :

**8.3.1 NMI Masks Further NMIs**

- While an NMI handler is executing, the processor ignores further interrupt signals at the NMI pin until the next IRET instruction is executed.

**8.3.2 IF Masks INTR**

- The IF (interrupt-enable flag) controls the acceptance of external interrupts signalled via the INTR pin. When IF = 0, INTR interrupts are inhibited; when IF = 1, INTR interrupts are enabled.
- As with the other flag bits, the processor clears IF in response to a RESET signal. The instructions CLI (Clear Interrupt-Enable Flag) and STI (Set Interrupt-Enable Flag) alter the setting of IF.
- These instructions may be executed only if CPL  $\leq$  IOPL. A protection exception occurs if they are executed when CPL > IOPL.

**8.3.3 RF Masks Debug Faults**

- The RF bit in EFLAGS controls the recognition of debug faults. This permits debug faults to be raised for a given instruction at most once, no matter how many times the instruction is restarted.

**8.3.4 MOV or POP to SS Masks Some Interrupts and Exceptions**

- Software that needs to change stack segments often uses a pair of instructions; for example :
 

```
MOV SS, AX
      MOV ESP, StackTop
```
- If an interrupt or exception is processed after SS has been changed but before ESP has received the corresponding change, the two parts of the stack pointer SS:ESP are inconsistent for the duration of the interrupt handler or exception handler.

- To prevent this situation, the 80386, after both a MOV to SS and a POP to SS instruction, inhibits NMI, INTR, debug exceptions and single-step traps at the instruction boundary following the instruction that changes SS. Some exceptions may still occur; namely, page fault and general protection fault.

### Review Question

- List different sources of interrupts and explain different ways by which 80386 can enable and disable interrupts.*

SPPU : Dec.-19, Marks 3

## 8.4 Priority Among Simultaneous Interrupts and Exceptions

- If more than one interrupt or exception is pending at an instruction boundary, the processor services one of them at a time. The priority among classes of interrupt and exception sources is shown in Table 8.4.1.
- The processor first services a pending interrupt or exception from the class that has the highest priority, transferring control to the first instruction of the interrupt handler.
- Lower priority exceptions are discarded; lower priority interrupts are held pending. Discarded exceptions will be rediscovered when the interrupt handler returns control to the point of interruption.

Priority	Interrupt /Exception
HIGHEST	Faults except debug faults
	Trap instructions INTO, INT n, INT 3
	Debug traps for this instruction
	Debug faults for next instruction
	NMI interrupt
LOWEST	INTR interrupt

Table 8.4.1 Priority among simultaneous interrupts and exceptions

### Review Question

- State the priorities of different interrupts of 80386.*

## 8.5 Interrupt Descriptor Table

SPPU : May-17, Dec.-18

- In protection mode, each interrupt or exception is associated with a descriptor which gives the information about interrupt service routine. These descriptors are stored in a special descriptor table called the **Interrupt Descriptor Table** or IDT. This table can be located anywhere in memory.

- Like the GDT and LDTs, the IDT is an array of 8 byte descriptors.
- The base address and limit for the interrupt descriptors table are loaded into the **Interrupt Descriptor Table Register** (IDTR) as shown in Fig. 8.5.1.
- LIDT (Load IDT register)/ SIDT (Store IDT register) instructions are used to load/store the IDT register with the linear base address and limit values, respectively.
- Because there are only 256 identifiers, the IDT need not contain more than 256 descriptors.

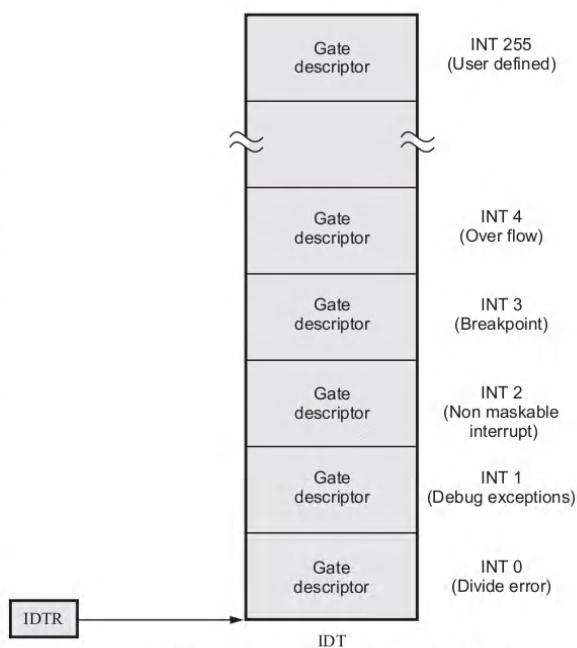


Fig. 8.5.1 Interrupt descriptor table and IDTR

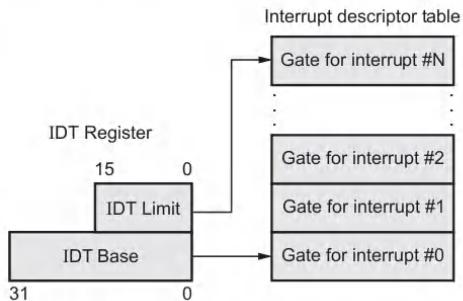


Fig. 8.5.2 Interrupt descriptor register and table

- There are three types of descriptors can be used in the IDT.
  - Trap gate descriptor
  - Interrupt gate descriptor
  - Task gate descriptor.
- If any other type of descriptor is found in the IDT when an exception occurs, the 80386 generates a general protection fault.

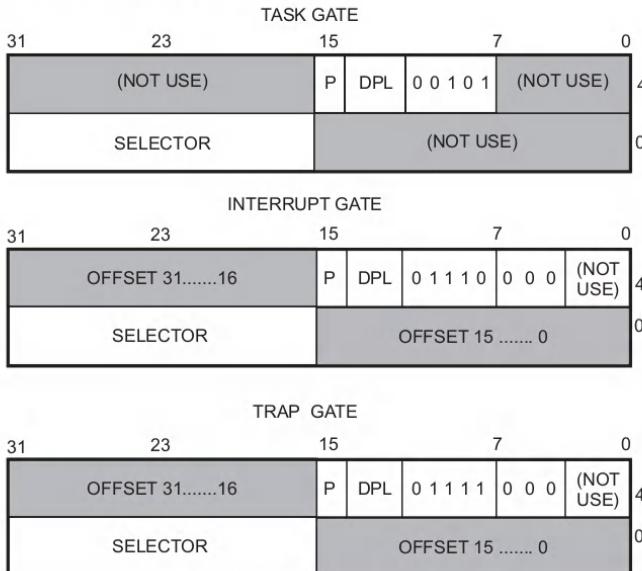


Fig. 8.5.3 IDT gate descriptors

**Review Questions**

1. Write a note on interrupt descriptor table.
2. Give the format of interrupt gate and trap gate descriptions.
3. Give the format of task gate descriptor.
4. What is IDT and how to locate IDT ?
5. Draw the format of interrupt gate and trap gate descriptor. What is the difference between them ?

**SPPU : May-17, Marks 4****SPPU : Dec.-18, Marks 6**

## 8.6 Interrupt Tasks and Interrupt Procedures SPPU : May-17, Dec.-17,18

- Interrupt or exception can "call" an interrupt handler that is either a procedure or a task. When responding to an interrupt or exception, the processor uses the interrupt or exception identifier to index a descriptor in the IDT.
- If the processor indexes to an interrupt gate or trap gate, it invokes the handler in a manner similar to a CALL to a call gate.
- If the processor finds a task gate, it causes a task switch in a manner similar to a CALL to a task gate.

### 8.6.1 Interrupt Procedures

- An interrupt gate or trap gate points indirectly to a procedure which will execute in the context of the currently executing task as illustrated by Fig. 8.6.1.
- The selector of the gate points to an executable-segment descriptor in either the GDT or the current LDT.

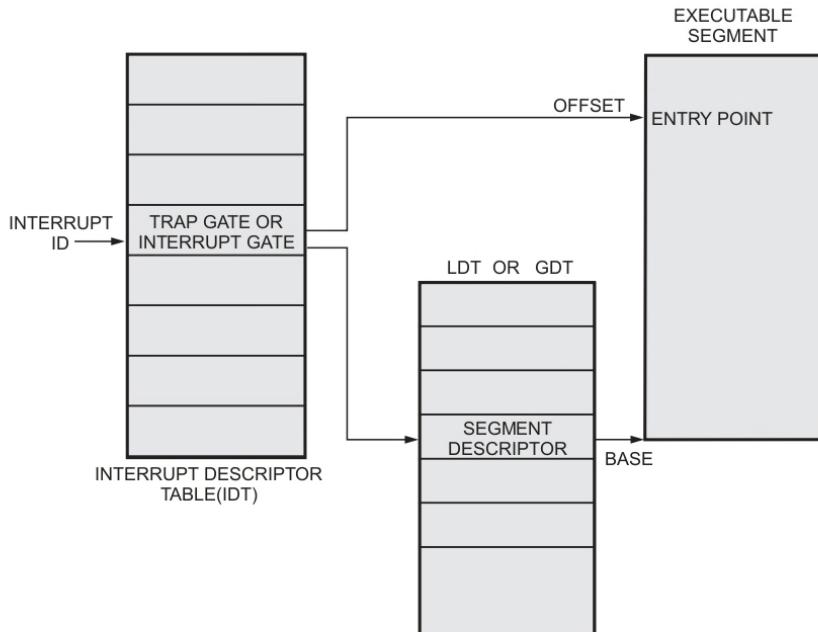


Fig. 8.6.1 Interrupt vectoring for procedures

- The offset field of the gate points to the beginning of the interrupt or exception handling procedure.

### 8.6.1.1 Stack of Interrupt Procedure

- Fig. 8.6.2 shows the information that is stacked before control is transferred to interrupt or exception handling procedure.

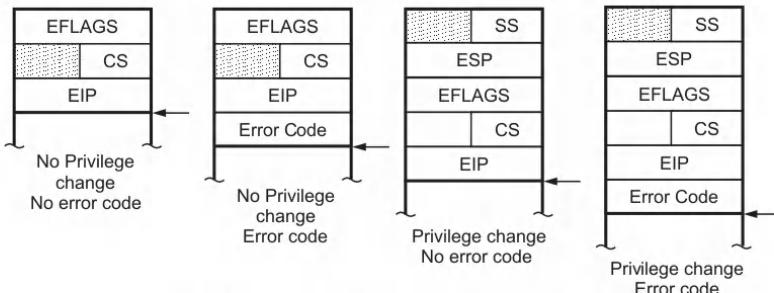


Fig. 8.6.2 Information stored onto the stack

- As shown in Fig. 8.6.2 if an exception cause a special error code it is saved on the stack. If exception requires a privilege change then old stack segment and pointer are also saved on the stack.

### 8.6.1.2 Trap Gate Vs Interrupt Gate

#### Trap Gate Vs Interrupt Gate

- Trap gate operates exactly like an interrupt gate in all respect except one. When an exception vectors through a trap gate all flags remain exactly as they were when the exception occurred (No change in flags status).
- When an exception vectors through an interrupt gate, the 80386/486/Pentium resets IF (Interrupt Flag) to disable further hardware interrupts, after it pushes the return address and EFLAGS but before it executes first instruction of the ISR.

#### Difference between Real Mode IVT and Protected Mode IDT

Sr. No.	Real Mode Interrupt Vector Table (IVT)	Protected Mode Interrupt Descriptor Table (IDT)
1.	The 80386 uses an IVT with 256 entries to store interrupt vectors.	The 80386 uses an IDT to store 256 entries of interrupt descriptors.
2.	In IVT, vectors specify the segment and offset of the interrupt handler.	Descriptor specify the selector, offset, type of segment and its privilege level.

3.	IVT is initialized with a base 0 and a limit of 03FFH.	IDT is relocatable.
4.	Each entry in the interrupt vector table is of 4 bytes.	Each descriptor entry in the IDT is of 8 bytes.

**Table 8.6.1****8.6.1.3 Returning from an Interrupt Procedure**

- An interrupt procedure slightly differs from the normal procedure, as its method of leaving the procedure is different from normal procedure. While returning from the interrupt it is necessary to read EFLAGS from the stack. Thus the IRET instruction is used to exit from an interrupt procedure instead of RET instruction.
- The IRET instruction increments EIP by an extra four bytes and loads the saved flags into the EFLAGS register.
- The IRET instruction then loads CS, and EIP pointers to point previous procedure from where it is interrupted. In case privilege change it also loads old stack segment and pointer.

**Processing Interrupt Service Routines**

- IDT stores the descriptors for interrupt service routines.
- Only trap gate, interrupt gate and task gate descriptors are allowed.
- Operate like programmed procedures/subroutines.
- Before transferring saves all register that are used.
- ISRs are invoked by interrupts or exceptions instead of CALL instructions.
- ISRs terminate with IRET instead of RET instructions.

**Privilege levels**

- Like call gates, interrupt and trap gates have privilege levels associated with them.
- The DPL field of a trap, task or interrupt gate determines the minimum privilege level required to pass through the gate. The CPL must be equal or higher privileged than the gate's DPL.
- It is recommended that DPL field always be kept at privilege level 3. Due to this any privilege level program can handle exceptions.
- The another condition must be satisfied to handle the exception is that the exception code's DPL must have equal or less privilege level than the CPL.

**Exception handler privilege levels**

- Exception gate's DPL must be less privilege than CPL.
- Exception codes DPL must be equal or less privilege than CPL.

### 8.6.2 Interrupt Tasks

- The third alternative for an exception gate is a task gate. When an exception identifier selects a task gate, the 80386 performs an immediate task switch. The task activated is determined by the TSS selector stored in the task gate descriptor. The TSS selector of the current task, which is now dormant, is copied into the Back Link field of the new task. The new task will have its NT (nested task) bit set in EFLAGS. When the exception handling task completes and executes an IRET instruction, the 80386 activates the interrupted task based on the back link information.

**Advantages of using task gate over trap and interrupt gates :**

- The entire context of the interrupted task is saved automatically.
- The exception handler does not need to be concerned with contaminating the interrupted code.
- The exception handler can run at any privilege level.
- The exception handler can use its own private code and data space because it can have its own LDT.

**Drawbacks of using task gate over trap and interrupt gates :**

- More time is required to perform task switch.
- A task gate cannot specify where in the task to begin execution. Dormant tasks always resume where they left off.
- It is difficult to retrieve any information about the interrupted code when it is in a different task.

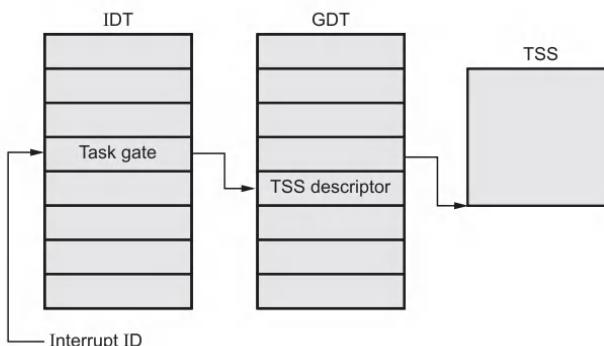


Fig. 8.6.3 Interrupt vectoring for tasks

**Review Questions**

1. Explain what happens when an interrupt calls a procedure as an interrupt handler.

**SPPU : May-17, Dec.-17, Marks 6**

2. Explain the procedure of handling interrupts in protected mode.

**SPPU : Dec.-18, Marks 6**

3. Draw the format of interrupt gate and trap gate descriptor. What is the difference between them ?

**SPPU : Dec.-18, Marks 6**

**8.7 Error Code**

- With exceptions that relate to a specific segment, the processor pushes an error code onto the stack of the exception handler (whether procedure or task). The format of error code is as shown in Fig. 8.7.1.

**EX (External, bit 0)**

When this bit is set, it indicates that the exception was initially caused by an external (hardware) interrupt or that it occurred when the 80386 was already handling another exception.

**I (IDT, bit 1)**

This bit is like a second TI bit. When set, it indicates that the index portion of the error code refers to a descriptor in the IDT.



**Fig. 8.7.1 Error code format**

**TI (Table Indicator, bit 2)**

This is just like the flag of the same name in a segment selector. When set, the index portion of the error code refers to a descriptor in the current LDT at the time the exception occurred. When clear, the index selects a GDT descriptor. This bit is undefined when I is set.

**INDEX (Descriptor Index, bits 3 through 15)**

This field uniquely identifies a descriptor in the GDT, LDT or IDT, depending on the status of the I and TI bits.

**Review Question**

1. Draw and explain the error code format of 80386.

## 8.8 Exception Conditions

SPPU : May-18

### Interrupt 0 : Divide Error

- When the quotient from either a DIV or IDIV instruction is too large to fit in the result register; 80386/486/Pentium will automatically trigger type 0 interrupt.

### Interrupt 1 : Debug Exceptions

- The 80386/486/Pentium triggers this interrupt for one of the conditions; whether the exception is a fault or a trap depends on the condition :
  - Instruction address breakpoint fault.
  - Data address breakpoint trap.
  - General detect fault.
  - Single-step trap.
  - Task-switch breakpoint trap.
- The 80386/486/Pentium does not push an error code for this exception. An exception handler can examine the debug registers to determine which condition caused the exception.

### Interrupt 2 : Non Maskable Interrupt

- As the name suggests, this interrupt can not be disabled by any software instruction. This interrupt is activated by low to high transition on 80386/486/Pentium NMI input pin. In response, 80386/486/Pentium triggers a type 2 interrupt.

### Interrupt 3 : Breakpoint

- The type 3 interrupt is used to implement BREAK POINT function in the system. The type 3 interrupt is produced by execution of the INT 3 instruction. Break point function is often used as a debugging aid in cases where single stepping provides more detail than wanted. When you insert a breakpoint, the system executes the instructions upto the breakpoint, and then goes to the breakpoint procedure. In the break point procedure you can write a program to display register contents, memory contents and other information that is required to debug your program. You can insert as many breakpoints as you want in your program.

### Interrupt 4 : Overflow Interrupt

- The type 4 interrupt is used to check overflow condition after any signed arithmetic operation in the system. The 80386/486/Pentium overflow flag, OF, will be represented in the destination register or memory location.
- For example, if you add the 8-bit signed number 0111 1000 (+ 120 decimal) and the 8 bit signed number 0110 1010 (+ 106 decimal), result is 1110 0010 (- 98 decimal). In signed numbers, MSB (Most Significant Bit) is reserved for sign and other bits represent magnitude of the number. In the previous example, after

addition of two 8-bit signed numbers result is negative, since it is too large to fit in 7 bits. To detect this condition in the program, you can put interrupt on overflow instruction, INTO, immediately after the arithmetic instruction, the instruction will simply function as NOP (no operation). However, if the overflow flag is set, indicating an overflow error, the 80386/486/Pentium will trigger a type 4 interrupt after executing the INTO instruction.

- Another way to detect and respond to an overflow error in a program is to put the jump if overflow instruction (JO) immediately after the arithmetic instruction. If the overflow flag is set as a result of arithmetic operation, execution will jump to the address specified in the JO instruction. At this address you can put an error routine which responds in the way you want to the overflow.

#### **Interrupt 5 : Bounds Check**

- The 80386/486/Pentium triggers interrupt 5 if it notices that the operand has crossed the limits specified by the previously executed BOUND instruction.

#### **Interrupt 6 : Invalid Opcode**

- This fault occurs when an invalid opcode is detected by the execution unit. (The exception is not detected until an attempt is made to execute the invalid opcode; i.e., prefetching an invalid opcode does not cause this exception). No error code is pushed on the stack. The exception can be handled within the same task.
- This exception also occurs when the type of operand is invalid for the given opcode. Examples include an intersegment JMP referencing a register operand or an LES instruction with a register source operand.

#### **Interrupt 7 : Coprocessor Not Available**

- This exception occurs in either of two conditions :
  - The 80386/486/Pentium encounters an ESC (escape) instruction and the EM (emulate) bit of CR0 (control register zero) is set.
  - The 80386/486/Pentium encounters either the WAIT instruction or an ESC instruction and both the MP (monitor coprocessor) and TS (task switched) bits of CR0 are set.

#### **Interrupt 8 : Double Fault**

- Normally, when the 80386/486/Pentium detects an exception while trying to invoke the handler for a prior exception, the two exceptions can be handled serially. If, however, the 80386/486/Pentium cannot handle them serially, it signals the double-fault exception instead. To determine when two faults are to be signaled as a double fault, the 80386/486/Pentium divides the exceptions into three classes : Exceptions, contributory exceptions and page faults. Table 8.8.1

shows this classification. It also shows which combinations of exceptions cause a double fault and which do not.

Class	ID	Description
Benign Exceptions	1	Debug exceptions
	2	NMI
	3	Breakpoint
	4	Overflow
	5	Bounds check
	6	Invalid opcode
	7	Coprocessor not available
	16	Coprocessor error
Contributory Exceptions	0	Divide error
	9	Coprocessor segment overrun
	10	Invalid TSS
	11	Segment not present
	12	Stack exception
	13	General protection
Page Faults	14	Page fault

Table 8.8.1 Double-fault detection classes

		SECOND EXCEPTION		
		Benign Exception	Contributory Exception	Page Fault
FIRST EXCEPTION	Benign Exception	OK	OK	OK
	Contributory Exception	OK	DOUBLE	OK
	Page Fault	OK	DOUBLE	DOUBLE

Table 8.8.2 Double-fault definition

**Interrupt 9 :** Reserved by Intel

**Interrupt 10 : Invalid TSS**

- Interrupt 10 occurs if during a task switch the new TSS is invalid. A TSS is considered invalid in the cases shown in Table 8.8.3. An error code is pushed onto the stack to help identify the cause of the fault. The EXT bit indicates whether the exception was caused by a condition outside the control of the program; e.g., an external interrupt via a task gate triggered a switch to an invalid TSS.

Error code	Condition
TSS id + EXT	The limit in the TSS descriptor is less than 103
LTD id + EXT	Invalid LDT selector or LDT not present
SS id + EXT	Stack segment selector is outside table limit
SS id + EXT	Stack segment is not a writeable segment
SS id + EXT	Stack segment DPL does not match new CPL
SS id + EXT	Stack segment selector RPL < > CPL
CS id + EXT	Code segment selector is outside table limit
CS id + EXT	Code segment selector does not refer to code segment
CS id + EXT	DPL of non-conforming code segment < > new CPL
CS id + EXT	DPL of conforming code segment > new CPL
DS/ES/FS/GS id + EXT	DS, ES, FS, or GS segment selector is outside table limits
DS/ES/FS/GS id + EXT	DS, ES, FS, or GS is not readable segment

**Table 8.8.3 Conditions that invalidate the TSS****Interrupt 11 : Segment Not Present**

- Exception 11 occurs when the 80386/486/Pentium detects that the present bit of a descriptor is zero. The 80386/486/Pentium triggers this fault in any of these cases :
  - While attempting to load the CS, DS, ES, FS or GS registers; loading the SS register however, causes a stack fault.
  - While attempting loading the LDT register with an LLDT instruction; loading the LDT register during a task switch operation, however, causes the "invalid TSS" exception.
  - While attempting to use a gate descriptor that is marked not-present.

- This fault is restartable. If the exception handler makes the segment present and returns, the interrupted program will resume execution.

**Interrupt 12 :** Stack Exception

- A stack fault occurs in either of two general conditions :
  - As a result of a limit violation in any operation that refers to the SS register. This includes stack-oriented instructions such as POP, PUSH, ENTER and LEAVE, as well as other memory references that implicitly use SS (for example, MOV AX, [BP + 8]). ENTER causes this exception when the stack is too small for the indicated local - variable space.
  - When attempting to load the SS register with a descriptor that is marked not-present but is otherwise valid. This can occur in a task switch, an interlevel CALL, an interlevel return, an LSS instruction or a MOV or POP instruction to SS.
- When the 80386/486/Pentium detects a stack exception, it pushes an error code onto the stack of the exception handler. If the exception is due to a not-present stack segment or to overflow of the new stack during an interlevel CALL, the error code contains a selector to the segment in question (the exception handler can test the present bit in the descriptor to determine which exception occurred); otherwise the error code is zero.

**Interrupt 13 :** General Protection Exception

- All protection violations that do not cause another exception cause a general protection exception. This includes
  1. Exceeding segment limit when using CS, DS, ES, FS or GS
  2. Exceeding segment limit when referencing a descriptor table
  3. Transferring control to a segment that is not executable
  4. Writing into a read-only data segment or into a code segment
  5. Reading from an execute-only segment
  6. Loading the SS register with a read-only descriptor (unless the selector comes from the TSS during a task switch, in which case a TSS exception occurs).
  7. Loading SS, DS, ES, FS or GS with the descriptor of a system segment
  8. Loading DS, ES, FS or GS with the descriptor of an executable segment that is not also readable.
  9. Loading SS with the descriptor of an executable segment
  10. Accessing memory via DS, ES, FS or GS when the segment register contains a null selector

11. Switching to a busy task
12. Violating privilege rules
13. Loading CR0 with PG = 1 and PE = 0.
14. Interrupt or exception via trap or interrupt gate from V86 mode to privilege level other than zero.
15. Exceeding the instruction length limit of 15 bytes (this can occur only if redundant prefixes are placed before an instruction)
- The general protection is a fault. In response to a general protection exception, the 80386/486/Pentium pushes an error code onto the exception handler's stack. If loading a descriptor causes the exception, the error code contains a selector to the descriptor; otherwise, the error code is null.

#### **Interrupt 14 : Page Fault**

- This exception occurs when paging is enabled (PG = 1) and the 80386/486/Pentium detects one of the following conditions while translating a linear address to a physical address :
  - The page-directory or page-table entry needed for the address translation has zero in its present bit.
  - The current procedure does not have sufficient privilege to access the indicated page.

#### **Interrupt 15 : Reserved by Intel.**

#### **Interrupt 16 : Coprocessor Error.**

- The 80386/486/Pentium reports this exception when it detects a signal from the 80287 or 80387 on the 80386/486/Pentium's ERROR input pin. The 80386/486/Pentium tests this pin only at the beginning of certain ESC instructions and when it encounters a WAIT instruction while the EM bit of the MSW is zero (no emulation).
- Table 8.8.4 shows the interrupt and exception summary

Description	Interrupt number	Return address points to faulting instruction	Type	Function that can generate the exception
Divide error	0	YES	FAULT	DIV, IDIV
Debug exceptions	1	YES	TRAP	Any Instruction
NMI	2	NO	NMI	INT 2 or NMI

Break point	3	NO	TRAP	One-byte INT 3
Overflow	4	NO	TRAP	INTO
Bounds check	5	YES	FAULT	BOUND
Invalid opcode	6	YES	FAULT	Any illegal instruction
Coprocessor not available	7	YES	FAULT	ESC, WAIT
Double fault	8	YES	ABORT	Any instruction that can generate an exception
Coprocessor segment overrun	9	NO	ABORT	Any operand of an ESC instruction that wraps around the end of a segment.
Invalid TSS	10	YES	FAULT	JMP, CALL, IRET, any interrupt
Segment not present	11	YES	FAULT	Any segment-register modifier
Stack exception	12	YES	FAULT	Any memory reference through SS
General protection	13	YES	FAULT/ABORT	Any memory reference or code fetch
Page fault	14	YES	FAULT	Any memory reference or code fetch
Reserved by Intel	15	—	—	—
Coprocessor error	16	YES	FAULT	ESC, WAIT
Two-byte SW Interrupt	0-255	NO	TRAP	INT n

**Table 8.8.4 Interrupt and exception summary****Review Question**

1. Explain interrupt no. 0 and 4.

**SPPU : May-18, Marks 4**



## *Notes*

## **UNIT - VI**

**9**

# **Introduction to Microcontrollers**

### **Syllabus**

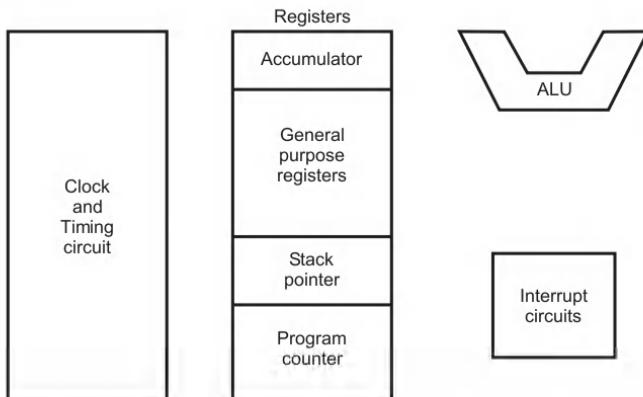
*Architecture of typical Microcontroller, Difference between Microprocessor and Microcontroller, Characteristics of microcontrollers, Application of Microcontrollers.*

### **Contents**

- |     |  |                         |
|-----|--|-------------------------|
| 9.1 | <i>Microcontrollers and Embedded Processors</i>      |                         |
| 9.2 | <i>Features of 8051 Microcontroller</i>              | <i>Dec.-18, Marks 2</i> |
| 9.3 | <i>Block Diagram of 8051 Microcontroller</i>         |                         |
| 9.4 | <i>Register Organization of 8051 Microcontroller</i> |                         |
| 9.5 | <i>Pin Diagram of 8051</i>                           |                         |
| 9.6 | <i>Memory Organization</i>                           |                         |
| 9.7 | <i>External Memory Interfacing</i>                   |                         |

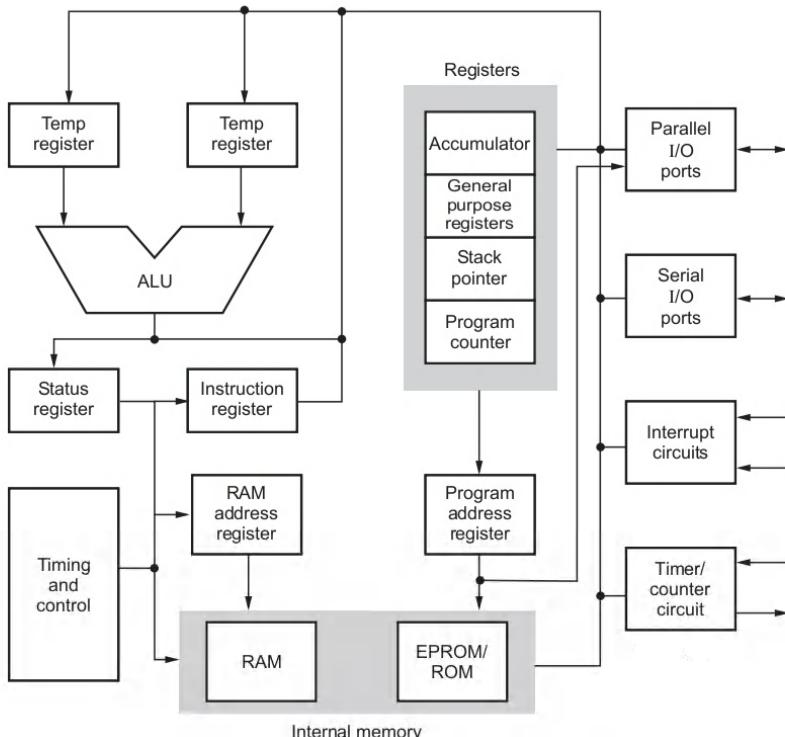
## 9.1 Microcontrollers and Embedded Processors

- The Fig. 9.1.1 shows the simplified block diagram of a microprocessor. As shown in the Fig. 9.1.1 it consists of an Arithmetic and Logic Unit (ALU), general purpose registers, Stack Pointer (SP), Program Counter (PC), clock timing circuit and interrupt circuit.



**Fig. 9.1.1 Simplified block diagram of a microprocessor**

- To make a complete microcomputer system only microprocessor is not sufficient. It is necessary to add other peripherals such as read only memory (ROM), read/write memory (RAM), decoders, drivers, number of input/output devices to make a complete microcomputer system. In addition, special purpose devices, such as interrupt controller, programmable timers, programmable I/O devices, DMA controllers may be added to improve the capacity and performance and flexibility of a microcomputer system.
- The key feature of microprocessor based computer system is that it is possible to design a system with a great flexibility. It is possible to configure a system as large system or small system by adding suitable peripherals.
- On the other hand, the microcontroller incorporates all the features that found in microprocessor. However, it has also added features to make a complete microcomputer system on its own. The microcontroller has built-in ROM, RAM, parallel I/O, serial I/O, counters and a clock circuit. Fig. 9.1.2 shows the simplified block diagram of a microcontroller. (Refer Fig. 9.1.2 on next page.)
- As shown in the Fig. 9.1.2, the microcontroller has on-chip (built-in) peripheral devices. These on-chip peripherals make it possible to have single-chip microcomputer system. There are few more advantages of built-in peripherals :



**Fig. 9.1.2 Block diagram of microcontroller**

- Built-in peripherals have smaller access times hence speed is more.
- Hardware reduces due to single chip microcomputer system.
- Less hardware, reduces PCB size and increases reliability of the system.

### 9.1.1 Comparison between Microprocessor and Microcontroller

The Table 9.1.1 shows the comparison between microprocessor and microcontroller.

No.	Microprocessor	Microcontroller
1.	Microprocessor contains ALU, general purpose registers, stack pointer, program counter, clock timing circuit and interrupt circuit.	Microcontroller contains the circuitry of microprocessor and in addition it has built-in ROM, RAM, I/O devices, timers and counters.
2.	It has many instructions to move data between memory and CPU.	It has one or two instructions to move data between memory and CPU.

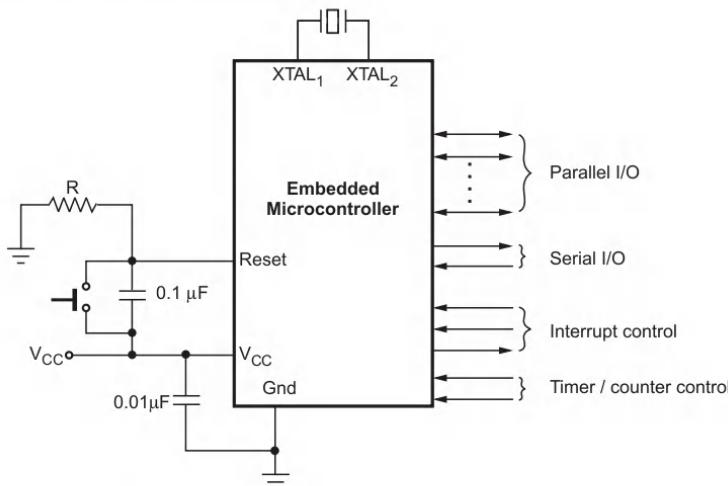
3.	It has one or two bit handling instructions.	It has many bit handling instructions.
4.	Access times for memory and I/O devices are more.	Less access times for built-in memory and I/O devices.
5.	Microprocessor based system requires more hardware.	Microcontroller based system requires less hardware reducing PCB size and increasing the reliability.
6.	Microprocessor based system is more flexible in design point of view.	Less flexible in design point of view.
7.	It has single memory map for data and code.	It has separate memory map for data and code.
8.	Less number of pins are multifunctioned.	More number pins are multifunctioned.

**Table 9.1.1**

### 9.1.2 Different Types of Microcontrollers

- Like microprocessors, the microcontrollers have family of microcontrollers. Different microcontrollers require different support chips and resources to develop particular microcontroller system. To choose an appropriate device to meet system requirements we must understand differences, different options and features of various microcontrollers.

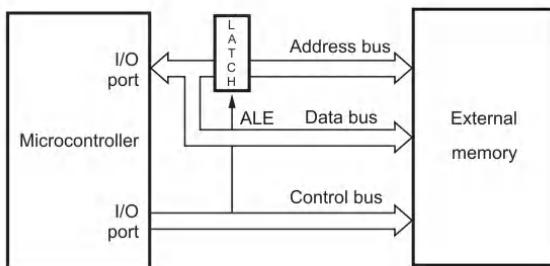
#### 9.1.2.1 Embedded Microcontrollers

**Fig. 9.1.3 Typical microcontroller system with embedded microcontroller**

- When a complete hardware required to run a particular application is provided on the microcontroller chip, it is referred to as an embedded microcontroller. Embedded microcontrollers only require power, reset circuit and clock. Embedded microcontrollers communicate with external devices with its digital I/O pins. Fig. 9.1.3 shows the typical microcontroller system with embedded microcontroller.

### 9.1.2.2 External Memory Microcontrollers

- Sometimes, for large systems, the built-in program memory and data memory are insufficient. To overcome this problem some microcontrollers allow the connection of external memory. For the connection of external memory some parallel port pins are used as address and data lines. Thus connecting external memory to the microcontroller reduces its parallel input/output capabilities. Fig. 9.1.4 shows the microcontroller with external memory connections. As shown in the Fig. 9.1.4, many times address and data lines are multiplexed and separated by external latch and ALE signal from the microcontroller.



**Fig. 9.1.4 Microcontroller with external memory connections**

- The 8051 can work very effectively as an embedded device or with external memory. Its architecture is also very thoughtful and practical combination of different philosophies. Before going to study architecture of 8051 we will see different processor's architectures.

### 9.1.3 Criteria for Selecting Microcontroller

Criteria that designer should consider in choosing microcontrollers are :

- It should satisfy the computing needs of the task efficiently and cost effectively. It should also satisfy the speed requirements, packaging format, RAM and ROM capacity, number I/O pins, on-chip timers and power consumption needs of the application.

- Availability of software development tools such as compilers, assemblers and debuggers.
- Availability in needed quantities both now and in the future.
- Its ability to upgrade to higher-performance or lower power consumption versions.

### 9.1.4 Applications of Microcontroller

Microcontrollers are preferred in embedded products. Some applications of microcontroller are :

- Home appliances : Washing machine, refrigerators, microwave ovens etc.
- Calculators, keyboards, printers, modems, mobile phones etc.
- Industrial controllers, data acquisition systems communication systems etc.
- Automobile engines, flight control systems, traffic light control systems etc.
- Military applications.

#### Review Questions

1. Give the comparison between microprocessor and microcontroller.
2. List out the typical applications of microprocessors and microcontrollers.
3. Bring out the architectural differences between a microprocessor and a microcontroller.
4. Differentiate between microprocessor and microcontroller with respect to their architecture and instructions.
5. Give the basic block diagrams, of a microprocessor and a microcontroller and justify that a microcontroller is an onchip computer.
6. Write a short notes on a) Embedded microcontrollers b) External memory microcontrollers.
7. What criteria do designers consider in choosing microcontrollers ? Mention the typical applications of microcontrollers.
8. Mention any two applications of 8051 microcontroller.

### 9.2 Features of 8051 Microcontroller

SPPU : Dec.-18

- The 8051 is an 8-bit microcontroller designed by Intel. It was optimized for 8-bit math and single bit Boolean operations. Its family includes 8031, 8051, 8052 and 8751 microcontrollers. Let us see the features of 8051 microcontroller.
- The features of the 8051 family are as follows :
  1. 4096 bytes on - chip program memory.
  2. 128 bytes on - chip data memory.
  3. Four register banks.

4. 128 user-defined software flags.
5. 64 kilobytes each program and external RAM addressability.
6. One microsecond instruction cycle with 12 MHz crystal.
7. 32 bidirectional I/O lines organized as four 8-bit ports (16 lines on 8031).
8. Multiple mode, high-speed programmable serial port.
9. Two multiple mode, 16-bit timers/counters.
10. Two-level prioritized interrupt structure.
11. Full depth stack for subroutine return linkage and data storage.
12. Direct byte and bit addressability.
13. Binary or decimal arithmetic.
14. Signed-overflow detection and parity computation.
15. Hardware multiple and divide in 4  $\mu$ sec.
16. Integrated boolean processor for control applications.
17. Upwardly compatible with existing 8084 software.
- The Table 9.2.1 gives the comparison of MCS-51 family microcontrollers.

Sr. No.	Feature	8031	8051	8052	8751
1.	Program memory (in bytes)	None	4 K ROM	8 K ROM	4 K EPROM
2.	Data memory (in Bytes)	128 RAM	128 RAM	256 RAM	128 RAM
3.	Timers / Counters (16-bit)	2	2	3	2
4.	I/O pins	32	32	32	32
5.	Serial Port	1	1	1	1
6.	Interrupt Sources (Reset not included)	5	5	6	5

**Table 9.2.1 Comparison of MCS-51 family microcontrollers**

- As shown in the Table 9.2.1, the 8052 has an extra 128 bytes of RAM, 4 K extra ROM, extra timer and one more interrupt source than the 8051 microcontroller. The 8052 maintains the source compatibility with 8051. This means that all programs written for the 8051 will run on 8052; however, reverse is not true.
- The 8751 microcontroller has 4 K of EPROM instead of ROM. This allows to erase and reprogram the contents of program memory within 8751. It takes around 20 minutes to erase the 8751 before it can be programmed again. This feature is very useful in the program development stage.

**Review Question**

1. List the salient features of 8051 microcontroller.

### 9.3 Block Diagram of 8051 Microcontroller

- Fig. 9.3.1 shows the internal block diagram of 8051. It consists of a CPU, two kinds of memory sections (data memory - RAM and program memory - ROM), input/output ports and control logic needed for a timer/counter, serial port and interrupt functions. These elements communicate through an eight bit data bus which runs throughout the chip referred as internal data bus. This bus is buffered to the outside world through an I/O port when memory or I/O expansion is desired. (Refer Fig. 9.3.1 on next page)
- Central Processor Unit (CPU) :** It monitors and controls all operations that are performed by microcontroller. The CPU of 8051 consists of eight-bit arithmetic and logic unit with associated registers like A, B, PSW, SP, the sixteen bit program counter and "Data pointer" (DPTR) registers. Along with these registers it has a set of special function registers and control unit.
- ROM :** A code of 4K memory is incorporated as on-chip ROM in 8051. The 8051 ROM is a non-volatile memory meaning that its contents cannot be altered.
- RAM :** The 8051 microcontroller is composed of 128 bytes of internal RAM. This is a volatile memory since its contents will be lost if power is switched off. These 128 bytes of internal RAM are divided into 32 working registers which in turn constitute 4 register banks (Bank 0-Bank 3) with each bank consisting of 8 registers (R0 - R7). There are 128 addressable bits in the internal RAM.
- I/O Ports :** The 8051 microcontroller has four 8-bit input/output ports : P0, P1, P2 and P3. All Ports can used as general purpose ports. In the presence of external memory, Port 0 functions as a multiplexed address and data bus and Port2 functions as a higher order byte address bus. All port pins of port 3 are multifunctional. Therefore, each pin of port 3 can be programmed to use as I/O or as one of the alternate function.
- Interrupt Control :** It supports both an internal (software) and external (hardware) interrupts. In 8051, 5 sources of interrupts are provided.
- Timers :** 8051 supports two multiple mode, 16-bit timers/counters. In timer mode, they can be used to generate a delay of particular and in counter mode they can be used to count external pulses.

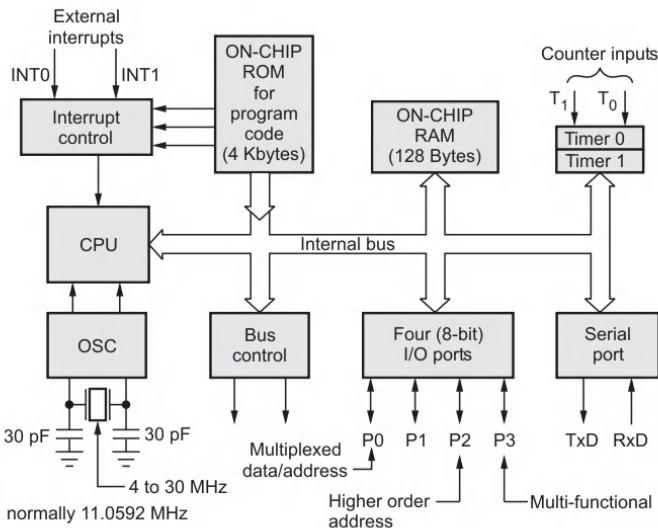


Fig. 9.3.1 Block diagram of 8051 microcontroller

- **Serial Port** : Provides a method of establishing serial communication by transmitting and receiving data bits. It uses TxD and RxD pins to transmit and receive bits, respectively.
- **Oscillator** : It is used for providing the clock to 8051 and decide baud rate for serial communication.

### Review Question

1. With the help of neat diagram explain the internal block diagram of 8051.

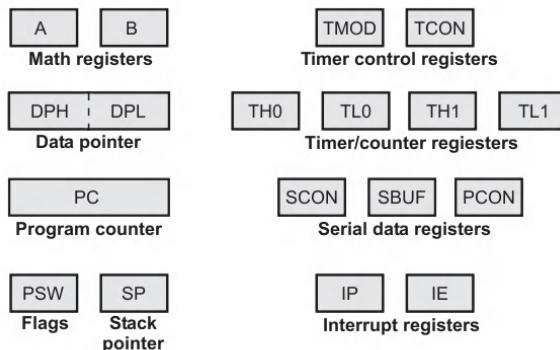
## 9.4 Register Organization of 8051 Microcontroller

- Fig. 9.4.1 shows the register organization of 8051. It shows all CPU registers along with the registers used for timers, interrupts, and serial communication.

### 9.4.1 A and B Registers

#### Register A (Accumulator)

- It is an 8-bit register called **accumulator**. It holds a source operand and receives the result of the arithmetic instructions (addition, subtraction, multiplication and division).



**Fig. 9.4.1 Register organization of 8051**

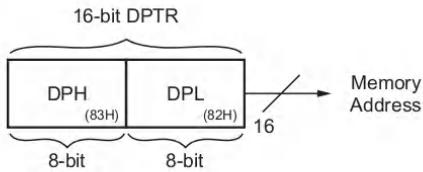
- Several functions apply exclusively to the accumulator : Rotate, parity computation, testing for zero and so on.

### Register B

- In addition to accumulator, an 8-bit B-register is available as a general purpose register. It is used for the hardware multiply/divide operation.

### 9.4.2 Data Pointer (DPTR)

- The data pointer (DPTR) consists of a high byte (DPH) and a low byte (DPL). Its function is to hold a 16-bit address. It may be manipulated as a 16-bit data register or as two independent 8-bit registers. It serves as a base register in indirect jumps, lookup table instructions and external data transfer. The DPTR does not have a single internal address; DPH (83H) and DPL (82H) have separate internal addresses.



### 9.4.3 Program Counter

- The 8051 has a 16-bit program counter. It is used to hold the address of memory location from which the next instruction is to be fetched.

#### 9.4.4 8051 Flag Bits/PSW Registers

- The Fig. 9.4.2 shows the bit pattern of Program Status Word (PSW) of 8051. PSW is also known as **flag register**.

B <sub>7</sub>	B <sub>6</sub>	B <sub>5</sub>	B <sub>4</sub>	B <sub>3</sub>	B <sub>2</sub>	B <sub>1</sub>	B <sub>0</sub>
CY	AC	FO	RS1	RS0	OV	-	P

Fig. 9.4.2

- The 8051 consists of following flags.
- CY-Carry Flag :** This flag is set if there is an overflow out of bit 7. The carry flag also serves as a borrow flag for subtraction. In both the examples shown below, the carry flag is set.
- AC-Auxiliary Carry Flag :** This flag is set if there is an overflow out of bit 3 i.e., carry from lower nibble to higher nibble (D<sub>3</sub> bit to D<sub>4</sub> bit).
- FO - Available for user for general purpose.**
- RS1 - RS0 (Register Bank Select) :** They select the working register bank as follows :

ADDITION				SUBTRACTION			
+ 9B H	+ 75 H	1001 1011	+ 0111 0101		- 89 H	- AB H	- 1000 1001
Carry <input type="checkbox"/> 1	10 H	<input type="checkbox"/> 0001 0000		Borrow <input type="checkbox"/> 1	DE H	<input type="checkbox"/> 1101 1110	- 1010 1011

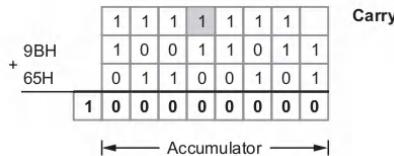
RS1	RS0	Bank Selection	
0	0	00H - 07H	Bank 0
0	1	08H - 0FH	Bank 1
1	0	10H - 17H	Bank 2
1	1	18H - 1FH	Bank 3

- OV-Over Flow Flag :** This flag is set whenever the result of a signed number operation is too large, causing the high-order bit to overflow into the sign bit.
- P-Parity Flag :** Parity is defined by the number of ones present in the accumulator. P = 0, if number of ones are even and P = 1, if number of ones are odd.

**Example :** The status of CY, AC and P flags after the addition of 9BH and 65H is as follows :

$$\text{CY} = 1, \text{AC} = 1 \text{ and } \text{P} = 0$$

- There are instructions in 8051, that tests the condition of flags in the PSW register and make decision based on the status of flags. Thus, programmer use these flags to perform some arithmetic operations which involves carry or borrow, or to change the program control (using conditional branching).



- As mention earlier, programmer can select register bank by setting corresponding bits in PSW.

#### 9.4.5 Special Function Registers

- The group of registers, implemented to perform special functions and are located immediately above the 128 bytes of RAM are called **special function registers**. All access to the four I/O ports, the CPU registers, interrupt control registers, the timer/counter, UART and power control are performed through registers between 80H and FFH.
- Special Function Registers (SFRs) are a sort of control table used for running and monitoring the operation of the microcontroller. Each of these registers as well as each bit they include, has its name, address in the scope of RAM and precisely defined purpose such as timer control, interrupt control, serial communication control etc.
- Even though there are 128 memory locations intended to be occupied by them, the basic core, shared by all types of 8051 microcontrollers, has only 21 such registers. Rest of locations are intentionally left unoccupied in order to enable the manufacturers to further develop microcontrollers keeping them compatible with the previous versions.
- Fig. 9.4.3 shows special function bit addresses. (Refer Fig. 9.4.3 on page 9-14)
- Table 9.4.1 contains a list of all the SFRs and their addresses and their value in binary at reset.

Symbol	Name	Address	Value in Binary
*ACC	Accumulator	0E0H	0 0 0 0 0 0 0 0
*B	B Register	0F0H	0 0 0 0 0 0 0 0

*PSW	Program Status Word	0D0H	0 0 0 0 0 0 0
SP	Stack Pointer	81H	0 0 0 0 0 1 1
DPTR	Data Pointer 2 Bytes		
DPL	Low Byte	82H	0 0 0 0 0 0 0
DPH	High Byte	83H	0 0 0 0 0 0 0
*P0	Port 0	80H	1 1 1 1 1 1 1
*P1	Port 1	90H	1 1 1 1 1 1 1
*P2	Port 2	0A0H	1 1 1 1 1 1 1
*P3	Port 3	0B0H	1 1 1 1 1 1 1
*IP	Interrupt Priority Control	0B8H	8051 X X X 0 0 0 0 8052 X X 0 0 0 0 0
*IE	Interrupt Enable Control	0A8H	8051 0 X X 0 0 0 0 8052 0 X 0 0 0 0 0
TMOD	Timer/Counter Mode Control	89H	0 0 0 0 0 0 0
*TCON	Timer/Counter Control	88H	0 0 0 0 0 0 0
* + T2CON	Timer/Counter 2 Control	0C8H	0 0 0 0 0 0 0
TH0	Timer/Counter 0 High Byte	8CH	0 0 0 0 0 0 0
TL0	Timer/Counter 0 Low Byte	8AH	0 0 0 0 0 0 0
TH1	Timer/Counter 1 High Byte	8DH	0 0 0 0 0 0 0
TL1	Timer/Counter 1 Low Byte	8BH	0 0 0 0 0 0 0
+ TH2	Timer/Counter 2 High Byte	0CDH	0 0 0 0 0 0 0
+ TL2	Timer/Counter 2 Low Byte	0CCH	0 0 0 0 0 0 0
+ RCAP2H	T/C 2 Capture Reg. High Byte	0CBH	0 0 0 0 0 0 0
+ RCAP2L	T/C 2 Capture Reg. Low Byte	0CAH	0 0 0 0 0 0 0
* SCON	Serial Control	98H	0 0 0 0 0 0 0
SBUF	Serial Data Buffer	99H	Indeterminate
PCON	Power Control	87H	HMOS 0 X X X X X X X CHMOS 0 X X X 0 0 0

Table 9.4.1 List of all SFRs ( \* – Bit addressable, + – 8052 only )

\* before register name indicates that it is a bit addressable.

+ before register name indicates that it is supported by only 8052.

Direct byte address (MSB)	Bit address (LSB)								Hardware register symbol
0FFH									B
0F0H	F7 F6 F5 F4 F3 F2 F1 F0								
0E0H	E7 E6 E5 E4 E3 E2 E1 E0								ACC
0D0H	D7 D6 D5 D4 D3 D2 D1 D0								PSW
0B8H	-- -- BC BB BA B9 B8								IP
0B0H	B7 B6 B5 B4 B3 B2 B1 B0								P3
0A8H	AF --- AC AB AA A9 A8								IE
0A0H	A7 A6 A5 A4 A3 A2 A1 A0								P2
98H	9F 9E 9D 9C 9B 9A 99 98								SCON
90H	97 96 95 94 93 92 91 90								P1
88H	8F 8E 8D 8C 8B 8A 89 88								TCON
80H	87 86 85 84 83 82 81 80								P0

Fig. 9.4.3 SFR bit address

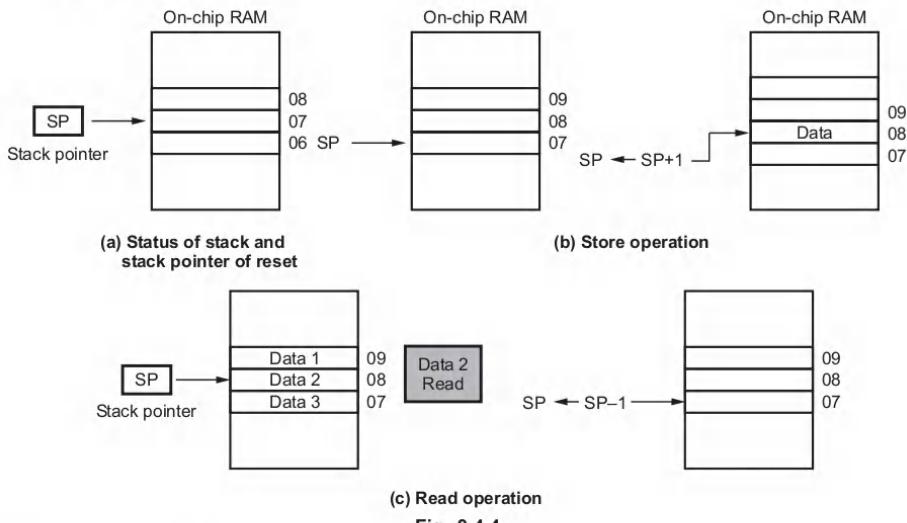
#### 9.4.6 Stack and Stack Pointer

- The stack refers to an area of internal RAM that is used to store and retrieve data quickly. The stack pointer register is used by the 8051 to hold an internal RAM address that is called **top of stack**. The stack pointer register is 8-bit wide. It is increased **before** data is stored during PUSH and CALL instructions and decremented **after** data is restored during POP and RET instructions.

- The stack array can reside anywhere in on-chip RAM. The stack pointer is initialized to 07H after a reset. This causes the stack to begin at location 08H. We can modify default location of stack by loading new location in stack pointer. For example,

MOV SP, # 13H

- The operation of stack and stack pointer is illustrated in Fig. 9.4.4.
- The stack may overwrite data in the register banks, bit-addressable RAM and scratch-pad RAM. Thus to avoid conflict with the register, bit-addressable RAM and scratch-pad RAM data, the stack is initialized at a higher location in the internal RAM.



### Review Questions

- Explain the function of register A.
- Explain the function of register B.
- Explain the function of program counter.
- List out the different bit addressable SFR's available in 8051.
- Explain the significance of Processor Status Word. Briefly discuss PSW register of 8051.
- What is the necessity of a flag register in a microprocessor/microcontroller?
- Explain the utility of bit F0 in the status register of 8051 microcontroller.
- For what condition the OV flag of 8051 is set after the addition instruction.
- What is a stack?

10. Discuss the need for stack memory in microcontrollers. How stack is operated in 8051? What is the default location of stack? How programmer can modify it?
11. Briefly explain the dual functions of port-3 pins of 8051.
12. How stack operates in 8051 CPU?
13. Discuss the need for stack memory in microcontrollers.
14. Write a note on SFR's.

## 9.5 Pin Diagram of 8051

- The 8051 is packaged in a 40-pin DIP. The Fig. 9.5.1 shows the pin diagram of 8051. It is important to note that many pins of 8051 are used for more than one function. The alternative functions of pins are shown in bold letters. Fig. 9.5.1 shows the pin diagram of 8051.

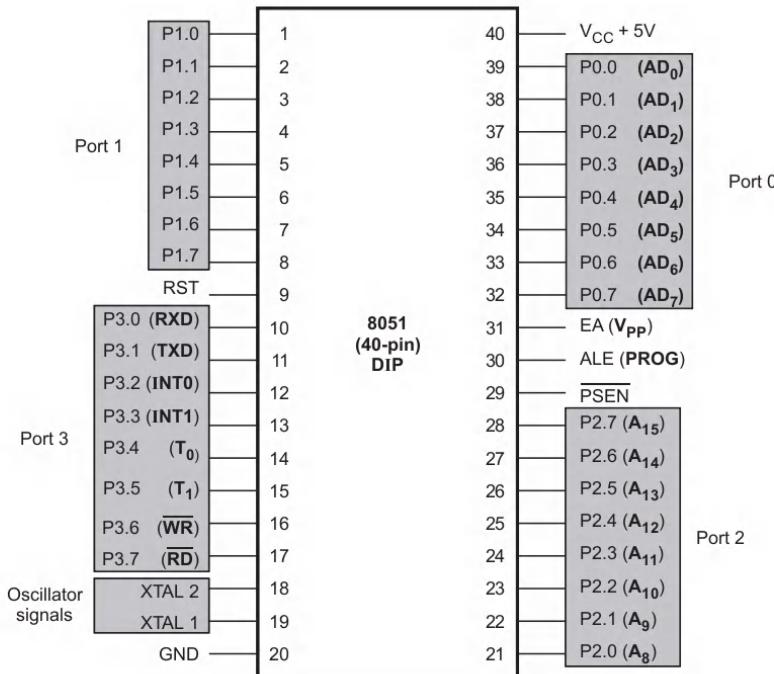


Fig. 9.5.1 Pin-out of 8051

- The 8051 has 32 I/O pins configured as four eight-bit parallel ports (P0, P1, P2 and P3). All four ports are bidirectional i.e. each pin will be configured as input or output (or both). All port-pins are multiplexed except the pins of port 1. Each port consists of a latch, an output driver and an input buffer.

#### **Port 0 (Pins 32 - 39)**

- Port 0 pins can be used as I/O pins. The output drives and input buffers of port 0 are used to access external memory. Port 0 outputs the low order byte of the external memory address, time multiplexed with the data being written or read. Thus, port 0 can be used as a multiplexed address/data bus.

#### **Port 1 (Pins 1 - 8)**

- Port 1 pins can be used only as I/O pins.

#### **Port 2 (Pins 21 - 28)**

- The output drives of port 2 are used to access external memory. Port 2 outputs the high order byte of the external memory address when the address is 16 bits wide. Otherwise, port 2 is used as an I/O port.

#### **Port 3 (Pins 10 - 17)**

- All port pins of port 3 are multifunctional. Therefore, each pin of port 3 can be programmed to use as I/O or as one of the alternate function. They have special functions as shown below including two external interrupts, two counter inputs, two special data lines and two timing control strobes.

Symbol	Position	Alternate Use
$\overline{RD}$	P3.7	External memory read signal.
$\overline{WR}$	P3.6	External memory write signal.
T1	P3.5	External timer 1 input.
T0	P3.4	External timer 0 input.
$\overline{INT1}$	P3.3	External interrupt 1 input.
$\overline{INT0}$	P3.2	External interrupt 0 input.
TXD	P3.1	Serial data output.
RXD	P3.0	Serial data input.

**Table 9.5.1**

#### **Power-supply Pins $V_{CC}$ (Pin 40) and $V_{SS}$ (Pin 20)**

- 8051 operates on d.c. power supply of +5 V with respect to ground. The +5 V is to be connected to pin  $V_{CC}$  and ground to pin  $V_{SS}$  with rated power supply current of 125 mA.

**Oscillator Pins XTAL2 (Pin 18) and XTAL1 (Pin 19)**

- For generating an internal clock signal, the external oscillator is connected at these two pins.

**ALE (Address Latch Enable, Pin 30)**

- AD<sub>0</sub> to AD<sub>7</sub> lines are multiplexed. To demultiplex these lines and for obtaining lower half of an address, an external latch and ALE signal of 8051 is used.

**RST (Reset, Pin 9)**

- This pin is used to reset 8051. For proper reset operation, reset signal must be held high at least for two machine cycles, while oscillator is running.

**PSEN (Program Store Enable, Pin 29)**

- It is the active low output control signal used to activate the enable signal of the external ROM/EPROM. It is activated every six oscillator periods while reading the external memory. Thus, this signal acts as the read strobe to external program memory.

**EA (External Access, Pin 31)**

- When the  $\overline{EA}$  pin is high (connected to V<sub>CC</sub>), program fetches to addresses 0000H through 0FFFH are directed to the internal ROM and program fetches to addresses 1000H through FFFFH are directed to external ROM/EPROM. When  $\overline{EA}$  is low (grounded), all addresses (0000H to FFFFH) fetched by program are directed to the external ROM/EPROM.

**Review Questions**

- Explain the functions of the following pins of 8051  
i) EA ii) ALE iii) RST.
- Explain the functions of following pins of 8051.  
i) ALE ii)  $\overline{EA}$  iii)  $\overline{PSEN}$  iv) RST
- Explain the following pins and its functions of 8051 microcontrollers :  
i) ALE ii)  $\overline{PSEN}$  ii)  $\overline{EA}$  iv)  $\overline{RD}$
- Explain the I/O ports of 8051.
- Briefly explain the dual functions of port-3 pins of 8051.

**9.6 Memory Organization**

- Fig. 9.6.1 shows the basic memory structure for 8051. It can access up to 64 K program memory and 64 K data memory. The 8051 has 4 K bytes of internal program memory and 256 bytes of internal data memory.

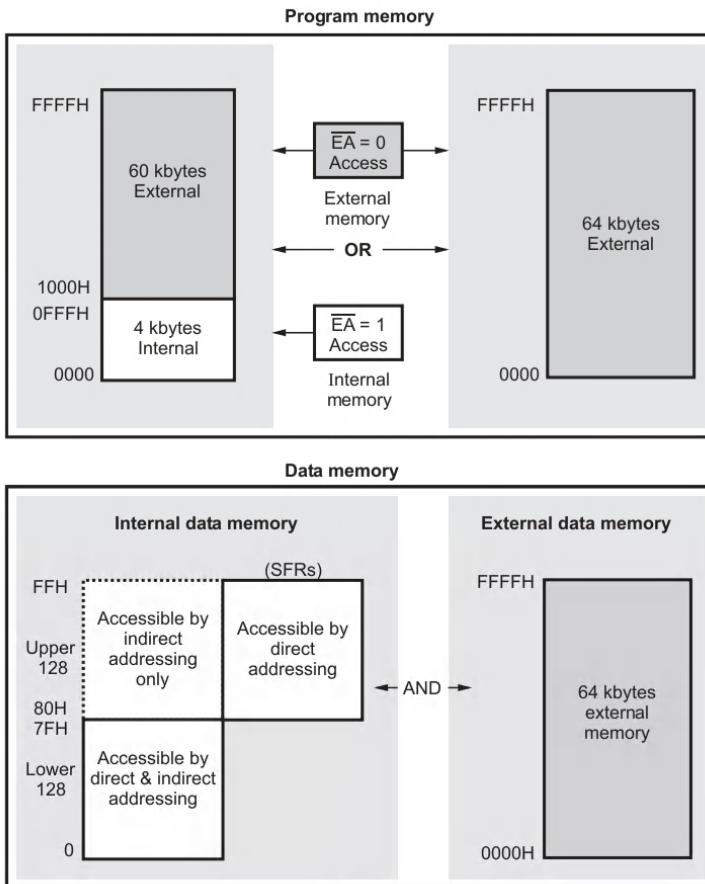


Fig. 9.6.1

### 9.6.1 Internal RAM Organization

- The 8051 has 128-byte internal RAM. It is accessed using RAM address register. The Fig. 9.6.2 (See Fig. 9.6.2 on next page) shows the organization of internal RAM. As shown in the Fig. 9.6.2, internal RAM of 8051 is organized into three distinct areas :
  - Register bank
  - Bit addressable
  - General purpose.

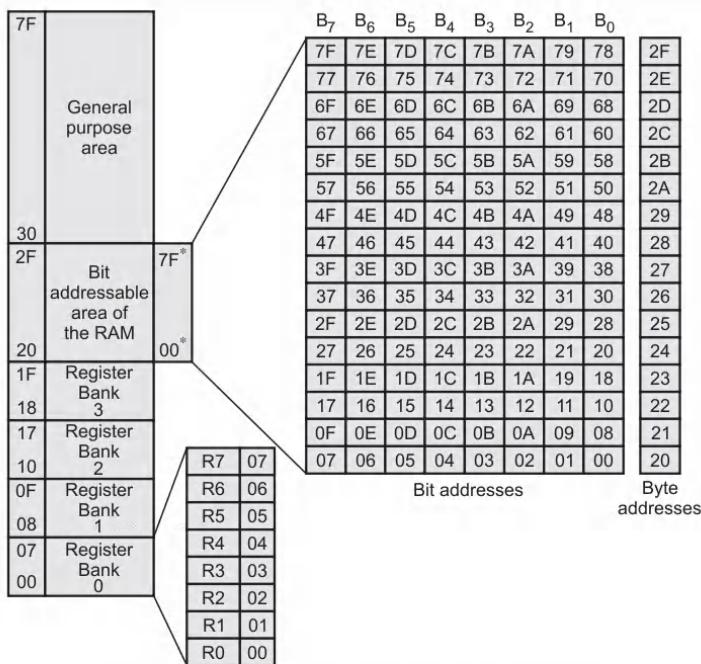


Fig. 9.6.2 Organization of internal RAM of 8051

### 9.6.1.1 8051 Register Banks (Working Registers)

- The first 32-bytes from address 00H to 1FH of internal RAM constitute 32 working registers. They are organized into four banks of eight registers each. The four register banks are numbered 0 to 3 and are consists of eight registers named R<sub>0</sub> to R<sub>7</sub>.
- Each register can be addressed by name or by its RAM address.
- Only one register bank is in use at a time. Bits RS0 and RS1 in the PSW determine which bank of registers is currently in use.

RS1 (PSW.4)	RS0 (PSW.3)	Bank selection
0	0	Bank 0
0	1	Bank 1
1	0	Bank 2
1	1	Bank 3

- On reset, the bank 0 is selected and hence it is a default register bank. Register banks when not selected can be used as general purpose RAM.

#### 9.6.1.2 Bit / Byte Addressable

- The 8051 provides 16 bytes of a bit-addressable area. It occupies RAM byte addresses from 20H to 2FH, forming a total of 128 ( $16 \times 8$ ) addressable bits.
- An addressable bit may be specified by its bit address of 00H to 7FH, or 8 bits may form any byte address from 20H to 2FH.
- For example, bit address 4EH refers bit 6 of the byte address 29H.

#### 9.6.1.3 General Purpose RAM

The RAM area above bit addressable area from 30H to 7FH is called general purpose RAM. It is addressable as byte.

#### 9.6.2 ROM Space in the 8051

- The 8051 has 4 kbyte of internal ROM with address space from 0000H to 0FFFH. It is programmed by manufacturer when the chip is built. This part cannot be erased or altered after fabrication. This is used to store final version of the program.
- It is accessed using program address register.

#### Review Questions

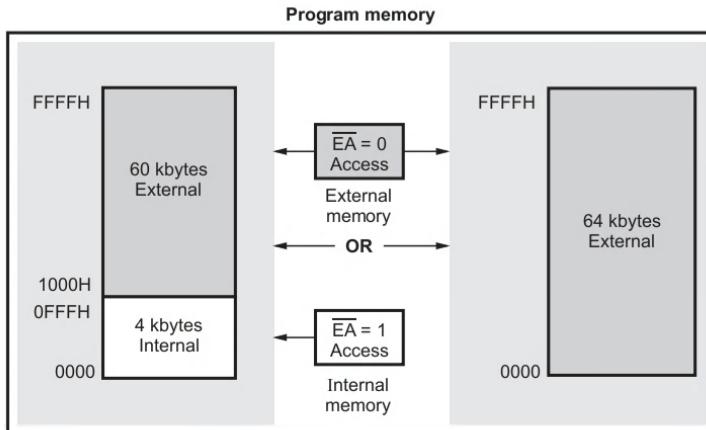
- Draw and explain the memory structure of 8051.
- Explain the internal RAM organization of 8051 microcontroller.
- Comment on ROM space in the 8051.
- With neat diagram, with the programming model of 8051 with addresses of SFR's and ports. Also give 128 bytes RAM allocation.

### 9.7 External Memory Interfacing

- We have seen that 8051 has internal data and code memory with limited memory capacity. This memory capacity may not be sufficient for some applications. In such situations, we have to connect external ROM/EPROM and RAM to 8051 microcontroller to increase the memory capacity. We also know that ROM is used as a program memory and RAM is used as a data memory. Let us see how 8051 accesses these memories.

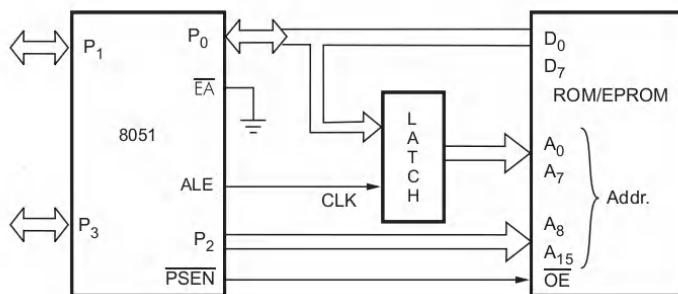
### 9.7.1 External Program Memory

- Fig. 9.7.1 shows a map of the 8051 program memory.



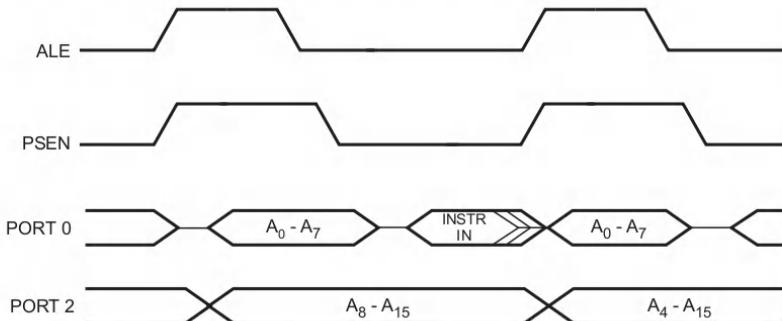
**Fig. 9.7.1 The 8051 program memory**

- In 8051, when the  $\overline{EA}$  pin is connected to  $V_{CC}$ , program fetches to addresses 0000H through 0FFFH are directed to the internal ROM and program fetches to addresses 1000H through FFFFH are directed to external ROM/EPROM. On the other hand when  $\overline{EA}$  pin is grounded, all addresses (0000H to FFFFH) fetched by program are directed to the external ROM/EPROM. The  $\overline{PSEN}$  signal is used to activate output enable signal of the external ROM/EPROM, as shown in the Fig. 9.7.2.



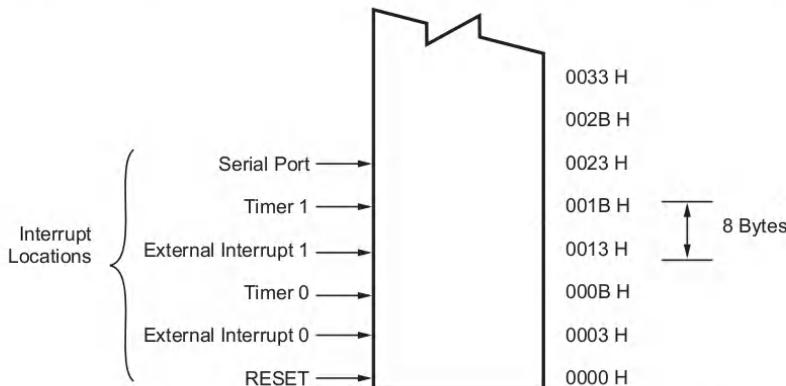
**Fig. 9.7.2 Accessing external program memory**

- As shown in the Fig. 9.7.2, the port 0 is used as a multiplexed address/bus. It gives lower order 8-bit address in the initial T-cycle and later it is used as a data bus. The 8-bit address is latched using external latch and ALE signal generated by 8051.
- The port 2 provides the higher order 8-bit address. Fig. 9.7.3 shows the timing waveforms for external program memory read cycle.



**Fig. 9.7.3 Timing waveforms for external program memory read cycle**

- The lower part of program memory stores the vector addresses for various interrupt service routines. Fig. 9.7.4 shows the vector address map. Each interrupt is assigned with a fixed location in program memory. For example, external interrupt 0 is assigned to location 0003H. The interrupt service locations are spaced at 8-byte intervals such as 0003H for External Interrupt 0, 000BH for Timer 0, 0013H for External Interrupt 1, 001BH for Timer 1, etc. If interrupt is going to be



**Fig. 9.7.4 Interrupt/Vector locations in the lower part of program memory**

used, its service routine must begin at corresponding location. If the interrupt is not going to be used, its service location is available as general purpose program memory.

### Instructions to Access External ROM/Program Memory

The Table 9.7.1 explains the instructions to access external ROM/program memory.

Mnemonic	Operation
MOVC A, @ A + DPTR	Copy the contents of the external ROM address formed by adding A and the DPTR, to A.
MOVC A, @ A + PC	Copy the contents of the external ROM address formed by adding A and the PC, to A.

Table 9.7.1

### 9.7.2 External Data Memory

- Fig. 9.7.5 shows a map of the 8051 data memory.

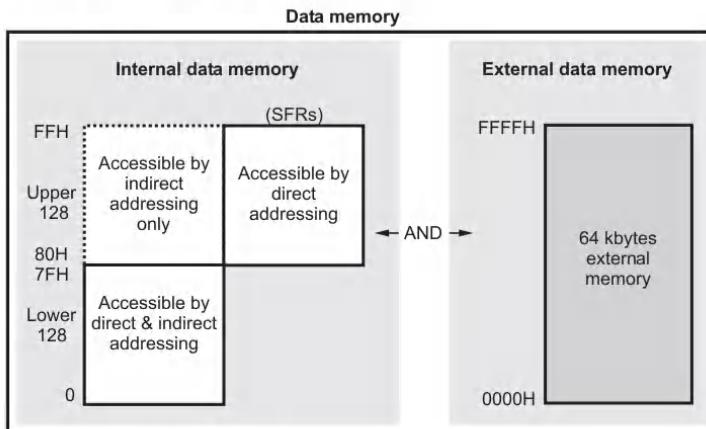


Fig. 9.7.5 A map of the 8051 data memory

- The 8051 can address up to 64 kbytes of external data memory. The "MOVX" instruction is used to access the external data memory. The internal data memory space for 8051 is divided into three blocks : Lower 128 bytes, Upper 128 bytes and SFRs. The upper addresses and SFRs occupy the same block of address space, 80H through FFH, although they are physically separate entities. As shown in the Fig. 9.7.5, the upper address space is accessible by indirect addressing only and SFRs are accessible by direct addressing only. On the other hand, lower address space can be accessed either by direct addressing or by indirect addressing.

- Fig. 9.7.6 shows the circuit diagram for connecting external data memory. The multiplexed address/data bus provided by port 0 is demultiplexed by external latch and ALE signal. Port 2 gives the higher order address bus. The  $\overline{RD}$  and  $\overline{WR}$  signals from 8051 selects the memory read and memory write operation, respectively.
- Fig. 9.7.7 (a) and (b) show the timing waveforms for external data memory read and write cycles, respectively.

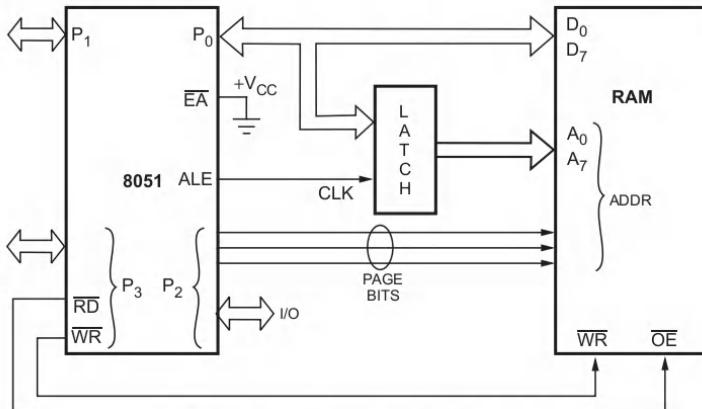


Fig. 9.7.6 Accessing external data memory

#### Instructions to Access External Data Memory

The Table 9.7.2 explains the instruction to access external data memory.

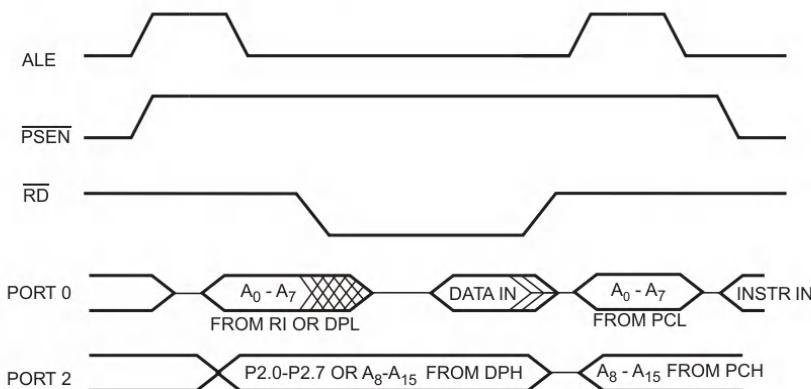


Fig. 9.7.7 (a) Timing waveforms for external data memory read cycle

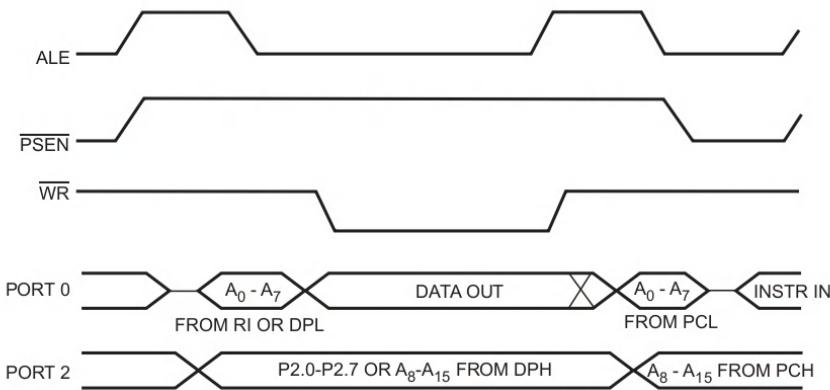


Fig. 9.7.7 (b) Timing waveforms for external data memory write cycle

Mnemonic	Operation
MOVX A, @Rp	Copy the contents of the external address in Rp to A.
MOVX A, @DPTR	Copy the contents of the external address in DPTR to A.
MOVX @Rp, A	Copy data from A to the external address in Rp.
MOVX @DPTR, A	Copy data from A to the external address in DPTR.

Table 9.7.2

### 9.7.3 Accessing External Data Memory in 8051C

- All external data moves with external ROM or external RAM involve the A register.
- While accessing external memory, R<sub>p</sub> can address 256 bytes and DPTR can address 64 kbytes.
- MOVX instruction is used to access external RAM or I/O addresses.
- When PC is used to access external ROM, it is incremented by 1 (to point to the next instruction) before it is added to A to form the physical address of external ROM.

**Example 9.7.1** Draw and explain 8051 connection to external RAM (8K × 8). Write a program to read 100 bytes of data from P1 and save the data in external RAM starting at 5100H location.

**Solution :**

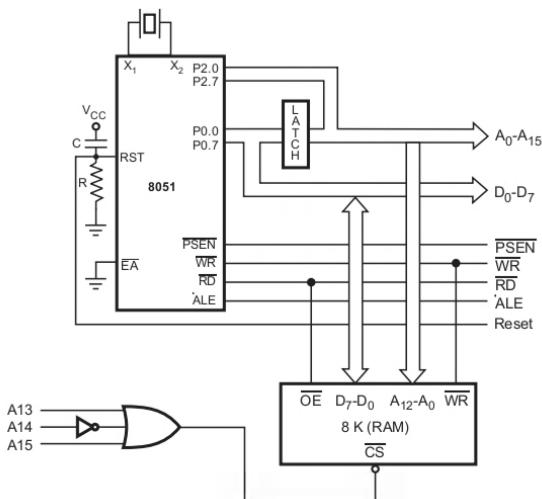


Fig. 9.7.8

### RAM Map

A <sub>15</sub>	A <sub>14</sub>	A <sub>13</sub>	A <sub>12</sub>	A <sub>11</sub>	A <sub>10</sub>	A <sub>9</sub>	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	RAM Address
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Start
0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFFH

```

MOV R0, # 100          ; Initialize counter
MOV DPTR, # 5100H
MOV P1, # OFFH          ; Configure P1 as an input port
BACK :                 ; Get data from P1
    MOV A, P1
    MOVX @ DPTR, A        ; Send it to external RAM
    INC DPTR
    DNZ R0, BACK

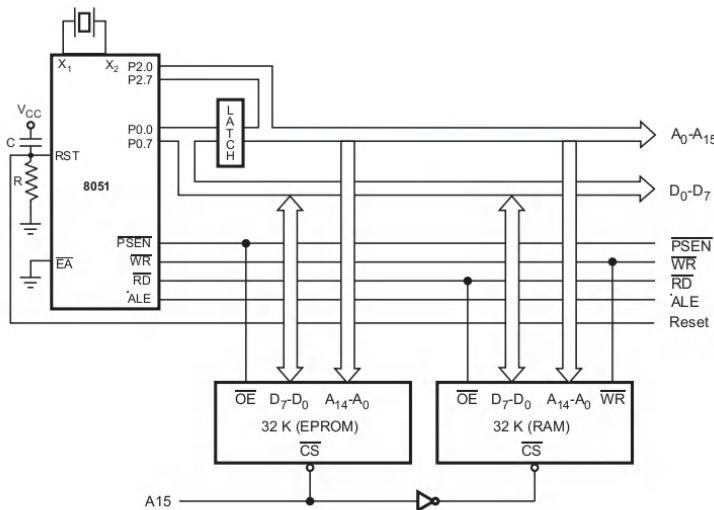
```

**Example 9.7.2** Discuss interfacing of external 32 K EPROM and 32 K RAM with the microcontroller. Draw diagram and explain.

**Solution :** To address 32 K EPROM and RAM needs 15 address lines (A<sub>0</sub> - A<sub>14</sub>). The remaining address line A<sub>15</sub> is used for selection of EPROM or RAM. When A<sub>15</sub> is low EPROM is selected and when A<sub>15</sub> is high RAM is selected.

**Memory Map**

Memory	A <sub>15</sub>	A <sub>14</sub>	A <sub>13</sub>	A <sub>12</sub>	A <sub>11</sub>	A <sub>10</sub>	A <sub>9</sub>	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	Address
RAM	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8000H (Start)
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFFH (End)
EPROM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000H (Start)
	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	7FFFH (End)

**Fig. 9.7.9**

**Example 9.7.3** An 8051 based system requires external memory of four 4 kbytes of SRAM each and two chips of EPROM of size 2 kbytes. The EPROM starts at address 2000H. SRAM address map follows EPROM map. Give the complete interface.

**Solution :** See Fig. 9.7.10 on next page.

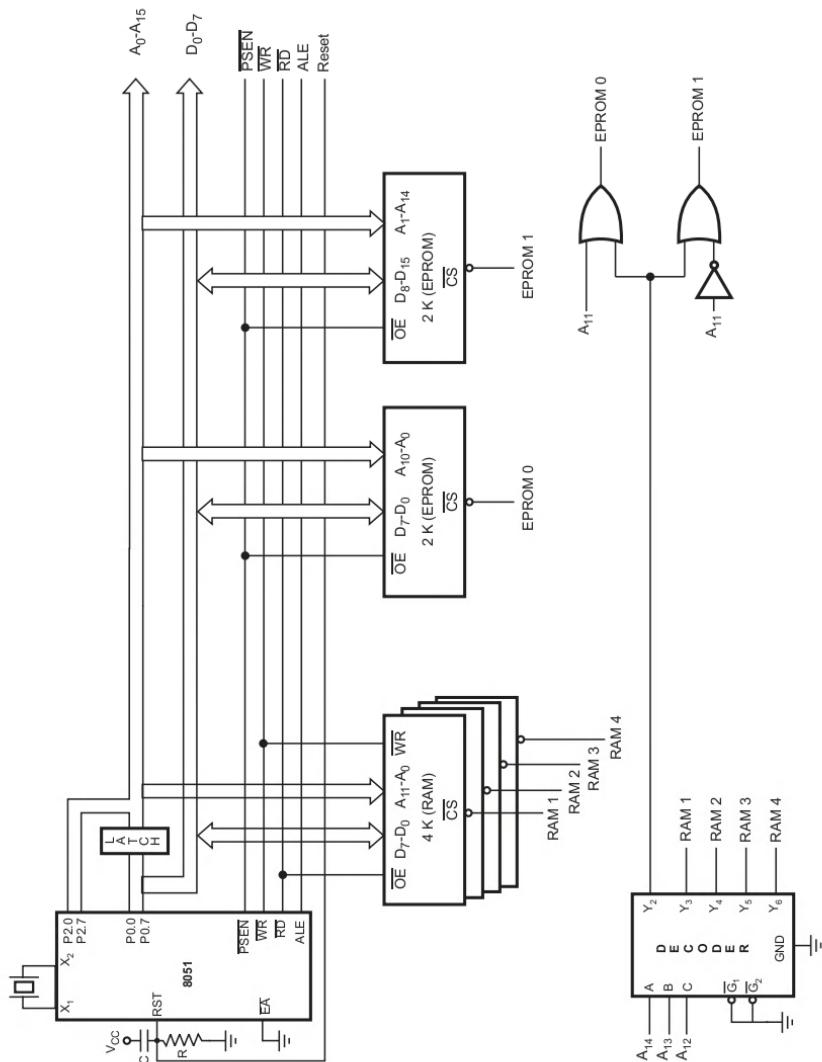


Fig. 9.7.10

A <sub>15</sub>	A <sub>14</sub>	A <sub>13</sub>	A <sub>12</sub>	A <sub>11</sub>	A <sub>10</sub>	A <sub>9</sub>	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	Address	
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2000H	EPROM0
0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	27FFH	
0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	2800H	EPROM1
0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	2FFFH	
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	3000H	RAM0
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3FFFH	
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000H	RAM1
0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	4FFFH	
0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	5000H	RAM2
0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	5FFFH	
0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	6000H	RAM3
0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	6FFFH	

**Example 9.7.4** Give the complete block schematic of an 8051 based system having following specifications.

64 kB of program memory.

64 kB of data memory.

Make use of 16 K × 8-bit memory chips and 74 LS 138 decoders.

Indicate clearly, the addresses selected for the memory chips.

**Solution :**

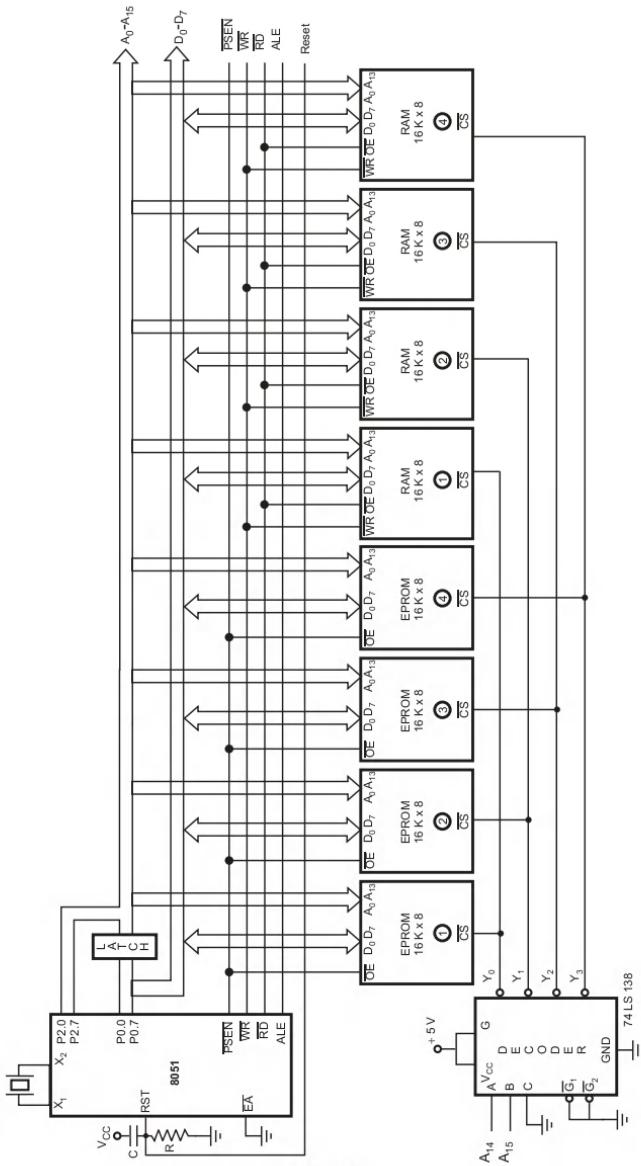


Fig 9.7.11

**Memory Map :**

Memory	A <sub>15</sub>	A <sub>14</sub>	A <sub>13</sub>	A <sub>12</sub>	A <sub>11</sub>	A <sub>10</sub>	A <sub>9</sub>	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	Address
EPROM1 Start	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000H
EPROM1 End	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3FFFH
EPROM2 Start	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000H
EPROM2 End	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	7FFFH
EPROM3 Start	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8000H
EPROM3 End	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	BFFFH
EPROM4 Start	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	C000H
EPROM4 End	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFFH
RAM1 Start	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000H
RAM1 End	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3FFFH
RAM2 Start	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000H
RAM2 End	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	7FFFH
RAM3 Start	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8000H
RAM3 End	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	BFFFH
RAM4 Start	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	C000H
RAM4 End	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	FFFFH

**Table 9.7.3****Review Questions**

1. Explain interfacing and timing diagrams for external program memory interfacing.
2. Explain interfacing and timing diagrams for external data memory interfacing.
3. Interface the external ROM and RAM to 8051. Explain how to access them.
4. Interface 8051 to 8K external RAM and 32K external ROM and explain how 8051 access them ?
5. With the help of a diagram, explain how to interface 8 KB EPROM and 8 KB RAM, to 8051 Microcontroller.



# Microprocessor Laboratory

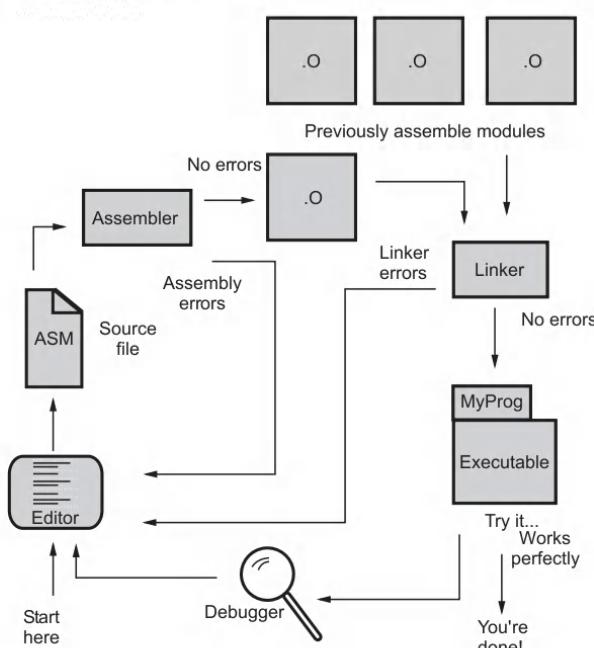
## Contents

- Assignment 1 : Write an X86/64 ALP to accept five 64 bit Hexadecimal numbers from user and store them in array and display the accepted numbers.
- Assignment 2 : Write an X86/64 ALP to accept a string and to display its length.
- Assignment 3 : Write an X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers.
- Assignment 4 : Write a switch case driven X86/64 ALP to perform 64-bit hexadecimal arithmetic operations (+,-,\*,/) using suitable macros. Define procedure for each operation.
- Assignment 5 : Write X86/64 ALP to count number of positive and negative numbers from the array.
- Assignment 6 : Write X86/64 ALP to convert 4-digit Hex number into its equivalent BCD number and 5-digit BCD number into its equivalent HEX number. Make your program user friendly to accept the choice from user for : (a) HEX to BCD b) BCD to HEX (c) EXIT. Display proper strings to prompt the user while accepting the input and displaying the result. (wherever necessary, use 64-bit registers).
- Assignment 7 : Write X86/64 ALP to switch from real mode to protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers.
- Assignment 8 : Write X86-64 ALP to perform non-overlapped block transfer without string specific instructions. Block containing data can be defined in the data segment.
- Assignment 9 : Write X86/64 ALP to perform overlapped block transfer with string specific instructions block containing data can be defined in the data segment.
- Assignment 10 : Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use successive addition and add and shift method. (use of 64-bit registers is expected).
- Assignment 11 : Write X86 menu driven Assembly Language Program (ALP) to implement OS (DOS) commands TYPE, COPY and DELETE using file operations. User is supposed to provide command line arguments in all cases.
- Assignment 12 : Write X86 ALP to find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character. Accept the data from the text file. The text file has to be accessed during Program\_1 execution and write FAR PROCEDURES in Program\_2 for the rest of the processing. Use of PUBLIC and EXTERN directives is mandatory.
- Assignment 13 : Write x86 ALP to find the factorial of a given integer number on a command line by using recursion. Explicit stack manipulation is expected in the code.

## Assembly Language Programming using NASM

### Assembly Language Development Process

1. Create your assembly language source code file in a text editor.
2. Use your assembler to create an object module from your source code file.
3. Use your linker to convert the object module (and any previously assembled object modules that are part of the project) into a single executable program file.
4. Test the program file by running it, using a debugger if necessary.
5. Go back to the text editor in step 1, fix any mistakes you may have made earlier, and write new code as necessary.
6. Repeat steps 1-5 until done.



**Fig. L.1 Assembly language development process**

### What Is NASM ?

The Netwide Assembler, NASM, is an 80x86 assembler designed for portability and modularity. It supports a range of object file formats, including Linux and

NetBSD/FreeBSD a.out, ELF, COFF, Microsoft 16-bit OBJ and Win32. It will also output plain binary files.

### Command to Install NASM on Ubuntu Linux

```
sudo apt-get install nasm
```

### Assemble the Program with NASM

We use NASM assembler to assemble our .asm program. The NASM assembler does not have a user interface. NASM works via text only, and you communicate with it through a terminal and a Linux console session. It's like those old DOS days when everything had to be entered on the command line.

- f elf :** The -f command tells NASM which format to use for the object code file it's about to generate. In 32-bit IA-32 Linux work, the format is ELF32, which can be specified on the command line as simply elf.
- g :** The -g command tells NASM to include debugging information in the output file.
- f stabs :** The command specifies that debug information is to be generated in the stabs format.

**Note :** -g and -f stabs commands are optional.

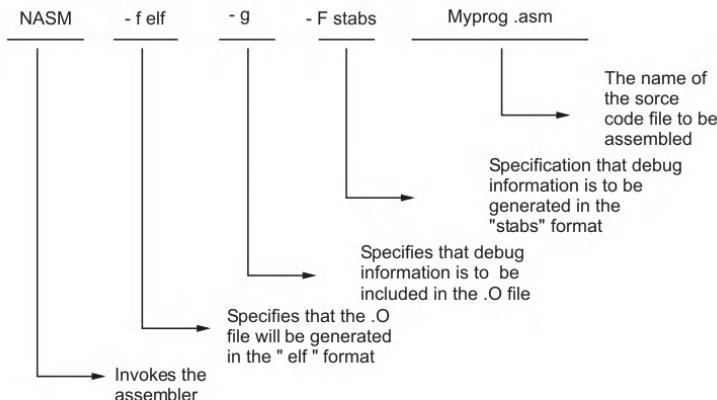


Fig. L.2 Command line to assemble the .asm file

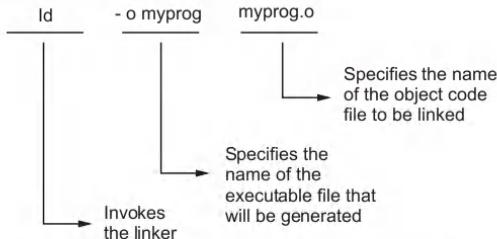
### Simple Command to Assemble .asm File

1. nasm -f elf myprog.asm or  
nasm-f elf 32 myprog.asm.      ... for 32-bit program on  
    32-bit OS
2. nasm -f elf64 myprog.asm      ... for 64-bit program on  
    64-bit OS

The above command will assemble myfile.asm into an ELF object file myprog.o.

### Link the Program with LD

The command line for linking is simpler than the one for assembling, as shown in Fig. L.3. The "ld" runs the linker program itself. The -o command specifies an output filename. In this case it is myprog.



**Fig. L.3 Command for linking the object file**

In the DOS and Windows world, executable files almost always use the .exe file extension. In the Linux world, executables generally have **no file extension at all**.

### Command Line to Link 32-bit Program on 64-bit OS

ld-m elf\_i386 -s -o myprog myprog.o

### Test the Executable File

Once the linker completes an error-free pass, our finished executable file will be waiting for us in our working directory. It's error-free if the assembler and linker processed it without displaying any error messages. However, error-free does not imply bug-free. To make sure it works, just name it on the terminal command line :

./myprog

### Linux System Calls

We can make use of Linux system calls in our assembly programs. We need to take the following steps for using Linux system calls in our program.

1. Put the system call number in the EAX register.
2. Store the arguments to the system call in the registers EBX, ECX, etc.
3. Call the relevant interrupt (80h)
4. The result is usually returned in the EAX register.

There are six registers that store the arguments of the system call used. These are the EBX, ECX, EDX, ESI, EDI and EBP. These registers take the consecutive arguments, starting with the EBX register. If there are more than six arguments then the memory location of the first argument is stored in the EBX register.

The INT 80H instruction is a special one. It generates a Linux system call (also referred to as a syscall).

#### Sending Data/Message to the Currently Active Terminal Window

```
mov eax,4          ; 04H specifies the sys_write kernel call  
mov ebx,1          ; 01H specifies stdout  
mov ecx, Message    ; Load starting address of display string into  
                      ; ECX  
mov edx, MessageLength ; Load the number of chars to display into EDX  
int 80H           ; Make the kernel call
```

#### Exiting a Program via INT 80h

The sys\_exit call is used shutting down the program and returning control to Linux. Here we need to place the number of the sys\_exit service in EAX, place a return code in EBX, and then execute INT 80h:

```
mov eax, 1          ; Specify Exit syscall  
mov ebx, 0          ; Return a code of zero  
int 80H            ; Make the syscall to terminate the  
                      ; program
```

The return code is a numeric value that we can define. Technically, there are no restrictions on what it is (aside from having to fit in a 32-bit register), but by convention a return value of 0 means "everything worked OK; shutting down normally." Return values other than 0 typically indicate an error of some sort.

#### Read a character from standard input (stdin)

```
mov eax, 3          ; Specify sys_read call  
mov ebx, 0          ; Specify File Descriptor 0: Standard Input  
mov ecx, Buff       ; Pass address of the buffer to read to  
mov edx, 1          ; Tell sys_read to read one char from stdin  
int 80h            ; Call sys_read  
sys_read's return value in EAX.
```

Buff is an uninitialized variable. It has no initial value, and contains nothing until we read a character from stdin and store it there.

When a call to sys\_read returns a 0, sys\_read has reached the end of the file it's reading from. If it returns a positive value, this value is the number of characters it has read from the file. In this case, since we only requested one character, sys\_read returns either a count of 1, or a 0 indicating that we are out of characters.

### **Linux SYSCALL (for 64 bit execution)**

#### **Steps for using Linux system call for 64 bit execution**

1. Write the system call number in RAX.
2. Set up the arguments to the system call in RDI, RSI, RDX, etc.
3. Make a system call with "SYSCALL" instruction.
4. The result is usually returned in RAX.

#### **Sending Data / Message to the Currently Active Terminal Window**

```
mov    rax, 1          ; 01 specifies sys_write kernel call
mov    rdi, 1          ; 01 specifies stdout
mov    rsi, message    ; Load starting address of message into rsi
mov    rdx, length     ; Load message string length into rdx
syscall
```

#### **Exiting a Program**

```
mov    rax, 60         ; sys_exit
mov    rdi, 0          ; return 0 (success)
syscall
```

#### **Read a Character(s) from standard input**

```
mov rax, 0            ; Specifies system call number (sys_read)
mov rdi, 0            ; Specifies (stdin)
mov rdx, n            ; Tell sys_read to read n characters
syscall              ; call kernel
```

#### **Structure of ALP (NASM Syntax)**

An assembly program can be divided into three sections :

- The data section
- The bss section
- The text section

#### **The data section**

The data section is used for declaring initialized data and constants. This data does not change at runtime. You can declare various constant values, file names or buffer size, etc. in this section. The syntax for declaring data section is : **section .data**.

## The bss section

The bss section is used for declaring variables. The syntax for declaring bss is : **section .bss**.

## The text section

The text section is used for keeping the actual code. This section must begin with the declaration global main, which tells kernel where the program execution begins. The syntax declaring text section is : **section .text**.

```
global_start
```

```
_start :
```

## Structure of program

```
section.data
```

```
declaring initialized data or constants
```

```
section.bss
```

```
declaration of variables
```

```
section.text
```

```
actual logic of program code section
```

```
exit : mov ebx, 0 ; sys_exit system call
      mov eax, 1      ; terminate and return to
      int 80h          ; Linux
```

## Editing, Assembling, Linking and Executing First Program : hello.asm

1. Boot the machine with ubuntu
2. Select and click on <dash home> icon from the toolbar.
3. Start typing "terminal".
4. Click on "terminal" icon. A terminal window will open showing command prompt.
5. Give the following command at the prompt to invoke the editor gedit hello.asm

## ; ALP to Print "Hello World" using 32-bit model (64-bit OS)

```
section .data
msg db 'Hello, world!',0xa ; message and newline
len equ $ - msg           ; length of message string

section .text

global _start              ; must be declared for linker (ld)
_start:                   ; tell linker entry point

mov eax,4                 ; Specifies system call number (sys_write)
mov ebx,1                 ; Specifies (stdout)
```

```

mov ecx,msg           ; Address of message to write
mov edx,len           ; Message length
int 0x80              ; call kernel

mov eax,1              ; Specifiy sys_exit call number
mov ebx,0              ; Return a code of zero
int 0x80              ; Make the syscall to terminate the program

```

**; ALP to Print "Hello World" using 64-bit model**

```

section .data

message: db 'Hello, world', 0x0A ; message and newline
length: equ    $-message        ; length of message string

section .text

global _start             ; global entry point export for ld

_start:

mov    rax, 1               ; 01 specifies sys_write kernel call
mov    rdi, 1               ; 01 specifies stdout
mov    rsi, message         ; Load starting address of message into rsi
mov    rdx, length          ; Load message string length into rdx
syscall

mov    rax, 60               ; sys_exit
mov    rdi, 0               ; return 0 (success)
syscall

```

6. Type in the program in gedit window, save and exit
7. To assemble the program write the command at the prompt as follows and press enter key
 

```
nasm -f elf32 hello.asm -o hello.o (for 32 bit)
```

```
nasm -f elf64 hello.asm -o hello.o (for 64 bit)
```
8. If the execution is error free, it implies hello.o object file as been created.
9. To link and create the executable give the command as

```
ld -o hello hello.o
```

10. To execute the program write at the prompt

```
./hello
```

11. Hello, world will be displayed at the prompt.

## RESB and Friends : Declaring Uninitialized Data

RESB, RESW, RESD, RESQ, REST, RESO, RESY and RESZ are designed to be used in the BSS section of a module: they declare uninitialized storage space. Each takes a single operand, which is the number of bytes, words, doublewords or whatever to reserve.

For example :

buffer:	resb	64	; reserve 64 bytes
wordvar:	resw	1	; reserve a word
realarray	resq	10	; array of ten reals
ymmvval:	resy	1	; one YMM register
zmmvals:	resz	32	; 32 ZMM registers

**Assignment 1 :** Write an X86/64 ALP to accept five 64 bit Hexadecimal numbers from user and store them in array and display the accepted numbers.

```
;;
section .data
msg db 10,'Enter Five 64-bit Hex numbers :,10
msg_len equ $-msg
msg1 db 10,'The numbers are : '
msg1_len equ $-msg1
msg2 db 13
msg2_len equ $-msg2
array DQ 0, 0, 0, 0, 0
arraycnt db 0
;;

section .bss
dispbuff resb 2
buf resb 18
buf_len: equ $-buf
len resb 1
%macro print 2
    mov rax,4      ; Specifies system call number (sys_write)
    mov rbx,1      ; Specifies (stdout)
    mov rcx,%1     ; Address of message to write
    mov rdx,%2     ; Message length
    int 0x80       ; call kernel
%endmacro
%macro accept 2
    mov rax,3      ; System call for Read
    mov rbx,0      ; from standard input
    mov rcx,%1     ; Start address
```

```
    mov rdx,%2          ; size
    int 80h             ; Invoke operating system to Read
%endmacro

;-----


section .text
    global _start
_start:
    print msg,msg_len   ; Display message
    mov byte[arraycnt], 5 ; Initialize number count
    mov edi, array        ; Initialize array pointer
back:
    accept buf, buf_len ; Read 64-bit Hex number
    mov esi, buf          ; Initialize Buffer pointer
    mov rcx, 8            ; Initialize byte counter
bk:   mov al, [esi]       ; Get ASCII Character
      cmp al, 39h         ; Connvert to hex as upper nibble
      jbe skip2
      sub al, 07h
skip2: sub al, 30h
      shl al, 4
      mov bl, al
      inc esi             ; Increment buffer point to read next byte
      mov al, [esi]         ; Get ASCII character
      cmp al, 39h         ; Connvert to hex as lower nibble
      jbe skip3
      sub al, 07h
skip3: sub al, 30h
      add al, bl           ; Pack two nibbles
      mov [edi], al         ; Save byte in array
      inc edi              ; Increment array pointer
      inc esi              ; Increment buffer pointer
      loop bk              ; If byte count is not zero repeat
      dec byte[arraycnt]  ; Decrement number count
      jnz back              ; If number count is not zero repeat

    print msg1,msg1_len   ; display message
    mov byte[arraycnt], 5 ; Initialize number count
    mov esi, array        ; Initialize array counter
back1: mov edi, buf        ; Initialize buffer counter
      print msg2,msg2_len ; newline
      mov rcx, 8           ; Initialize byte counter
```

```
bk1:    mov al, [esi]          ; Get packed byte
        mov bl, al              ; Unpack nibbles and Convert them to ASCII
        and al, 0f0h
        shr al, 4
        cmp al, 09
        jbe ss1
        add al, 07h
ss1:    add al, 30h
        mov [edi], al           ; Save ASCII value of upper nibble
        inc edi
        and bl, 0fh
        cmp bl, 09
        jbe ss2
        add bl, 07h
ss2:    add bl, 30h
        mov [edi], bl           ; Save ASCII value of lower nibble
        inc edi                ; Increment buffer pointer
        inc esi                ; Increment array pointer
        loop bk1               ; If byte count is not zero repeat
        print buf, 16           ; Display 64- bit Hex number
        dec byte[arraycnt]     ; If number count is not zero repeat
        jnz back1
exit:   mov rax,1             ; Specifiy sys_exit call number
        mov rbx,0              ; Return a code of zero
        int 0x80                ; Make the syscall to terminate the program
```

### Output

Enter Five 64-bit Hex numbers :

1234567898a65432

9234567898765499

5234567898765436

9234567898765488

1234567898765477

The numbers are :

1234567898A65432

9234567898765499

5234567898765436

9234567898765488

1234567898765477

**Assignment 2 : Write an X86/64 ALP to accept a string and to display its length.**

```
;-----  
section .data  
    msg db 10,'Enter the string : ',10  
    msg_len equ $-msg  
    msg1 db 10,'Length of string in Hex = '  
    msg1_len equ $-msg1  
;  
  
section .bss  
    dispbuff resb 2  
    buf resb 50  
    buf_len: equ $-buf  
    len resb 1  
%macro print 2  
    mov rax,4 ; Specifies system call number (sys_write)  
    mov rbx,1 ; Specifies (stdout)  
    mov rcx,%1 ; Address of message to write  
    mov rdx,%2 ; Message length  
    int 0x80 ; call kernel  
%endmacro  
%macro accept 2  
    mov rax,3 ; System call for Read  
    mov rbx,0 ; from standard input  
    mov rcx,%1 ; Start address  
    mov rdx,%2 ; size  
    int 80h ; Invoke operating system to Read  
%endmacro  
;  
  
section .text  
    global _start  
_start:  
    print msg,msg_len ; display message  
    accept buf, buf_len  
    dec al ; Adjust the count of number bytes read  
    dec al ; Decrement count for string terminating Character  
    mov [len], al ; Save length  
    print msg1,msg1_len ; display message  
    mov bl, [len] ; Restore length
```

```

call disp8num    ; display string length

exit:
    mov rax,1      ; Specifiy sys_exit call number
    mov rbx,0      ; Return a code of zero
    int 0x80       ; Make the syscall to terminate the program

disp8num:
    mov rdi,dispbuff ; point edi to buffer
    mov rcx,2        ; load number of digits to display

dispup1:
    rol bl,4        ; rotate number left by four bits
    mov dl,bl        ; move lower byte in dl
    and dl,0fh       ; mask upper digit of byte in dl
    add dl,30h       ; add 30h to calculate ASCII code
    cmp dl,39h       ; compare with 39h
    jbe dispskip1    ; if less than 39h skip adding 07 more
    add dl,07h       ; else add 07

dispskip1:
    mov [rdi],dl     ; store ASCII code in buffer
    inc rdi          ; point to next byte
    loop dispup1     ; decrement the count of digits to display
                      ; if not zero jump to repeat
    print dispbuff,2 ; display maximum number
    ret              ; return to calling program

```

**Output**

Enter the string : Microprocessor

Length of string in Hex = 0E

**Assignment 3 :** Write an X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers.

; ALP to find the maximum number

---

```

section .data
welmsg db 10,'The maximum number in the array is :',
welmsg_len equ $-welmsg
array dq
8abc123456781234h,90ff123456781234h,7700123456781234h,8800123456781234h,8a9fdd345
6781234h
arrcnt equ 5
;
```

---

```
section .bss
    dispbuff resb 2
    buf resb 16
%macro print 2
    mov eax,4      ; Specifies system call number (sys_write)
    mov ebx,1      ; Specifies (stdout)
    mov ecx,%1    ; Address of message to write
    mov edx,%2    ; Message length
    int 0x80      ; call kernel
%endmacro

;-----



section .text
    global _start
_start:
    print welmsg,welmsg_len ; display message
    mov esi,array           ; initialize array pointer
    mov rax, [esi]           ; get first number as maximum
    mov ecx,arrcnt          ; initialize counter
up1:   add esi, 8           ; Increment array pointer to point next number

        mov rbx, [esi]           ; Get next number
        cmp rax, rbx             ; Compare two numbers
        jnc skip
        mov rax, rbx             ; New maximum number
        mov edi, buf              ; Initialize buffer pointer
skip:  loop up1             ; If number count is not zero repeat
        mov rbx, rax              ; save maximum
        mov edi, buf              ; Initialize buffer pointer
        mov ecx,16                ; load number of digits to display
dispup1:
        rol rbx,4                ; rotate number left by four bits
        mov dl,bl                 ; move lower byte in dl
        and dl,0fh                ; mask upper digit of byte in dl
        add dl,30h                ; add 30h to calculate ASCII code
        cmp dl,39h                ; compare with 39h
        jbe dispskip1             ; if less than 39h skip adding 07 more
        add dl,07h                ; else add 07
dispskip1:
        mov [edi],dl               ; store ASCII code in buffer
        inc edi                  ; point to next byte
```

```

loop dispup1 ; if byte count not zero repeat
print buf,16 ; display maximum number
ret          ; return to calling program

```

**Output**

The maximum number in the array is : 90FF123456781234

**Assignment 4 :** Write a switch case driven X86/64 ALP to perform 64-bit hexadecimal arithmetic operations (+,-,\*,/ ) using suitable macros. Define procedure for each operation.

```

section .data
menu db "=====,10
        db "      MENU      ",10
        db "=====,10
        db " 1. Perform Addition",10
        db " 2. Perform Subtraction",10
        db " 3. Perform multiplication",10
        db " 4. Perform Division",10
        db " 5. Exit",10
        db "Enter your choice : "
menu_len: equ    $-menu

msg      db     10,"Enter First 64-bit Hex number : "
msg_len:  equ    $-msg
msg1     db     10,"Enter Second 64- bit Hex number : "
msg1_len: equ    $-msg1
msg3     db     "The result is : "
msg3_len: equ    $-msg3

msg4     db     "Exit1"
msg4_len: equ    $-msg4
msg5     db     "The quotient is : "
msg5_len: equ    $-msg5
msg6     db     10, "The remainder is : "
msg6_len: equ    $-msg6
NL       db     10,13
NL_len:  equ    $-NL
array dq 8000000000001234h,00000000000000004h
dividend dq 0000000080000055h
divisor dd 10000000h
remainder dq 0

```

```
=====
=====section .bss
dispbuff resb 2
buf resb 16
len resb 1
%macro print 2
    mov rax,4 ; Specifies system call number (sys_write)
    mov rbx,1 ; Specifies (stdout)
    mov rcx,%1 ; Address of message to write
    mov rdx,%2 ; Message length
    int 0x80 ; call kernel
%endmacro
%macro accept 2
    mov rax,3 ; System call for Read
    mov rbx,0 ; from standard input
    mov rcx,%1 ; Start address
    mov rdx,%2 ; size
    int 80h ; Invoke operating system to Read
%endmacro

%macro addm 2
    mov rax,%1 ; load number 1
    mov rbx,%2 ; load number 2
    add rax,rbx ; perform addition
%endmacro
%macro subm 2
    mov rax,%1 ; load number 1
    mov rbx,%2 ; load number 2
    sub rax,rbx ; perform subtraction
%endmacro

%macro mulm 2
    mov rax,%1 ; load number 1
    mov rbx,%2 ; load number 2
    mul rbx ; perform multiplication
%endmacro

%macro divm 2
    mov rdx, 0
    mov rax,%1 ; load dividend
    mov ebx,%2 ; load divisor
    div ebx ; perform division
%endmacro
```

```
%macro exit 0
    mov rax, 1      ; System call for exit
    int 80h         ; Invoke operating system to exit
%endmacro exit
=====
section .text
    global _start
_start:
    menu:
        print menu, menu_len
        mov esi, array
        accept buf, 1
        mov al, [buf]
;*****
case1:   cmp byte[buf],'1'       ; Comparison of choice
        jne case2
        call Add
        exit

case2:   cmp byte[buf],'2'       ; Comparison of choice
        jne case3
        call Sub
        exit

case3:   cmp byte[buf],'3'       ; Comparison of choice
        jne case4
        call Multiply
        exit

case4:   cmp byte[buf],'4'       ; Comparison of choice
        jne case5
        call Divide
        exit

case5:   cmp byte[buf],'5'       ; Comparison of choice
        jne menu
        exit

Add:    print msg3, msg3_len
        addm [esi], [esi +8]
        mov rbx, rax
```

```
call dispup1
ret
Sub:  print msg3, msg3_len
      subm [esi], [esi +8]
      mov rbx, rax
      call dispup1
      ret
Multiply: print msg3, msg3_len
          mulm [esi], [esi +8]
          mov rbx, rdx
          call dispup1
          mov rbx, rax
          call dispup1
          ret
Divide:
        print msg5, msg5_len
        divm [dividend], [divisor]
        mov rbx, rax
        mov [remainder], rdx
        call dispup1
        print msg6, msg6_len
        mov rbx, [remainder]
        call dispup1
        ret
dispup1:
        mov rcx , 16
        mov edi, buf
        st:   rol rbx,4           ; rotate number left by four bits
              mov dl,bl           ; move lower byte in dl
              and dl,0fh          ; mask upper digit of byte in dl
              add dl,30h          ; add 30h to calculate ASCII code
              cmp dl,39h          ; compare with 39h
              jbe dispskip1        ; if less than 39h skip adding 07 more
              add dl,07h          ; else add 07
dispskip1:
        mov [edi],dl           ; store ASCII code in buffer
        inc edi               ; point to next byte
        loop st               ; if byte count not zero repeat
        print buf,16           ; display maximum number
        ret                   ; return to calling program
```

**Output**

```
=====
 MENU
=====
```

- 1. Perform Addition
- 2. Perform Subtraction
- 3. Perform multiplication
- 4. Perform Division
- 5. Exit

Enter your choice :1

The result is : 8000000000001238

```
=====
 MENU
=====
```

- 1. Perform Addition
- 2. Perform Subtraction
- 3. Perform multiplication
- 4. Perform Division
- 5. Exit

Enter your choice :2

The result is : 8000000000001230

```
=====
 MENU
=====
```

- 1. Perform Addition
- 2. Perform Subtraction
- 3. Perform multiplication
- 4. Perform Division
- 5. Exit

Enter your choice :3

The result is : 000000000000000200000000000000010

```
=====
 MENU
=====
```

- 1. Perform Addition
- 2. Perform Subtraction
- 3. Perform multiplication
- 4. Perform Division
- 5. Exit

```
Enter your choice : 4
The quotient is : 0000000000000008
The remainder is : 0000000000000055
```

**Assignment 5 : Write X86/64 ALP to count number of positive and negative numbers from the array.**

; ALP to count number of positive and negative numbers from the array

```
;
```

```
-----
```

```
section .data
welmsg db 10,'Count +ve and -ve numbers in an array',10
welmsg_len equ $-welmsg
pmmsg db 10,'Count of +ve numbers::'
pmmsg_len equ $-pmmsg
nmsg db 10,'Count of -ve numbers::'
nmsg_len equ $-nmsg
nwline db 10
array dw 8505h,90ffh,8700h,8800h,8a9fh,0a0dh,0200h
arrcnt equ 7
pcnt db 0
ncnt db 0
```

```
;
```

```
-----
```

```
section .bss
disbuff resb 2
%macro print 2
    mov eax,4      ; Specifies system call number (sys_write)
    mov ebx,1      ; Specifies (stdout)
    mov ecx,%1    ; Address of message to write
    mov edx,%2    ; Message length
    int 0x80       ; call kernel
%endmacro
```

```
;
```

```
-----
```

```
section .text
    global _start
_start:
    print welmsg,welmsg_len    ; display message
    mov esi,array               ; initialize array pointer
```

```

        mov ecx,arrcnt          ; initialize counter
up1:   bt word[esi],31         ; Check MSB of the number
        jnc pnxt               ; if MSB = 0 skip next two
                                ; instructions
        inc byte[ncnt]          ; increment negative number counter
        jmp pskip
pnxt:  inc byte[pcnt]          ; increment positive number counter
pskip: inc esi                ; increment array pointer by 2
      inc esi
      loop up1               ; if ecx is not zero check next number
      print pmsg,pmsg_len     ; display message
      mov bl,[pcnt]             ; load positive number count
      call disp8num            ; display positive number count

      print nmsg,nmsg_len
      mov bl,[ncnt]
      call disp8num
      print nwline,1           ; New line char

exit:   mov eax,1               ; Specifiy sys_exit call number
        mov ebx,0               ; Return a code of zero
        int 0x80                ; Make the syscall to terminate the program

disp8num:
        mov edi,dispbuff         ; point edi to buffer
        mov ecx,2               ; load number of digits to display

dispup1:
        rol bl,4                ; rotate number left by four bits
        mov dl,bl                ; move lower byte in dl
        and dl,0fh               ; mask upper digit of byte in dl
        add dl,30h               ; add 30h to calculate ASCII code
        cmp dl,39h               ; compare with 39h
        jbe dispskip1            ; if less than 39h akip adding 07 more
        add dl,07h               ; else add 07

dispskip1:
        mov [edi],dl              ; store ASCII code in buffer
        inc edi
        loop dispup1             ; point to next byte
                                ; decrement the count of digits to
                                ; display
                                ; if not zero jump to repeat
        print dispbuff,2          ; display maximum number
        ret                      ; return to calling program

```

### Assemble and Link

```

neha@neha-Inspiron-3537:~/ATUL/final$ nasm -f elf GB8.asm
neha@neha-Inspiron-3537:~/ATUL/final$ ld -m elf_i386 -s -o GB8 GB8.o
neha@neha-Inspiron-3537:~/ATUL/final$ ./GB8

```

**Output**

Count +ve and -ve numbers in an array  
 Count of +ve numbers::02  
 Count of -ve numbers::05

**Assignment 6 :** Write X86/64 ALP to convert 4-digit Hex number into its equivalent BCD number and 5-digit BCD number into its equivalent HEX number. Make your program user friendly to accept the choice from user for : (a) HEX to BCD b) BCD to HEX (c) EXIT. Display proper strings to prompt the user while accepting the input and displaying the result. (wherever necessary, use 64-bit registers).

section .data

```

menu    db "=====,10
db "      MENU      ",10
db "=====,10
db " 1]. HEX to BCD conversion",10
db " 2]. BCD to HEX conversion",10
db " 3]. Exit",10
db "Enter your choice"
menu_len:equ   $-menu

hbmsg     db  10,"Hex to BCD "
          db  10,"Enter Maximum 4-digit Hex number: "
hbmsg_len:  equ  $-hbmsg
bhmsg     db  10,  "BCD to Hex "
          db  10,"Enter maximum 5-digit BCD number: "
bhmsg_len:  equ  $-bhmsg
hmsg      db  "Equivalent Hex number is: "
hmsg_len:  equ  $-hmsg

bmsg      db  "Equivalent BCD number is: "
bmsg_len:  equ  $-bmsg
msg       db  10,13
msg_len:  equ  $-msg
=====
```

section .bss

```

buf      resb   6
buf_len: equ    $-buf
digitcount  resb   1
ans       resw   1
char_ans  resb   4
%macro dispmsg 2
```

```
mov rax,4      ; System call for write
mov rbx,1      ; Specifies standard Output
mov rcx,%1    ; Address of message to write
mov rdx,%2    ; Message length
int 80h        ; Invoke operating system to write
%endmacro
%macro accept 2
    mov rax,3    ; System call for Read
    mov rbx,0    ; from standard input
    mov rcx,%1    ; Start address
    mov rdx,%2    ; size
    int 80h        ; Invoke operating system to Read
%endmacro
%macro exit 0
    mov rax, 1   ; System call for exit
    int 80h        ; Invoke operating system to exit
%endmacro
;=====
section .text
    global _start
_start:
menu:
    dispmsg menu, menu_len
    accept buf,buf_len
    mov al,[buf]
;*****
case1: cmp     byte[buf],'1'      ;Comparison of choice
        jne     case2
        call    hex_bcd
        dispmsg msg, msg_len
        exit
case2: cmp     byte[buf],'2'      ;Comparison of choice
        jne     case3
        call    bcd_hex
        dispmsg msg, msg_len
        exit
case3: cmp     byte[buf],'3'      ;comparison of choice
        je ext
        dispmsg msg, msg_len
        jmp menu
ext:   exit
;*****
hex_bcd:
    dispmsg hbmsg, hbmsg_len
    call Hex_Num
```

```

    mov rax,rbx          ; Save hex number
    mov rbx,10            ; Load divisor
back:
    xor rdx,rdx          ; Clear remainder
    div rbx              ; number = number/divisor
    push rdx             ; Save remainder

    inc byte[digitcount] ; Increment counter
    cmp rax,0h            ; Check whether quotient = 0
    jne back              ; If not zero repeat
    dispmsg   bmsg, bmsg_len

print_bcd:
    pop rdx              ; Restore remainder
    add dl,30h            ; Convert to ASCII
    mov [char_ans].dl      ; Save ASCII value
    dispmsg   char_ans,1  ; Display ASCII number
    dec byte[digitcount] ; Decrement counter
    jnz print_bcd         ; If counter not zero repeat
    ret                  ; Return

Hex_Num:
    accept buf, buf_len
    mov rbx,00            ; Clear result
    dec rax              ; Adjust the count of number of bytes read
    mov rcx,rax           ; counter = number of bytes read
    mov esi,buf           ; Initialize pointer to buffer
    mov rax,00            ; Clear rax

L3:
    shl rbx,4             ; Read byte
    mov al,[esi]
    cmp al,39h            ; Compare with 39H
    jbe skip2              ; If it is less or equal skip next instruction
    sub al,07h             ; else subtract 7

skip2:
    sub al,30h            ; subtract 30 to get hex equivalent
    add rbx,rax           ; result = result + hex number
    inc esi               ; increment pointer
    loop L3                ; Decrement counter and if not zero repeat
    ret                  ; Return
*****
bcd_hex:
    dispmsg   bhmsg, bhmsg_len
    accept buf,buf_len
    mov rsi,buf           ; Initialize pointer to number
    dec rax              ; Adjust the count of number of bytes read
    mov rcx, rax           ; counter = number of bytes read
    xor rax,rax            ; Result = 0
    mov rbx,10             ; Multiplier 10 decimal

```

```

back1: xor rdx,rdx
       mul rbx
       xor rdx,rdx
       mov dl,[rsi]      ; Get the ASCII digit
       sub dl,30h        ; convert to BCD
       add rax,rdx        ; Result = Result + Digit
       inc rsi           ; Increment pointer
       dec rcx           ; Decrement counter
       jnz back1         ; If not zero repeat
       mov [ans],ax        ; save result
       dispmsg hmsg, hmsg_len
       mov ax,[ans]        ; Get result
       call display_16
       ret
display_16:
       mov rsi,char_ans+3 ; load last byte address of char_ans in rsi
       mov rcx,4          ; number of digits
cnt:
       mov rdx,0          ; make rdx=0 (as in div instruction rdx:rax/rbx)
       mov rbx,16         ; divisor=16 for hex
       div rbx            ; rax/rbx
       cmp dl,09h         ; check for remainder in RDX
       jbe add30
       add dl,07h
add30:
       add dl,30h         ; Calculate ASCII code
       mov [rsi],dl        ; Store it in buffer
       dec rsi            ; Point to one byte back
       dec rcx            ; Decrement count
       jnz cnt            ; If not zero repeat
       dispmsg char_ans,4 ; Display result on screen
       ret

```

**Assemble, Link and Execute Commands**

```

neha@neha-VirtualBox:~/Documents/M$ nasm -f elf64 hex_bcd.asm
neha@neha-VirtualBox:~/Documents/M$ ld -o hex_bcd hex_bcd.o
neha@neha-VirtualBox:~/Documents/M$ ./hex_bcd

```

**Output1**

```

=====
MENU
=====
1]. HEX to BCD conversion
2]. BCD to HEX conversion
3]. Exit
Enter your choice1

```

Hex to BCD

Enter Maximum 4-digit Hex number: FFFF

Equivalent BCD number is: 65535

**Output2**

=====

MENU

=====

1]. HEX to BCD conversion

2]. BCD to HEX conversion

3]. Exit

Enter your choice2

BCD to Hex

Enter maximum 5-digit BCD number: 65535

Equivalent Hex number is: FFFF

**Assignment 7 : Write X86/64 ALP to switch from real mode to protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers.**

; ALP to use GDTR, LDTR, and IDTR

-----

```
section .data
rmodemsg db 10,'Processor is in Real Mode'
rmsg_len:equ $-rmodemsg
pmodemsg db 10,'Processor is in Protected Mode'
pmsg_len:equ $-pmodemsg
gdtmsg db 10,'GDT Contents are:'
gmsg_len:equ $-gdtmsg
ldtmsg db 10,'LDT Contents are:'
lmsg_len:equ $-ldtmsg
idtmsg db 10,'IDT Contents are:'
imsg_len:equ $-idtmsg
trmsg db 10,'Task Register Contents are:'
tmsg_len: equ $-trmsg
mswmsg db 10,'Machine Status Word:'
mmsg_len:equ $-mswmsg
colmsg db ?
nwline db 10
```

-----

section .bss

```

gdt resd 1      ; Reserve double word
    resw 1      ; Reserve word
ldt resw 1      ; Reserve word
idt resd 1      ; Reserve double word
    resw 1      ; Reserve word
tr resw 1      ; Reserve word
cr0_data resd 1
dnum_buff resb 04

%macro disp 2
    mov eax,4      ; Specifies system call number (sys_write)
    mov ebx,1      ; Specifies (stdout)
    mov ecx,%1      ; Address of message to write
    mov edx,%2      ; Message length
    int 0x80      ; call kernel
%endmacro

;-----;

section .text
global _start
_start:
    smsw eax      ; Reading CR0
    mov [cr0_data],eax
    bt eax,0      ; Checking PE bit(LSB), if 1=Protected Mode, else
                  ; Real Mode
    jc prmode
    disp rmodemsg,rmsg_len ; dispaly message when PE = 0
    jmp nxt1

prmode: disp pmodemsg,pmsg_len; dispaly message when PE = 1
nxt1: sgdt [gdt] ; Store the contents of the descriptor table register
            ; the six bytes of memory indicated by the operand
    sldt [ldt]      ; Store the segment selector from the local descriptor
            ; table register (LDTR) in the destination operand
    sidt [idt]      ; Stores the content the interrupt descriptor table
            ; register (IDTR) in the destination operand
    str [tr]        ; Stores the visible portion of the task
            ; registerin the destination operand
    disp gdtdmsg,gmsg_len ; display message

    mov bx,[gdt+4]      ; display the higher word contents
    call disp_num        ; of the descriptor table register
    mov bx,[gdt+2]      ; display the middle word contents
    call disp_num        ; of the descriptor table register
    disp colmsg,1        ; display colon
    mov bx,[gdt]        ; display the lower word contents
    call disp_num        ; of the descriptor table register
    disp ldtmsg,lmsg_len ; display message

```

```

mov bx,[ldt]           ; display the contents of segment selector
call disp_num          ; from the local descriptor table register
                        ; (LDTR)
disp idtmsg,imsg_len  ; display message

mov bx,[idt+4]         ; display the higher word contents
call disp_num          ; of the interrupt descriptor table register
mov bx,[idt+2]         ; display the middle word contents
call disp_num          ; of the interrupt descriptor table register
disp colmsg,1          ; display colon
mov bx,[idt]           ; display the lower word contents
call disp_num          ; of the interrupt descriptor table register
disp trmsg,tmsg_len   ; display message

mov bx,[tr]             ; display the visible portion of the task register
call disp_num          ; display message

mov bx,[cr0_data+2];   ; display the higher word contents
call disp_num          ; of the machine status word (CR0)
mov bx,[cr0_data]       ; display the lower word contents
call disp_num          ; of the machine status word (CR0)
disp nnewline,1         ; Print on the next line
exit:    mov eax,1      ; Specifiy sys_exit call number
        mov ebx,0      ; Return a code of zero
        int 80h        ; Make the syscall to terminate the program
disp_num:
        mov esi,dnum_buff ; point esi to buffer
        mov ecx,04        ; load number of digits to display

up1:
        rol bx,4         ; rotate number left by four bits
        mov dl,bl          ; move lower byte in dl
        and dl,0fh         ; mask upper digit of byte in dl
        add dl,30h         ; add 30h to calculate ASCII code
        cmp dl,39h         ; compare with 39h
        jbe skip1          ; if less than 39h skip adding 07 more
        add dl,07h         ; else add 07

skip1:
        mov [esi],dl        ; store ASCII code in buffer
        inc esi            ; point to next byte
loop up1               ; decrement the count of digits to display
                        ; if not zero jump to repeat
disp dnum_buff,4       ; display the number from buffer

ret

```

**Assemble and Link**

```
neha@neha-Inspiron-3537:~/ATUL/final$ nasm -f elf64 A7.asm
neha@neha-Inspiron-3537:~/ATUL/final$ ld -o A7 A7.o
neha@neha-Inspiron-3537:~/ATUL/final$ ./A7
```

**Output**

```
Processor is in Protected Mode
GDT Contents are::DF244000:007F
LDT Contents are::0000
IDT Contents are::81DD6000:0FFF
Task Register Contents are::0040
Machine Status Word::8005FFFF
```

**Assignment 8 :** Write X86-64 ALP to perform non-overlapped block transfer without string specific instructions. Block containing data can be defined in the data segment.

```
; Objective :-Block Transfer operation without using String operations(32 bit pointers)
%macro print 2
    mov rax, 1
    mov rdi, 1
    mov rsi, %1
    mov rdx, %2
    syscall
%endmacro
%macro read 2
    mov rax, 0
    mov rdi, 0
    mov rsi, %1
    mov rdx, %2
    syscall
%endmacro
section .data
msg1 db "Source Block is:",10
len1 equ $-msg1
msg2 db "Destination Block is:",10
len2 equ $-msg2
colon db "."
msg3 db "Menu:",10,"1.Non-overlapped Block Transfer",10,"2.Overlapped Block Transfer",10,"3.Exit",10,"Enter choice:",10
len3 equ $-msg3
msg4 db "Invalid choice",10
len4 equ $-msg4
msg5 db "Enter offset value for destination block to be started from(Block starts from 1)",10
len5 equ $-msg5
msg6 db "Before Transfer:",10
```

```
len6 equ $-msg6
msg7 db "After Transfer.",10
len7 equ $-msg7
newline db 10
num db 10h,20h,30h,40h,50h,00h,00h,00h,00h,00h

section .bss
array resb 5
var resb 1
avar resb 2
count resb 1
rch resb 4
rca resb 8
cnt resb 1
choice resb 1
offset resb 1
global _start
section .text
_start:
_start:
    print msg3,len3
    read choice,2
    cmp byte[choice],33h
    je exit
    cmp byte[choice],32h
    je overlapped
    cmp byte[choice],31h
    je non_overlapped
    print msg4,len4
    jmp start
non_overlapped:
    mov esi,num
    mov edi,array
    mov cx,05h
back1:mov al,[esi]
    mov [edi],al
    inc esi
    inc edi
    dec cx
    jnz back1
    print msg1,len1      ;To print source block
    mov esi,num
    mov byte[cnt],05
back2:
    mov al,byte[esi]
    mov byte[var],al
    mov dword[rch],esi
```

```
call ascii
print rca,8
print colon,1
call asciiconvert2
print avar,2
print newline,1
mov esi,dword[rch]
inc esi
dec byte[cnt]
jnz back2
print msg2,len2           ;To print destination block
mov edi,array
mov byte[cnt],05
back3:
    mov al,byte[edi]
    mov byte[var],al
    mov dword[rch],edi
    call ascii
    print rca,8
    print colon,1
    call asciiconvert2
    print avar,2
    print newline,1
    mov edi,dword[rch]
    inc edi
    dec byte[cnt]
    jnz back3
    jmp exit
overlapped:
    print msg5,len5
    read offset,2

    print msg6,len6
    mov esi,num

    mov byte[cnt],05
back4:
    mov al,byte[esi]
    mov byte[var],al
    mov dword[rch],esi
    call ascii
    print rca,8
    print colon,1
    call asciiconvert2
    print avar,2
    print newline,1
    mov esi,dword[rch]
```

```
inc esi
dec byte[cnt]
jnz back4

mov esi,num
mov edi,num
cmp byte[offset],34h
je l4
cmp byte[offset],33h
je l3
cmp byte[offset],32h
je l2
cmp byte[offset],31h
je l1
print msg4,len4
jmp overlapped
l4: add edi,8
add esi,4
jmp transfer
l3: add edi,7
add esi,4
jmp transfer
l2: add edi,6
add esi,4
jmp transfer
l1: add edi,5
add esi,4

transfer:
    mov cx,05h
back5:mov al,[esi]
        mov [edi],al
        dec esi
        dec edi
        dec cx
        jnz back5
        print msg7,len7
        mov edi,num
        cmp byte[offset],34h
        je l_4
        cmp byte[offset],33h
        je l_3
        cmp byte[offset],32h
        je l_2
        cmp byte[offset],31h
        je l_1
l_1:
```

```
    add edi,1
    je l_1
    jmp printblock
l_2:
    add edi,2
    jmp printblock
l_3:
    add edi,3
    jmp printblock
l_4:
    add edi,4
    jmp printblock
printblock:
    mov byte[cnt],05
back6:
    mov al,byte[edi]
    mov byte[var],al
    mov dword[rch],edi
    call ascii
    print rca,8
    print colon,1
    call asciiconvert2
    print avar,2
    print newline,1
    mov edi,dword[rch]
    inc edi
    dec byte[cnt]
    jnz back6

exit:
    mov rax,60          ;Exit system call
    mov rdi,00
    syscall

asciiconvert2:           ;Procedure to covert into ascii(8 bits)
    mov r8,avar
    mov byte[count], 2
    mov al, [var]
back:rol al, 04h
    mov bl,al
    AND bl, 0Fh
    cmp bl, 09h
    jbe next
    add bl, 7h
next:
```

```
add bl, 30h
mov [r8], bl
inc r8
dec byte[count]
jnz back
ret
ascii:           ;Procedure to convert Hex to Ascii(32 bits)
    mov r9,rca
    mov byte[count],8
    mov eax,dword[rch]
bck2: mov ebx,eax
    rol ebx,04
    mov eax,ebx
    and bl,0Fh
    cmp bl,9h
    jbe next_1
    add bl,7h
next_1:
    add bl,30h
    mov [r9],bl
    inc r9
    dec byte[count]
    jnz bck2
    ret
```

#### Output

```
neha@neha-Inspiron-3537:~/Neha_prog$ nasm -f elf64 A8_1.asm
neha@neha-Inspiron-3537:~/Neha_prog$ ld -o h A8_1.o
neha@neha-Inspiron-3537:~/Neha_prog$ ./h
```

Menu:

- 1.Non-overlapped Block Transfer
- 2.Overlapped Block Transfer
- 3.Exit

Enter choice:

1

Source Block is:

006006DD:10

006006DE:20

006006DF:30

006006E0:40

006006E1:50

Destination Block is:

006006E8:10

006006E9:20

006006EA:30

```
006006EB:40
006006EC:50
neha@neha-Inspiron-3537:~/Neha_prog$ ./h
Menu:
1.Non-overlapped Block Transfer
2.Overlapped Block Transfer
3.Exit
Enter choice:
2
Enter offset value for destination block to be started from(Block starts from 1)
2
Before Transfer:
006006DD:10
006006DE:20
006006DF:30
006006E0:40
006006E1:50
After Transfer:
006006DF:10
006006E0:20
006006E1:30
006006E2:40
006006E3:50
neha@neha-Inspiron-3537:~/Neha_prog$ ./h
Menu:
1.Non-overlapped Block Transfer
2.Overlapped Block Transfer
3.Exit
Enter choice:
2
Enter offset value for destination block to be started from(Block starts from 1)
1
Before Transfer:
006006DD:10
006006DE:20
006006DF:30
006006E0:40
006006E1:50
After Transfer:
006006DE:10
```

```
006006DF:20
006006E0:30
006006E1:40
006006E2:50
neha@neha-Inspiron-3537:~/Neha_prog$ ./h
Menu:
1.Non-overlapped Block Transfer
2.Overlapped Block Transfer
3.Exit
Enter choice:
3
neha@neha-Inspiron-3537:~/Neha_prog$
```

**Assignment 9 :** Write X86/64 ALP to perform overlapped block transfer with string specific instructions block containing data can be defined in the data segment.

```
; Objective :-Block Transfer operation using String operations(32 bit pointers)
%macro print 2 ; Macro for displaying string
    mov rax, 1
    mov rdi, 1
    mov rsi, %1
    mov rdx, %2
    syscall
%endmacro
%macro read 2 ; Macro for reading string
    mov rax, 0
    mov rdi, 0
    mov rsi, %1
    mov rdx, %2
    syscall
%endmacro
section .data
msg1 db "Source Block is:",10
len1 equ $-msg1
msg2 db "Destination Block is:",10
len2 equ $-msg2
colon db "."
msg3 db "Menu:",10,"1.Non-overlapped Block Transfer",10,"2.Overlapped Block
Transfer",10,"3.Exit",10,"Enter choice:",10
len3 equ $-msg3
msg4 db "Invalid choice",10
len4 equ $-msg4
msg5 db "Enter offset value for destination block to be started from(Block starts from
1)",10
len5 equ $-msg5
```

```
msg6 db "Before Transfer.",10
len6 equ $-msg6
msg7 db "After Transfer.",10
len7 equ $-msg7
newline db 10
num db 10h,20h,30h,40h,50h,00h,00h,00h,00h,00h

section .bss
array resb 5
var resb 1
avar resb 2
count resb 1
rch resb 4
rca resb 8
cnt resb 1
choice resb 1
offset resb 1
global _start
section .text
_start:
start:
    print msg3,len3
    read choice,2
    cmp byte[choice],33h
    je exit
    cmp byte[choice],32h
    je overlapped
    cmp byte[choice],31h
    je non_overlapped
    print msg4,len4
    jmp start
non_overlapped:
    mov esi,num
    mov edi,array
    mov cx,05h
    CLD
back1:movsb
    dec cx
    jnz back1
    print msg1,len1           ;To print source block
    mov esi,num
    mov byte[cnt],05
back2:
    mov al,byte[esi]
    mov byte[var],al
    mov dword[rch],esi
    call ascii
```

```
print rca,8
print colon,1
call asciiconvert2
print avar,2
print newline,1
mov esi,dword[rch]
inc esi
dec byte[cnt]
jnz back2
print msg2,len2 ;To print destination block
mov edi,array
mov byte[cnt],05
back3:
    mov al,byte[edi]
    mov byte[var],al
    mov dword[rch],edi
    call ascii
    print rca,8
    print colon,1
    call asciiconvert2
    print avar,2
    print newline,1
    mov edi,dword[rch]
    inc edi
    dec byte[cnt]
    jnz back3
    jmp exit
overlapped:
    print msg5,len5
    read offset,2

    print msg6,len6
    mov esi,num

    mov byte[cnt],05
back4:
    mov al,byte[esi]
    mov byte[var],al
    mov dword[rch],esi
    call ascii
    print rca,8
    print colon,1
    call asciiconvert2
    print avar,2
    print newline,1
    mov esi,dword[rch]
    inc esi
```

```
dec byte[cnt]
jnz back4

mov esi,num
mov edi,num
cmp byte[offset],34h
je l4
cmp byte[offset],33h
je l3
cmp byte[offset],32h
je l2
cmp byte[offset],31h
je l1
print msg4,len4
jmp overlapped

l4: add edi,8
add esi,4
jmp transfer
l3: add edi,7
add esi,4
jmp transfer
l2: add edi,6
add esi,4
jmp transfer
l1: add edi,5
add esi,4

transfer:
    mov cx,05h
std
back5: movsb
    dec cx
    jnz back5
    print msg7,len7
    mov edi,num
    cmp byte[offset],34h
    je l_4
    cmp byte[offset],33h
    je l_3
    cmp byte[offset],32h
    je l_2
    cmp byte[offset],31h
    je l_1
l_1:
    add edi,1
    je l_1
    jmp printblock
l_2:
    add edi,2
```

```
        jmp printblock
l_3:
    add edi,3
    jmp printblock
l_4:
    add edi,4
    jmp printblock
printblock:
    mov byte[cnt],05
back6:
    mov al,byte[edi]
    mov byte[var],al
    mov dword[rch],edi
    call ascii
    print rca,8
    print colon,1
    call asciiconvert2
    print avar,2
    print newline,1
    mov edi,dword[rch]
    inc edi
    dec byte[cnt]
    jnz back6

exit:
    mov rax,60          ;Exit system call
    mov rdi,00
    syscall

asciiconvert2:           ;Procedure to covert into ascii(8 bits)
    mov r8,avar
    mov byte[count], 2
    mov al, [var]
back:rol al, 04h
    mov bl,al
    AND bl, 0Fh
    cmp bl, 09h
    jbe next
    add bl, 7h
next:
    add bl, 30h
    mov [r8], bl
    inc r8
    dec byte[count]
    jnz back
```

```
        ret
ascii:           ;Procedure to convert Hex to Ascii(32 bits)
    mov r9,rca
    mov byte[count],8
    mov eax,dword[rch]
bck2:mov ebx,eax
    rol ebx,04
    mov eax,ebx
    and bl,0Fh
    cmp bl,9h
    jbe next_1
    add bl,7h
next_1:
    add bl,30h
    mov [r9],bl
    inc r9
    dec byte[count]
    jnz bck2
    ret
```

**Output**

neha@neha-Inspiron-3537:~/Neha\_prog\$ nasm -f elf64 A9.asm

neha@neha-Inspiron-3537:~/Neha\_prog\$ ld -o h A9.o

neha@neha-Inspiron-3537:~/Neha\_prog\$ ./h

Menu:

1.Non-overlapped Block Transfer

2.Overlapped Block Transfer

3.Exit

Enter choice:

1

Source Block is:

006006CD:10

006006CE:20

006006CF:30

006006D0:40

006006D1:50

Destination Block is:

006006D8:10

006006D9:20

006006DA:30

006006DB:40

006006DC:50

neha@neha-Inspiron-3537:~/Neha\_prog\$ ./h

Menu:

```
1.Non-overlapped Block Transfer
```

```
2.Overlapped Block Transfer
```

```
3.Exit
```

```
Enter choice:
```

```
2
```

```
Enter offset value for destination block to be started from(Block starts from 1)
```

```
3
```

```
Before Transfer:
```

```
006006CD:10
```

```
006006CE:20
```

```
006006CF:30
```

```
006006D0:40
```

```
006006D1:50
```

```
After Transfer:
```

```
006006D0:10
```

```
006006D1:20
```

```
006006D2:30
```

```
006006D3:40
```

```
006006D4:50
```

```
neha@neha-Inspiron-3537:~/Neha_prog$ ./h
```

```
Menu:
```

```
1.Non-overlapped Block Transfer
```

```
2.Overlapped Block Transfer
```

```
3.Exit
```

```
Enter choice:
```

```
2
```

```
Enter offset value for destination block to be started from(Block starts from 1)
```

```
4
```

```
Before Transfer:
```

```
006006CD:10
```

```
006006CE:20
```

```
006006CF:30
```

```
006006D0:40
```

```
006006D1:50
```

```
After Transfer:
```

```
006006D1:10
```

```
006006D2:20
```

```
006006D3:30
```

```
006006D4:40
```

```
006006D5:50
```

```
neha@neha-Inspiron-3537:~/Neha_prog$ ./h
```

Menu:

1.Non-overlapped Block Transfer

2.Overlapped Block Transfer

3.Exit

Enter choice:

3

neha@neha-Inspiron-3537:~/Neha\_prog\$

**Assignment 10 : Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use successive addition and add and shift method. (use of 64-bit registers is expected).**

```
; Objective :- To multiply two 8-bit hexadecimal numbers
%macro print 2                                ;Macro for displaying string
    mov rax, 1
    mov rdi, 1
    mov rsi, %1
    mov rdx, %2
    syscall
%endmacro
%macro read 2                                    ;Macro for reading string
    mov rax, 0
    mov rdi, 0
    mov rsi, %1
    mov rdx, %2
    syscall
%endmacro
section .data
msg1 db "Enter value for multiplicand(2 digits):",10
len1 equ $.-msg1
msg2 db "Enter value for multiplier(2 digits):",10
len2 equ $.-msg2
msg3 db "1.Successive Addition",10,"2.Add and Shift",10,"3.Exit",10,"Enter choice:",10
len3 equ $.-msg3
msg4 db "Invalid choice!",10
len4 equ $.-msg4
result dw 0000000000000000h
newline db 10
msg5 db "Result:",10
len5 equ $.-msg5
section .bss
multiplicanda resb 3
multiplicandh resb 1
multipliera resb 3
multiplierh resb 1
choice resb 2
```

```
avar resb 4
var resb 2
count resb 1
resulta resb 4
mulcopy resb 1
global _start
section .text
_start:
    print msg1,len1
    read multiplicanda,3
    print msg2,len2
    read multipliera,3
    mov ax,word[multiplicanda]           ;Convert multiplicand
                                            ; to hex
    mov word[avar],ax
    call packnum1
    mov al,byte[var]
    mov byte[multiplicandh],al

    mov ax,word[multipliera]           ;Convert multiplier to
                                            ; hex
    mov word[avar],ax
    call packnum1
    mov al,byte[var]
    mov byte[multiplierh],al

start:
    print msg3,len3
    read choice,2
    cmp byte[choice],31h
    je sadd
    cmp byte[choice],32h
    je addnshift
    cmp byte[choice],33h
    je exit
    print msg4,len4
    jmp start

sadd:
    mov al,byte[multiplierh]          ;Perform multiplication by
                                            ; successive addition
    mov byte[count],al

bac:
    mov edx,0h
    mov dl,byte[multiplicandh]
    add word[resulth],dx
    dec byte[count]
    jnz bac
    jmp printresult
```

```
addnshift:           ;Perform multiplication by add
                    ; and shift
    mov al,byte[multiplierh]
    mov byte[mulcopy],al

    mov byte[count],8
back1:
    shl word[resulth],01      ;Perform shifting
    shl byte[mulcopy],01
    jnc s1
    xor ax,ax
    mov al,byte[multiplicandh]
    add word[resulth],ax      ;Perform addition
s1:
    dec byte[count]
    jnz back1
printresult:         ;Printing the final result
    print msg5,len5
    mov ax,word[resulth]
    mov word[var],ax
    call asciiconvert1
    mov eax,dword[avar]
    mov dword[resulta],eax
    print resulta,4
    print newline,1
exit:
    mov rax,60                ;Exit system call
    mov rdi,00
    syscall
packnum1:            ;Procedure to convert Ascii to
                    ; Hex
    mov rsi,avar
    mov ebx,0
    mov byte[count],2
    mov eax,0
bck11:
    mov al,[rsi]
    rol bl,4
    cmp al,39h
    jbe next11
    sub al,7h
next11:
    sub al,30h
    or bl,al
    inc rsi
    dec byte[count]
    jnz bck11
```

```
        mov byte[var],bl
        ret
asciiconvert1:           ;Procedure to covert into
                           ; ascii(16 bits)
        mov r8,avar
        mov byte[count], 4
        mov ax, [var]
back:rol ax, 04h
        mov bl,al
        AND bl, 0Fh
        cmp bl, 09h
        jbe next
        add bl, 7h
next:
        add bl, 30h
        mov [r8], bl
        inc r8
        dec byte[count]
        jnz back
        ret
```

**Output**

neha@neha-Inspiron-3543:~/Neha/MIT/assign6\$ nasm -f elf64 Assign10.asm

neha@neha-Inspiron-3543:~/Neha/MIT/assign6\$ ld -o h Assign10.o

neha@neha-Inspiron-3543:~/Neha/MIT/assign6\$ ./h

Enter value for multiplicand(2 digits):

12

Enter value for multiplier(2 digits):

12

1.Successive Addition

2.Add and Shift

3.Exit

Enter choice:

1

Result:

0144

neha@neha-Inspiron-3543:~/Neha/MIT/assign6\$ ./h

Enter value for multiplicand(2 digits):

FF

Enter value for multiplier(2 digits):

FF

1.Successive Addition

2.Add and Shift

3.Exit

Enter choice:

2

Result:

```
FE01
neha@neha-Inspiron-3543:~/Neha/MIT/assign6$ ./h
Enter value for multiplicand(2 digits):
03
Enter value for multiplier(2 digits):
02
1.Successive Addition
2.Add and Shift
3.Exit
Enter choice:
4
Invalid choice!
1.Successive Addition
2.Add and Shift
3.Exit
Enter choice:
3
neha@neha-Inspiron-3543:~/Neha/MIT/assign6$
```

**Assignment 11 :** Write X86 menu driven Assembly Language Program (ALP) to implement OS (DOS) commands TYPE, COPY and DELETE using file operations. User is supposed to provide command line arguments in all cases.

### Copy

;Aim: Write a Program to simulate DOS COPY commands

```
section .data
fnotmsg db 'FILE NOT FOUND...',10
fnmsg_len equ $-fnotmsg
msg1 db 'Copied Successfully!',10
len1 equ $-msg1

newline db 10
section .bss
fd_in resd 1
fd_out resd 1
fbuff resb 20
fb_len equ $-fbuff
act_len resd 1
srcfile resb 80
destfile resb 80

%macro print 2
    mov eax,4
```

```
mov ebx,1
mov ecx,%1
mov edx,%2
int 0x80
%endmacro
section .text
global _start
_start:

    pop ecx          ;pop no. of arguments
    pop ecx          ;pop exec command

    pop ecx          ;pop source filename
    mov edx,00

up:
    cmp byte[ecx+edx],0
    jz l1
    inc edx
    jmp up

l1:
    xor edi,edi
    mov esi,ecx
    ;save source filename

l2:
    mov al,byte[esi]
    mov byte[srcfile + edi],al
    inc edi
    inc esi
    dec edx
    jnz l2

    pop ecx          ;pop destination filename
    mov edx,00

up1:
    cmp byte[ecx+edx],0
    jz l11
    inc edx
    jmp up1

l11:
    xor edi,edi
    mov esi,ecx
```

```
;save destination filename
l21:
    mov al,byte[esi]
    mov byte[destfile+edi],al
    inc edi
    inc esi
    dec edx
    jnz l21

;open the file for reading

    mov eax,5
    mov ebx,srcfile
    mov ecx,0
    mov edx,0777
    int 80h
    mov [fd_in],eax
    bt eax,31
    jnc conti1
    print frootmsg,fnmsg_len
    jmp exit

conti1:

;open or create the file for writing
    mov eax, 8
    mov ebx, destfile
    mov ecx, 0777      ;read, write and execute by all
    int 0x80          ;call kernel
    mov [fd_out],eax  ;store the file descriptor

readfile:
    mov eax,3
    mov ebx,[fd_in]
    mov ecx,fbuff
    mov edx,fb_len
    int 80h
    mov [act_len],eax
    cmp eax,0
    je nxt1
                ;write to the file
    mov eax,4          ;system call number (sys_write)
```

```

    mov ebx, [fd_out]      ;file descriptor
    mov ecx, fbuff         ;message to write
    mov edx,[act_len]      ;number of bytes
    int 0x80                ;call kernel

    jmp readfile

nxt1:   mov eax,6          ;close source file
        mov ebx,[fd_in]
        int 80h
        mov eax,6          ;close destination file
        mov ebx,[fd_out]
        int 80h
        print msg1,len1

exit:           ;exit system call
    mov eax,1
    mov ebx,0
    int 0x80

```

**Output**

```

neha@neha-Inspiron-3537:~/Neha_prog$ nasm -f elf64 copy.asm
neha@neha-Inspiron-3537:~/Neha_prog$ ld -m elf_i386 -s -o h copy.o
neha@neha-Inspiron-3537:~/Neha_prog$ ./h myfile2.txt myfile1.txt
Copied Successfully!
neha@neha-Inspiron-3537:~/Neha_prog$

```

**Delete**

;Aim: Write a Program to simulate DOS DELETE commands

```

section .data
fnotmsg db 'FILE NOT FOUND...',10
fnmsg_len equ $-fnotmsg
msg1 db 'Deleted Successfully!',10
len1 equ $-msg1
newline db 10
section .bss
fd_in resd 1
fbuff resb 20
fb_len equ $-fbuff

act_len resd 1
counter resb 1

```

```
srcfile resb 80
%macro print 2
    mov eax,4
    mov ebx,1
    mov ecx,%1
    mov edx,%2
    int 0x80
%endmacro
section .text
global _start
_start:

    pop ecx          ;pop no. of arguments
    pop ecx          ;pop exec command

    pop ecx          ;pop source filename
    mov edx,00

up:
    cmp byte[ecx+edx],0
    jz l1
    inc edx
    jmp up

l1:
    xor edi,edi
    mov esi,ecx
    ;save source filename

l2:
    mov al,byte[esi]
    mov byte[srcfile+edi],al
    inc edi
    inc esi
    dec edx
    jnz l2
    ;delete file
    mov eax, 10      ; system call 10: unlink
    mov ebx, srcfile ; file name to unlink
    int 80h          ; call into the system

    print msg1,len1
exit:                 ;exit system call
    mov eax,1
```

```
mov ebx,0  
int 0x80
```

**Output**

```
neha@neha-Inspiron-3537:~/Neha_prog$ nasm -f elf64 delete.asm  
neha@neha-Inspiron-3537:~/Neha_prog$ ld -m elf_i386 -s -o h delete.o  
neha@neha-Inspiron-3537:~/Neha_prog$ ./h myfile2.txt  
Deleted Successfully!  
neha@neha-Inspiron-3537:~/Neha_prog$
```

**Type**

:Aim: Write a Program to simulate DOS TYPE commnds

```
section .data  
fnotmsg db 'FILE NOT FOUND...',10  
fmmsg_len equ $-fnotmsg  
newline db 10  
section .bss  
fd_in resd 1  
fbuff resb 20  
fb_len equ $-fbuff  
act_len resd 1  
counter resb 1  
srcfile resb 80  
%macro print 2  
    mov eax,4  
    mov ebx,1  
    mov ecx,%1  
  
    mov edx,%2  
    int 0x80  
%endmacro  
section .text  
    global _start  
_start:  
  
    pop ecx          ;pop no. of arguments  
    pop ecx          ;pop exec command  
  
    pop ecx          ;pop source filename  
    mov edx,00
```

```
up:
    cmp byte[ecx+edx],0
    jz l1
    inc edx
    jmp up

l1:
    xor edi,edi
    mov esi,ecx
    ;save source filename

l2:
    mov al,byte[esi]
    mov byte[srcfile+edi],al
    inc edi
    inc esi
    dec edx
    jnz l2

;open the file for reading

    mov eax,5
    mov ebx,srcfile
    mov ecx,0
    mov edx,0777
    int 80h
    mov [fd_in],eax
    bt eax,31
    jnc readfile
    print fnotmsg,fnmsg_len
    jmp exit

readfile:
    mov eax,3
    mov ebx,[fd_in]
    mov ecx,fbuff
    mov edx,fb_len
    int 80h
    mov [act_len],eax
    cmp eax,0
    je nxt1
    ;write to the terminal

    print fbuff,[act_len]
```

```

jmp readfile

nxt1: mov eax,6      ;close source file
       mov ebx,[fd_in]
       int 80h
exit:           ;exit system call
       mov eax,1
       mov ebx,0
       int 0x80

```

**Output**

```

neha@neha-Inspiron-3537:~/Neha_prog$ nasm -f elf64 type.asm
neha@neha-Inspiron-3537:~/Neha_prog$ ld -m elf_i386 -s -o h type.o
neha@neha-Inspiron-3537:~/Neha_prog$ ./h myfile2.txt
Neha Atul Godse
neha@neha-Inspiron-3537:~/Neha_prog$
```

**myfile1**

Neha Atul Godse

**myfile2**

Neha Atul Godse

**Assignment 12 :** Write X86 ALP to find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character. Accept the data from the text file. The text file has to be accessed during Program\_1 execution and write FAR PROCEDURES in Program\_2 for the rest of the processing. Use of PUBLIC and EXTERN directives is mandatory.

;Aim:

- ; Write x86 ALP to find, a)Number of Blank spaces b)Number of lines c)Occurrence of a particular character.
- ; Accept data from a file.

```

section .data
    file_name db 'myfile.txt',0
    fnotmsg db 'FILE NOT FOUND...',10
    fmmsg_len equ $-fnotmsg
    openmsg db 'FILE OPENED SUCESSFULLY!',10
    omsg_len equ $-openmsg
    filemsg db 'FILE CONTENTS ARE:-',10
    fmsg_len equ $-filemsg
    opmsg db 'Enter character:',10
    opmsg_len equ $-opmsg
    msg1 db 'Number of blank spaces:'
```

```
len1 equ $-msg1
msg2 db 'Number of lines :'
len2 equ $-msg2
msg3 db 'Number of Occurrences :'
len3 equ $-msg3

newline db 10
bs dd 0h
nl dd 0h
chc dd 0h

section .bss
    final resb 8
    result resb 4
    counter resb 4
    fd_in resd 1
    fbuff resb 20
    fb_len equ $-fbuff
    act_len resd 1
    ch1 resb 1

;macro to write
%macro print 2
    mov eax,4
    mov ebx,1
    mov ecx,%1
    mov edx,%2
    int 0x80
%endmacro

;macro to read
%macro read 2
    mov eax,3
    mov ebx,0
    mov ecx,%1
    mov edx,%2
    int 0x80
%endmacro

section .text
    global _start
_start:
    print opmsg,opmsg_len
    read ch1,1
        ;open the file for reading
    mov eax,5
    mov ebx,file_name
    mov ecx,0
    mov edx,0777
    int 80h
    mov [fd_in],eax
```

```
        bt eax,31
        jnc conti1
        print fnotmsg,fmsg_len
        jmp exit
conti1:
        print openmsg,omsg_len
        print filemsg,fmsg_len
                           ;read from the file
readfile:
        mov eax,3
        mov ebx,[fd_in]
        mov ecx,fbuff
        mov edx,fb_len
        int 80h
        mov [act_len],eax
        mov dword[counter],eax
        cmp eax,0
        je nxt1
        mov esi,0
back2:
        cmp byte[fbuff+esi],' '
                           ;count no. of blank
                           ; spaces
        jne nxt3
        inc byte[bs]
nxt3:
        mov bl,byte[ch1]           ;count no. of
                           ; occerences of
                           ; particular character
        cmp byte[fbuff+esi],bl
        jne nxt2
        inc byte[chc]
nxt2:
        cmp byte[fbuff+esi],10      ;count no. of newlines
        jne nxt
        inc byte[nl]
nxt:
        inc esi
        dec byte[counter]
        jnz back2
        print fbuff,[act_len]
        jmp readfile
                           ;close file
nxt1:
        mov eax,6
        mov ebx,[fd_in]
```

```
int 80h

print msg1,len1
mov eax,dword[bs]           ;print no. of blank
                                ; spaces
mov dword[result],eax
call ascii
print final,8
print newline,1
print msg2,len2
mov eax,dword[nl]           ;print no. of newlines
mov dword[result],eax
call ascii
print final,8
print newline,1
print msg3,len3
mov eax,dword[chc]           ;print no. of occurrences
                                ;of particular character
mov dword[result],eax
call ascii
print final,8
print newline,1
exit:                         ;exit system call
                                ;function to convert
                                ;hex to ascii
mov eax,1
mov ebx,0
int 0x80

ascii:
                                ;function to convert
                                ;hex to ascii
mov esi,final
mov dword[counter],8
mov eax,dword[result]

bck2:
mov ebx,eax
rol ebx,04
mov eax,ebx
and bl,0Fh
cmp bl,9h
jbe next1
add bl,7h

next1:
add bl,30h
mov [esi],ebx
inc esi
dec dword[counter]
jnz bck2
ret
```

**myfile**

Hello World  
Microprocessor:

A microprocessor is a computer processor which incorporates the functions of a computer's central processing unit (CPU) on a single integrated circuit (IC), or at most a few integrated circuits.

**Output**

```
neha@neha-Inspiron-3537:~/Neha_prog$ nasm -f elf Assign12.asm
neha@neha-Inspiron-3537:~/Neha_prog$ ld -m elf_i386 -s -o h Assign12.o
neha@neha-Inspiron-3537:~/Neha_prog$ ./h
Enter character:
a
FILE OPENED SUCESSFULLY!
FILE CONTENTS ARE:-
Hello World
Microprocessor:
A microprocessor is a computer processor which incorporates the functions of a computer's central processing unit (CPU) on a single integrated circuit (IC), or at most a few integrated circuits.
Number of blank spaces:0000001B
Number of lines :00000007
Number of Occurrences :00000009
neha@neha-Inspiron-3537:~/Neha_prog$
```

**Assignment 13 :** Write x86 ALP to find the factorial of a given integer number on a command line by using recursion. Explicit stack manipulation is expected in the code.

;Aim:

; Calculate factorial of a given number(command line input)

section .data

msg1 db 'Factorial of a given number is :'

len1 equ \$-msg1

msg2 db 'Given number is :'

len2 equ \$-msg2

msg0 db 'Factorial of a given number is :1',10

```
len0 equ $-msg0
msg3 db 'Error',10
len3 equ $-msg3
newline db 10
factorial dd 1
section .bss
    final resb 8
    result resb 4
    counter resb 4
    num resb 4
    numh resb 4
    numlen resb 4
        ;macro to write
%macro print 2
    mov eax,4
    mov ebx,1
    mov ecx,%1
    mov edx,%2
    int 0x80
%endmacro
        ;macro to read
%macro read 2
    mov eax,3
    mov ebx,0
    mov ecx,%1
    mov edx,%2
    int 0x80
%endmacro
section .text
    global _start
_start:
    xor eax, eax
    mov dword[num],eax

    print msg2,len2
    pop ecx          ;pop no. of arguments
    pop ecx          ;pop exec command
    pop ecx          ;pop given number
    mov edx,00

up:
    cmp byte[ecx+edx],0
```

```
jz l1
inc edx
jmp up

l1:
mov esi,ecx           ;printing the number from command line
mov eax,dword[esi]
mov dword[num],eax
mov dword[numlen],edx
print num,dword[numlen]
print newline,1

mov esi,num
call packnum          ;convert the ascii number to hex
mov dword[numh],ebx
cmp dword[numh],0
jne next2
print msg0,len0       ;print factorial of 0
jmp exit

next2:
mov ecx,dword[numh]
call facto

mov edx,dword[factorial] ;print factorial of the number
mov dword[result],edx
call ascii
print msg1,len1
print final,8
print newline,1

exit:                  ;exit system call
mov eax,1
mov ebx,0
int 0x80

packnum:              ;function to convert from ascii to hex
mov eax,0
mov ecx,dword[numlen]
mov dword[counter],ecx
mov ebx,0

back:
mov al,[esi]
rol ebx,4
```

```
cmp al,39h
jbe next
sub al,7h
next:
sub al,30h
or bl,al
inc esi
dec dword[counter]
jnz back
ret
ascii:           ;function to convert hex to ascii
mov esi,final
mov dword[counter],8
mov eax,dword[result]
bck2:
mov ebx,eax
rol ebx,04
mov eax,ebx
and bl,0Fh
cmp bl,9h
jbe next1
add bl,7h
next1:
add bl,30h
mov [esi],ebx
inc esi
dec dword[counter]
jnz bck2
ret
facto:          ;function to find factorial of given number
push ecx
;push value from ecx into stack
cmp ecx,01
;if value equal to 1, stop recursion
jne next3
jmp exit1
next3:dec ecx
call facto      ;recursive call
exit1: pop ecx
;pop value from stack into ecx
mov eax,ecx
mul dword[factorial] ;perform multiplication
mov dword[factorial],eax ;store product into factorial
ret
```

**Output**

```
neha@neha-Inspiron-3537:~/Neha_prog$ nasm -f elf A13.asm
neha@neha-Inspiron-3537:~/Neha_prog$ ld -m elf_i386 -s -o h A13.o
neha@neha-Inspiron-3537:~/Neha_prog$ ./h 7
Given number is :7
Factorial of a given number is :0000013B0
neha@neha-Inspiron-3537:~/Neha_prog$
```



**May-2017**  
**Microprocessor**

S.E. (Computer) Semester - II (Course-2015)

**SPPU**  
**Solved Paper**  
**[5152]-569**

Time : Two Hours]

[Maximum Marks : 50]

N. B. :

- i) Attempt Q.1 or Q.2, Q.3 or Q.4, Q.5 or Q.6, Q.7 or Q.8.
- ii) Neat diagrams must be drawn wherever necessary.
- iii) Figures to the right side indicate full marks.
- iv) Assume suitable data, if necessary.

**Q.1 a) What is the use of following instructions ?**

[2]

- i) Wait (Refer page 2-54)
- ii) Lock (Refer page 2-31)

**b) Explain segment address translation in detail. (Refer section 4.2)**

[4]

**c) Draw and explain segment descriptor. (Refer section 4.2.1)**

[6]

**OR**

**Q.2 a) What is the use of Direction Flag ? (Refer section 3.8.1)**

[2]

**b) Draw and explain the system address and system segment registers.  
(Refer section 3.8)**

[4]

**c) Explain the following instructions, mention flags affected :**

[6]

- i) CWD (Refer page 2-16)
- ii) BT (Refer page 2-11)
- iii) LAHF (Refer page 2-27)

**Q.3 a) List the registers and data structures that are used in multitasking.  
(Refer section 6.1)**

[2]

**b) Differentiate between memory mapped I/O and I/O mapped I/O.  
(Refer section 3.4)**

[4]

**c) Explain what happens when an interrupt calls a procedure as an interrupt handler.  
(Refer section 8.6.1)**

[6]

**OR**

- Q.4** a) Write the two mechanisms that provide protection for I/O functions.  
**(Refer section 3.4)** [2]
- b) What is IDT and how to locate IDT ? **(Refer section 8.5)** [4]
- c) Explain the different exception conditions-Faults, Traps and Aborts.  
**(Refer section 8.1)** [6]
- Q.5** a) Write short note on "Task Switch Breakpoint".  
**(Not in New Syllabus)** [3]
- b) Write short note on "Protection within a V86 task". **(Refer section 7.3.2)** [4]
- c) Explain various debugging features of 80386.  
**(Not in New Syllabus)** [6]

**OR**

- Q.6** a) Write short note on "General Detect Fault". **(Not in New Syllabus)** [3]
- b) Which bit of EFLAGS indicates V86 mode ? Explain, how hardware and software cooperate with each other to emulate V86 mode ? **(Refer section 7.2)** [4]
- c) Explain, how test registers are used in testing TLB ?  
**(Not in New Syllabus)** [6]
- Q.7** a) Explain following signals **(Refer section 3.3.1)** [3]
- i) ADS#  
ii) READY#  
iii) NA#
- b) Write note on CLK2 and internal processor clock.  
**(Not in New Syllabus)** [4]
- c) Which data types are supported by 80387 ? **(Not in New Syllabus)** [6]

**OR**

- Q.8** a) Explain following signals  
BE0# through BE3#. **(Refer section 3.3.1)** [3]
- b) Explain following signals **(Not in New Syllabus)** [4]
- i) PEREQ  
ii) BUSY#  
iii) ERROR#
- c) Draw read cycle with pipelined address timing. **(Refer section 3.6)** [6]

**December-2017****Microprocessor**

S.E. (Computer) Semester - II (Course-2015)

**SPPU  
Solved Paper  
[5252]-569**

**Time : Two Hours****[Maximum Marks : 50]**

- Q.1** a) Explain immediate and register addressing mode with an examples.  
 (Refer section 1.6) [2]
- b) Explain with example SHL and ROL instructions.  
 (Refer pages 2 - 45 and 2 - 42) [4]
- c) Explain in detail the control registers of 80386. (Refer section 3.8.3) [6]
- OR**
- Q.2** a) Explain MSW. (Refer section 3.8.3) [2]
- b) Explain paging mechanism. (Refer section 4.3) [4]
- c) Explain the following instructions, mention flags affected : [6]
- i) LIDT (Refer page 2 - 29)
- ii) CLD (Refer page 2 - 14)
- iii) MOVS (Refer page 2 - 34)
- Q.3** a) What is CPL and RPL ? (Refer section 5.3.4.2) [2]
- b) Differentiate between memory mapped I/O and I/O mapped I/O.  
 (Refer section 3.4) [4]
- c) Draw and briefly explain task state segment. (Refer section 6.2) [6]
- OR**
- Q.4** a) When does a page fault occur ? [2]
- Ans. :** A page fault is a type of exception raised by computer hardware when a running program accesses a memory page that is not currently mapped by the Memory Management Unit (MMU) into the virtual address space of a process.
- b) Explain any two I/O privilege instructions. (Refer section 5.7) [4]
- c) Explain what happens when an interrupt calls a procedure as an interrupt handler.  
 (Refer section 8.6.1) [6]
- Q.5** a) What are the contents of various registers of processor 80386 after reset ?  
 (Refer section 3.2) [3]

- b)** How many debug registers are present in 80386 ? List and draw all of them.  
 (Not in New Syllabus) [4]
- c)** With neat diagram explain the process of linear address formation in V86 mode.  
 (Refer section 7.2.2) [6]

**OR**

- Q.6 a)** Write short note on "instruction address breakpoint".  
 (Not in New Syllabus) [3]

- b)** What all initializations required to start processor in real mode after reset ? [4]

**Ans. :** By default, after reset, the processor starts in real mode.

- c)** With neat diagram explain "entering and leaving V86 mode".  
 (Refer section 7.4) [6]

- Q.7 a)** Explain HOLD and HLDA signals of 80386DX. (Refer section 3.3.3) [3]

- b)** List various bus states when address pipelining is used. [4]

**Ans. :** Bus states : T1 - First clock of a non - pipelined bus cycle  
 (386<sup>TM</sup> DX drives new address and asserts  $\overline{\text{ADS}}$ ).

T2 - Subsequent clocks of a bus cycle when  $\overline{\text{NA}}$  has not been sampled asserted in the current bus cycle.

T21 - Subsequent clocks of a bus cycle when  $\overline{\text{NA}}$  has been sampled asserted in the current bus cycle but there is not yet an internal bus request pending (386 DX will not drive new address or asserts  $\overline{\text{ADS}}$ ).

T2P - Subsequent clocks of a bus cycle when  $\overline{\text{NA}}$  has been sampled asserted in the current bus cycle and there is an internal bus request pending (386 DX drives new address and asserts  $\overline{\text{ADS}}$ ).

T1P - First clock of a pipelined bus cycle.

Ti - Idle state

Th - Hold acknowledge state (386 DX asserts HLDA).

- c)** Draw read cycle with non - pipelined address timing. (Refer section 3.6) [6]

**OR**

- Q.8 a)** Explain the following signals :  
 i) NMI ii) INTR iii) RESET (Refer section 3.3.2) [3]

- b)** Draw and explain 80387 register stack. (Not in New Syllabus) [4]

- c)** Draw 'write cycle with pipelined address timing'. (Refer section 3.6) [6]

**May-2018  
Microprocessor**

S.E. (Computer) Semester - II (Course-2015)

**SPPU  
Solved Paper  
[5352]-569**

Time : Two Hours]

[Maximum Marks : 50

- Q.1** a) Explain immediate and register addressing mode with an example. (Refer section 1.6) [2]
- b) Draw and explain the flag register of 80386. (Refer section 3.8) [4]
- c) Draw and explain segment descriptor. (Refer section 4.2.1) [6]
- OR**
- Q.2** a) What is the use of interrupt flag. (Refer section 3.8.1) [2]
- b) Explain paging mechanism. (Refer section 4.3) [4]
- c) Draw and explain the 80386 address translation mechanism considering PG bit in CR0 in set. (Refer section 4.3) [6]
- Q.3** a) What is CPL and RPL ? (Refer section 5.3.4) [2]
- b) Explain interrupt no. 0 and 4. (Refer section 8.8) [4]
- c) Explain the role of task register in multitasking and the instructions used to modify and read TR. (Refer section 6.4) [6]
- OR**
- Q.4** a) List five aspects of protection in the 80386. (Refer section 5.2) [2]
- b) Write a short note on 'I/O permission bit map'. (Refer section 5.6) [3]
- c) Draw and explain TSS. (Refer section 6.2) [7]
- Q.5** a) Write short note on Virtual 8086 mode. (Refer section 7.1) [3]
- b) Explain software initializations required for protected mode. (Not in New Syllabus) [4]
- c) Draw and explain structure of the TLB. (Not in New Syllabus) [6]
- OR**
- Q.6** a) What are the contents of various registers of processor 80386 after reset ? (Refer section 3.2) [3]
- b) Explain entering and leaving V86 mode. (Refer section 7.4) [4]
- c) Draw and explain debug registers of the 80386. (Not in New Syllabus) [6]

**Q.7 a) Explain the following signals :**

i) W/R#   ii) D/C#   iii) M/IO# (Refer section 3.3) [3]

**b) Explain any four 80387 constant instructions. (Not in New Syllabus)** [4]

**c) Draw read cycle with non - pipelined address timing. (Refer section 3.6)** [6]

**OR**

**Q.8 a) Explain the following signals :**

i) INTR#   ii) NMI#   iii) RESET# (Refer section 3.3) [3]

**b) Draw and explain 80387 register stack. (Not in New Syllabus)** [4]

**c) Explain any six 80387 data transfer instructions. (Not in New Syllabus)** [6]

**December-2018**

**Microprocessor**

S.E. (Computer) Semester - II (Course-2015)

**SPPU  
Solved Paper  
[5459]-193**

**Time : 2 Hours**

**[Maximum Marks : 50]**

**Q.1 a) With the help of neat diagram explain how logical address is converted into physical address ? Assume paging mechanism is disabled. (Refer section 4.2)** [6]

**b) Explain any three control transfer instructions of 80386. (Refer section 2.2)** [6]

**OR**

**Q.2 a) Explain how linear address is converted into physical address by 80386 memory management. (Refer section 4.4)** [6]

**b) What is the use of following instructions in 80386 ? Mention which flags gets effected with each instruction :** [6]

**ADC, DIV, CMP (Refer section 2.2)**

**Q.3 a) With the help of suitable diagram, explain how call gate descriptor is used to change the privilege levels in protected mode ? (Refer section 5.3.4.2)** [6]

**b) Explain the procedure of handling interrupts in protected mode.  
(Refer section 8.6)** [6]

**OR**

**Q.4 a) What is the role of TSS in multitasking ? Explain I/O permission bitmap in TSS.  
(Refer sections 6.2 and 5.7)** [6]

**b) Draw the format of interrupt gate and trap gate descriptor. What is the difference between them ? (Refer sections 8.5 and 8.6)** [6]

**Q.5 a)** What is the role of DR0 to DR3 registers in debugging ? Explain task switch breakpoint. (Refer section 3.8) [4]

**b)** What are content of CR0 register after RESET in 80386 ? Explain all related bits. (Refer section 9.2) [3]

**c)** Explain linear address formation in virtual mode of 80386. (Refer section 7.3) [6]

**OR**

**Q.6 a)** Explain any four debugging features of 80386. (Not in New Syllabus) [4]

**b)** List any three differences between Virtual 86 mode and 8086. (Refer section 7.1) [3]

**c)** With the help of neat diagram explain format of DR6 register. (Refer section 3.8) [6]

**Q.7 a)** Compare pipelined and Non-pipelined bus cycle. [2]

**Ans.** : Non-pipelined machine cycles do not overlap. In pipelined machine cycles, the address of the next cycle is overlapped with the data transfer of the current cycle. The pipelining increases the amount of time required for the memory or I/O device to respond.

**b)** Explain the following signals of 80386 : [6]

M/IO#, W/R#, READY# (Refer section 3.3)

**c)** Explain the following instructions of 80387 : [5]

FLD, FSQRT, FLDZ, FBSTP (Not in New Syllabus)

**OR**

**Q.8 a)** Explain any two instructions used in 80387 to pop data from its stack registers. (Not in New Syllabus) [2]

**b)** With the help of neat diagram, explain the pipelined read bus cycle. (Refer section 3.6) [6]

**c)** When WAIT state is required in 80386 read bus cycle ? Explain with neat diagram. (Refer section 3.6) [6]

**May-2019  
Microprocessor**

S.E. (Computer) Semester - II (Course-2015)

**SPPU  
Solved Paper  
[5559]-193**

Time : 2 Hours

[Maximum Marks : 50]

**Q.1 a)** List fundamental data types of 80386.  
(Refer section 1.7) [2]

b) *Describe following different flags defined in 80386 processor*  
a) DF b) VM c) NT d) RF (Refer section 1.4) [4]

c) *Explain shift and rotate instructions of 80386.*  
(Refer pages 2-31, 2-32, 2-34, 2-35 and 2-42) [6]

OR

**Q.2** a) *Draw and explain the format of a selector.*  
(Refer section 4.2) [2]

b) *List and explain control registers of 80386.*  
(Refer section 3.8.3) [4]

c) *With help of diagram explain the 80386 mechanism to translate logical address to linear address.* (Refer section 4.2) [6]

**Q.3** a) *List aspects of protection related to pages.* (Refer section 5.4) [2]

b) *With appropriate diagram explain the concept of privilege levels in 80386.*  
(Refer section 5.3.3) [4]

c) *How call gate descriptor is used to locate the procedure in another code segment ?  
How protection is provided ?*  
(Refer section 5.3.5.2) [6]

OR

**Q.4** a) *Define "Faults".* (Refer section 8.1) [2]

b) *Explain "How 80386 identifies interrupts ?"* (Refer section 8.2) [4]

c) *By which two ways, 80386 allows input/output to be performed ? Explain each in details.* (Refer section 3.4) [6]

**Q.5** a) *Explain features of "Virtual 8086 mode".* (Refer section 7.1) [3]

b) *Explain 80386 processor state after RESET.* (Refer section 3.2) [4]

c) *What all initializations required to start processor in protected mode after reset ?*  
(Not in New Syllabus) [6]

OR

**Q.6** a) *Write a short note on "Switching to protected mode".*  
(Not in New Syllabus) [2]

b) *List the features of 80386 architecture that supports debugging.*  
(Not in New Syllabus) [5]

c) *With the necessary diagrams explain entering and leaving V86 mode ?*  
(Refer section 7.4) [6]

- Q.7** a) Draw and explain read cycle with non-pipelined address timing.  
 (Refer section 3.6) [8]
- b) Which data types are supported by 80387 ? (Not in New Syllabus) [5]
- OR
- Q.8** a) Draw and explain write cycle with pipelined address timing.  
 (Refer section 3.6) [8]
- b) The 80387 instructions are divided into which functional groups ? Explain with one example of each. (Not in New Syllabus) [5]

**December-2019**  
**Micropocessor**

S.E. (Computer) Semester - II (Course-2015)

**SPPU**  
**Solved Paper**  
**[5668]-189**

Time : Two Hours

[Maximum Marks : 50]

- Q.1** a) List and explain coprocessor interface instructions of 80386. (Refer section 3.8) [2]
- b) With the help of diagram explain 80386 applications register set.  
 (Refer section 1.4) [4]
- c) Explain how linear address 0080400A H will be translated into physical address using paging mechanism. Whether the address generated will be the same to linear address ? (Refer section 4.3) [6]

**Ans. :** Linear address

31	Directory	22 21	Page	12 11	Offset	0
0000	0000	10	00 0000	0100	0000 0000	1010

- Index to page directory = 10. This entry in the page directory gives the base address of the page table.
- Index to page table = 100. This entry in the page table gives the base address of the page frame.
- Index to page frame = 1010. Selects 1010<sup>th</sup> (10<sup>th</sup> byte) of memory from the page frame.
- The physical address generated will be different from the linear address.

OR

- Q.2** a) Explain LEA and XLAT instructions. (Refer Pages 2-27 and 2-55) [2]
- b) Draw and explain EFLAGS register of 80386. (Refer section 3.8) [4]

- c) With the help of diagram explain the 80386 mechanism to translate logical address to linear address and linear to physical address. (Refer section 4.3) [6]
- Q.3**
- a) List aspects of protection related to pages. (Refer section 5.4) [2]
  - b) Write a short note on "Multitasking" feature of 80386. (Refer section 6.1) [4]
  - c) List different sources of interrupts and explain different ways by which 80386 can enable and disable interrupts. (Refer sections 8.1 and 8.3 ) [6]
- OR
- Q.4**
- a) Define DPL, RPL and CPL. (Refer section 5.3.4) [2]
  - b) Write a short note on "Task Linking". (Refer section 6.7) [4]
  - c) List mechanism which provide protection for I/O functions and explain the role of IOPL in providing protection for I/O functions. (Refer section 5.6) [6]
- Q.5**
- a) Write a short note on "Virtual 8086 mode". (Refer section 7.1) [3]
  - b) Explain 80386 processor state after RESET.  
(Refer section 3.2) [4]
  - c) What all initializations required to start processor in real mode after reset ?  
(Not in New Syllabus) [6]
- OR
- Q.6**
- a) Explain, how test registers are used in testing TLB ? (Not in New Syllabus) [7]
  - b) What all initializations required to start processor in protected mode after rest ?  
(Not in New Syllabus) [6]
- Q.7**
- a) Explain HOLD and HLDA signals of 80386 DX. (Refer section 3.3) [4]
  - b) Draw and explain 80387 register stack. (Not in New Syllabus) [4]
  - c) Draw and explain bus states and transitions when address pipelining is not used.  
(Refer section 3.6) [5]
- OR
- Q.8**
- a) List various bus states when address pipelining is used. (Refer section 3.6) [4]
  - b) Which data types are supported by 80387 ? (Not in New Syllabus) [4]
  - c) Draw write cycle with non-pipelined address timing. (Refer section 3.6) [5]



# **SOLVED MODEL QUESTION PAPER (In Sem)**

## **Microprocessor**

S.E. (Computer) Semester - IV (As Per 2019 Pattern)

Time : 1 Hour]

[Maximum Marks : 30

N. B. :

- i) Attempt Q.1 or Q.2, Q.3 or Q.4.
- ii) Neat diagrams must be drawn wherever necessary.
- iii) Figures to the right side indicate full marks.
- iv) Assume suitable data, if necessary.

**Q.1** a) With the help of diagram explain 80386 applications register set.

(Refer section 1.4) [4]

b) List the features of 80386DX. (Refer section 1.2) [3]

c) Describe 80386 flag register with significance of each and every bit in detail.  
(Refer section 1.4) [8]

**OR**

**Q.2** a) Enlist and explain any three addressing modes of 80386. (Refer section 1.6) [6]

b) What is BIU in 80386 processor ? What are the functions of BIU ?  
(Refer section 1.3) [4]

c) Explain shift and rotate instructions of 80386. (Refer section 2.2) [5]

**Q.3** a) Explain 80386 processor state after RESET. (Refer section 3.2) [3]

b) Explain HOLD and HLDA signals of 80386DX. (Refer section 3.3) [4]

c) Draw read cycle with non - pipelined address timing. (Refer section 3.6) [8]

**OR**

**Q.4** a) Explain the significance of the following signals with relevance to 80386

a.  $\overline{BE}_0$  to  $\overline{BE}_3$       b.  $\overline{BS}_{16}$  to NA with their interdependency  
c. PEREQ      d.  $\overline{ERROR}$       e. READY (Refer section 3.3) [5]

b) Differentiate between memory mapped I/O and I/O mapped I/O.  
(Refer section 3.4) [3]

c) What is MSW (Machine Status Word) in 80386 ? Draw its format.  
(Refer section 3.8) [7]

# **SOLVED MODEL QUESTION PAPER (End Sem)**

## **Microprocessor**

S.E. (Computer) Semester - IV (As Per 2019 Pattern)

Time :  $2\frac{1}{2}$  Hours]

[Maximum Marks : 70]

N. B. :

- i) Attempt Q.1 or Q.2, Q.3 or Q.4, Q.5 or Q.6, Q.7 or Q.8.
- ii) Neat diagrams must be drawn wherever necessary.
- iii) Figures to the right side indicate full marks.
- iv) Assume suitable data, if necessary.

**Q.1** a) Draw and explain the protected mode addressing mechanism.  
(Refer section 4.1) [8]

b) What do you mean by paging ? Assume that paging is enabled. Explain the virtual address translation process to access operand from the memory.  
(Refer section 4.3) [10]

**OR**

**Q.2** a) List and explain the different descriptor table register used in protected mode of 80386 processor. (Refer section 4.2) [10]

b) Draw and explain how 80386 microprocessor translates logical address into linear address. (Refer section 4.2) [8]

**Q.3** a) List five aspects of protection in the 80386. (Refer section 5.2) [5]  
b) 80386 is currently executing program from code segment having PL as 2. If it needs to access the code from PL0, is it possible ? If yes, which are the methods used for the same ? If no, justify your answer. (Refer section 5.3) [12]

**OR**

**Q.4** a) What is DPL, RPL and CPL ? (Refer section 5.3) [3]  
b) What are different privileged and IOPL sensitivity instructions of 80386 processor ? Explain in short each instruction. (Refer section 5.7) [8]

c) List mechanism which provide protection for I/O functions and explain the role of IOPL in providing protection for I/O functions. (Refer section 5.6) [6]

**Q.5** a) Write a short note on "Multitasking" feature of 80386. (Refer section 6.1) [4]  
b) Explain TSS with the help of diagram in detail. (Refer section 6.2) [8]

- c) With neat diagram explain "entering and leaving V86 mode". (Refer section 7.4) [6]

OR

- Q.6 a) What is the difference between TSS descriptor and task gate descriptor ? Explain with respect to multitasking. (Refer section 6.5) [8]
- b) With neat diagram explain the process of linear address formation in V86 mode. (Refer section 7.2) [6]
- c) When is the 'back link' entry in TSS valid ? What is the purpose of this entry ? (Refer section 6.7) [4]

- Q.7 a) Explain the different exception conditions-Faults, Traps and Aborts. (Refer section 8.1) [6]
- b) Explain the procedure of handling interrupts in protected mode. (Refer section 8.6) [6]
- c) Give the comparison between microprocessor and microcontroller. (Refer section 9.1) [5]

OR

- Q.8 a) Draw the format of interrupt gate and trap gate descriptor. What is the difference between them ? (Refer section 8.5) [6]
- b) With the help of neat diagram explain the internal block diagram of 8051. (Refer section 9.3) [8]
- c) Mention any two applications of 8051 microcontroller. (Refer section 9.1) [3]



## *Notes*